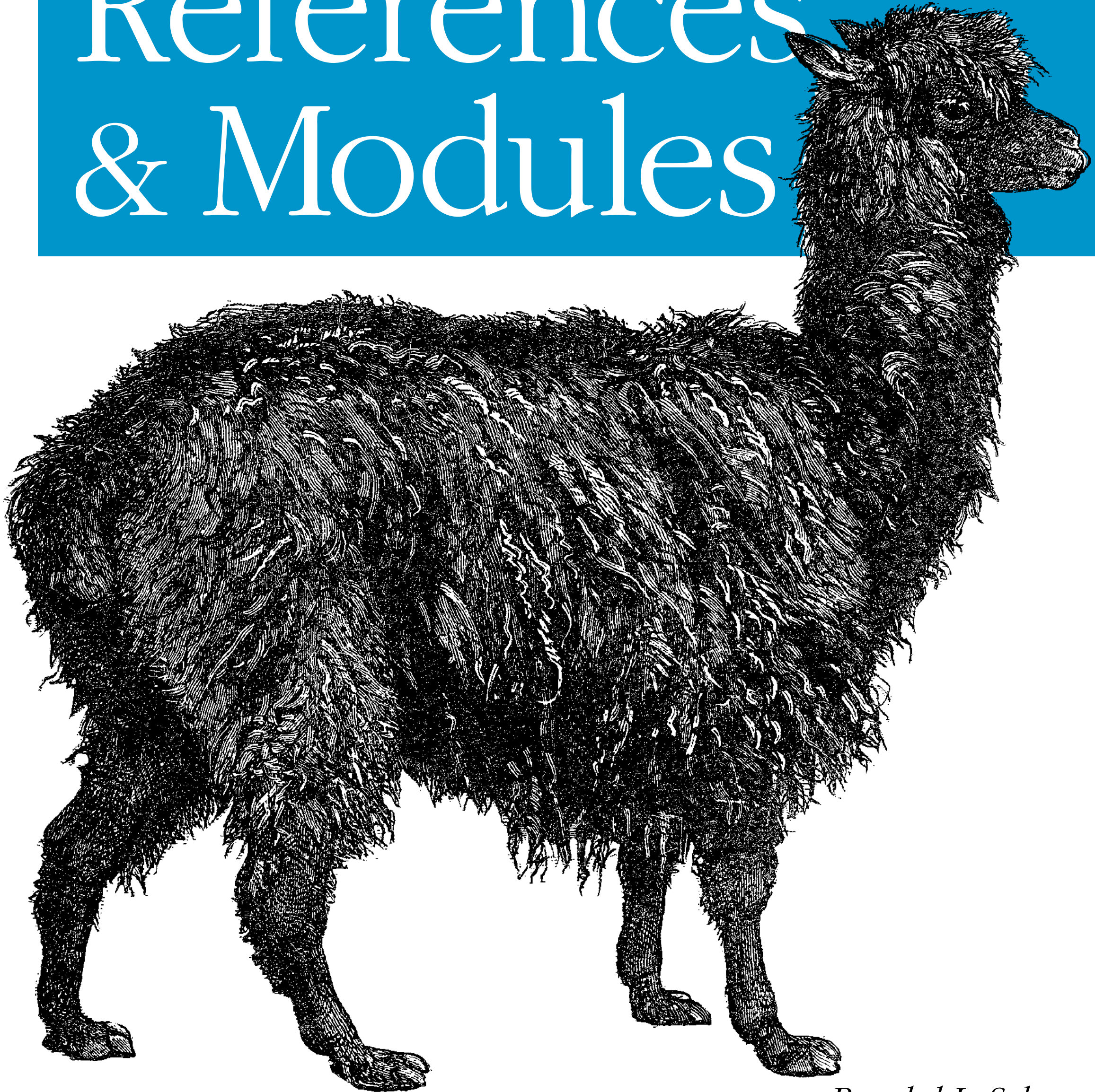
The Long-awaited Sequel to Learning Perl

