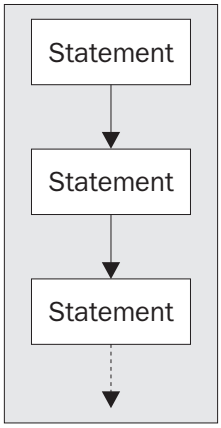
4

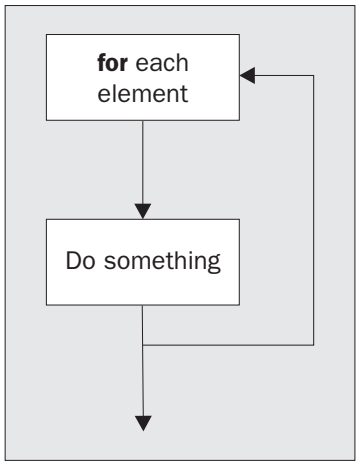
Loops and Decisions

Most of the programs so far have been very simply structured – they've done one statement after another in turn. If we were to represent statements as boxes, our programs would look like this:

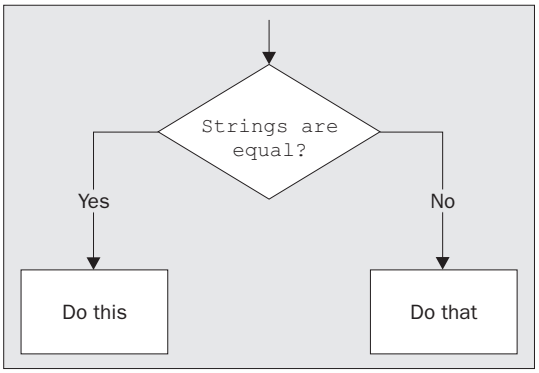


This sort of diagram is called a **flow chart**, and programmers have used them for a long time to help design their programs. They're considered a bit passé these days, but they're still useful. The path Perl (or any other language) takes by following the arrows is called the **flow of execution** of the program. Boxes denote statements (or a single group of statements), and diamonds denote tests. There are also a whole host of other symbols for magnetic tapes, drum storage, and all sorts of wonderful devices, now happily lost in the mists of time.

In the last chapter, we introduced the **for loop** to our growing repertoire of programming tools. Instead of taking a straight line through our program, perl did a block of statements again and again for each element of the list. The for loop is a **control structure** – it controls the flow of execution. Instead of going in a straight line, we can now go around in a loop:



Not only that, but we can choose our path through the program depending on certain things, like the comparisons we looked at in Chapter 2. For instance, we'll do something **if** two strings are equal:



We'll take a look at the other sorts of control structures we have in Perl, for example, structures that do things **if** or **unless** something is true. We'll see structures that do things **while** something is true or **until** it is true. Structures that loop **for** a certain number of times, or **for each** element in a list. Each of the words in bold is a Perl keyword, and we'll examine each of them in this chapter.

Deciding If...

Let's first extend our previous currency conversion program a little, using what we learned about hashes from the previous chapter.

Try It Out : Multiple Currency Converter

We'll use a hash to store the exchange rates for several countries. Our program will ask for a currency to convert from, then a currency to convert to, and the amount of currency we would like to convert:

```
#!/usr/bin/perl
# convert1.plx
use warnings;
use strict;

my ($value, $from, $to, $rate, %rates);
%rates = (
    pounds          => 1,
    dollars         => 1.6,
    marks           => 3.0,
    "french francs" => 10.0,
    yen             => 174.8,
    "swiss francs"  => 2.43,
    drachma         => 492.3,
    euro            => 1.5
);

print "Enter your starting currency: ";
$from = <STDIN>;
print "Enter your target currency: ";
$to = <STDIN>;
print "Enter your amount: ";
$value = <STDIN>;

chomp($from,$to,$value);
$rate = $rates{$to} / $rates{$from};

print "$value $from is ",$value*$rate," $to.\n";
```

Here's a sample run of this program:

```
> perl convert1.plx
Enter your starting currency: dollars
Enter your target currency: euro
Enter your amount: 200
200 dollars is 187.5 euro.
>
```

Let's first see how this all works, and then we'll see what's wrong with it.

How It Works

The first thing we do is to declare all our variables. You don't have to do this at the start of the program, but it helps us organize:

```
my ($value, $from, $to, $rate, %rates);
```

Note that you **do** need to put brackets when you're declaring more than one variable at once. Once again it's a question of precedence – `my` has a lower precedence than a comma. Now we can set out our rates table:

```
%rates = (
    pounds          => 1,
    dollars         => 1.6,
    marks           => 3.0,
    "french francs" => 10.0,
    yen             => 174.8,
    "swiss francs"  => 2.43,
    drachma         => 492.3,
    euro            => 1.5
);
```

Using `<STDIN>` as we did in the last chapter to read a line from the console, we'll ask for two currencies and the amount:

```
print "Enter your starting currency: ";
$from = <STDIN>;
print "Enter your target currency: ";
$to = <STDIN>;
print "Enter your amount: ";
$value = <STDIN>;
```

Now we have a problem. We read in entire lines, and lines end with a new line character. In order for us to look up the currencies in the hash and to display the currencies back properly, we have to get rid of this. The way we do this is to use the `chomp` operator on the values we've just read in – `chomp` gets rid of a final new line if one is present but does nothing if there is no new line. For instance, it turns `"euro\n"`, which isn't in the hash, into `"euro"`, which is:

```
chomp($from,$to,$value);
```

Note that it actually changes the value of the variables passed to it. Instead of returning the new string, it modifies the variable, and actually returns the number of new lines removed. Now we don't actually *need* to remove the new line from `$value`. All we do is use it in calculation and perl will convert it to a number. When we do that the new line will automatically be lost. However, since we might want to print out, '200 marks', we need to make sure there is no new line after '200'.

