8

Subroutines

When programming, there'll naturally be processes you want to do again and again: adding up the values in an array, stripping extraneous blank spaces from a string, getting information into a hash in a particular format, and so on. It would be tedious to write out the code for each of these little processes every time we need to use one, and it would be horrific to maintain too: if there are bugs in the way we've specified it, we'll have to go through and find each one of them and fix it. It would be better if we could define a particular process just once, and then be able to call on that just like we've been calling on Perl's built-in operators.

This is exactly what **subroutines** allow us to do. Subroutines (or just **subs**) give us the ability to give a name to a section of code. Then when we need to use that code in our program, we just call it by name.

Subroutines help our programming for two main reasons. First, they let us reuse code, as we've described above. This makes it easier to find and fix bugs and makes it faster for us to write programs. The second reason is that they allow us to chunk our code into organizational sections. Each subroutine can, for example, be responsible for a particular task.

So, when is it appropriate to use subroutines in Perl? I would say there would be two cases when a piece of code should be put into a subroutine: first, when you know it will be used to perform a calculation or action that's going to happen more than once. For instance, putting a string into a specific format, printing the header or footer of a report, turning an incoming data record into a hash, and so on.

One thing we'll see later on is that we can use subroutines in a similar way to the way we've been using Perl's built-in operators. We can give them arguments and get scalars and lists returned to us.

Second, if there are logical units of your program that you want to break up to make your program easier to understand. I can imagine few things worse than debugging several thousand lines of Perl that are not broken up in any way (well, maybe one or two things). As an extreme example, sometimes – and only sometimes – I like to have a 'main program', which consists entirely of calls to subroutines, like this:

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

setup();
get_input();
process_input();
output();
```

This immediately shows me the structure of my program. Each of those four subroutines would, of course, have to be defined, and they'd probably call on other subroutines themselves. This allows us to partition up our programs, to change our single, monolithic piece of code into manageable chunks for ease of understanding, ease of debugging, and ease of maintaining the program.

The 'Difference' Between Functions and Subroutines

Instead of the term 'subroutine', you're sure to come across the word 'function' many times as you deal with Perl and Perl resources. So let's have a look at the difference between 'function', 'subroutine', and 'operator'. The problem is that other programming languages use the terms ever so slightly differently.

Usually

In most programming languages, and in computer science in general, the following definitions apply:

- A **function** is something that takes a number of arguments (possibly zero), does something with them, and returns a value. A function can either be built into the programming language or it can be supplied by the user.
- An **operator** is a function that is usually represented by a symbol rather than a name and is almost always built into the programming language.
- A **subroutine** is some code provided by the user that performs an action and doesn't return a value. Unfortunately, languages like C have functions that can return nothing. These 'void functions' could be called subroutines – but they're not. That's life.

In Perl

Because some people who know other languages use the usual terms, Perl's definitions are a little confusing:

- If someone mentions a **function** in Perl, they almost certainly mean something built into Perl. However, they might be coming from C and mean a subroutine. The main reference documentation for Perl built-ins is called `perlfunc`. You can also find the complete list in Appendix C.
- An **operator** in Perl can have a name instead of a symbol, so it can look very much like a function. Hence, some people tend to use the terms interchangeably. Those built-ins that have symbols instead of names are documented in `perlop`, which also refers to 'named operators'. `perl` itself speaks about the 'print operator', so we've used that terminology in this book. However, you're equally likely to hear Perl people talk about 'the print function'.
- **Subroutines** in Perl are akin to C's functions – they are sections of code that can take arguments, perform some operations with them, and may return a meaningful value, but don't have to. However, they're always user-defined rather than built-ins:

> **Simply put: Subroutines are chunks of code you give Perl; Functions and Operators are things that Perl provides.**

Understanding Subroutines

Now we know what subroutines are, it's time to look at how to define them and how to use them. First, we'll learnhow to create subroutines.

Defining a Subroutine

So, we can give Perl some code, and we can give it a name, and that's our subroutine. Here's how we do it:

```
sub marine {
   ...
}
```

There are three sections to this declaration:

- The keyword `sub`. This is case-sensitive and needs to be in lower case.
- The name we're going to give it. The rules for naming a subroutine are exactly those for naming variables; names must begin with an alphabetic character or an underscore, to be followed by one or more alphanumerics or underscores. Upper case letters are allowed, but we tend to reserve all-uppercase names for special subroutines. And again, as for variables, you can have a scalar `$fred`, an array `@fred`, a hash `%fred`, a filehandle `fred`, and a subroutine `fred`, and they'll all be distinct.
- A block of code delimited by curly brackets, just as we saw when we were using `while` and `if`. Notice that we don't need a semicolon after the closing brace.

After we've done that, we can use our subroutine.

Before we go any further, it's worth taking a quick time out to ponder how we name our subroutines. You can convey a lot about a subroutine's purpose with its name, much like that of a variable. Here are some guidelines – not hard-and-fast rules – about how you should name subroutines.

- If they're primarily about performing an activity, name them with a verb, for example, `summarize` or `download`.
- If they're primarily about returning information, name them after what they return, for example, `greeting` or `header`.
- If they're about testing whether a statment is true or not, give them a name that makes sense in an `if` statement; starting with `is_`... or `can_`... helps, or if that isn't appropriate, name them with an adjective: for example, `is_available`, `valid`, or `readable`.
- Finally, if you're converting between one thing and another, try and convey both things. Traditionally this is done with an `2` or `_to_` in the middle: `text2html`, `metres_to_feet`. That way you can tell easily what's being expected and what's being produced.

Try It Out : Version Information

It's traditional for programs to tell you their version and name either when they start up or when you ask them with a special option. It's also convenient to put the code that prints this information into a subroutine to get it out of the way. Let's take our very first program and update it for this traditional practice.

Here's what we started with, version 1:

```
#!/usr/bin/perl
use warnings;
print "Hello, world.\n";
```

And here it is with warnings and strict modes turned on and version information:

```
#!/usr/bin/perl
# hello2.plx
use warnings;
use strict;

sub version {
   print "Beginning Perl's \"Hello, world.\" version 2.0\n";
}

my $option = shift;
version if $option eq "-v" or $option eq "--version";
print "Hello, world.\n";
```

Now, we're starting to look like a real utility:

```
>perl hello2.plx -v
Beginning Perl's "Hello, world." version 2.0
Hello, world.
```

How It Works

As before, we have the `sub` keyword, a name, `version`, and then the block of code. We've defined the `version` subroutine as follows:

```
sub version {
   print "Beginning Perl's \"Hello, world.\" version 2.0\n";
}
```

It's a simple block of code that calls the `print` statement. It didn't have to – it could have done anything. Any code that's valid in the main program is valid inside a subroutine, including:

- Calling other subroutines
- Calling the current subroutine again – see the section 'Recursion' at the end of the chapter on this very subject.

We call this block the **body** of the subroutine, just like we had the body of a loop; similarly, it stretches from the open curly bracket after the subroutine name to the matching closing bracket.

Now we've defined it, we can use it. We just give the name, and Perl runs that block of code, albeit with the proviso that we've added the right flag on the command line:

```
version if $option eq "-v" or $option eq "--version";
```

When it's finished doing `version`, it comes back and carries on with the next statement:

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

No doubt version th3ree will address the warnings that Perl gives if you call this program without appending -v or --version to its name.

Order of Declaration

If we just call our subroutines by name, as we did above, we're forced to declare them before we use them. This may not sound much of a limitation, but there are times when we'll want to declare our subroutines after the main part of the program. In fact, that's the usual way to structure a program. This is because when you open up the file in your editor, you can see what's going on right there at the top of the file, without having to scroll through a bunch of definitions first. Take the extreme example at the beginning of this chapter:

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

setup();
get_input();
process_input();
output();
```

That would then be followed, presumably, by something like this:

```
sub setup {
   print "This is some program, version 0.1\n";
   print "Opening files...\n";
   open_files();
   print "Opening network connections...\n";
   open_network();
   print "Ready!\n";
}

sub open_files {
   ...
}
```

That's far easier to understand than trawling through a pile of subroutines before getting to the four lines that constitute our main program. It also encourages the 'top-down' school of programming.