Learning Perl Objects, References & Modules



Randal L. Schwartz
with Tom Phoenix

O'REILLY®

Learning Perl Objects, References, and Modules

Other Perl resources from O'Reilly

Related titles

- Programming Perl
- Learning Perl
- Perl Cookbook
- CGI Programming with Perl
- Computer Science & Perl Programming
- Embedding Perl in HTML with Mason
- Mastering Regular Expressions
- Perl for System Administration
- Programming Web Services with Perl
- Perl Pocket Reference
- Perl in a Nutshell
- Perl Graphics Programming

Perl Books Resource Center

perl.oreilly.com is a complete catalog of O'Reilly's books on Perl and related technologies, including sample chapters and code examples.

Perl.com is the central web site for the Perl community and providing a starting place for finding out everything about Perl.

Conferences

O'Reilly & Associates bring diverse innovators together to nurture the ideas that spark revolutionary industries. We specialize in documenting the latest tools and systems, translating the innovator's knowledge into useful skills for those in the trenches. Visit *conferences.oreilly.com* for our upcoming events.

Safari Bookshelf (*safari.oreilly.com*) is the premier online reference library for programmers and IT professionals. Conduct searches across more than 1,000 books. Subscribers can zero in on answers to time-critical questions in a matter of seconds. Read the books on your Bookshelf from cover to cover or simply flip to the page you need. Try it today with a free trial.

Learning Perl Objects, References, and Modules

Randal L. Schwartz with Tom Phoenix

O'REILLY®

Beijing • Cambridge • Farnham • Köln • Paris • Sebastopol • Taipei • Tokyo

CHAPTER 3

Introduction to References

A Perl scalar variable holds a single value. An array holds an ordered list of one or more scalars. A hash holds a collection of scalars as values, keyed by other scalars.

Although a scalar can be an arbitrary string, which allows complex data to be encoded into an array or hash, none of the three data types are well-suited to complex data interrelationships. This is a job for the *reference*. Let's look at the importance of references by starting with an example.

Performing the Same Task on Many Arrays

Before the Minnow can leave on an excursion (e.g., a three-hour tour), every passenger and crew member should be checked to ensure they have all the required trip items in their possession. Let's say that for maritime safety, every person on board the Minnow needs to have a life preserver, some sunscreen, a water bottle, and a rain jacket. You can write a bit of code to check for the Skipper's supplies:

```
my @required = qw(preserver sunscreen water_bottle jacket);
my @skipper = qw(blue_shirt hat jacket preserver sunscreen);
for my $item (@required) {
  unless (grep $item eq $_, @skipper) { # not found in list?
    print "skipper is missing $item.\n";
  }
}
```

The `grep` in a scalar context returns the number of times the expression `$item eq $_` returns true, which is 1 if the item is in the list and 0 if not.[*] If the value is 0, it's false, and you print the message.

[*] There are more efficient ways to check list membership for large lists, but for a few items, this is probably the easiest way to do so with just a few lines of code.

Of course, if you want to check on Gilligan and the Professor, you might write the following code:

```
my @gilligan = qw(red_shirt hat lucky_socks water_bottle);
for my $item (@required) {
  unless (grep $item eq $_, @gilligan) { # not found in list?
    print "gilligan is missing $item.\n";
  }
}

my @professor = qw(sunscreen water_bottle slide_rule batteries radio);
for my $item (@required) {
  unless (grep $item eq $_, @professor) { # not found in list?
    print "professor is missing $item.\n";
  }
}
```

You may start to notice a lot of repeated code here and decide that it would be served best in a subroutine:

```
sub check_required_items {
  my $who = shift;
  my @required = qw(preserver sunscreen water_bottle jacket);
  for my $item (@required) {
    unless (grep $item eq $_, @_) { # not found in list?
      print "$who is missing $item.\n";
    }
  }
}

my @gilligan = qw(red_shirt hat lucky_socks water_bottle);
check_required_items("gilligan", @gilligan);
```

The subroutine is given five items in its `@_` array initially: the name `gilligan` and the four items belonging to Gilligan. After the `shift`, `@_` will have only the items. Thus, the `grep` checks each required item against the list.