Next we calculate the exchange rate, which is just the rate of the target currency divided by the rate of the initial currency:

```
$rate = $rates{$to} / $rates{$from};
```

Finally, we multiply the value by the exchange rate and print out the results:

```
print "$value $from is ",$value*$rate," $to.\n";
```

Now, this is all well and good, but watch what happens if one of the currencies we ask for isn't in the hash:

```
> perl convert1.plx
Enter your starting currency: dollars
Enter your target currency: lira
Enter your amount: 300
Use of uninitialized value in division (/) at convert1.plx line 26, <STDIN> line 3.
300 dollars is 0 lira.
```

What was that warning? Well, the message tells us that something we used during the division at line 26:

```
$rate = $rates{$to} / $rates{$from};
```

was not defined. We know in this case it's the target currency. `$rates{lira}` is not in the hash. When the other currency is undefined, then we get more serious problems:

```
> perl convert1.plx
Enter your starting currency: lira
Enter your target currency: dollars
Enter your amount: 132000
Use of uninitialized value in division (/) at convert1.plx line 26, <STDIN> line 3.
Illegal division by zero at convert.plx line 26, <STDIN> line 3.
```

This time the other side of the division is undefined, and Perl converts the undefined value to zero. Unfortunately, you can't divide by zero. To solve both these problems we really want to be able to stop the program when an unknown currency is entered – that is, if a certain string does not exist in the hash.

We now need to find out if something happened, and perform a certain action if it did. This expression is called an **if statement**. Here's what an if statement looks like in Perl:

```
if ( <some test> ) {
   <do something>
}
```

In our case, we want to ensure a hash key exists. Now Perl isn't a difficult language; to sort a list, you use the `sort` keyword, to find the length of a string, you use the `length` keyword. To see if a hash key exists, we use the aptly named `exists` keyword for our test – `exists $rates{$to}` and `exists $rates{$from}`:

Try It Out : Testing Invalid Keys

Let's now put the `if` statement to use and test to make sure we are given valid, existing keys:

```
#!/usr/bin/perl
# convert2.plx
use warnings;
use strict;
```

```
my ($value, $from, $to, $rate, %rates);
%rates = (
    pounds          => 1,
    dollars         => 1.6,
    marks           => 3.0,
    "french francs" => 10.0,
    yen             => 174.8,
    "swiss francs"  => 2.43,
    drachma         => 492.3,
    euro            => 1.5
);

print "Enter your starting currency: ";
$from = <STDIN>;
print "Enter your target currency: ";
$to = <STDIN>;
print "Enter your amount: ";
$value = <STDIN>;

chomp($from,$to,$value);

if (not exists $rates{$to}) {
   die "I don't know anything about $to as a currency\n";
}
if (not exists $rates{$from}) {
   die "I don't know anything about $from as a currency\n";
}

$rate = $rates{$to} / $rates{$from};

print "$value $from is ",$value*$rate," $to.\n";
```

Now if we enter a currency that is unknown, we get our own error message and the program ends:

```
> perl convert2.plx
Enter your starting currency: dollars
Enter your target currency: lira
Enter your amount: 300
I don't know anything about lira as a currency
>
```

How It Works

After we've got the currency names, and before we try to divide, we use the following code to see if the currencies are valid entries in the hash. We do two very similar comparisons, one for the start currency and one for the target, so let's just examine one of them:

```
if (not exists $rates{$to}) {
   die "I don't know anything about $to currency\n";
}
```

This is our `if` statement: if the entry `$to` does not exist in the `%rates` hash, then we give an error message. `die` is a way of making Perl print out an error message and finish the program. It also reports to the operating system – Windows, Unix, or whatever it may be, that the program finished with an error. The part in brackets, `not exists $rates{$to}` is known as the **condition**. If that condition is true, we do the action in braces and terminate the program.

How do we construct conditions, then?

Logical Operators Revisited

The `if` statement, and all the other control structures we're going to visit in this chapter, test to see if a condition is true or false. They do this using the Boolean logic mentioned in Chapter 2, together with Perl's ideas of true and false. To remind you of these:

- An empty string, `""`, is false.
- The number zero and the string `"0"` are both false.
- An empty list, `()`, is false.
- The undefined value is false.
- Everything else is true.

However, you need to be careful for a few traps here. A string containing invisible characters, like spaces or new lines, is true. A string that isn't `"0"` is true, even if its numerical value is zero, so `"0.0"` for instance, is true.

Larry Wall has said that programming Perl is an empirical science – you learn things about it by trying it out. Is `(())` a true value? You can look it up in books and the online documentation, or you can spend a few seconds writing a program like this:

```
#!/usr/bin/perl
use strict;
use warnings;

if ( (()) ) {
   print "Yes, it is.\n";
}
```

This way you get the answer right away, with a minimum of fuss. (If you're interested, it isn't a true value.) We'll see in later chapters how to make this sort of test program easier and faster to write, but what we know now is sufficient to test the hypothesis. I'm continually writing these little programs to check out facets of Perl I'm not sure about. Try getting into the habit of doing it, too.

We've also seen that conditional operators can test things out, returning `1` if the test was successful and the undefined value if it was not. Let's see more of the things we can test.

Comparing Numbers

We can test whether one number is bigger, smaller, or the same as another. Assuming we have two numbers, stored in the variables `$a` and `$b`, here are the operators we can use for this:

`$a > $b`	`$a` is greater than `$b`
`$a < $b`	`$a` is less than `$b`
`$a == $b`	`$a` has the same numeric value as `$b`
`$a != $b`	`$a` does not have the same numeric value as `$b`

Don't forget that the numeric comparison needs a doubled equals sign (==), so that Perl doesn't think you're trying to set `$a` to the value of `$b`:

Also remember that Perl converts `$a` and `$b` to numbers in the usual way. It reads numbers or decimal points from the left for as long as possible, ignoring initial spaces, and then drops the rest of the string. If no numbers were found, the value is set to zero.