> *Traditional programming methodology, which I've been using here, states that we should start at the highest level of our program and break it down into smaller and smaller problems – starting at the top and working down. There's also a bottom-up school of thought that dictates you should write your basic operations first, then glue them together. There's even been the suggestion of a 'middle-out' style that starts at a middle layer and adds smaller operations and higher-level structure at the same time. I encourage you to start with top-down programming until something else becomes natural.*

However, in order to get this to work, we need to provide hints to Perl as to what we're doing. That's why the calls to subroutines above have a pair of brackets around them: `setup()`, `open_files()`, and so on. This helps to tell Perl that it should be looking for a subroutine somewhere instead of referring to a filehandle or anything else it could have been. What happens if we don't do this?

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

setup;
sub setup {
   print "This is some program, version 0.1\n";
}
```

```
>perl subdec1.plx
Bareword "setup" not allowed while "strict subs" in use at subdecl.plx line 6.
Execution of subdec1.plx aborted due to compilation errors.
>
```

Perl didn't know what we meant at the time, so it complained. To tell it we're talking about a subroutine, we use brackets, just like when we want the parameters to an operator like `print` to be unambiguous.

There's another way we can tell Perl that we're going to refer to a subroutine and that's to provide a **forward definition** – also known as **pre-declaring** the subroutine. This means 'we're not going to define this right now, but look out for it later.'

We do this by just saying `sub NAME;`. Note that this does require a semicolon at the end. Here's another way of writing the above:

```
#!/usr/bin/perl
use warnings;
use strict;
sub setup; sub get_input; sub process_input; sub output;
sub open_files; sub open_network;
...
```

From now on, we can happily use the subroutines without the brackets:

```
setup;
get_input;
process_input;
output;

sub setup {
   print "This is some program, version 0.1\n";
   print "Opening files...\n";
   open_files;
   print "Opening network connections...\n";
   open_network;
   print "Ready!\n";
}

sub open_files {
   ...
}
```

Alternatively, you can ask Perl to provide the forwards for you. If we say `use subs (...)`, we can provide a list of subroutine names to be pre-declared:

```
#!/usr/bin/perl
use warnings;
use strict;
use subs qw(setup get_input process_input output pen_files open_network);
...
```

Personally, however, I tend to leave in the brackets to remind me I'm dealing with subroutines. You may also see yet another way of calling subroutines:

```
&setup;
&get_input;
&process_input;
&output;
```

This was popular in the days of Perl 4, and we'll see later why the ampersand is important. For the time being, think of the ampersand as being the 'type symbol' for subroutines.

Subroutines for Calculation

As we mentioned at the beginning of the chapter, as well as being set pieces of code to be executed whenever we need them, we can also use subroutines just like Perl's built-in functions and operators. We can pass parameters to the subroutine and expect an answer back.

Parameters and Arguments

Just like with Perl's built-ins, we pass parameters by placing them between the brackets:

```
my_sub(10,15);
```

What happens to them there? Well, they end up in one of Perl's special variables, the array `@_` and from there we can get at them:

Try It Out : Totalling a List

We'll write a subroutine that takes a list of values, adds them up, and prints the total:

```
#!/usr/bin/perl
# total1.plx
use warnings;
use strict;

total(111, 107, 105, 114, 69);
total(1...100);

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

And to see it in action:

```
> perl total1.plx
The total is 506
The total is 5050
>
```

How It Works

We can pass any list to a subroutine, just like we can to `print`. When we do so, the list ends up in `@_` where it's up to us to do something with it. Here, we go through each element of it and add them up:

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

This is a little cryptic, but it's how you're likely to see it done in real Perl code. You could write this a little less tersely as follows:

```
my @args = @_;
foreach my $element (@args) {
   $total = $total+$element;
}
```

In the first example, `@_` would contain `(111, 107, 105, 114, 69)`, and we'd add each value to `$total` in turn.

Return Values

However, sometimes we don't want to perform an action like printing out the total, but instead we want to return a result. We may also want to return a result to indicate whether what we were doing succeeded. This will allow us to say things like:

```
$sum_of_100 = total(1...100);
```

There are two ways to do this: implicitly or explicitly. The implicit way is nice and easy. We just make the value we want to return the last thing in our subroutine:

```
#!/usr/bin/perl
# total2.plx
use warnings;
use strict;

my $total      = total(111, 107, 105, 114, 69);
my $sum_of_100 = total(1...100);

sub total {
   my $total = 0;
   $total += $_ for @_;
   $total;
}
```

It doesn't need to be a variable: we could use any expression there. We can also return a list instead of a single scalar.

Try It Out : Splitting Time

Let's split a time in seconds up to hours, minutes, and seconds. We give a subroutine a time in seconds, and it returns a three-element list with the hours, minutes, and remaining seconds:

```
#!/usr/bin/perl
# seconds1.plx
use warnings;
use strict;

my ($hours, $minutes, $seconds) = secs2hms(3723);
print "3723 seconds is $hours hours, $minutes minutes and $seconds seconds";
print "\n";

sub secs2hms {
   my ($h,$m);
   my $seconds = shift;
   $h = int($seconds/(60*60)); $seconds %= 60*60;
   $m = int($seconds/60);      $seconds %= 60;
   ($h,$m,$seconds);
}
```

This tells us that:

```
>perl seconds1.plx
3723 seconds is 1 hours, 2 minutes and 3 seconds
>
```

How It Works

Just like a built-in function, when we're expecting a subroutine to return a list, we can use an array or list of variables to collect the return values:

```
my ($hours, $minutes, $seconds) = secs2hms(3723);
```

When `secs2hms` returns, this'll be equivalent to:

```
my ($hours, $minutes, $seconds) = (1,2,3);
```

Now let's look at how the subroutine works. We start in the usual way: `sub`, the name, and a block:

```
sub secs2hms {
```

We have two variables to represent hours and minutes, and we read the parameters in from `@_`. If you don't tell `shift` which array to take data from, it'll read from `@_` if you're in a subroutine or `@ARGV` if you're not:

```
   my ($h,$m);
   my $seconds = shift;
```

Then the actual conversion: There are 3600 (60*60) seconds in an hour, and so the number of hours is the number of seconds divided by 3600. However, that'll give us a floating-point number – if we divided 3660 by 3600, we'd get 1.0341666… we'd rather have 'one and a bit', so we use `int()` to get the integer value, the '1' part of the division, and use the modulus operator to get the remainder. After dealing with the first 3600 seconds, we want to carry on looking at the next 123:

```
$h = int($seconds/(60*60)); $seconds %= 60*60;
```

The second statement on this line sets `$seconds` to `$seconds % (60*60)`. If it was 3723 before, it'll be 123 now.

The same goes for minutes: we divide to get 'two and a bit', and the remainder tells us that there are three seconds outstanding. Hence, our values are 1 hour, 2 minutes, and 3 seconds:

```
$m = int($seconds/60);      $seconds %= 60;
```

We return this just by leaving a list of the values as the last thing in the subroutine:

```
($h,$m,$seconds);
```

The return Statement

The explicit method of returning something from a subroutine is to say `return(...)`. The first `return` statement we come across will immediately return that list to the caller. So, for instance:

```
sub secs2hms {
   my ($h,$m);
   my $seconds = shift;
   $h = int($seconds/(60*60)); $seconds %= 60*60;
   $m = int($seconds/60);      $seconds %= 60;
   return ($h,$m,$seconds);
   print "This statement is never reached.";
}
```

This also means we can have more than one `return` statement, and it's often useful to do so.