So far, so good. You can check the Skipper and the Professor with just a bit more code:

```
my @skipper = qw(blue_shirt hat jacket preserver sunscreen);
my @professor = qw(sunscreen water_bottle slide_rule batteries radio);
check_required_items("skipper", @skipper);
check_required_items("professor", @professor);
```

And for the other passengers, you repeat as needed. Although this code meets the initial requirements, you've got two problems to deal with:

- To create `@_`, Perl copies the entire contents of the array to be scanned. This is fine for a few items, but if the array is large, it seems a bit wasteful to copy the data just to pass it into a subroutine.
- Suppose you want to modify the original array to force the provisions list to include the mandatory items. Because you have a copy in the subroutine ("pass

by value”), any changes made to `@_` aren’t reflected automatically in the corresponding provisions array.[*]

To solve either or both of these problems, you need pass by reference rather than pass by value. And that’s just what the doctor (or Professor) ordered.

Taking a Reference to an Array

Among its many other meanings, the backslash (\) character is also the “take a reference to” operator. When you use it in front of an array name, e.g., `\@skipper`, the result is a *reference* to that array. A reference to the array is like a pointer: it points at the array, but is not the array itself.

A reference fits wherever a scalar fits. It can go into an element of an array or a hash, or into a plain scalar variable, like this:

```
my $reference_to_skipper = \@skipper;
```

The reference can be copied:

```
my $second_reference_to_skipper = $reference_to_skipper;
```

or even:

```
my $third_reference_skipper = \@skipper;
```

All three references are completely interchangeable. You can even say they’re identical:

```
if ($reference_to_skipper == $second_reference_to_skipper) {
  print "They are identical references.\n";
}
```

This equality compares the numeric forms of the two references. The numeric form of the reference is the unique memory address of the `@skipper` internal data structure, unchanging during the life of the variable. If you look at the string form instead, with `eq` or `print`, you get a debugging string:

```
ARRAY(0x1a2b3c)
```

which again is unique for this array because it includes the hexadecimal (base 16) representation of the array’s unique memory address. The debugging string also notes that this is an array reference. Of course, if you ever see something like this in your output, it almost certainly means there’s a bug; users of your program have little interest in hex dumps of storage addresses!

[*] Actually, assigning new scalars to elements of `@_` after the `shift` modifies the corresponding variable being passed, but that still wouldn’t let you extend the array with additional mandatory provisions.

Because a reference can be copied, and passing an argument to a subroutine is really just copying, you can use this code to pass a reference to the array into the subroutine:

```
my @skipper = qw(blue_shirt hat jacket preserver sunscreen);
check_required_items("The Skipper", \@skipper);

sub check_required_items {
  my $who = shift;
  my $items = shift;
  my @required = qw(preserver sunscreen water_bottle jacket);
  ...
}
```

Now `$items` in the subroutine will be a reference to the array of `@skipper`. But how do you get from a reference back into the original array? By *dereferencing* the reference.

Dereferencing the Array Reference

If you look at `@skipper`, you'll see that it consists of two parts: the `@` symbol and the name of the array. Similarly, the syntax `$skipper[1]` consists of the name of the array in the middle and some syntax around the outside to get at the second element of the array (index value 1 is the second element because you start counting index values at 0).