Try It Out : Guess My Number

This is a very simple guessing game. The computer has a number, and the user has to guess what it is. If the user doesn't guess correctly, the computer gives a hint. As we learn more about Perl, we'll add the opportunity to give more than one try and to pick a different number each game:

```
#!/usr/bin/perl
# guessnum.plx
use warnings;
use strict;

my $target = 12;
print "Guess my number!\n";
print "Enter your guess: ";
my $guess = <STDIN>;

if ($target == $guess) {
   print "That's it! You guessed correctly!\n";
   exit;
}
if ($guess > $target) {
   print "Your number is bigger than my number\n";
   exit;
}
if ($guess < $target){
   print "Your number is less than my number\n";
   exit;
}
```

Let's have a few go's at it:

```
> perl guessnum.plx
Guess my number!
Enter your guess: 3
Your number is less than my number
> perl guessnum.plx
Guess my number!
Enter your guess: 15
Your number is bigger than my number
> perl guessnum.plx
Guess my number!
Enter your guess: 12
That's it! You guessed correctly!
>
```

How It Works

First off, we set up our secret number. OK, at the moment it's not very secret, since it's right there in the source code, but we can improve on this later. After this, we get a number from the user:

```
my $guess = <STDIN>;
```

Then we do three sorts of comparisons with the numeric operators we've just seen. We use the basic pattern of the if statement again, `if ( <condition> ) { <action> }`:

```
if ($target == $guess) {
   print "That's it! You guessed correctly!\n";
   exit;
}
```

Since only one of the tests can be true – the user's number can't be both smaller than our number and the same as it – we may as well stop work after a test was successful. The `exit` operator tells perl to stop the program completely. You can optionally give `exit` a number from 0 to 255 to report back to the operating system. Traditionally, 0 denotes success and anything else is failure. By default, `exit` reports success.

Comparing Strings

When we're comparing strings, we use a different set of operators to do the comparisons:

`$a gt $b`	`$a` sorts alphabetically after `$b`
`$a le $b`	`$a` sorts alphabetically before `$b`
`$a eq $b`	`$a` is the same as `$b`
`$a ne $b`	`$a` is not the same as `$b`

Here's a very simple way of testing if a user knows a password. Don't use this for anything you value, since the user can just read the source code to find it!

```
#!/usr/bin/perl
# password.plx
use warnings;
use strict;

my $password = "foxtrot";
print "Enter the password: ";
my $guess = <STDIN>;
chomp $guess;
if ($password eq $guess) {
   print "Pass, friend.\n";
}
if ($password ne $guess) {
   die "Go away, imposter!\n";
}
```

Here's our security system in action:

```
> perl password.plx
Enter the password: abracadabra
Go away, imposter!
> perl password.plx
Enter the password: foxtrot
Pass, friend.
>
```

How It Works

As before, we ask the user for a line:

```
my $guess = <STDIN>;
```

Just a warning: this is a horrendously bad way of asking for a password, since it's echoed to the screen, and everyone looking at the user's computer would be able to read it. Even though you won't be using a program like this, if you ever do need to get a password from the user, the Perl FAQ provides a better method. In `perlfaq8`, type `perldoc -q password` to find it.

```
chomp $guess;
```

We must never forget to remove the new line from the end of the user's data. We didn't need to do this for numeric comparison, because Perl would remove that for us anyway during conversion to a number. Otherwise, even if the user had put the right password in, Perl would have tried to compare `"foxtrot"` with `"foxtrot\n"` and it could never be the same:

```
if ($password ne $guess) {
   die "Go away, imposter!\n";
}
```

Then if the password we have isn't the same as the user's input, we send out a rude message and terminate the program.

Other Tests

What other tests can we perform? We've seen `exists` at the beginning of this chapter, for determining whether a key exists in a hash. We can test if a variable is defined (It must contain something other than the undefined value), by using `defined`:

```
#!/usr/bin/perl
# defined.plx
use warnings;
use strict;

my ($a, $b);
$b = 10;
if (defined $a) {
   print "\$a has a value.\n";
}
if (defined $b) {
   print "\$b has a value.\n";
}
```

Not surprisingly, the result we get is this:

```
>perl defined.plx
$b has a value.
>
```

You can use this to avoid the warnings that occur when you try and use a variable that doesn't have a value. If we'd tried to say `if ($a == $b)`, Perl would have said:

```
Use of uninitialized value in numeric eq (==)
```

So we have our basic comparisons. Don't forget that some functions will return a true value if they were successful and the undefined value if they were not. You will often want to check whether the return value of an operation (particularly one that relates to the operating system) is true or not.

How do you actually test whether something is a true value or not? You may want to see if a user's input isn't empty after being chomped, for example. Well, don't do it like this:

```
my $true = (1 == 1);
if ($a == $true) { … }
```

The whole point of `if` is that it does the action if something is true. You should just say `if ($a) { ... }`

Logical Conjunctions

We also saw in Chapter 2 that we can join together several tests into one, by the use of the logical operators. Here's a summary of those:

`$a and $b`	True if both `$a` and `$b` are true.
`$a or $b`	True if either of `$a` or `$b`, or both are true.
`not $a`	True if `$a` is not true.

In fact, we saw `not` earlier:

```
if (not exists $rates{$to})
```

There is also another set of logical operators: `&&` for `and`, `||` for `or`, and `!` for `not`. However, I find the first set easier to read and understand. Don't forget there is a difference in precedence between the two – `and`, `or`, and `not` all have lower precedence than their symbolic representations.

Running Unless...

There's another way of saying `if (not exists $rates{$to})`. As always in Perl, there's more than one way to do it. Some people prefer to think 'if this doesn't exist, then { ... }', but other people think 'unless this does exist, then { ... }'. Perl caters for both sets of thought patterns, and we could just as easily have written this:

```
unless (exists $rates{$to}) {
   die "I don't know anything about {$to} as a currency\n";
}
```

The psychology is different, but the effect is the same: `unless ($a)` is effectively `if (not ($a))`. We'll see later how Perl provides a few alternatives for these control structures to help them more effectively fit the way you think.

Statement Modifiers

When we're talking in English, it's quite normal for us to say

- If this doesn't exist, then this happens, or
- Unless this exists, this happens.

Similarly, it's also quite natural to reverse the two phrases, saying

- This happens, if this doesn't exist, or
- This happens unless this exists.

Going back to our currency converter example, convert2.plx, we could turn around the if statements within to read:

```
die "I don't know anything about $rates{$to} as a currency\n"
   if not exists $rates{$to};
```

Notice how the syntax here is slightly different, it's *<action> if <condition>*. There is no need for brackets around the condition, and there are no braces around the action. Indeed, the indentation isn't part of the syntax, so we could even put the whole statement on one line. Only a single statement will be covered by the condition. The condition modifies the statement, and so is called a **statement modifier**.

We can turn `unless` into a statement modifier, too. So instead of this:

```
if (not exists $rates{$to}) {
   die "I don't know anything about {$to} as a currency\n";
}
if (not exists $rates{$from}) {
   die "I don't know anything about{$from} as a currency\n";
}
```

you may find it more natural to write this:

```
die "I don't know anything about $to as a currency\n"
   unless exists $rates{$to};
die "I don't know anything about $from as a currency\n"
   unless exists $rates{$from};
```

Sure enough, if you swap those lines into `convert2.plx`, you'll get the same results.

