7

References

Way back in Chapter 2 we learned that we couldn't get away with putting one list inside another. Perl flattens lists, and an inner list would get subsumed into whatever we try to put it inside. Similarly, hashes have a single scalar key attached to a single scalar value; there's apparently no way we can put several pieces of data in one hash key.

However, these are both things we'll want to do from time to time. For instance, we might want to represent a chessboard as eight lists of eight squares so that we can address each square by row and column. We might also want to store information about someone – their address, phone number, and occupation – and key it to their name.

Of course, we've seen ways we could do this already: We could store our chessboard as an array of 64 squares, and write some code to convert between row-and-column co-ordinates and a number from 0 to 63. For the address book, we could just use three hashes, each using the same set of names as keys – not a terribly elegant solution but one that does the job with the techniques we've seen so far.

However, in this chapter, we're going to be looking at a very powerful facility in Perl that lets us do this sort of thing and a whole lot more besides – **references**.

What Is a Reference?

Put at its very simplest, a reference is a piece of data that tells us the location of another piece of data. If I told you to "see the first paragraph on page 130", I'd effectively be giving you a reference to the text in that paragraph. It wouldn't be the text itself, but it would tell us where to find it. This would also let us talk about (refer to) the text right away, despite the fact that it's somewhere else in the book. That's why references are so useful – we can specify data once, and they let us access it from wherever else we are.

In Perl, a reference is always a scalar, although the data it refers to may not be: our cross-reference above wasn't even a sentence, but referred to an entire paragraph. Likewise, a reference, even though it's only a scalar, can talk about the data stored in an array or hash.

Languages like C and C++ have a feature that's similar to references, called **pointers**. Now if you're familiar with pointers, please try and put the knowledge aside while you're going through this chapter. They are similar to references in that both point us to locations in the computer's memory. However, pointers tend to leave interpretation of what's there for the programmer to disentangle. References, on the other hand, only store memory locations for specific, clearly defined data structures – maybe not *pre*defined, but defined nevertheless. They allow you to leave the arrangement of computer memory to the computer itself. For me, this is a huge relief, as the machine's far better at that sort of thing than I am.

The main use we have for references is the one we discussed above – as flat-pack storage for arrays and hashes. We can now refer *unambiguously* to the contents of an array or a hash, using a single scalar, so we're now in a position to do things like putting hashes inside hashes, lists inside lists, even hashes in lists, and vice-versa. But that's not all…

Anonymity

We can also use references to create **anonymous data**. Anonymous data, as you might have guessed, is data that doesn't have a variable name attached to it. Instead, it's placed at a certain memory location, and we're given a simple reference to that location. Our list (or hash or whatever) has no name to speak of, but we know exactly where to find it, should we need to use it.

This is a bit like literal data, where we had literal scalars and lists in our program, but not quite – literal data was constant: we couldn't change it.

For example, instead of creating an array `(1,2,3)` called `@array` and then creating a reference to `@array,` we can cut out the middle man, by referencing `(1,2,3)` directly.

This lets us create real scalars, arrays, and hashes, containing data that we can refer to and modify, just as if it were a normal variable. This doesn't mean that we leave arrays and hashes floating about randomly in our program to be plucked out of the air whenever we need them. We know where to find this anonymous data (we have a reference that's telling us just this), and it only exists for as long as part of our program is using it.

The Lifecycle of a Reference

To understand how we deal with references, let's look at the three areas of a reference's life cycle – creation, use, and destruction. After that, we'll see how we can practically use references to create more complicated data structures than simple arrays and hashes.

Reference Creation

There are two ways to create a reference, one for each of the following situations:

- You've already got the data in a variable.
- You want to use anonymous data to go straight to a reference.

The simple rule for the first situation where the variable is already defined is:

You create a reference by putting a backslash in front of the variable.

That's it. Let's see some examples:

```
my @array   = (1, 2, 3, 4, 5);
my $array_r = \@array;
```

We create a perfectly normal array variable and then take a reference to it by putting a backslash before the variable's name. That's literally all there is to it. In the same way, we can create a reference to a hash:

```
my %hash   = ( apple => "pomme", pear => "poire" );
my $hash_r = \%hash;
```

or a scalar:

```
my $scalar   = 42;
my $scalar_r = \$scalar;
```

We can treat our references just like ordinary scalars, so we can put them in an array:

```
my $a = 3;
my $b = 4;
my $c = 5;
my @refs = (\$a, \$b, \$c);
```

Or, if you don't like putting so many backslashes in your array definitions, you can declare this kind of array in a second way:

```
my @refs=\($a, $b, $c);
```

So, if you try referencing a list, you won't actually get a reference to the list, but rather a list of references to each element *in* the list. If this isn't what you want, you can always put the data into an array. We can also put references in a hash, but only as values. Perl doesn't yet support references as hash keys. You can certainly do this, though:

```
my @english = qw(January February March April May June);
my @french  = qw(Janvier Fevrier  Mars  Avril Mai Juin);
my %months  = ( english => \@english, french => \@french );
```

So what does this give us? We have a hash with two keys, `english` and `french`. The `english` key contains a reference to an array of English month names, while the `french` key contains a reference to an array of French month names. With these references, we can access and modify the original data, which means that, in effect, we've stored two arrays inside a single hash.