Here's the trick: any reference to an array can be placed in curly braces and written in place of the name of an array, ending up with a method to access the original array. That is, wherever you write `skipper` to name the array, you use the reference inside curly braces: `{ $items }`. For example, both of these lines refer to the entire array:

```
@  skipper
@{ $items }
```

whereas both of these refer to the second item of the array:[*]

```
$  skipper [1]
${ $items }[1]
```

By using the reference form, you've decoupled the code and the method of array access from the actual array. Let's see how that changes the rest of this subroutine:

```
sub check_required_items {
  my $who = shift;
  my $items = shift;
  my @required = qw(preserver sunscreen water_bottle jacket);
  for my $item (@required) {
    unless (grep $item eq $_, @{$items}) { # not found in list?
```

[*] Note that whitespace was added in these two displays to make the similar parts line up. This whitespace is legal in a program, even though most programs won't use it.

```
      print "$who is missing $item.\n";
    }
  }
}
```

All you did was replace @_ (the copy of the provisions list) with @{$items}, a dereferencing of the reference to the original provisions array. Now you can call the subroutine a few times as before:

```
my @skipper = qw(blue_shirt hat jacket preserver sunscreen);
check_required_items("The Skipper", \@skipper);
my @professor = qw(sunscreen water_bottle slide_rule batteries radio);
check_required_items("Professor", \@professor);
my @gilligan = qw(red_shirt hat lucky_socks water_bottle);
check_required_items("Gilligan", \@gilligan);
```

In each case, $items points to a different array, so the same code applies to different arrays each time it is invoked. This is one of the most important uses of references: decoupling the code from the data structure on which it operates so the code can be reused more readily.

Passing the array by reference fixes the first of the two problems mentioned earlier. Now, instead of copying the entire provision list into the @_ array, you get a single element of a reference to that provisions array.

Could you have eliminated the two shifts at the beginning of the subroutine? Sure, at the expense of clarity:

```
sub check_required_items {
  my @required = qw(preserver sunscreen water_bottle jacket);
  for my $item (@required) {
    unless (grep $item eq $_, @{$_[1]}) { # not found in list?
      print "$_[0] is missing $item.\n";
    }
  }
}
```

You still have two elements in @_. The first element is the passenger or crew member name and is used in the error message. The second element is a reference to the correct provisions array, used in the grep expression.

Dropping Those Braces

Most of the time, the dereferenced array reference is contained in a simple scalar variable, such as @{$items} or ${$items}[1]. In those cases, the curly braces can be dropped, unambiguously, forming @$items or $$items[1].

However, the braces cannot be dropped if the value within the braces is not a simple scalar variable. For example, for @{$_[1]} from that last subroutine rewrite, you can't remove the braces.

This rule also means that it's easy to see where the "missing" braces need to go. When you see `$$items[1]`, a pretty noisy piece of syntax, you can tell that the curly braces must belong around the simple scalar variable, `$items`. Therefore, `$items` must be a reference to an array.

Thus, an easier-on-the-eyes version of that subroutine might be:

```
sub check_required_items {
  my $who = shift;
  my $items = shift;
  my @required = qw(preserver sunscreen water_bottle jacket);
  for my $item (@required) {
    unless (grep $item eq $_, @$items) { # not found in list?
      print "$who is missing $item.\n";
    }
  }
}
```

The only difference here is that the braces were removed for `@$items`.

Modifying the Array

You've seen how to solve the excessive copying problem with an array reference. Now let's look at modifying the original array.

For every missing provision, push that provision onto an array, forcing the passenger to consider the item:

```
sub check_required_items {
  my $who = shift;
  my $items = shift;
  my @required = qw(preserver sunscreen water_bottle jacket);
  my @missing = ( );

  for my $item (@required) {
    unless (grep $item eq $_, @$items) { # not found in list?
      print "$who is missing $item.\n";
      push @missing, $item;
    }
  }

  if (@missing) {
    print "Adding @missing to @$items for $who.\n";
    push @$items, @missing;
  }
}
```

Note the addition of the `@missing` array. If you find any items missing during the scan, push them into `@missing`. If there's anything there at the end of the scan, add it to the original provision list.