Using Logic

There is yet another way to do something if a condition is true, and we saw it briefly in Chapter 2. By using the fact that perl stops processing a logical conjunction when it knows the answer for definite, we can create a sort of `unless` conditional:

```
exists $rates{$to}
   or die "I don't know anything about {$to} as a currency\n";
```

How does this work? Well, it's reliant on the fact that perl uses lazy evaluation to give a logical conjunction its value. If we have the statement `X or Y`, then if `X` is true, it doesn't matter what `Y` is, so perl doesn't look at it. If `X` isn't true, perl has to look at `Y` to see whether or not that's true. So if `$rates{$to}` exists in the hash, then our currency converter won't die with an error message. Instead, it will do nothing and continue executing the next statement.

This form of conditional is most often used when checking that something we did succeeded or returned a true value. We will see it often when we're handling files.

To create a positive `if` conditional this way, use `and` instead of `or`. For example, to add one to a counter if a test is successful, you may say:

```
$success and $counter++;
```

If you recall, `and` statements are reliant on both sub-statements being true. So, if `$success` is not true, perl won't bother evaluating `$counter++` and upping its value by one. If `$success` was true, then it would.

Multiple Choice

If you look back to when we did our password tester, you'll see the following lines:

```
if ($password eq $guess) {
   print "Pass, friend.\n";
}
if ($password ne $guess) {
   die "Go away, imposter!\n";
}
```

While this does what we want, we know that if the first one is true, then the second one will not be – we're asking exactly opposite questions: Are these the same? Are they not the same?

In which case, it seems wasteful to do two tests. It'd be much nicer to be able to say 'if the strings are the same, do this. Otherwise, do that.' And in fact we can do exactly that, although the keyword is not 'otherwise' but '**else**':

```
if ($password eq $guess) {
   print "Pass, friend.\n";
} else {
   die "Go away, imposter!\n";
}
```

That's:

```
if ( <condition> ) { <action> } else { <alternative action> }
```

if elsif else

In some cases, we'll want to test more than one condition. When looking at several related possibilities, we'll want to ask questions like "Is this true? If this isn't, then is that true? If that's not true, how about the other?" Note that this is distinct from asking three independent questions; whether we ask the second depends on whether or not the first was true. In Perl, we could very easily write something like this:

```
if ( <condition 1> ) {
   <action>
} else {
   if ( <condition 2> ) {
      <second action>
   } else {
      if ( <condition 3> ) {
         <third action>
      } else {
         <if all else fails>
      }
   }
}
```

I hope you'll agree though that this looks pretty messy. To make it nicer, we can combine the `else` and the next `if` into a single word: `elsif`. Here's what the above would look like when rephrased in this way:

```
if ( <condition 1> ) {
   <action>
} elsif ( <condition 2> ) {
   <second action>
} elsif ( <condition 3> ) {
   ...
} else {
   <if all else fails>
}
```

Much neater! We don't have an awful cascade of closing brackets at the end, and it's easier to see what we're testing and when we're testing it.

Try It Out : Want To Go For A Walk?

I'll certainly not go outside if it's raining, but I'll always go out for a walk in the snow. I'll not go outside if it's less than 18 degrees Celsius. Otherwise, I'll probably go out unless I've got too much work to do. Do I want to go for a walk?

```
#!/usr/bin/perl
# walkies.plx
use warnings;
use strict;

print "What's the weather like outside? ";
my $weather = <STDIN>;
print "How hot is it, in degrees? ";
my $temperature = <STDIN>;
print "And how many emails left to reply to? ";
my $work = <STDIN>;
chomp($weather, $temperature);

if ($weather eq "snowing") {
   print "OK, let's go!\n";
} elsif ($weather eq "raining") {
   print "No way, sorry, I'm staying in.\n";
} elsif ($temperature < 18) {
   print "Too cold for me!\n";
} elsif ($work > 30) {
   print "Sorry - just too busy.\n";
} else {
   print "Well, why not?\n";
}
```

It's 20 degrees, I've got 27 emails to reply to, and it's cloudy out there. Let's see what the Simulated Simon would do:

```
> perl walkies.plx
What's the weather like outside? cloudy
How hot is it, in degrees? 20
And how many emails left to reply to? 27
Well, why not?
>
```

Looks like I can fit a walk in after all. Maybe after I show you how this program works.

How It Works

The point of this rather silly little program is that once it has gathered the information it needs, it runs through a series of tests, each of which could cause it to finish. First, we check to see if it's snowing:

```
if ($weather eq "snowing") {
   print "OK, let's go!\n";
```

If so, then we print our message and, this is the important part, do no more tests. If not, then we move onto the next test:

```
} elsif ($weather eq "raining") {
   print "No way, sorry, I'm staying in.\n";
```

Again, if this is true, we stop testing; otherwise, we move on. Finally, if none of the tests are true, we get to the `else`:

```
} else {
   print "Well, why not?\n";
}
```

Please remember that this is very different to what would happen if we used four separate `if` statements. The tests overlap, so it is possible for more than one condition to be true at once. For example, if it were snowing and I have over 30 emails to reply to, we'd get two conflicting answers. `elsif` tests should be read as 'Well, how about if…?'

Just in case you were curious, there is no `elsunless`. This is a Good Thing.

More Elegant Solutions

For three or four tests, it's reasonable to use `if-elsif-elsif-…-else`. But for any more than that, it starts to look ugly. What happens if we get input from the user and there are ten options? There are two general solutions to this, the first of which is to use a hash. We'll see in a few chapters time how you can store code to be executed inside a hash. If you can't use a hash, you're pretty much stuck with a chain of `elsifs`. You may, however, find it easier to do it like this:

```
print "Please enter your selection (1 to 10): ";
my $choice = <STDIN>;
for ($choice) {
   $_ == 1 && print "You chose number one\n";
   $_ == 2 && print "You chose number two\n";
   $_ == 3 && print "You chose number three\n";
   ...
}
```

We're using a `for` loop just like in the last chapter, but with a list of one thing. Why? Two reasons really:

- To give our program a bit of structure – brackets and indenting should make you realize there's a control structure going on.
- To alias `$choice` to `$_` for more convenient access.