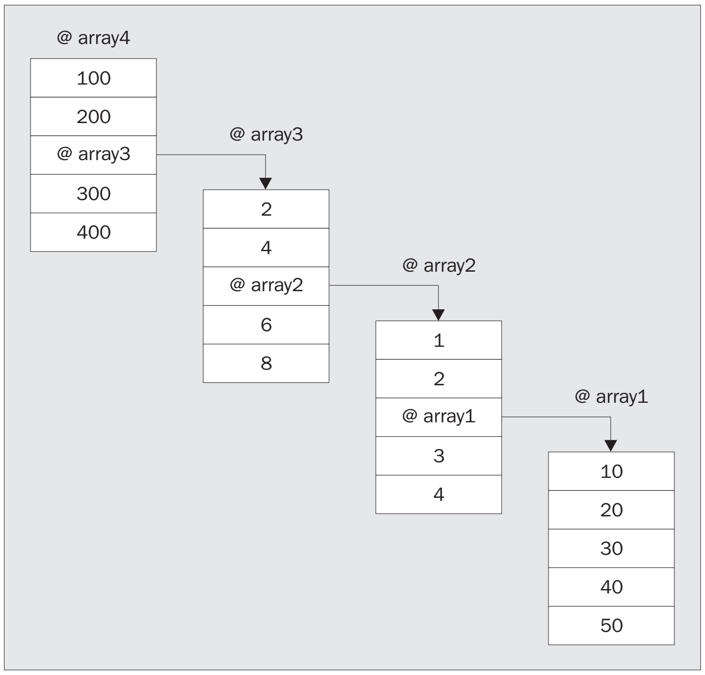
We can use the same trick to store arrays inside arrays:

```
my @array1 = ( 10, 20, 30, 40 );
my @array2 = ( 1, 2, \@array1, 3, 4);
```

Now `@array2` is made up of five scalars, and the middle one is a reference to another array. We can do this over and over again, if we want to:

```
my @array3 = (2, 4, \@array2, 6, 8);
my @array4 = (100, 200, \@array3, 300, 400);
```

This gives us a very versatile way to store complex data structures. What we've just done is to store a structure that looks like this:



Anonymous References

Our next step is to do all this without having to go through the interim stages of creating the variables. Anonymous references will let us go straight from our raw data to a reference, and the rules here are just as simple:

To get an array reference instead of an array, use square brackets `[]` instead of parentheses.

To get a hash reference instead of a hash, use curly braces `{}` instead of parentheses.

So, referring to our examples above, instead of doing this:

```
my @array   = (1, 2, 3, 4, 5);
my $array_r = \@array;
```

we can go straight to an array reference like this:

```
my $array_r = [1, 2, 3, 4, 5];
```

Likewise, to get a hash reference, instead of doing this:

```
my %hash   = ( apple => "pomme", pear => "poire" );
my $hash_r = \%hash;
```

we say:

```
my $hash_r = { apple => "pomme", pear => "poire" };
```

We can put anonymous references inside hashes and arrays, just like references created from variables:

```
my %months  = (
   english => ["January", "February", "March", "April", ",May", ",June"],
   french  => ["Janvier", "Fevrier", "Mars", "Avril", "Mai", "Juin"]
);
```

And we can put references inside references:

```
my @array = ( 100,200,[ 2,4,[ 1,2,[ 10,20,30,40,50 ],3,4 ],6,8 ],300,400 );
```

That's exactly the same structure as we created above. Here it is again, with a lot more spacing added:

```
my @array = ( 100, 200,
               [ 2, 4,
                  [ 1, 2,
                     [ 10, 20, 30, 40, 50 ],
                  3, 4 ],
               6, 8 ],
            300, 400 );
```

What about creating an anonymous scalar – what happens if we try this? Well, as we saw above, trying to create a reference to a list gives us a list of references to the list's elements. So if we did this:

```
my @refs = \(1, 2, 3, 4);
```

we'd expect it to give us four references, to `1`, `2`, `3`, and `4`. Perl does in fact do this, but while it will let us retrieve the numbers, it won't allow us to change them – it's almost like trying to modify a literal in your variable. If we ever want to get a scalar reference, it's best to use a temporary variable.

Using References

Once we've created our references (whether to real variables or anonymous data), we're going to want to use them. So how do we access the data? The operation we use to get data back from a reference is called **dereferencing**, and once again, the rule's very simple:

To dereference data, put the reference in curly braces wherever you would normally use a variable's name.

First, we'll see how to do this with arrays. Say we've got an array and a reference:

```
my @array   = (1, 2, 3, 4, 5);
my $array_r = \@array;
```

We can get at the array like this:

```
my @array2  = @{$array_r};
```

We put the reference, `$array_r`, inside curly braces, and use that instead of our original array variable `@array`. We can use this dereferenced array, `@{$array_r}`, anywhere we might otherwise use an array:

Try It Out : Constructing and Dereferencing

For our first attempt, we'll do something simple. We'll just create a reference to an array, then use it as we'd normally use an array:

```
#!/usr/bin/perl
# deref1.plx
use warnings;
use strict;

my @array   = (1, 2, 3, 4, 5);
my $array_r = \@array;

print "This is our dereferenced array: @{$array_r}\n";
for (@{$array_r}) {
   print "An element: $_\n";
}
print "The highest element is number $#{$array_r}\n";
print "This is what our reference looks like: $array_r\n";
```