The key is in the last line of that subroutine. You're dereferencing the `$items` array reference, accessing the original array, and adding the elements from `@missing`. Without passing by reference, you'd modify only a local copy of the data, which has no effect on the original array.

Also, `@$items` (and its more generic form `@{$items}`) works within a double-quoted string. Do not include any whitespace between the `@` and the immediately following character, although you can include nearly arbitrary whitespace within the curly braces as if it were normal Perl code.

Nested Data Structures

In this example, the array `@_` contains two elements, one of which is also an array. What if you take a reference to an array that also contains a reference to an array? You end up with a complex data structure, which can be quite useful.

For example, iterate over the data for the Skipper, Gilligan, and the Professor by first building a larger data structure holding the entire list of provision lists:

```
my @skipper = qw(blue_shirt hat jacket preserver sunscreen);
my @skipper_with_name = ("Skipper", \@skipper);
my @professor = qw(sunscreen water_bottle slide_rule batteries radio);
my @professor_with_name = ("Professor", \@professor);
my @gilligan = qw(red_shirt hat lucky_socks water_bottle);
my @gilligan_with_name = ("Gilligan", \@gilligan);
```

At this point, `@skipper_with_name` has two elements, the second of which is an array reference, similar to what was passed to the subroutine. Now group them all:

```
my @all_with_names = (
  \@skipper_with_name,
  \@professor_with_name,
  \@gilligan_with_name,
);
```

Note that you have just three elements, each of which is a reference to an array, each of which has two elements: the name and its corresponding initial provisions. A picture of that is in Figure 3-1.

Therefore, `$all_with_names[2]` will be the array reference for the Gilligan's data. If you dereference it as `@{$all_with_names[2]}`, you get a two-element array, `"Gilligan"` and another array reference.

How would you access that array reference? Using your rules again, it's `${$all_with_names[2]}[1]`. In other words, taking `$all_with_names[2]`, you dereference it in an expression that would be something like `$DUMMY[1]` as an ordinary array, so you'll place `{$all_with_names[2]}` in place of `DUMMY`.

How do you call the existing `check_required_items()` with this data structure? The following code is easy enough.

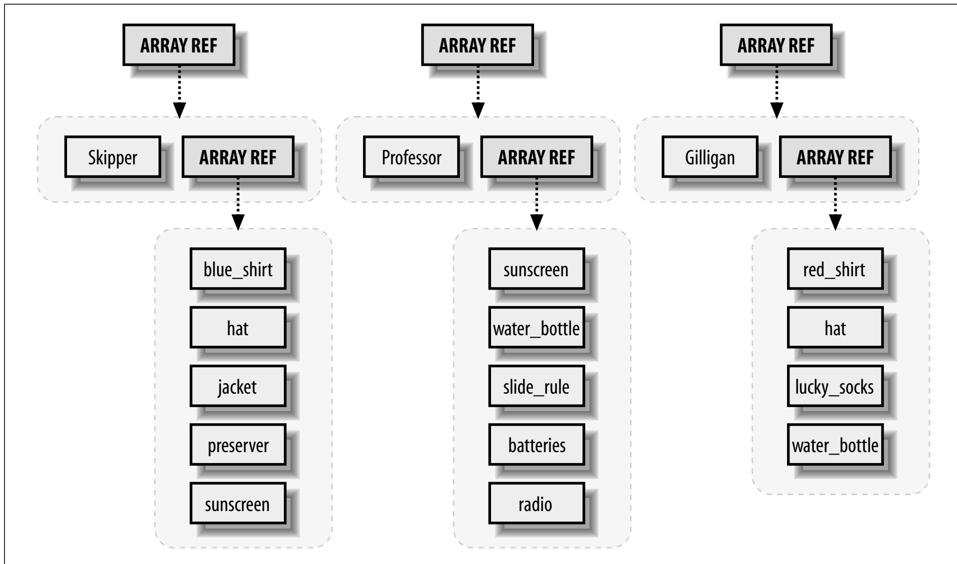


Figure 3-1. The array @all_with_names holds a multilevel data structure containing strings and references to arrays

```
for my $person (@all_with_names) {
  my $who = $$person[0];
  my $provisions_reference = $$person[1];
  check_required_items($who, $provisions_reference);
}
```