Caching

One particularly effective use of this is called **caching**, and it's a technique we can use to make subroutines that do calculations work faster. To use caching, we store each answer we generate from a set of parameters into a cache, usually a hash. If we see those parameters again, we can fetch the answer from the cache rather than work it all out from scratch. For example, here's a subroutine that gets the first line in a file:

```
sub first_line {
   my $filename = shift;
   open FILE, $filename or return "";
   my $line = <FILE>;
   return $line;
}
```

And here's that subroutine with caching:

```
my %cache;
sub first_line {
   my $filename = shift;
   return $cache{$filename} if exists $cache{$filename}
   open FILE, $filename or return "";
   my $line = <FILE>;
   $cache{filename} = $line;
   return $line;
}
```

Although it's possible that the first lines of those files change while we're running the program, it's not likely. So, we check to see if we've seen a file before; if we have, we give the answer we got last time and return. If we haven't seen it before, we open the file, check it out, and then store the answer in the cache for next time.

If you've got subroutines where the answer is likely to be the same every time you call with a given parameter, and where you're doing significantly more work than a simple lookup, consider using a cache like this.

Context

Some of Perl's built-ins do different things in different contexts: `localtime`, for instance, returns a string in scalar context and a breakdown of the time in list context. As `perlfunc` puts it, *'There is no rule that relates the behavior of an expression in list context to its behavior in scalar context, or vice versa. It might do two totally different things.'*

We can make our subs sensitive to context as well. Perl provides two functions to allow us to examine how we were called. The more complex one is `caller`, and the one we'll look at is `wantarray`. Strictly speaking, it tells us whether our caller wants a list. If so, it will be true. If a single scalar is required, then it will be false. If the caller isn't planning to do anything with what we give it, it will be the undefined value. So, for instance, we can emulate `localtime` like this:

```
#!/usr/bin/perl
# seconds2.plx
use warnings;
use strict;
my ($hours, $minutes, $seconds) = secs2hms(3723);
print "3723 seconds is $hours hours, $minutes minutes and $seconds seconds\n";
my $time = secs2hms(6868);
print "6868 seconds is $time\n";

sub secs2hms {
   my ($h,$m);
   my $seconds = shift;
   $h = int($seconds/(60*60)); $seconds %= 60*60;
   $m = int($seconds/60);      $seconds %= 60;
   if (wantarray) {
      return ($h,$m,$seconds);
   }
   return "$h hours, $m minutes and $seconds seconds";
}
```

```
>perl seconds2.plx
3723 seconds is 1 hours, 2 minutes and 3 seconds
6868 seconds is 1 hours, 54 minutes and 28 seconds
>
```

To be honest, however, it's pretty unlikely that you'll ever do this: It's best to have a subroutine that returns the same thing all the time, unless it's being used by someone other than yourself.

Subroutine Prototypes

If your subroutines are likely to be used by someone else, you might want to consider using subroutine prototypes. You'll also need to think about these if you're planning on passing more than one array to a subroutine. We'll look later at how that is done.

A subroutine prototype tells Perl what sort of arguments it's expecting. This can be used to check to ensure that the user is passing the right number of parameters, and it can also change the way Perl reads your program. For instance, you can make it possible to leave off the brackets from around your parameters, in the same way that `print "one", "two";` is the same as `print("one","two");` and you can chose whether:

```
print mysub "one", "two";
```

means:

```
print( mysub("one", "two") );
```

or:

```
print( mysub("one"), "two" );
```

That is, how many arguments your subroutine should swallow up.

Prototypes talk about the number of scalars we allow, and we use a dollar sign for each one. So, the prototype for a subroutine that takes two arguments would be `$$`. Prototypes come between the name and the block of the subroutine definition, in brackets, like this:

```
sub sum_of_two_squares ($$) {
   my ($a,$b) = (shift, shift);
   return $a**2+$b**2;
}
```

The problem is, just like when we wanted to use subroutines without the brackets, Perl hadn't read as far as their definition when it came across the call and so didn't know what to expect. When using prototypes we need to ensure that Perl gets to read the prototype before we use the subroutine, and to do this, we can use a forward definition at the top of the program, like so:

```
#!/usr/bin/perl
# sumsquare.plx
use warnings;
use strict;
sub sum_of_two_squares ($$);
```

Try It Out : Using Prototypes

Now if we try to give any more or less than two parameters, Perl complains even before the program starts:

```
#!/usr/bin/perl
# sumsquare.plx
use warnings;
use strict;
sub sum_of_two_squares ($$);

my ($first, $second) = @ARGV;
print "The sum of the squares of $first and $second is ";
print sum_of_two_squares($first, $second),"\n";

print sum_of_two_squares($first, $second, 0),"\n";

sub sum_of_two_squares ($$) {
   my ($a,$b) = (shift, shift);
   return $a**2+$b**2;
}
```

We try to use three parameters, but Perl won't allow it because we've told it only to accept two:

```
>perl sumsquare.plx 10 20
Too many arguments for main::sum_of_two_squares at sumsquare.plx line 11, near "0)"
Execution of sumsquare.plx aborted due to compilation errors.
>
```

If we comment out that line, it works as expected:

```
> perl sumsquare.plx 10 20
The sum of the squares of 10 and 20 is 500
>
```

You can specify that the number may vary by the use of a semicolon in the prototype. Everything after the semicolon is tentative; you can also use an `@_` sign to denote 'any number of parameters'.

Understanding Scope

It's now time to have a serious look at what we're doing when we declare a variable with `my`. The truth, as we've briefly glimpsed it, is that Perl has two types of variable. One type is the **global variable** (or **package variable**), which can be accessed anywhere in the program. The second type is the **lexical variable,** which we declare with `my`.

Global Variables

Global variables are what you get if you don't use `my`. If we were to say:

```
#!/usr/bin/perl
$x = 10;
```

then `$x` would be a global variable. They're also called package variables because they live inside a package (a package is just a convenient place to put subroutines and variables).

When we start programming, we're in a package called `main`. If we assign `$x`, as above, then we create a package variable `$x` in package `main`. Perl knows it by its full name, `$main::x` – the variable `$x` in the `main` package. But because we're in the `main` package when we make the assignment, we can just call it by its short name, `$x`. It's like the phone system – you don't have to dial the area code when you call someone in the same region as you.

We can create a variable in another package by using a fully-qualified name. Instead of the main package, we can have a package called `Fred`. Here we'll store all of Fred's variables and subroutines. So, to get at the `$name` variable in package Fred, we say `$Fred::name`, like this:

```
$x = 10;
$Fred::name = "Fred Flintstone";
```

The fact that it's in a different package doesn't mean we can't get at it. Remember that these are global variables, available from anywhere in our program. All packages do is give us a way of subdividing the namespace.