Let's have a look in more detail about how the `for` loop works.

1, 2, Skip A Few, 99, 100

Now we know how to do everything once. But what if we need to repeat an operation or series of operations? Of course, there are methods available to specify this in perl too. We saw the `for` loop in Chapter 3, and this is one example of a class of control structures called loops.

In programming, there are various types of loop. Some loop forever and are called **infinite loops**, while most, in contrast, are finite loops. We say that a program 'gets into' or 'enters' a loop and then 'exits' or 'gets out' when finished. Infinite loops may not sound very useful, but they certainly can be – particularly because most languages, Perl included, provide you with a 'site door' by which you can exit

the loop. They will also be useful for when you just want the program to continue running until the user stops it manually, the computer powers down, or the heat death of the universe occurs, whichever is sooner.

There's also a difference between 'definite' loops and 'indefinite' loops. In a definite loop, you know how many times the block will be repeated in advance – a `for` loop is definite, because it will always iterate for each item in the array. An indefinite loop will check a condition in each iteration to whether it should do another or not.

There's also a difference between an indefinite loop that checks before the iteration and one that checks afterward. The latter will always go through at least one iteration in order to get to the check, whereas the former checks first and so may not go through any iterations at all.

Perl supports ways of expressing all of these types of loop. First, let's examine again the `for` loops we saw in the previous chapter.

for Loops

The `for` loop executes the statements in a block for each element in a list. Because of this, it's also known as the `foreach` loop, and you can use `foreach` anywhere you'd use `for`. For example:

```
#!/usr/bin/perl
#forloop1.plx
use warnings;
use strict;

my @array = (1, 3, 5, 7, 9);
my $i;
for $i (@array) {
   print "This element: $i\n";
}
```

This does exactly the same thing, and gives exactly the same output as this:

```
#!/usr/bin/perl
#forloop2.plx
use warnings;
use strict;

my @array = (1, 3, 5, 7, 9);
my $i;
foreach $i (@array) {
   print "This element: $i\n";
}
```

It's mainly a question of personal style – you won't go wrong if you use `foreach` all the time when talking about arrays. There's another form of `for` that does something completely different, and we'll see that a bit later on. We borrowed the syntax from a language called C, and so people who are used to programming in C can sometimes be confused by seeing `for` used with an array. If you use `foreach`, you'll keep them happy.

However, `foreach` is longer to type, and Perl programmers are notoriously lazy. And what's more, this is Perl, not C. Personally, I try and use `for` for constant lists and ranges like `(1...10)`, and `foreach` for arrays, but I'm not really consistent in that. Use whatever suits you.

As we mentioned above, the `for` loop is definite. You can work out, before you enter the loop, how many times you are going to repeat. It's also finite, since it's not possible to construct an infinitely long list.

Choosing an Iterator

We can specify the iterator variable ourselves as we did in the examples above, or we can use the default one, `$_` . Furthermore, if we're being good and using `strict`, we can make our iterator variable a lexical, `my` variable as we go along. That is, we could write the above like this:

```
#!/usr/bin/perl
#forloop3.plx
use warnings;
use strict;

my @array = (1, 3, 5, 7, 9);
foreach my $i (@array) {
   print "This element: $i\n";
}
```

There's actually a very subtle difference between declaring your iterator inside and outside of the loop. If you declare your iterator outside the loop, any value it had then will be restored afterwards. We can check this out by setting the variable and testing it afterwards:

```
#!/usr/bin/perl
#forloop4.plx
use warnings;
use strict;

my @array = (1, 3, 5, 7, 9);
my $i="Hello there";
foreach $i (@array) {
   print "This element: $i\n";
}
print "All done: $i\n";
```

This will produce the following output:

```
> perl forloop4.plx
This element: 1
This element: 3
This element: 5
This element: 7
This element: 9
All done: Hello there
>
```

Meanwhile declaring the iterator within the loop, as in `forloop3.plx`, will create a new variable `$i` each time, which will only exist for the duration of the loop.

As a matter of style, it's usual to keep the names of iterator variables very short. The traditional iterator is `$i`, as I've used here. The length of a variable name should be related to the importance of the variable; iterators are throwaway variables that only exist for one block, so they shouldn't be prominently named.

What We Can Loop Over

We can use `foreach` and `for` loops on any type of list whatsoever: A constant list:

```
my @array = qw(the quick brown fox ran over the lazy dog);
for (6, 3, 8, 2, 5, 4, 0, 7) {
   print "$array[$_] ";
}
```

an array:

```
my @array = qw(the quick brown fox ran over the lazy dog);
my $word;
for $word (@array) {
   print "$word ";
}
```

even a list generated by a function, like `sort` or `keys`:

```
my %hash = ( car => 'voiture', coach => 'car', bus => 'autobus' );
for (keys %hash) {
   print "English: $_\n";
   print "French: $hash{$_}\n\n";
}
```

It's a very powerful tool for those of you who need to list or 'enumerate' the contents of a hash or array, but there is a proviso before you go and use your `for` loops unwisely.

Aliases and Values

Be aware that the `for` loop creates an alias, rather than a value. Any changes you make to the iterator variable, whether it be `$_` or one you supply, will be reflected in the original array. For instance:

```
#!/usr/bin/perl
# forloop5.plx
use warnings;
use strict;

my @array = (1..10);
foreach (@array) {
   $_++;
}

print "Array is now: @array\n";
```

will change the actual contents of the array, as follows:

```
> perl forloop5.plx
Array is now: 2 3 4 5 6 7 8 9 10 11
>
```

Naturally, you can't change things that are constant, so doing the following will give an error:

```
#!/usr/bin/perl
# forloop6.plx
use warnings;
use strict;

foreach (1, 2, 3) {
   $_++;
}
```

```
> perl forloop6.plx
Modification of a read-only value attempted at forloop6.plx line 7
>
```

This means exactly what it says – we tried to modify (by adding one) to something that we could only read from, and not write to – in this case, the literal value of one. If you need to change the iterator for any reason, make a local copy, like this:

```
#!/usr/bin/perl
# forloop7.plx
use warnings;
use strict;

foreach (1, 2, 3) {
   my $i = $_;
   $i++;
}
```

Because `$i` is unrelated to the original list, we don't run into this problem.