This requires no changes to the subroutine. `$person` will be each of `$all_with_names[0]`, `$all_with_names[1]`, and `$all_with_names[2]`, as the loop progresses. When you dereference `$$person[0]`, you get "Skipper," "Professor," and "Gilligan," respectively. `$$person[1]` is the corresponding array reference of provisions for that person.

Of course, you can shortcut this as well, since the entire dereferenced array matches the argument list precisely:

```
for my $person (@all_with_names) {
  check_required_items(@$person);
}
```

or even:

```
check_required_items(@$_) for @all_with_names;
```

As you can see, various levels of optimization can lead to obfuscation. Be sure to consider where your head will be a month from now when you have to reread your own code. If that's not enough, consider the new person who takes over your job after you have left.

Simplifying Nested Element References with Arrows

Look at the curly-brace dereferencing again. As in the earlier example, the array reference for Gilligan's provision list is `${$all_with_names[2]}[1]`. Now, what if you want to know Gilligan's first provision? You need to dereference *this* item one more level, so it's Yet Another Layer of Braces: `${${$all_with_names[2]}[1]}[0]`. That's a really noisy piece of syntax. Can you shorten that? Yes!

Everywhere you write `${DUMMY}[$y]`, you can write `DUMMY->[$y]` instead. In other words, you can dereference an array reference, picking out a particular element of that array by simply following the expression defining the array reference with an arrow and a square-bracketed subscript.

For this example, this means you can pick out the array reference for Gilligan with a simple `$all_with_names[2]->[1]`, and Gilligan's first provision with `$all_with_names[2]->[1]->[0]`. Wow, that's definitely easier on the eyes.

If *that* wasn't already simple enough, there's one more rule: if the arrow ends up between "subscripty kinds of things," like square brackets, you can also drop the arrow. `$all_with_names[2]->[1]->[0]` becomes `$all_with_names[2][1][0]`. Now it's looking even easier on the eye.

The arrow has to be *between* subscripty things. Why wouldn't it be between? Well, imagine a reference to the array `@all_with_names`:

```
my $root = \@all_with_names;
```

Now how do you get to Gilligan's first item?

```
$root -> [2] -> [1] -> [0]
```

More simply, using the "drop arrow" rule, you can use:

```
$root -> [2][1][0]
```

You cannot drop the first arrow, however, because that would mean an array `@root`'s third element, an entirely unrelated data structure. Let's compare this to the full curly-brace form again:

```
${${${$root}[2]}[1]}[0]
```

It looks much better with the arrow. Note, however, that no shortcut gets the entire array from an array reference. If you want all of Gilligan's provisions, you say:

```
@{$root->[2][1]}
```

Reading this from the inside out, you can think of it like this:

1. Take `$root`.
2. Dereference it as an array reference, taking the third element of that array (index number 2).

3. Dereference that as an array reference, taking the second element of that array (index number 1).
4. Dereference that as an array reference, taking the entire array.

The last step doesn't have a shortcut arrow form. Oh well.*

References to Hashes

Just as you can take a reference to an array, you can also take a reference to a hash. Once again, you use the backslash as the "take a reference to" operator:

```
my %gilligan_info = (
  name => 'Gilligan',
  hat => 'White',
  shirt => 'Red',
  position => 'First Mate',
);
my $hash_ref = \%gilligan_info;
```

You can dereference a hash reference to get back to the original data. The strategy is similar to dereferencing an array reference. Write the hash syntax as you would have without references, and then replace the name of the hash with a pair of curly braces surrounding the thing holding the reference. For example, to pick a particular value for a given key, use:

```
my $name = $ gilligan_info { 'name' };
my $name = $ { $hash_ref } { 'name' };
```

In this case, the curly braces have two different meanings. The first pair denotes the expression returning a reference, while the second pair delimits the expression for the hash key.