What do we mean by 'subdividing the namespace'? Well, the namespace is the set of names we can give our variables. Without packages, we could only have one `$name`. What packages do is help us make `$name` in package `Fred` different to `$name` in package `Barney` and `$name` in package `main`.

```
#!/usr/bin/perl
# globals.plx
use warnings;
$main::name   = "Your Name Here";
$Fred::name   = "Fred Flintstone";
$Barney::name = "Barney Rubble";

print "\$name in package main   is $name\n";
print "\$name in package Fred   is $Fred::name\n";
print "\$name in package Barney is $Barney::name\n";
```

```
> perl globals.plx
$name in package main   is Your Name Here
$name in package Fred   is Fred Flintstone
$name in package Barney is Barney Rubble
```

You can change what package you're currently working in with the aptly named `package` operator. We could write the above like this:

```
#!/usr/bin/perl
# globals2.plx
use warnings;
$main::name   = "Your Name Here";
$Fred::name   = "Fred Flintstone";
$Barney::name = "Barney Rubble";

print "\$name in package main   is $name\n";
package Fred;
print "\$name in package Fred   is $name\n";
package Barney;
print "\$name in package Barney is $name\n";
package main;
```

When `use strict` is in force, it makes us use the full names for our package variables. If we try and say this:

```
#!/usr/bin/perl
#strict1.plx
use warnings;
use strict;
$x = 10;
print $x;
```

Perl will give us an error – Global symbol "$x" requires explicit package name. The package name it's looking for is `main`, and it wants us to say `$main::x`

```
#!/usr/bin/perl
#strict2.plx
use warnings;
use strict;
$main::x = 10;
print $main::x;
```

As we've seen before, we can also use the `our` operator to tell Perl that a given variable should be treated as a package variable in the current package. This works just as well:

```
#!/usr/bin/perl
#strict3.plx
use warnings;
use strict;
our $x;
$x = 10;
print $x;
```

Global variables can be accessed and altered at any time by any subroutine or assignment that you care to apply to it. Of course, this is handy if you want to store a value – for instance, the user's name – and be able to get it anywhere.

It's also an absolute pain in the neck when it comes to subroutines. Here's why:

```
$a = 25;
$b = some_sub(10);
print $a;
```

Looks innocent, doesn't it? Looks like we should see the answer 25. But what happens if `some_sub` uses and changes the global `$a`? Any variable anywhere in your program can be wiped out by another part of your program. We call this 'action at a distance', and it gets real spooky to debug. Packages alleviate the problem, but to make sure that we never get into this mess, you have to ensure that every variable in your program has a different name. In small programs, that's feasible, but in huge team efforts, it's a nightmare. It's far clearer to be able to restrict the possible effect of a variable to a certain area of code, and that's exactly what lexical variables do.

Lexical Variables

The range of effect that a variable has is called its **scope**, and lexical variables declared with `my` are said to have **lexical scope,** that is, they exist from the point where they're declared until the end of the enclosing block, brackets, subroutine, or file. The name 'lexical' comes from the fact that they're confined to a well-defined chunk of text.

Each block has got a 'pad' in which it keeps its current lexical variables, if any. If Perl doesn't find the variable you're referring to in the current pad, it'll look to the surrounding blocks until it finds it – or doesn't. Every time you say `my`, you're creating a new variable attached to the current pad. It's completely independent of any variables in other pads, and you use can use it to 'hide' similarly-named lexicals that exist outside of the current block:

```
my $x;
$x = 30;
{
   my $x; # New $x
   $x = 50;
   # We can't see the old $x, even if we want to.
}
# This $x is, and always has been, 30.
```

Great. We can now use variables in our subroutines in the knowledge that we're not going to upset any behavior outside them. We know that if we say:

```
sub strip {
   my $input = shift;
   $input =~ s/^\s+//;
   $input =~ s/\s+$//;
   return $input;
}
```

that we're not going to clobber any other `$input` in the program. The highlighted part shows you the lifespan of the variable: It comes into existence at the `my` statement and goes away at the end of the nearest set of braces. We say that it 'goes out of scope' at the end of the subroutine. Once it's out of scope, we shouldn't expect to be able to get to it again. In a sense, we've created a temporary variable.

Runtime Scope

However, we can't use this trick for global variables, and Perl's special variables such as `$_` and `$/` are globals. What can we do to temporarily set their value? One way to do it is like this:

```
sub slurp {
   my $save = $/;
   undef $/;
   my $file = <>;
   $/ = $save;
   return $file;
}
```

That is, we can save away the current contents to a separate variable, and replace `$/` with its old contents when we're finished. Alternatively, we can get Perl to do the saving and restoring for us automatically: to give a global variable a specific local value, use the `local` operator:

```
sub slurp {
   local $/ = undef;
   my $file = <>;
   return $file;
}
```

`local` gives a variable **runtime scope**. This means that any statement executed between `local` and the end of the block will see the new value of the variable. How does this differ from lexical scope? The key is that, as we've seen in this chapter, program flow doesn't just go straight through blocks of code We can temporarily bounce off into subroutines, too. So, the difference is:

> **Runtime scope means a variable has a temporary value for the duration of the current block, inclusive of any side trips into other subroutine blocks, that is seen everywhere in the program – because it's a global. Lexical scope, on the other hand, creates a variable that is only visible to the statements inside the block.**

Try It Out : Runtime Scope

This program uses `local` to give `$_` a runtime scope. You should be able to see how `local` differs from `my`:

```
#!/usr/bin/perl
# runtime.plx
use strict;
use warnings;
my $x = 10;              # Line 5
$_ = "alpha";
{
   my $x = 20;
   local $_ = "beta";
   somesub();            # Line 10
}
somesub();

sub somesub {
   print "\$x is $x\n";
   print "\$_ is $_\n";
}
```

```
>perl runtime.plx
$x is 10
$_ is beta
$x is 10
$_ is alpha
>
```

How It Works

Can you see what's happening? Although we say `my $x = 20;` on line 8, that only affects statements between line 8 and the end of the block, which is line 11. It's a lexical variable that is constrained by the actual text, not by the order of execution. It doesn't have any effect when we call `somesub` on line 10. `local`, on the other hand, affects everything we do between lines 9 and 11, and that includes calling `somesub`. Its scope is determined by the statements that get executed.

When to Use my() and When to Use local

Mark-Jason Dominus gives simple but effective advice:

> **Don't use `local`. Always use `my`.**

This is somewhat of an overstatement, but it's a justified one. Unless you're dealing with special variables like `$/`, you usually want to use `my`. If you need to lie to Perl for some period of time about a global's value, try rethinking your design.

Passing More Complex Parameters

Sometimes we want to pass things other than an ordinary list of scalars, so it's important to understand how passing parameters works.

@_ Provides Aliases!

Remember when we did something like this:

```
@array = (1, 2, 3, 4);
for (@array) {
   $_++;
}
print "@array\n";
```

We found that this would print "2, 3, 4, 5". The elements of the array had been affected. We said then that the iterator variable is an alias to the elements of the list. Well, the same goes for the elements of `@_`. They're actually aliases for the things we pass. That's why we've got to be careful when we're dealing with `@_` directly. It's dangerous to say, for example:

```
sub add_one_and_double {
   $_[0]++;
   return $_[0]*2;
}
```

because if we tried:

```
add_one_and_double(1);
```

Perl would try to modify a constant, which is by definition impossible. Hence, we tend to avoid using @_ directly and instead make local copies of the arguments, either wholesale into an array:

```
my @args = @_;
```

into named variables as a group:

```
my ($filename, $title, $description) = @_;
```

or individually by calling shift (especially if the number of parameters can vary):

```
my $filename    = shift;
my $title       = shift;
my $description = shift;
```

@_ has, effectively, runtime scope. Each subroutine has its own copy of @_, meaning that if one subroutine calls another, we have not lost the argument values to one of them:

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

first(1,2,3);

sub first {
   print "In first, arguments are @_\n";
   second(4,5,6);
   print "Back in first, arguments are @_\n";

}
sub second {
   print "In second, arguments are @_\n";
}
```

In first, arguments are 1 2 3
In second, arguments are 4 5 6
Back in first, arguments are 1 2 3

The question of which variable has scope to where can often be quite tricky to answer, but remember that a lot of trouble may be avoided by naming your variables wisely in the first place.

Lists Always Collapse

We've seen this before, but it's worth saying it again: when you put an array inside a list, the list collapses. The original structure of the array is lost, even before we start putting anything in the parameter array @_. That's why you can't say something like:

```
check_same(@a, @b)
```

and expect to work out where @a ends and @b starts. As far as Perl's concerned there's just one list there. To get around this, you can use references.

Passing References to a Subroutine

There's actually nothing special about passing references into a subroutine, so long as we remember that we can modify the original value when we dereference:

```
#!/usr/bin/perl
# subrefs1.plx
use warnings;
use strict;

my $a = 5;
increment(\$a);
print $a;

sub increment {
   my $reference = shift;
   $$reference++;
}
```

However, what we can do is use prototypes to take a reference behind the scenes. If in a prototype, instead of a dollar sign, we give a type symbol followed by a backslash, Perl will automatically take a reference to that type of variable. So, `sub something (\$)` will look for a single scalar variable and take a reference to it. `sub something ($\%$)` looks for a scalar, a hash, and a scalar and will take a reference to the hash.

For instance, if we change the above to:

```
#!/usr/bin/perl
# subrefs2.plx
use warnings;
use strict;
sub increment (\$);


my $a = 5;
increment($a);
print $a;


sub increment (\$) {
   my $reference = shift;
   $$reference++;
}
```

Notice how we no longer need to take the reference ourselves. We can just say `increment($a)` instead of `(\$a)`. Other languages call this **pass by reference**, as opposed to **pass by value**. Actually, all we're doing is passing a reference and Perl constructs that for us.

This is exactly how we get arrays and hashes to keep their structure when we're passing them to a subroutine.

Passing Arrays and Hashes to a Subroutine

Because the prototype can make a reference for us, we can actually take arrays, hashes and more complicated data structures and let them keep their structure.

Try It Out : Passing Arrays

So, to see if two arrays have the same contents, you could do this:

```
sub check_same (\@\@) {
   my ($ref_one, $ref_two) = @_;
   # Same size?
   return 0 unless @$ref_one == @$ref_two;
   for my $elem (0..$#$ref_one) {
      return 0 unless $ref_one->[$elem] eq $ref_two->[$elem];
   }
   # Same if we got this far
   return 1;
}
```

Putting that into a program looks like this:

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

sub check_same (\@\@);

my @a = (1, 2, 3, 4, 5);
my @b = (1, 2, 4, 5, 6);
my @c = (1, 2, 3, 4, 5);
print "\@a is the same as \@b" if check_same(@a,@b);
print "\@a is the same as \@c" if check_same(@a,@c);


sub check_same (\@\@) {
   my ($ref_one, $ref_two) = @_;
   # Same size?
   return 0 unless @$ref_one == @$ref_two;
   for my $elem (0..$#$ref_one) {
      return 0 unless $ref_one->[$elem] eq $ref_two->[$elem];
   }
   # Same if we got this far
   return 1;
}
```

As expected:

```
>perl passarray.plx
@a is the same as @c
>
```

How It Works

Using the prototype here and at the top of the program means that Perl will take references to two arrays. Hence, what we'll see in @_ are two array references:

```
sub check_same (\@\@) {
   my ($ref_one, $ref_two) = @_;
```

> **If you use a prototype at the start of your program as a forward definition, you must explicitly use the same prototype again at the definition proper, or Perl will complain of a prototype mismatch.**

We can special-case check the size: if our arrays aren't the same size, there's no way they can be the same.

```
      return 0 unless @$ref_one == @$ref_two;
```

Now we come to the comparison. We're going to stop as soon as we find something that differs, since that proves that they're not the same:

```
      for my $elem (0..$#$ref_one) {
         return 0 unless $ref_one->[$elem] eq $ref_two->[$elem];
      }
```

If we got to the end of the array and we didn't return, then they didn't differ:

```
      return 1;
```

> **This only works when we're passing something to a subroutine. We can't do a similar trick for returning arrays, and hence**
>
> ```
> (@a, @b) = somesub();
> ```
>
> **will never work. The list will be flattened, there'll be no way to tell where `@a` ends and `@b` begins, and everything will end up in `@a`. If you need to do this, pass references to the arrays and have the subroutine fill them.**

Passing Filehandles to a Subroutine

Passing filehandles to a subroutine is somewhat special. You can actually either pass a glob or a reference to a glob. It doesn't make any difference. You can then collect the filehandle into a glob, like this:

```
sub say_hello {
   *WHERE = shift;
   print WHERE "Hi there!\n"
}
say_hello(*STDOUT);
```

Alternatively, you can also collect the filehandle into an ordinary scalar and use that in place of a filehandle, as we do below:

```
sub say_hello {
   my $fh = shift;
   print $fh "Hi there!\n"
}
sub get_line {
   my $fh = shift;
   my $response = <$fh>;
   chomp $response;
   $response =~ s/^\s+//;
   return $response;
}

say_hello(*STDOUT);
get_line (*STDIN );
```

Default Parameter Values

One thing that's occasionally useful is the ability to give the parameters for your subroutine a default value, that is, give the parameter a value to run through the subroutine with if one is not specified when the subroutine is called. This is very easily done with the `||` operator.

The logical or operator, `||`, has a very special feature: it returns the last thing it saw. So, for instance, if we say `$a = 3 || 5`, then `$a` will be set to 3. Because 3 is a true value, it has no need to examine anything else, and so 3 is the last thing it sees. If, however, we say `$a = 0 || 5`, then `$a` will be set to 5; 0 is not a true value, so it looks at the next one, 5, which is the last thing it sees.

Hence, anything we get from `@_` that doesn't have a true value can be given a default with the `||` operator. We can create subroutines with a flexible number of parameters and have Perl fill in the blanks for us:

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

sub log_warning {
   my $message = shift || "Something's wrong";
   my $time    = shift || localtime; # Default to now.
   print "[$time] $message\n";
}

log_warning("Klingons on the starboard bow", "Stardate 60030.2");
log_warning("/earth is 99% full, please delete more people");
log_warning();
```

```
>perl defaults.plx
[Stardate 60030.2] Klingons on the starboard bow
[Wed May 3 04:07:50 2000] /earth is 99% full, please delete more people
[Wed May 3 04:07:51 2000] Something's wrong
>
```

One by-product of specifying defaults for parameters is the opportunity to use those parameters as flags. Your subroutine can then alter its functionality based on the number of arguments passed to it.

Named Parameters

One of the more horrid things about calling subroutines is that you have to remember which order the parameters are set. Was it username first and then password, or host first and then username, or…?

Named parameters are a neat way of solving this. What we'd rather say is something like this:

```
logon( username => $name, password => $pass, host => $hostname);
```

and then give the parameters in any order. Now, Perl makes this really, really easy because that set of parameters can be thought of as a hash:

```
sub logon {
   die "Parameters to logon should be even" if @_ % 2;
   my %args = @_;
   print "Logging on to host $args{hostname}\n";
   ...
}
```

Whether and how often you use named parameters is a matter of style. For subroutines that take lots of parameters, some of which may be optional, it's an excellent idea; For those that take two or three parameters, it's probably not worth the hassle.

References to Subroutines

Just like variables, you can take references to subroutines. That's where the ampersand (&) type symbol comes in.

Declaring References to Subroutines

The same rules apply here as for taking references to variables. Put a backslash before the name, but include the ampersand:

```
sub something { print "Wibble!\n" }

my $ref = \&something;
```

Alternatively, we can create an anonymous subroutine by saying `sub {BLOCK}`:

```
my $ref = sub { print "Wibble!\n" }
```

Calling a Subroutine Reference

Just like before, there are two ways to call subroutine references. Directly:

```
&{$ref};
&{$ref}(@parameters);
&$ref(@parameters);
```

Or through an arrow notation:

```
$ref->();
$ref->(@parameters);
```

Callbacks

OK, now we can create and use subroutine references. Why would we want to? The usual thing we do with them is pass them to another subroutine. This is called a **callback**, because it allows the subroutine to 'call back' our code at certain times. This means we can turn a very general subroutine into something that does exactly what we want.

Try It Out : Using a Callback

For instance, the core module `File::Find` will give us a subroutine called `find`. This takes two (or more) parameters: a callback and a list of directories. All it does – and this is a harder task than it sounds – is go through every file underneath each directory in the list, walk into any directories it finds, and call the callback with certain variables set. We can use this to create a directory browser:

```
#!/usr/bin/perl
# biglist.plx
use warnings;
use strict;
use File::Find;
find ( \&callback, "/") ; # Warning: Lists EVERY FILE ON THE DISK!

sub callback {
   print $File::Find::name, "\n";
}
```

Or we could delete every file whose name ends in `.bak`: (a typical extension for temporary backup files):

```
#!/usr/bin/perl
# backupkill.plx
use warnings;
use strict;
use File::Find;
find ( \&callback, "/") ;

sub callback {
   unlink $_ if /\.bak$/;
}
```

or indeed, anything we want. We'll see more of `File::Find` in Chapter 10, where we'll explain how these examples work. We'll also see at the end of the book that callbacks are particularly important for graphical applications.

Arrays and Hashes of References to Subroutines

Another use for subroutine references is to allow us to call one of a selection of subroutines. For instance, if we're writing a menu system that calls a subroutine related to each menu option. We could naturally write it like this:

```
print "Type c for customer menu, s for sales menu and o for orders menu.\n";
chomp (my $choice = <>);
if     ($choice eq "c") {
   customer_menu();
} elsif ($choice eq "s") {
   sales_menu();
} elsif ($choice eq "o") {
   orders_menu();
} else {
   print "Unknown option.\n";
}
```

However, that's messy. What we're doing is relating a string to a subroutine, and relating one thing to another in Perl should always make you think of a hash. Here's how we could use a hash of subroutine references:

```
my %menu = (
   c => \&customer_menu,
   s => \&sales_menu,
   o => \&orders_menu
}
print "Type c for customer menu, s for sales menu and o for orders menu.\n";
chomp (my $choice = <>);
if (exists $menu{$choice}) {
   # Call it!
   $menu{$choice}->();
} else {
   print "Unknown option.\n";
}
```

Much neater.

Recursion

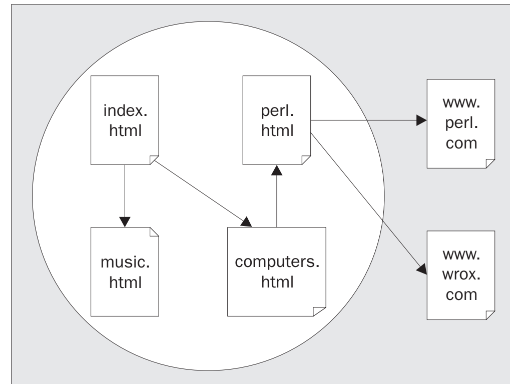
recursion, *n.*: *See* recursion

The above joke, so old it has hair on it, gives you an idea as to what recursion is – it's something that refers to itself in its definition. Specifically, recursion in computer programming is a subroutine that calls itself as part of its operation.

Of course, we have to be careful when we're doing this: we've got to make sure we stop somewhere and that our programs don't loop away into oblivion. The thing that tells us when to stop is called the **terminating condition**.

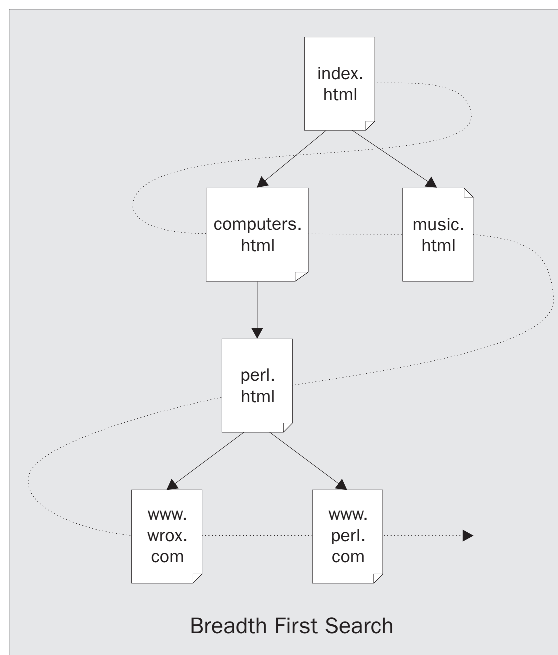
Try It Out : Spidering a Web Site

A web site is a collection of pages linked together in some way. If you're running a web site, you might want to ensure that all the links work properly: that the pages inside your site can be read and that links to other sites on the Internet are still valid. The general procedure we need to follow is something like this: to check a page, get the web page, extract all links, get those pages to ensure that they are valid and reachable, and then check those pages still on our site. So, let's say we had the following set of pages:



We'd start at http://www.mysite.org/index.html, and from there we'd find links to `computers.html` and `music.html` – we'd want to check each of these. Examining each of those for links would give us a link to `perl.html`, where we'd find links to http://www.perl.com/ and http://www.wrox.com/. We'd want to make sure that these pages were reachable, but since these were off our site, we wouldn't examine them for further links. Any broken pages beyond that are not something we can do anything about.

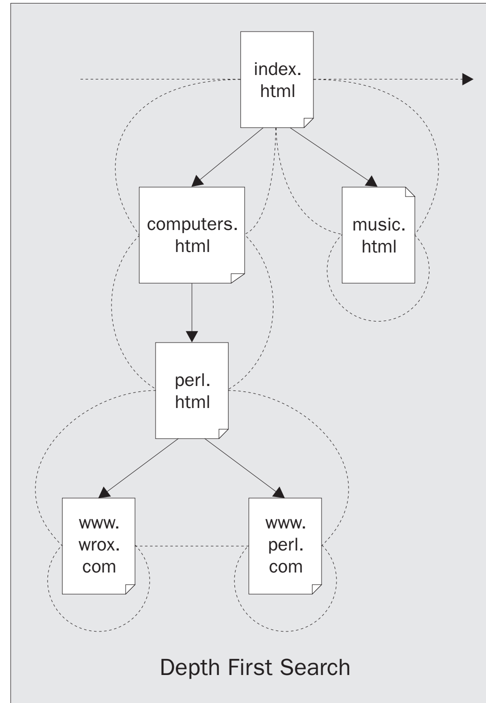
Now, you should notice that there are two routes we can take to do this, starting from `index.html`: We could see extract links on the first level, `computing.html` and `music.html`. Then we could visit the links we got from that level, `perl.html` Then we could go to the external sites, where we'd have to stop. That's called a **breadth-first search**, and it looks like this:



Breadth First Search

The important thing about a breadth-first search is that for each 'level' we need to keep track of which links to visit on the next level. It's what we did when traversing the tree of references at the end of Chapter 6. This isn't recursive, because we're not doing exactly the same thing with each site we get to.

However, there's another way we could do this which avoids the need to explicitly keep track of where we're going next time: we could go first to `computers.html`, then follow the link to `perl.html`, then follow the external links, and then back up and visit `music.html`. If there's a link, we visit it. If not, we go back to where we were. This is a **depth-first search**, and we can implement it recursively:



Depth First Search

Notice how each visit to a page is a similar shape to the ones above? That's the recursion. And notice how on external sites and pages with no links, we close the loop and head off to the next page? Those are our terminating conditions. We'll also add another terminating condition, of course – don't investigate a page if the link is dead.

Here's what it looks like in Perl:

```
sub traverse {
   my $url = shift;
   return if $seen{$url}++;        # Break circular links
   my $page = get($url);
   if ($page) {
      print "Link OK : $url\n";
   } else {
      print "Link dead : $url\n";
      return;                     # Terminating condition : if dead.
   }
   return unless in_our_site($url); # Terminating condition : if external.
   my @links = extract_links($page);
   return unless @links;            # Terminating condition : no links
   for my $link (@links) {
      traverse($link) # Recurse;
   }
}
```

Now let's turn that into a full program:

```
#!/usr/bin/perl
# webchecker.plx
use warnings;
use strict;
my %seen;

print "Web Checker, version 1.\n";
die "Usage: $0 <starting point> <site base>\n"
   unless @ARGV == 2;

my ($start, $base) = @ARGV;
$base .= "/" unless $base=~m|/$|;

die "$start appears not to be in $base\n"
   unless in_our_site($start);
traverse($start);

sub traverse {
   my $url = shift;
   $url =~ s|/$|/index.html|;
   return if $seen{$url}++;        # Break circular links
   my $page = get($url);
   if ($page) {
      print "Link OK : $url\n";
   } else {
      print "Link dead : $url\n";
      return;                     # Terminating condition : if dead.
   }
   return unless in_our_site($url); # Terminating condition : if external.
   my @links = extract_links($page, $url);
   return unless @links;            # Terminating condition : no links
   for my $link (@links) {
      traverse($link) # Recurse
   }
}

sub in_our_site {
   my $url = shift;
   return index($url, $base) == 0;
}

sub get {
   my $what = shift;
   sleep 5; # Be friendly
   return `lynx -source $what`;
}

sub extract_links{
   my ($page, $url) = @_;
   my $dir = $url;
   my @links;
   $dir =~ s|(.*)/.*?$|$1|;
   for (@links = ($page=~/<A HREF=["']?([^\s"'>]+)["']?/gi)) {
      $_ = $base.$_ if s|^/||;
      $_ = $dir."/".$_ if !/^(ht|f)tp:/;
   }
   return @links;
}
```

While it isn't very polished – it's quite primitive – it works:

```
> http://www.wrox.com/Default.asp http://www.wrox.com/
Web Checker, version 1.
Link OK : http://www.wrox.com/Default.asp
Link OK : http://www.wrox.com/Consumer/DJ.asp
Link OK : http://www.wrox.com/Consumer/Store/ListTitles.asp?By=105&Category=Consumer
Link OK : http://www.wroxconferences.com
Link OK : http://www.wrox.com/Consumer/Store/ListTitles.asp?By=104&Category=Consumer
Link OK : http://www.wrox.com/Consumer/Forums/Default.asp?Category=Consumer
Link OK : http://www.wrox.com/Consumer/Store/Download.asp?Category=Consumer
Link OK : http://www.wrox.com/Consumer/EditDetails.asp?Category=Consumer
Link OK : http://www.wrox.com/Consumer/Contacts.asp
…
>
```

Now, we'll see how it works, and then we'll see what's wrong with it.

How It Works

First, we need to know two things: the first URL we're going to start with and the base for the site, so we know when we're about to visit an external site:

```
die "Usage: $0 <starting point> <site base>\n"
   unless @ARGV == 2;

my ($start, $base) = @ARGV;
```

If the base URL doesn't end with a slash, we give it one, since we depend on this fact later:

```
$base .= "/" unless $base=~m|/$|;
```

Next, we'll check that the page we're starting from is actually part of the site. You never know…:

```
die "$start appears not to be in $base\n"
   unless in_our_site($start);
```

And then we kick off the action:

```
traverse($start);
```

Now here's the subroutine we saw above, slightly modified:

```
sub traverse {
   my $url = shift;
```

If the URL ends in a slash, we treat it as an index page:

```
   $url =~ s|/$|/index.html|;
```

This is our first problem. It's a bad assumption. Some sites have the index page as `index.html`, some as `index.htm`, some as `Default.asp` – in fact, it could be anything. The only way we can tell is to look at the exact response from the server when we ask for a URL ending in a slash.

Next we need to make sure we haven't seen the page before, because web sites can have circular links and we don't want to go whizzing around forever:

```
    return if $seen{$url}++;        # Break circular links
```

And we get our page. If we successfully retrieve it, we say so. If the link is dead, there's no point trying to find other links from it:

```
    my $page = get($url);
    if ($page) {
       print "Link OK : $url\n";
    } else {
       print "Link dead : $url\n";
       return;                    # Terminating condition : if dead.
    }
```

We don't look for links in external sites:

```
    return unless in_our_site($url); # Terminating condition : if external.
```

Now we extract the links and give up if we can't find any:

```
    my @links = extract_links($page, $url);
    return unless @links;            # Terminating condition : no links
```

Now we call ourselves on each of the links:

```
    for my $link (@links) {
       traverse($link) # Recurse
    }
```

Imagine how this would work for our example site above: the first call to `index.html` would put `computers.html` and `music.html` in `@links`. Then we'd call ourselves first on `computers.html`, which would in turn call ourselves on `perl.html`, which would then check the external links and return. There's nothing else on `computers.html`, so that'd return. Then we'd move onto `music.html` and return, and we'd be done – this is a depth-first search, just like in the diagram. We look at the first link we see, every time:

```
}
```

Now we come to the helper subroutines:

```
sub in_our_site {
   my $url = shift;
   return index($url, $base) == 0;
}
```

We just check that the URL we're about to look at starts with the same characters as the base. This isn't foolproof, but it's close enough. We know that `$base` has to end in a slash, so the only things we allow if we're looking at http://www.mysite.org/ are things that start http://www.mysite.org/… It counts out FTP, HTTPS, or any other protocol, but it'll do.

```
sub get {
   my $what = shift;
   sleep 5; # Be friendly
   return `lynx -source $what`;
}
```

We use `lynx` again to get our web pages. While we make the effort to be friendly to the web servers by not bombarding them with requests as fast as we can, we don't check that the page we get back is valid. Sometimes if we're behind a cache and we request a dud site, we'll get back a perfectly fine page – with an error message on it! We don't do any error checking here at all! Again, the only way to be really sure is to connect to the server directly and examine the exact response.

> **Thankfully, we don't have to do all that work. There's a module called `LWP::Simple` which provides a subroutine, also called `get`, which does the job properly. If you've got that installed, just add `use LWP::Simple;` after the `use strict;` line, and remove this subroutine. If not, we'll be looking at it and how it works in Chapter 10.**

Now we try and extract the links from the HTML file. There are two problems here: first, finding and extracting the links, and second turning them into real URLs. In order to prepare us for the second problem, we take the URL we've just looked up, and extract the directory name from it:

```
my $dir = $url;
$dir =~ s|(.*)/.*$|$1|;
```

This'll turn http://mysite.org/pictures/index.html into http://mysite.org/pictures, or so we hope.

Now we try and extract the links:

```
for (@links = ($page=~/<A HREF=["']?([^\s"'>]+)/gi)) {
```

We look for all examples of `<A HREF=` followed by an optional double or single quote mark, and then some text that isn't a space, quote mark, or closing tag that we extract. This should extract all the links, right?

I've said before that parsing HTML using regular expressions is a potentially risky operation, and I stand by it. This makes a couple of assumptions that may not always hold true:

- `HREF` always follows `A` with a single space and no elements in between.
- There are no spaces, greater-than signs or quotation marks in the URL. According to the standards, there won't be, but the standards aren't always adhered to.
- This piece of text won't be found inside a `<PRE>` tag, a comment tag, or anything else that changes it from the usual meaning.
- This URL doesn't contain a #-sign to point to a spot in the middle of the page.

And so on. A lot of these assumptions are usually going to be true, but we can't rely on them. As before, the only way to be sure is to go through and check the data piece by piece, and as before there's a module that does this for us. `HTML::LinkExtor` is designed to extract links from HTML files, but it's pretty tricky to use. Further, it's about a hundred times slower. When you must have the right answer, use that; when 'close enough is good enough', use the above.

Now, there are two types of filename that we'll find in there. Absolute URLs include the Internet host they're coming from: for example, http://www.mysite.org/perl.html. Relative URLs, on the other hand, speak about a file on the same server as the current on: for instance, from http://www.mysite.org/perl.html, we could say `/music.html` to get to http://www.mysite.org/music.html. Because relative URLs only give you directions from the current page, not from anywhere on the

Internet, we have to convert them to absolute URLs before looking them up. This is why we need to know the directory name of the file we're currently looking at. The rules for turning relative URLs into absolute ones are tricky, but we simplify them here:

- If the URL starts with a forward slash, it should be taken as from the base of the site. (This is another dangerous assumption – the base of the site may not be the base of the server, which is where relative URLs are really measured from.) Since we know `$base` ends with a forward slash, we string the initial slash from our relative URL and glue them together.
- Otherwise, if it doesn't start with `http://` or `ftp://`, it's a relative URL and we need to add the URL for the current directory to the beginning of it:

```
$_ = $base.$_ if s|^/||;
$_ = $dir."/".$_ if !/^(ht|f)tp:/;
```

> **Again, this may well work for some – or maybe even most – cases, but it's not a complete solution. It doesn't take into account things like the fact that saying "./" refers to another file in the current directory. As usual, there's a module – URI – which can help us convert between relative and absolute URLs.**

Hopefully, I've made you think about some of the assumptions that you can make in your programming and why you need to either cover every case you can think of – or get someone else to cover it for you. In Chapter 10, we'll be looking at modules like the ones mentioned here that can get these things right for us.

Style Point: Writing Big Programs

Subroutines give us the perfect opportunity to think about that it means to program, and how to approach the programming problem. Learning the bricks and mortar of a programming language is one thing, but learning how to put them all together into a complex program is quite another.

One approach, then, is to separate out the various components; if a program's going to be performing a variety of tasks, you're obviously going to need to write code for each one. So, stage one in building a large program:

- Identify what the program will do.

This turns the question from 'How do I write a program which handles my business?' into 'How do I write a program which does X, Y, and Z?' We've now identified individual goals and turned a very general problem into a little more specific one. Now we've got to work out how to achieve those goals. It might be useful at this stage to break the goals into manageable chunks; this is where subroutines can be a useful mirror of the development process.

- Break down goals into a series of ideas

Imagine you're directing a robot. You've got a chair in front of you, and a wardrobe over on one wall. On top of the wardrobe is a box, and you want the robot to bring you the box. Unfortunately, you can't just say 'bring me the box'; that's not a primitive enough operation for the robot. So you have to consider the stages involved and write them out explicitly. So, our draft program would go:

```
Put chair in front of wardrobe.
Stand on chair.
Pick up box.
Get down off chair.
Move to human.
Put down box.
```

That'd certainly be enough for most humans, but it probably wouldn't be enough for most robots. If they don't know about "put something somewhere", you're going to have to break it down some more. This is where subroutines come in, to break down the big tasks into simpler goals:

```
sub "Put chair in front of wardrobe" {
    Move to chair
    Pick up chair
    Move to wardrobe
    Put down chair
}
```

> **Incidentally, this way of interspersing English descriptions with programming terminology to describe the outline of a program is called "pseudocode" – it's one popular way to plan out a program.**

Of course, you may find you have to define things like 'move' and 'pick up' in terms of individual movements; this depends on the tools already provided for you. With Perl, you've got a reasonably high-level set of tools to play with; you don't have to break up strings yourself, for instance, as you do in some languages. Getting to the computer's level is our final stage:

- ❑ Specify each idea in a way the computer can understand.

Easier said than done, of course, because it means you need to know exactly what the computer can and can't understand; thankfully, though, computers are more than able to tell you when they can't understand something. Effectively, though, programming is just explaining how you want a task to be performed, in simple enough stages. Subroutines give you the ability to group those stages around individual tasks.

Summary

Subroutines are a bit of code with a name, and they allow us to do two things: chunk our program into organizational units and perform calculations and operations on pieces of data, possibly returning some more data. The basic format of a subroutine definition is:

```
sub name BLOCK
```

We can call a subroutine by just saying `name` if we've had the definition beforehand. If the definition is lower down in the program, we can say `name()`, and you may see `&name` used in older programs. Otherwise, we can use a forward definition to tell Perl that `name` should be interpreted as the name of a subroutine.

When we pass parameters to a subroutine, they end up in the special array `@_` – this contains aliases of the data that was passed. Prototypes allow us to specify how many parameters to accept, and they also allow us to pass references instead of aliases; this in turn allows us to pass arrays and hashes without them being flattened.

We can take references to subroutines by saying `\&name`, and use them by saying `$subref->()` or `&$subref`. We can get anonymous subroutines by saying `sub { BLOCK }` with no name. Subroutine references give us callbacks and the ability to fire off a subroutine from a set of several.

Ordinary subroutines are allowed to call other subroutines; they're also allowed to call themselves, which is called recursion. Recursion needs a terminating condition, or else the subroutinr will never end. Perl takes care of where it's going, where it came from, and the parameters that have been passed at each level.

Finally, we looked at how to divide up programs into subroutines, as well as the top-down level of programming: start with the goal, then subdivide into tasks and put these tasks into subroutines. Then subdivide again if necessary, until we've got to a level that the computer can understand.

Exercises

1. Go back to the seconds1.plx program seen earlier in the chapter. Rewrite it so that it contains a second subroutine that asks the user for a number, puts it into a global variable and converts that into hours, minutes, and seconds.

2. Create three subroutines such that each identify themselves on screen and then calls the next in the list – that is, sub1 calls sub2 which calls sub3 – until 300 subroutine calls have been made in total. When that does occur, break out of the loop and identify which was the last subroutine called. First do this using a global variable....

3. Repeat this exercise passing the current call number and call limit around as parameters

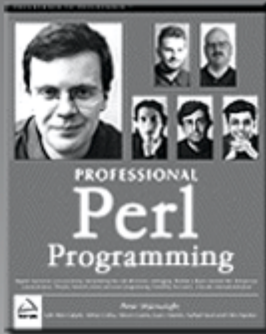
4. Write a subroutine that receives by reference an array containing a series of numbers, initially (1, 1). The subroutine then calculates the sum of the two most recent references and adds another element to the array that is the sum of both. Do this ten times and then print out your array. You should get the first twelve numbers of the Fibonacci sequence

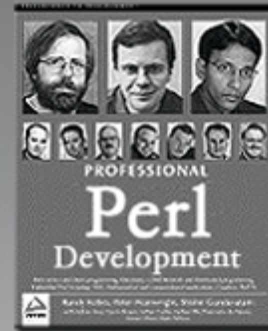
```
1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144
```

Source code available at : www.wrox.com

Peer discussion at : lamplists.com

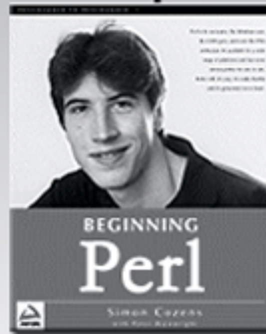
Also from Wrox





http://www.wrox.com/books/1861004494.htm

http://www.wrox.com/books/1861004389.htm



http://www.wrox.com/books/1861003145.htm

lamplists.com

The Open Source Programmer's Resource Centre

This work is licensed under the Creative Commons **Attribution-NoDerivs-NonCommercial** License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nd-nc/1.0 or send a letter to Creative Commons, 559 Nathan Abbott Way, Stanford, California 94305, USA.

The key terms of this license are:

Attribution: The licensor permits others to copy, distribute, display, and perform the work. In return, licensees must give the original author credit.

No Derivative Works: The licensor permits others to copy, distribute, display and perform only unaltered copies of the work -- not derivative works based on it.

Noncommercial: The licensor permits others to copy, distribute, display, and perform the work. In return, licensees may not use the work for commercial purposes -- unless they get the licensor's permission.