Statement Modifiers

Just as there was a statement modifier form of `if`, like this:

```
die "Something wicked happened" if $error;
```

there's also a statement modifier form of `for`. This means you can iterate an array, executing a single statement every time. Here, however, you don't get to choose your own iterator variable: it's always `$_`. Let's create a simple totalling program using this idiom:

Try It Out : Quick Sum

The aim of this program is to take a list of numbers and output the total. For ease of use, we'll take the numbers from the command line.

```
#!/usr/bin/perl
# quicksum.plx
use warnings;
use strict;

my $total=0;
$total += $_ for @ARGV;
print "The total is $total\n";
```

Now when we give the program a few numbers to sum, it does just that:

```
>perl quicksum.plx 15 62 3 8 4
The total is 92
>
```

How It Works - @ARGV Explained

The whole trickery of this program is in that one line:

```
$total += $_ for @ARGV;
```

The first key point is the `@ARGV` array. This contains everything that's on the command line after the name of the program we're running. Perl receives, from the system, an array containing everything on the command line. This is split up a little like a Perl array without the commas. A single word is one element, as is a number, or a string in quotes. Depending on the shell, which is the thing that talks to the system in the first place, you may be able to backslash a space to stop it separating. Let's quickly write something that prints out each element of `@ARGV` separately, so we can see how they're split up:

```
#!/usr/bin/perl
# whatsargv.plx
use warnings;
use strict;

foreach (@ARGV) {
   print "Element: |$_|\n";
}
```

> **Why the strange parallel bars? You'll be pleased to hear it's not some arcane Perl syntax for doing anything special. All we're doing is placing a symbol on either side of our element. This is an oft-used debugging trick: the idea is that it allows you to see if there are any spaces at the start of end of the data, and allows you to tell the difference between an empty string and a string consisting entirely of spaces.**

Now let's try a few examples:

```
> perl whatsargv.plx one two three
Element: |one|
Element: |two|
Element: |three|
> perl whatsargv.plx "a string"    12
Element: |a string|
Element: |12|
> perl whatsargv.plx
>
```

In the first case, the three words were split up into separate elements. In the second, we kept two words together by giving them as a string in quotes. We also had another element afterwards, and the amount of white space between them made no difference to the number of elements. Finally, if there is nothing after the name of the program, there's nothing in `@ARGV`.

Let's get back to our program. We've now got an array constructed from of the command line, and we're going over it with a for loop:

```
$total += $_ for @ARGV;
```

With these statement modifiers, if they're not obviously clear, think how they'd be written normally. In this case:

```
for (@ARGV) {
   $total += $_;
}
```

that is, for each element, add that element to the total. This is more or less exactly how you take a total.

Looping While...

Now we come to the indefinite loops. As mentioned above, these check a condition, then do an action. The first one is `while`. As you might be able to work out from the name, this means anaction continues while a condition is true. The syntax of `while` is much like the syntax of `if`:

```
while ( <condition> ) { <action> }
```

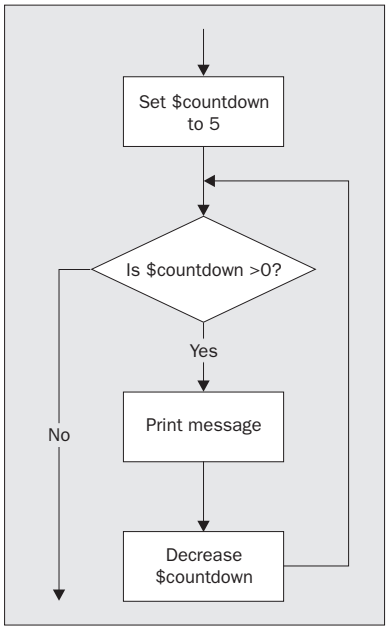
Here's a very simple while loop:

```
#!/usr/bin/perl
# while1.plx
use warnings;
use strict;

my $countdown = 5;

while ($countdown > 0) {
   print "Counting down: $countdown\n";
   $countdown--;
}
```

And here's what it produces:

```
>perl while1.plx
Counting down: 5
Counting down: 4
Counting down: 3
Counting down: 2
Counting down: 1
>
```

Let's see a flow chart for this program:



While there's still a value greater than zero in the `$counter` variable, we do the two statements:

```
print "Counting down: $countdown\n";
$countdown--;
```

Perl goes through the loop a first time when `$countdown` is 5 – the condition is met, so a message gets printed, and `$countdown` gets decreased to 4. Then, as the flow chart implies, back we go to the top of the loop. We test again: `$countdown` is still more than zero, so off we go again. Eventually, `$countdown` is 1, we print our message, `$countdown` is decreased, and it's now zero. This time around, the test fails, and we exit the loop.

while (<STDIN>)

Of course, another way to ensure that your loop is going to terminate is to make the condition do the change. This is sometimes thought of as bad style, but there's one example that is extremely widespread:

```
while (<STDIN>) {
   ...
}
```

Actually, this is a bit of shorthand; another example of a common Perl idiom. To write it out fully, it would look like this:

```
while (defined($_ = <STDIN>)) {
   ...
}
```

Since `<STDIN>` reads a new line from the user, the condition itself will depend on changing information. What we're doing is setting `$_` to each new line of input until we run out. We'll see in Chapter 6 how this is used to process files.

Infinite Loops

The important but obvious point is that what we're testing gets changed inside the loop. If our condition is always going to give a true result, we have ourselves an **infinite loop**. Let's just remove the second of those two statements:

```
#!/usr/bin/perl
# while2.plx
use warnings;
use strict;

my $countdown = 5;

while ($countdown > 0) {
   print "Counting down: $countdown\n";
}
```

`$countdown` never changes. It's always going to be 5, and 5 is, we hope, always going to be more than zero. So this program will keep printing its message until you interrupt it by holding down Ctrl and C. Hopefully, you can see why you need to ensure that what you do in your loop affects your condition.

Should we actually want an infinite loop, there's a fairly standard way to do it. Just put a true value – typically `1` – as the condition:

```
while (1) {
   print "Bored yet?\n";
}
```

The converse of course is to say `while(0)` in the loop's declaration, but nothing will ever happen because this condition is tested before any of the commands in the loop are executed. A bit silly, really.