Let's run this:

```
>perl deref1.plx
This is our dereferenced array: 1 2 3 4 5
An element: 1
An element: 2
An element: 3
An element: 4
An element: 5
The highest element is number 4
This is what our reference looks like: ARRAY(0xa063fbc)
>
```

How It Works

We've seen a few examples of creating references now, so you should be familiar with this syntax. First, we define an array variable and its contents and then backslash it to create a reference to it.

```
my @array   = (1, 2, 3, 4, 5);
my $array_r = \@array;
```

Now we can use `@{$array_r}` instead of `@array`. Both refer to exactly the same data, and both do exactly the same things. For instance, `@{$array_r}` will interpolate inside double quotes:

```
print "This is our dereferenced array: @{$array_r}\n";
```

Just as if we'd used the original `@array`, our dereferenced array prints out the contents of the array, separated by spaces:

```
This is our dereferenced array: 1 2 3 4 5
```

In the same way, we can use the array in a `for` loop, with no surprises:

```
for (@{$array_r}) {
   print "An element: $_\n";
}
```

Finally, we can also get the highest element number in the array, just as if we'd said `$#array`, like this:

```
print "The highest element is number $#{$array_r}\n";
```

Now, we take a look at what our reference actually looks like itself. After all, it's a scalar, so it must have a value that we can print out and look at. It does, and this is what we get if we print out the reference:

```
This is what our reference looks like: ARRAY(0xa063fbc)
```

Well, the ARRAY part obviously tells us that we have an array reference, but what about the part in brackets? Well, we know that a reference is a memory location, telling us where the data is stored in the computer's memory. We generally don't need to worry about this actual value, as we can't do that much with it. Note also that it's unlikely that you'll get exactly the same value as I have here. It will simply depend on what hardware your system has, what other software you're running, and what perl is doing.

There is one way you might want to make use of this value directly: to see if two references refer to the same piece of data, you can compare them as numbers using ==.

If we try and manipulate it, it ceases to be a reference and becomes an ordinary number – the value of the hexadecimal above. We can see that if we run the following program:

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

my $ref = [1, 2, 3];
print "Before: $ref\n";
print "@{$ref}\n";
$ref++;
print "After: $ref\n";
print "@{$ref}\n";
```

will give us something like this:

```
>perl noref.plx
Before: ARRAY(0xa041160)
1 2 3
After: 168038753
Can't use string ("168038753") as an ARRAY ref while "strict refs" in use at noref.plx line 11.
>
```

When we tried to modify our reference, it degenerated to the ordinary number `168038752`, which is the `0xa041160` mentioned above. Adding one to that gave us the number above, which is an ordinary string, rather than a reference. Perl then complains if we try and use it as a reference.

This is why we can't use references as hash keys – these can only be strings, so our references will get 'stringified' to something like the form above. Once that happens, we're not able to use them as references again.

Array Elements

What about the individual elements in an array? How do we access these? Well, the rule is pretty much the same as for the array as a whole; just use the reference in curly braces in the same way you would the array name:

```
#!/usr/bin/perl
# deref2.plx
use warnings;
use strict;

my @band = qw(Crosby Stills Nash Young);
my $ref  = \@band;
for (0..3) {
   print "Array    : ", $band[$_]  , "\n";
   print "Reference: ", ${$ref}[$_], "\n";
}
```

As you can see, these refer to the same thing:

```
>perl deref2.plx
Array       : Crosby
Reference   : Crosby
Array       : Stills
Reference   : Stills
Array       : Nash
Reference   : Nash
Array       : Young
Reference   : Young
>
```

The important thing to note here is that these are not two different arrays – they are two ways of referring to the *same* piece of data. This is very important to remember when we start modifying references.

Reference Modification

If we want to modify the data referred to by a reference, the same rule applies as before. Replace the name of the array with the reference in curly brackets. However, when we do this, the data in the original array will change, too:

```
#!/usr/bin/perl
# modify1.plx
use warnings;
use strict;

my @band = qw(Crosby Stills Nash Young);
my $ref  = \@band;
print "Band members before: @band\n";
pop @{$ref};
print "Band members after: @band\n";
```

```
>perl modify1.plx
Band members before: Crosby Stills Nash Young
Band members after: Crosby Stills Nash
>
```

We can still use `push`, `pop`, `shift`, `unshift` (and so on) to manipulate the array. However, in doing so, we'll also be changing what's stored in `@band`.

It's quite possible to have multiple references to the same data. Just as before, if you use one to change the data, you change it for the others, too. This will give the same results as before:

```
my @band = qw(Crosby Stills Nash Young);
my $ref1 = \@band;
my $ref2 = \@band;
print "Band members before: @band\n";
pop @{$ref1};
print "Band members after: @{$ref2}\n";
```

The same goes for anonymous references:

```
my $ref1 = [qw(Crosby Stills Nash Young)];
my $ref2 = $ref1;
print "Band members before: @{$ref2}\n";
pop @{$ref1};
print "Band members after: @{$ref2}\n";
```

Notice here that we're using `[qw(…)],` which is the same as saying

```
[('Crosby', 'Stills', 'Nash', 'Young')]
```

and the brackets inside get removed, just like when we said `((1,2,3))` back in Chapter 3.

Because anonymous references give us a reference straight away, it's possible to say things like:

```
@{[ 1, 2, 3 ]}
```

This little bit of trickery (thanks to Randal Schwartz) will, of course, give us the list 1, 2, 3. However, it's less useless than it seems. An array dereference will interpolate just like an ordinary array, so you can use this to make functions interpolate inside strings. For example:

```
print "The time is @{[scalar localtime]} according to my clock";
```

will display just the same as:

```
print "The time is ", scalar localtime, " according to my clock";
```

You can also modify individual elements, using the syntax `${$reference}[$element]`:

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

my @array = (68, 101, 114, 111, 117);
my $ref = \@array;
${$ref}[0] = 100;
print "Array is now : @array\n";
```

```
>perl modelem.plx
Array is now 100 101 114 111 117
>
```

And again, you can do the same with anonymous data:

```
my $ref = [68, 101, 114, 111, 117];
${$ref}[0] = 100;
print "Array is now : @{$ref}\n";
```

Hash References

For references to hashes, the rule is exactly the same. So, to access the hash that a reference points to, you use `%{$hash_r}`. If you want to get at a hash entry `$hash{green}`, you say `${hash_r}{green}`:

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

my %hash = (
   1 => "January",    2 => "February", 3 => "March",     4 => "April",
   5 => "May",        6 => "June",     7 => "July",      8 => "August",
   9 => "September", 10 => "October", 11 => "November", 12 => "December"
);

my $href = \%hash;
for (keys %{$href}) {
   print "Key: ", $_, "\t";
   print "Hash: ",$hash{$_}, "\t";
   print "Ref: ",${$href}{$_}, "\n";
}
```

As expected, we get the same data when using the hash as when using the reference:

```
>perl hash.plx
Key: 1    Hash: January      Ref: January
Key: 2    Hash: February     Ref: February
Key: 3    Hash: March        Ref: March
Key: 10   Hash: October      Ref: October
Key: 4    Hash: April        Ref: April
Key: 11   Hash: November     Ref: November
Key: 5    Hash: May          Ref: May
Key: 12   Hash: December     Ref: December
Key: 6    Hash: June         Ref: June
Key: 7    Hash: July         Ref: July
Key: 8    Hash: August       Ref: August
Key: 9    Hash: September    Ref: September
>
```

This should also help to remind you that Perl's hashes aren't ordered as you might expect!

Notation Shorthands

There are two more rules, but they're not essential for understanding and using references. They just make it easier for us to write programs manipulating references:

You don't have to write the curly brackets.

You may find that it makes your code a little clearer if you omit the curly brackets around the reference. For example, we could rewrite our original dereferencing example `deref1.plx` like this:

```
#!/usr/bin/perl
# dref1alt.plx
use warnings;
use strict;

my @array   = (1, 2, 3, 4, 5);
my $array_r = \@array;

print "This is our dereferenced array: @$array_r\n";
for (@$array_r) {
   print "An element: $_\n";
}
print "The highest element is number $#$array_r\n";
print "This is what our reference looks like: $array_r\n";
```

Our hash example `hash.plx` would then look like this:

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

```
my %hash = (
   1 => "January",    2 => "February", 3 => "March",     4 => "April",
   5 => "May",        6 => "June",     7 => "July",      8 => "August",
   9 => "September", 10 => "October", 11 => "November", 12 => "December"
);

my $href = \%hash;
for (keys %$href) {
   print "Key: ", $_, " ";
   print "Hash: ",$hash{$_}, " ";
   print "Ref: ",$$href{$_}, " ";
   print "\n";
}
```

However, it may sometimes be clearer to leave the curly brackets in. Consider these three assignments:

```
$$hashref{KEY}   = "VALUE"; # 1
${$hashref}{KEY} = "VALUE"; # 2
${$hashref{KEY}} = "VALUE"; # 3
```

Case 1 is the same as case 2, whereas case 3 dereferences the scalar reference stored in `$hashref{KEY}`.

You can also run into problems when you have one reference stored inside another. If we have the following array reference:

```
$ref = [ 1, 2, [ 10, 20 ] ];
```

we can get at the internal array reference by saying `${$ref[2]}`. But say we want to get at the second element of that array – the one containing the value 20. Well, we could store the reference inside another scalar and then dereference it, like this:

```
$inside  = ${$ref}[2];
$element = ${$inside}[1];
```

Or we could get the element directly, by repeatedly substituting references for array names:

```
$element = ${${ref}[2]}[1];
```

This gets very ugly, very quickly, especially if you're dealing with hash references, where it becomes hard to tell if the curly braces surround a reference or a hash key.

So, to help us clear it up again, we introduce another rule:

Instead of `${$ref}`, we can say `$ref->`

Let's demonstrate this, by taking one of our previous examples, `modelem.plx`, and incorporating this into the code. Here's the relevant piece of the original:

```
my @array = (68, 101, 114, 111, 117);
my $ref = \@array;
${$ref}[0] = 100;
print "Array is now : @array\n";
```

and here it is rewritten:

```
my @array = (68, 101, 114, 111, 117);
my $ref = \@array;
$ref->[0] = 100;
print "Array is now : @array\n";
```

Likewise for hashes, we can use this arrow notation to make things a bit clearer for ourselves. Recall `hash.plx` from a little while ago:

```
for (keys %{$href}) {
   print "Key: ", $_, " ";
   print "Hash: ",$hash{$_}, " ";
   print "Ref: ",${$href}{$_}, " ";
   print "\n";
}
```

Instead of that, we can write:

```
for (keys %{$href}) {
   print "Key: ", $_, " ";
   print "Hash: ",$hash{$_}, " ";
   print "Ref: ",$href->{$_}, " ";
   print "\n";
}
```

Now we can get at our array-in-an-array like this:

```
$ref = [ 1, 2, [ 10, 20 ] ];
$element = {$ref->[2]}->[1];
```

or more simply:

```
$element = $ref->[2]->[1];
```

However, we've got one more sub-rule that can simplify this even further:

Between sets of brackets, the arrow is optional.

We can therefore rewrite the above as:

```
$element = $ref->[2][1];
```

Personally, I never omit the arrow in this way – it's far too easy to confuse `$ref->[0][1]` with `$ref[0][1]`, which perl will interpret as a dereference of the first element in the ordinary array `@ref`.

Reference Counting and Destruction

We've now seen all the ways you can create and use references. So when and how are references destroyed? Well, every piece of data in Perl has something called a **reference count** attached to it. This keeps track of the number of instances of the executing code accessing that exact chunk of data.

When we create a reference to some data, the data's reference count goes up by one. When we stop referring to it – we reassign the reference variable or 'break' it (as we saw above, when we tried to modify its value) – the reference count goes down. When nobody's using the data, and the reference count gets down to zero, the data is removed. Consider the following example:

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

my $ref;
{
   my @array = (1, 2, 3);
   $ref = \@array;
   my $ref2 = \@array;
   $ref2 = "Hello!";
}
undef $ref;
```

Now, let's look at the references to the array `(1, 2, 3)` as we go through the program. To start with, the array is created, and the data `(1, 2, 3)` has one reference, which is in use by the array `@array`:

```
my $ref;
{
   my @array = (1, 2, 3);
```

Now we've created another reference to it, and the reference count increases to two:

```
   $ref = \@array;
```

Once again we create a reference, and the count goes up to three:

```
   my $ref2 = \@array;
```

However, we've now changed that reference to be an ordinary string – it's not pointing at our array any more, so the reference count on `(1, 2, 3)` goes back down to two. Note that changing `$ref2` doesn't affect the original array. That only happens when we dereference:

```
   $ref2 = "Hello!";
```

Now a block ends, and all the lexical variables – the `my` variables – inside that block go out of scope. That means that `$ref2` and `@array` are destroyed. The reference count of the data `(1, 2, 3)` goes down again because `@array` is no longer using it. However, `$ref` still has a reference to it, so the reference count is still one, and the data itself is not removed from the system. `$ref` still refers to `(1, 2, 3)` and can access and change this data as before, that is, of course, until we get rid of it:

```
}
```

Now the final reference to the data `(1, 2, 3)` is removed, that array is finally freed:

```
undef $ref;
```

Counting Anonymous References

Anonymous data works in the same way. However, it doesn't get its initial reference count from being attached to a variable, but rather from when its first explicit reference is created:

```
my $ref = [1, 2, 3];
```

This data therefore has a reference count of one, rather than:

```
my @array = (1, 2, 3);
my $ref = \@array;
```

which has a count of two.

Using References for Complex Data Structures

Now that we've looked at what references are, you might be asking: why on earth would we want to use them? Well, as we mentioned in the introduction, we often want to create data structures that are more complex than simple arrays or hashes. We may need to store arrays inside arrays, or hashes inside hashes, and References help us do this.

So let's now take a look at a few of the complex data structures we can create with references.It won't be exhaustive by any means, but it should serve to give you ideas as to how complex data structures look and work in Perl, and it should also help you to understand the most common data structures.

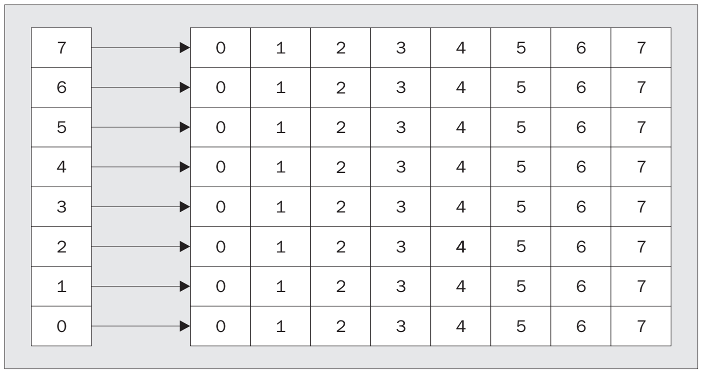
Matrices

What is a matrix? No, not the thing that Keanu Reeves wants out of. A matrix is simply an array of arrays. You can refer to any single element with a combination of two subscripts, which you might want to think of as a row number and a column number. It's harking back to the chessboard example we mentioned in the introduction to this chapter.

If you use the arrow syntax, matrices are very easy to use. You get at an element by saying:

```
$array[$row]->[$column]
```

`$array[$row]` is an array reference, and we're derefencing the `$column`'th element in it. With a chessboard example, it would look like this:



So, `$array[0]->[0]` would be the bottom left hand corner of our chessboard, and `$array[7]->[7]` would be the top right.