Try It Out : English – Sdrawkcab Translator

In this example, we'll use our newly-introduced `while(<STDIN>)` construction to take a line of text from the user and produce the equivalent sentence translated into a language called Sdrawkcab. Sdrawkcab is a word in Sdrawkcab meaning 'backwards' – I hope you can see why.

```
#!/usr/bin/perl
# sdrawkcab1.plx
use warnings;
use strict;

while (<STDIN>) {
   chomp;
   die "!enod llA\n" unless $_;
   my $sdrawkcab = reverse $_;
   print "$sdrawkcab\n";
}
```

And here's a sample run.

```
> perl sdrawkcab1.plx
Hello
olleH
How are you?
?uoy era woH

!enod llA
>
```

(!oot ,rotalsnart hsilgnE-backwardS a sa ti esu yllautca nac uoy taht si rotalsnart siht tuoba gniht taerg ehT)

How It Works

The main part of this program is a loop that takes in a line from the user and places it in `$_`:

```
while (<STDIN>) {
   ...
}
```

Inside that loop, what do we do? First, we remove the new line. If we're going to turn it into Sdrawkcab, we want the new line at the end, not the beginning:

```
chomp;
```

If, after removing that new line, there's nothing left, `$_` is an empty string, a false value, we then finish the program:

```
die "!enod llA\n" unless $_;
```

Next, we do our actual translation – we can't just do `print reverse $_` however, because `reverse` in a list context, such as supplied by `print`, treats its arguments as a list and reverses the order of the items. Since we've only got one item here, that wouldn't be very interesting. You'll just get what you typed in:

```
my $sdrawkcab = reverse $_;
```

Then, finally, we print it out, and go back to get another line.

```
print "$sdrawkcab\n";
```

It's not very elegant, granted, but it gets the job done. Of course, there's more than one way to do it as I'll show you in the next section. But before you read on, you might like to make the translator a little 'prettier', prompting the user for a phrase to translate and prefacing the translation with a suitable phrase. From streams of such small improvements to an established core, most programs came.

Running at Least Once

When we were categorizing our lists, we divided indefinite loops into two catagories: those that those that execute at least once and those that may execute zero times. The `while` loop we've seen so far tests the condition first; if the condition isn't true the first time around, the 'body' of the loop never gets executed. There's another way to write our loop to ensure that the body is always executed at least once:

```
do { <actions> } while (<condition>)
```

Now we do the test after the block. This is equivalent to moving the diamond in our flow chart from the top to the bottom.

You may find it more natural to write the previous program like this:

```
#!/usr/bin/perl
# sdrawkcab2.plx
use warnings;
use strict;

do {
   $_ = <STDIN>;
   chomp;
   my $sdrawkcab = reverse $_;
   print "$sdrawkcab\n";
} while ($_);
print "!enod llA\n";
```

This does more or less the same thing, but in a slightly different way. First a line is read, then the translation produced, then we see if we need to get another line. There's one slight problem with this, when we want to end, we input a blank line that Perl 'translates' and prints out. See if you can fix this, and then see if you prefer the end result with the first program.

Statement Modifying

As before, you can use `while` as a statement modifier. Following the pattern of `for` and `if`, here's what you'd do with `while`:

```
while ( <condition> ) { <statement> }
```

becomes:

```
<statement> while <condition>
```

So, here's a way of writing our countdown program in three lines (if you exclude '`use strict`' and '`use warnings`', of course):

```
#!/usr/bin/perl
my $countdown = 5;
print "Counting down: $countdown\n" while $countdown-- > 0;
```

Don't be confused by the fact that the `while` is at the end – the condition is tested first, just as an ordinary `while` loop.

Looping Until

The opposite of `if` is `unless`, and the opposite of `while` is `until`. It's exactly the same as `while (not <condition>) { ... }`:

```
#!/usr/bin/perl
# until.plx
use warnings;
use strict;

my $countdown = 5;

until ($countdown-- == 0) {
   print "Counting down: $countdown\n";
}
```

Controlling Loop Flow

When we wrote our Sdrawkcab translator, the only way we could stop the loop was to end the program with a `die` command. Of course, there is another way to do it – by keeping a variable set to tell us whether or not we want to go through another loop. We can test this in our `while` condition. This kind of Boolean variable is called a **flag**, because it indicates something about the status of our program. We **set a flag** when we change its value.

Here's a version of the Sdrawkcab program that sets a flag when it's time to finish:

```
#!/usr/bin/perl
# sdrawkcab3.plx
use warnings;
use strict;

my $stopnow = 0;
until ($stopnow) {
   $_ = <STDIN>;
   chomp;
if ($_) {
      my $sdrawkcab = reverse $_;
      print "$sdrawkcab\n";
   } else {
      $stopnow = 1;
   }
}
print "!enod llA\n";
```

When `$_` becomes the empty string, and hence a false value, the `if ($_)` test fails. This sets `$stopnow` to `1` and will end the `until` loop.

There's a school of thought, called 'structured programming' that urges strict adherence to these loops and conditionals. Unfortunately, you end up with code like on the previous page. Most programmers, though, take a less strict approach. When it's time to leave the loop, they don't wait for the test to come around again, they just leave.

Breaking Out

The keyword `last`, in the body of a loop, will make perl immediately exit, or 'break out of' that loop. The remaining statements are not processed, and you up right at the end. This is exactly what we want to do to make the above program easier to deal with:

```
#!/usr/bin/perl
# sdrawkcab4.plx
use warnings;
use strict;

while (<STDIN>) {
   chomp;
   last unless $_;
   my $sdrawkcab = reverse $_;
   print "$sdrawkcab\n";
}
# and now we can carry on with something else...
```

You can use this in a `for` loop as well:

```
#!/usr/bin/perl
# forlast.plx
use warnings;
use strict;

my @array = ( "red", "blue", "STOP THIS NOW", "green");
for (@array) {
   last if $_ eq "STOP THIS NOW";
   print "Today's colour is $_\n";
}
```

```
>perl forlast.plx
Today's colour is red
Today's colour is blue
>
```

If you try to do a `last` when you're not in a loop, perl will complain, even if you have forgotten to use `use warnings`:

```
#!/usr/bin/perl
# badlast.plx

last;
```

```
Can't "last" outside a block at badlast.plx line 4.
```

Going onto the Next

If you want to skip the rest of the processing of the body, but don't want to exit the loop, you can use `next` to immediately go back to the start of the loop, passing the next value to the iterator. This is an oft-used technique to process only selected elements:

```
#!/usr/bin/perl
# next.plx
use strict;
use warnings;

my @array = (8, 3, 0, 2, 12, 0);
for (@array) {
   if ($_ == 0) {
      print "Skipping zero element.\n";
      next;
   }
   print "48 over $_ is ", 48/$_, "\n";
}
```

In `next.plx` then, we have set a trap for all those dastardly zeroes that want to cause our divisions to fail:

```
>perl next.plx
48 over 8 is 6
48 over 3 is 16
Skipping zero element.
48 over 2 is 24
48 over 12 is 4
Skipping zero element.
>
```

> **Be careful: while `next` takes you to the next iteration of the loop, `last` doesn't take you to the last iteration, it takes you past it.**

On rare occasions, you'll want to go back to the top of the loop, but without testing the condition (in the case of a `for` loop) or getting the next element in the list (as in a `while` loop). If you feel you need to do this, the keyword to use is `redo`:

Try It Out - Debugging Loops 101

It's perfectly possible to have a loop inside a loop. The interesting part comes when you need to go to the end or the beginning of a external loop from an internal loop. For example, if you're reading some input from the user, and the input is any one of a series of pre-determined 'safe words', you end the loop. Here's what you might want to do:

```
#!/usr/bin/perl
# looploop1.plx
use warnings;
use strict;
my @getout = qw(quit leave stop finish);
```

```
while (<STDIN>) {
   chomp;
   for my $check (@getout) {
      last if $check eq $_;
   }
   print "Hey, you said $_\n";
}
```

The problem with this is that it doesn't work. Now, 'it doesn't work' is possibly the worst way to approach finding a bug. What do we mean by 'it doesn't work'? Does it sit on the couch all day watching TV? We need to be specific! What doesn't work about it?

How It Doesn't Work and Why

Well, even if we put in one of the words that's supposed to let us quit, Perl carries on, like this:

```
>perl looploop1.plx
Hello
Hey, you said Hello
quit
Hey, you said quit
stop
Hey, you said stop
leave
Hey, you said leave
finish
Hey, you said finish
```

We've specifically isolated the problem. Now, let's see if we can find any clues as to what's causing it. The fact that it's printing out means it's finished the `for` loop. Let's add in a couple of `print` statements to help us investigate what the `for` loop is actually doing:

```
for my $check (@getout) {
   print "Testing $check against $_\n";
   last if $check eq $_;
   print "Well, it wasn't $check\n";
}
```

Now run it again:

```
Hello
Testing quit against Hello
Well, it wasn't quit
Testing leave against hello
Well, it wasn't leave
Testing stop against Hello
Well, it wasn't stop
Testing finish against Hello
Well, it wasn't finish
Hey, you said Hello
quit
Testing quit against quit
Hey, you said quit
```

Aha, more clues. So it's testing properly, and it's finishing when it sees 'quit', which is one of the stop words. That's a relief to know, but it's only finishing the `for` loop, rather than finishing the `while` loop. This is the root of the problem:

> **'It doesn't work' is not a bug report. First you need to be specific about what doesn't work. Then you need to detail what doesn't work about it. Then you can start to examine why it doesn't work. When you've got over the 'doesn't work' feeling and fully investigated what it's really doing and how that differs from your expectations, only then can you begin to fix it.**

So, how do we fix this one? What we need to do is to distinguish between the two loops, the inner `for` loop and the outer `while` loop. The way we distinguish between them is by giving them names, or **labels**.

A **label** goes before the `for`, `while`, or `until` of a loop, and ends with a colon. The rules for naming labels are the same as for naming variables, but it's usual to construct labels from uppercase letters.

Here's our program with labels attached:

```
#!/usr/bin/perl
# looploop2.plx
use warnings;
use strict;

my @getout = qw(quit leave stop finish);

OUTER: while (<STDIN>) {
   chomp;
   INNER: for my $check (@getout) {
      last if $check eq $_;
   }
   print "Hey, you said $_\n";
}
```

Now for the finale, we can direct last, next, and redo to a particular loop by giving them the label. Here's the fixed version:

```
#!/usr/bin/perl
# looploop3.plx
use warnings;
use strict;

my @getout = qw(quit leave stop finish);

OUTER: while (<STDIN>) {
   chomp;
   INNER: for my $check (@getout) {
      last OUTER if $check eq $_;
   }
   print "Hey, you said $_\n";
}
```

Now when we find a matching word, we don't just jump out of the `for` loop – we go all the way to the end of the outer `while` loop as well, which is exactly what we wanted to do.

Goto

As a matter of fact, you can put a label before any statement whatsoever. If you want to really mess around with the structure of your programs, you can use `goto LABEL` to jump anywhere in your program. Whatever you do, don't do this. This is not to be used. Don't go that way.

I'm telling you about it for the simple reason that if you see it in anyone else's Perl, you can laugh heartily at them. There are other, more acceptable forms of `goto`, which we'll see when we come to subroutines. But `goto` with a label is to be avoided like the plague.

Why? Because not only does it turn the clock back thirty years (the structured programming movement started with the publication of a paper called 'Use of goto considered harmful'), but it tends to make your programs amazingly hard to follow. The flow of control can shoot off in any direction at any time, into any part of the file, perhaps into a different file. You can even find yourself jumping into the middle of loops, which really doesn't bear thinking about. Don't use it unless you really, really, really understand why you shouldn't. And even then, don't use it. Larry Wall has never used `goto` with a label in Perl, and he wrote it.

Don't. (He's watching - *Ed*)

Summary

Before this chapter, our programs plodded along in a straight line, following one statement with another.

We've now seen how we can react to different circumstances in our programs, which is the start of flexible and powerful programming. We can test whether something is true or false using `if` and `unless` and take appropriate action. We've also examined how to test multiple related conditions, using `elsif`.

We can repeat areas of a program, in several different ways: once per element of a list, using `for`, or continually while a condition is true or false, using `while` and `until`.

Finally, we've examined some ways to alter the flow of perl's execution through these loops. We can break out of a loop with `last`, skip to the next element with `next`, and start processing the current element again with `redo`.

Exercises

1. Modify the currency program convert2.plx to keep asking for currency names until a valid currency name is entered.

2. Modify the number-guessing program guessnum.plx so that it loops until the correct answer is entered.

3. Write your own program to capture all the prime numbers between 2 and a number the user gives you.