Autovivification

Now, there's one last thing we need to know about references before we go on. If we assign values to a reference, perl will automatically create all appropriate references necessary to make it work. So, if we say this:

```
my $ref;
$ref->{UK}->{England}->{Oxford}->[1999]->{Population} = 500000;
```

perl will automatically know that we need `$ref` to be a hash reference. So, it'll make us a nice new anonymous hash:

```
$ref = {};
```

Then we need `$ref->{UK}` to be a hash reference, because we're looking for the hash key `England`; that hash entry needs to be an array reference, and so on. Perl effectively does this:

```
$ref = {};
$ref->{UK} = {};
$ref->{UK}->{England} = {};
$ref->{UK}->{England}->{Oxford} = [];
$ref->{UK}->{England}->{Oxford}->[1999] = {};
$ref->{UK}->{England}->{Oxford}->[1999]->{Population} = 500000;
```

What this means is that we don't have to worry about creating all the entries ourselves. So we can just write:

```
my @chessboard;
$chessboard[0]->[0] = "WR";
```

This is called **autovivification** – things springing into existence. We can use it to greatly simplify the way we use references:

Try It Out : A Chess Game

Now that we can represent our chessboard, let's set up a chess game. This will consist of two stages: setting up the board, and making moves. The computer will have no idea of the rules, but will simply function as a board, allowing us to move pieces around. Here's our program:

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

my @chessboard;
my @back = qw(R N B Q K N B R);
for (0..7) {
   $chessboard[0]->[$_] = "W" . $back[$_]; # White Back Row
   $chessboard[1]->[$_] = "WP";            # White Pawns
   $chessboard[6]->[$_] = "BP";            # Black Pawns
   $chessboard[7]->[$_] = "B" . $back[$_]; # Black Back Row
}

while (1) {
   # Print board
   for my $i (reverse (0..7)) { # Row
       for my $j (0..7) {       # Column
          if (defined $chessboard[$i]->[$j]) {
            print $chessboard[$i]->[$j];
          } elsif ( ($i %2) == ($j %2) ) {
             print "..";
          } else {
             print "  ";
          }
         print " ";  # End of cell
      }
      print "\n";     # End of row
   }

   print "\nStarting square [x,y]: ";
   my $move = <>;
   last unless ($move =~ /^\s*([1-8]),([1-8])/);
   my $startx = $1-1; my $starty = $2-1;

   unless (defined $chessboard[$starty]->[$startx]) {
      print "There's nothing on that square!\n";
      next;
   }
   print "\nEnding square [x,y]: ";
   $move = <>;
   last unless ($move =~ /([1-8]),([1-8])/);
   my $endx = $1-1; my $endy = $2-1;

   # Put starting square on ending square.
   $chessboard[$endy]->[$endx] = $chessboard[$starty]->[$startx];
   # Remove from old square
   undef $chessboard[$starty]->[$startx];
}
```

Now let's see the first part of a game in progress:

```
> perl chess.plx
 BR  BN  BB  BQ  BK  BN  BB  BR
 BP  BP  BP  BP  BP  BP  BP  BP
     ..      ..      ..      ..
 ..      ..      ..      ..
     ..      ..      ..      ..
 ..      ..      ..      ..
 WP  WP  WP  WP  WP  WP  WP  WP
 WR  WN  WB  WQ  WK  WN  WB  WR

Starting square [x,y]: 4,2

Ending square [x,y]: 4,4
 BR  BN  BB  BQ  BK  BN  BB  BR
 BP  BP  BP  BP  BP  BP  BP  BP
     ..      ..      ..      ..
 ..      ..      ..      ..
     ..      WP      ..      ..
 ..      ..      ..      ..
 WP  WP  WP  ..  WP  WP  WP  WP
 WR  WN  WB  WQ  WK  WN  WB  WR

Starting square [x,y]: 4,7

Ending square [x,y]: 4,5
 BR  BN  BB  BQ  BK  BN  BB  BR
 BP  BP  BP  .   BP  BP  BP  BP
     ..      ..      ..      ..
 ..      ..  BP  ..      ..
     ..      WP      ..      ..
 ..      ..      ..      ..
 WP  WP  WP  ..  WP  WP  WP  WP
 WR  WN  WB  WQ  WK  WN  WB  WR
```

How It Works

Our first task is to set up the chessboard, with the pieces in their initial positions. Remember that we're assigning `$chessboard[$row]->[$column] = $thing`. First, we set up an array of pieces on the 'back row'. We'll use this to make it easier to put each piece in its appropriate column:

```
my @back = qw(R N B Q K N B R);
```

Now we'll go over each column:

```
for (0..7) {
```

In row zero, the back row for white, we want to place the appropriate piece from the array in each square:

```
    $chessboard[0]->[$_] = "W" . $back[$_]; # White Back Row
```

In row one of each column, we want a white pawn, WP:

```
      $chessboard[1]->[$_] = "WP";            # White Pawns
```

Now we do the same again for black's pieces on rows 6 and 7:

```
      $chessboard[6]->[$_] = "BP";            # Black Pawns
      $chessboard[7]->[$_] = "B" . $back[$_]; # Black Back Row
  }
```

What about the rest of the squares on board? Well, they don't exist right now, but will spring into existence when we try and read from them.

Next we go into our main loop, printing out the board and moving the pieces. To print the board, we obviously want to look at each piece. So we loop through each row and each column:

```
      for my $i (reverse (0..7)) { # Row
          for my $j (0..7) {       # Column
```

If the element is defined, it's because we've put a piece there, so we print it out:

```
             if (defined $chessboard[$i]->[$j]) {
               print $chessboard[$i]->[$j];
```

Note that at this point, we're accessing all 64 squares. this means any square that didn't exist before will do from now on. This next piece of prettiness prints out the "checkered" effect. On a checkerboard, dark squares come on odd rows in odd columns and even rows in even columns. `$x % 2` tests whether `$x` divides equally by two – whether it is odd or even. If the 'oddness' (or 'evenness') of the row and column is the same, we print a dark square:

```
             } elsif ( ($i %2) == ($j %2) ) {
                print "..";
```

Otherwise, we print a blank square consisting of two spaces:

```
             } else {
                print "  ";
             }
```

To separate the cells, we use a single space:

```
            print " ";  # End of cell
         }
```

And at the end of each row, we print a new line:

```
         print "\n";     # End of row
      }
```

Now we ask for a square to move from:

```
      print "\nStarting square [x,y]: ";
      my $move = <>;
```

We're looking for two digits with a comma in the middle:

```
      last unless ($move =~ /([1-8]),([1-8])/);
```

Now we convert human-style coordinates (1 to 8) into computer-style coordinates (0 to 7):

```
      my $startx = $1-1; my $starty = $2-1;
```

Next, check if there's actually a piece there. Note that a y coordinate is a row, so it goes first – look back at the diagram if you're not sure how this works:

```
      unless (defined $chessboard[$starty]->[$startx]) {
         print "There's nothing on that square!\n";
         next;
      }
```

We do the same for the ending square, and then move the piece. We copy the piece to the new square:

```
      $chessboard[$endy]->[$endx] = $chessboard[$starty]->[$startx];
```

And then we delete the old square:

```
      undef $chessboard[$starty]->[$startx];
```

We've now used a matrix, a two-dimensional array. The nice thing about perl's auto vivification is that we didn't need to say explicitly that we were dealing with references. Perl takes care of all that behind the scenes, and we just assigned the relevant values to the right places. However, if we were to look at the contents of the `@chessboard` array, we'd see eight array references.

Trees

We're now going to build on the principle of matrices, by introducing **tree**-like data structures, in which we use hashes as well as arrays. The classic example of one of these structures is an address book. Suppose we want to keep someone's address and phone number in a hash. We could say this:

```
   %paddy = (
      address => "23, Blue Jay Way",
      phone   => "404-6599"
   );
```

That's all very well, and it makes sense. The only problem is, you have to create a separate hash for each person in your address book and put each one in a separate variable. This isn't easy at all at run time, and is very messy to write. So instead, you use references.

What we do is create a main 'address book' hash, referenced as `$addressbook`, with everyone else's hashes as values off that:

```
$addressbook{"Paddy Malone"} = {
   address => "23, Blue Jay Way",
   phone   => "404-6599"
};
```

Note that if you've included the `use strict;` pragma, you'll have to declare this hash explicitly as `my %addressbook;` before using it.

It's now very easy to take new entries from the user and add them to our address book:

```
print "Give me a name:"; chomp $name   =<>;
print "Address:";        chomp $address=<>;
print "Phone number:";   chomp $phone  =<>;
$addressbook{$name} = {
   address => $address,
   phone   => $phone
};
```

To print out a single person, we'd use this:

```
if (exists $addressbook{$who}) {
   print "$who\n";
   print "Address:  ", $addressbook{$who}->{address}, "\n";
   print "Phone no: ", $addressbook{$who}->{phone},   "\n";
}
```

To print every address, we'd use this:

```
for $who (keys %addressbook) {
   print "$who\n";
   print "Address:  ", $addressbook{$who}->{address}, "\n";
   print "Phone no: ", $addressbook{$who}->{phone},   "\n";
}
```

Deleting an address is very simple:

```
delete $addressbook{$who};
```

How about adding another level to our tree. Can we have an array of 'friends' for each person? No problem. We just use an anonymous array:

```
$addressbook{"Paddy Malone"} = {
   address => "23, Blue Jay Way",
   phone   => "404-6599",
   friends => [ "Baba O'Reilly", "Mick Flaherty" ]
};
```

We can get at each person's friends by saying `$addressbook{$who}->{friends}`. That will give us an anonymous array. We can then dereference that to a real array and print it out:

```
for $who (keys %addressbook) {
   print "$who\n";
   print "Address:  ", $addressbook{$who}->{address}, "\n";
   print "Phone no: ", $addressbook{$who}->{phone},   "\n";
   my @friends = @{$addressbook{$who}->{friends}};
   print "Friends:\n";
   for (@friends) {
      print "\t$_\n";
   }
}
```

This would now give us something like:

```
Paddy Malone
Address:  23, Blue Jay Way
Phone no: 404-6599
Friends:
      Baba O'Reilly
      Mick Flaherty
```

What we now have is one hash (address book), containing another hash (peoples' details), in turn containing an array (each person's friends).

We can quite easily **traverse** the tree structure, that is, move from person to person by following links. We do this by visiting a link, then adding all of that person's friends onto a 'to do' array. We must be very careful here not to get stuck in a loop. If one person links to another, and the other links back again, we need to avoid bouncing about between them indefinitely. One simple way to keep track of the links we've already processed is to use a hash. Here's how we can do it:

```
$, = "\t"            # Set output field separator for tabulated display
my @todo = ("Paddy Malone"); # Start point
my %seen;
while (@todo) {
   my $who = shift @todo; # Get person from the end
   $seen{$who}++;         # Mark them as seen.
   my @friends = @{$addressbook{$who}->{friends}};
   print "$who has friends: ", @friends, "\n";
   for (@friends) {
      # Visit unless they're already visited
      push @todo, $_ unless exists $seen{$_};
   }
}
```

The reference `$seen` is used to build up a hash table of everyone whose name has been held in the variable `$who`. The `for` loop at the bottom only adds names to the `@todo` list if they're not defined in that hash, That is, if they've not been displayed already. Given a fairly closed community, we could see something like this:

```
Paddy Malone has friends Baba O'Reilly Mick Flaherty
Baba O'Reilly has friends Bob McDowell Mick Flaherty Andy Donahue
Mick Flaherty has friends Paddy Malone Timothy O'Leary
Bob McDowell has friends Andy Donahue Baba O'Reilly
Andy Donahue has friends Jimmy Callahan Mick Flaherty
Timothy O'Leary has friends Bob McDowell Mick Flaherty Paddy Malone
Jimmy Callahan has friends Andy Donahue Baba O'Reilly Mick Flaherty
```

Linked Lists

The last thing we're going to look at is creating **linked lists**. These actually cover quite a broad range of data structures, but all have one common feature:

One part of each record in the list refers to at least one other record in the list.

Just as any good page on the web will link to at least one other page, each record in a linked list will include a reference to another record in the list, and possibly several. That's all well and good, but what improvement does this give us on the structures we've seen already? We know how to use a value held in one record to reference another – rather handy, but not exactly earth-shattering.

The fact is, while this is how linked lists hang together, it's not quite the full story. The examples we've seen so far have been passing references to and from records in a single root data structure: the addressbook hash reference. We take the name of a friend and use that as a key in the hash to access that friend's details.

Now, what if I have a bunch of friends at work, where there's already a data structure in place containing just this sort of information. Now, I want to include colleagues in my list of friends, but it's not practical to copy all the data from one to the other. What's more, while the work system uses a similar structure to the addressbook one, `$work` (the root reference – equivalent to `$addressbook`) uses ID numbers as **indices in an array**. For example, my friend Dan is registered as employee 4109, so his details are referenced by `$work[4108]` – yes, array indices start at 0. Anyway, it seems I can't have "Dan Maharry" as one of my friends.

Maybe I could just put '4109' in as his name. What the heck, I'll know who it is. No, of course we'd still be trying to access the addressbook hash reference, and "4109" isn't in there.

What if we get the program to check *both* root references for a suitable match? That works fine, until I'm sending out Christmas mail (automatically, of course. It's what perl does best!), and he gets one starting:

```
Dear 4109,
Let me tell you all about this new book I've written….
```

Hmm. Not really ideal. What we really need is to have our 'friends' key reference a hash table (instead of a list), with the key "Dan Maharry" assigned the value of the appropriate reference. So, instead of:

```
      friends => [ "Baba O'Reilly", "Mick Flaherty" ]
```

we put:

```
friends => {   "Baba O'Reilly" => $addressbook("Baba O'Reilly"),
         "Mick Flaherty" => $addressbook("Mick Flaherty"),
         "Dan Maharry"   => $work[4109]
      }
```

The power and versatility (and some would say beauty) of a linked list derives from a very simple fact:

The internal structure of any record in a linked list can be independent of all others.

In the simplest case, all our references were *from* addressbook entries *to* addressbook entries. This belies the fact that each of them could actually refer to **any** data structure at all. As we saw though, the flexibility of Perl references allows us to link up all sorts of different structures.

Summary

We've looked at references, a way to put one type of data structure inside another. References work because they allow us to refer to another piece of data. They tell us where Perl stores it and give us a way to get at it. Because references are always scalars, you can think of them as flat-pack storage for arrays and hashes.

We can create a reference explicitly by putting a backslash in front of a variable's name: `\%hash` or `\@array`, for example. Alternatively, we can create an anonymous reference by using `{}` instead of `()` for a hash and `[]` instead of `()` for an array. Finally, we can create a reference by creating a need for one. If a reference needs to exist for what we're doing, Perl will spring one into existence by autovivification.

We can use a reference by placing it in curly brackets where a variable name should go. `@{$array_r}` can replace `@array` everywhere and we don't even need the brackets if it's clear what we mean. We can then access elements of array or hash references using the arrow notation: `$array_ref->[$element]` for an array and `$hash_ref->{$key}` for a hash.

We've also seen a few complex data structures: matrices, which are arrays of arrays; trees, which may contain hashes or arrays; and linked lists, which contain references to other parts of the data structure, or even other data structures. For more information on these kinds of data structure, consult the Perl 'Data Structures Cookbook' documentation (`perldsc`) or the Perl 'List of Lists' documentation. (`perllol`)

If you're really interested in data structures from a computer science point of view, *Mastering Algorithms in Perl* by Orwant et al. (*O'Reilly – ISBN 1-56592-398-7*) has some chapters on these kinds of structure, primarily, trees and tree traversal. The ultimate guide to data structures is still '*The Art Of Computer Programming, Volume 1*', by Donald Knuth (*Addison Wesley – ISBN 0201896834*) – affectionately known as 'The Bible'.

Exercises

1. Construct an array of arrays to form a multiplication table covering from one times one to six times six but as words. Then ask the user to query it and return the result in words only.

2. Take the chess program and revise it so it checks for the validity of the knight's moves. Remember that the knight cannot move off the board or take one of its own pieces. The knight moves in an L-shape – two squares horizontally or vertically and then one square at ninety degrees to that.