To perform an operation on the entire hash, you proceed similarly:

```
my @keys = keys % gilligan_info;
my @keys = keys % { $hash_ref };
```

As with array references, you can use shortcuts to replace the complex curly-braced forms under some circumstances. For example, if the only thing inside the curly braces is a simple scalar variable (as shown in these examples so far), you can drop the curly braces:

```
my $name = $$hash_ref{'name'};
my @keys = keys %$hash_ref;
```

Like an array reference, when referring to a specific hash element, you can use an arrow form:

```
my $name = $hash_ref->{'name'};
```

* It's not that it hasn't been discussed repeatedly by the Perl developers; it's just that nobody has come up with a nice backward-compatible syntax with universal appeal.

Because a hash reference fits wherever a scalar fits, you can create an array of hash references:

```
my %gilligan_info = (
  name => 'Gilligan',
  hat => 'White',
  shirt => 'Red',
  position => 'First Mate',
);
my %skipper_info = (
  name => 'Skipper',
  hat => 'Black',
  shirt => 'Blue',
  position => 'Captain',
);
my @crew = (\%gilligan_info, \%skipper_info);
```

Thus, `$crew[0]` is a hash reference to the information about Gilligan. You can get to Gilligan's name via any one of:

```
${ $crew[0] } { 'name' }
my $ref = $crew[0]; $$ref{'name'}
$crew[0]->{'name'}
$crew[0]{'name'}
```

On that last one, you can still drop the arrow between "subscripty kinds of things," even though one is an array bracket and one is a hash brace.

Let's print a crew roster:

```
my %gilligan_info = (
  name => 'Gilligan',
  hat => 'White',
  shirt => 'Red',
  position => 'First Mate',
);
my %skipper_info = (
  name => 'Skipper',
  hat => 'Black',
  shirt => 'Blue',
  position => 'Captain',
);
my @crew = (\%gilligan_info, \%skipper_info);

my $format = "%-15s %-7s %-7s %-15s\n";
printf $format, qw(Name Shirt Hat Position);
for my $crewmember (@crew) {
  printf $format,
    $crewmember->{'name'},
    $crewmember->{'shirt'},
    $crewmember->{'hat'},
    $crewmember->{'position'};
}
```

That last part looks very repetitive. You can shorten it with a hash slice. Again, if the original syntax is:

```
@ gilligan_info { qw(name position) }
```

the hash slice notation from a reference looks like:

```
@ { $hash_ref } { qw(name position) }
```

You can drop the first brace pair because the only thing within is a simple scalar value, yielding:

```
@ $hash_ref { qw(name position) }
```

Thus, you can replace that final loop with:

```
for my $crewmember (@crew) {
  printf $format, @$crewmember{qw(name shirt hat position)};
}
```

There is no shortcut form with an arrow (`->`) for array slices or hash slices, just as there is no shortcut for entire arrays or hashes.

A hash reference prints as a string that looks like `HASH(0x1a2b3c)`, showing the hexadecimal memory address of the hash. That's not very useful to an end user and only barely more usable to the programmer, except as an indication of the lack of appropriate dereferencing.

Exercises

The answers for all exercises can be found in the Appendix.

Exercise 1 [5 min]

How many different things do these expressions refer to?

```
$ginger->[2][1]
${$ginger[2]}[1]
$ginger->[2]->[1]
${$ginger->[2]}[1]
```

Exercise 2 [30 min]

Using the final version of `check_required_items`, write a subroutine `check_items_for_all` that takes a hash reference as its only parameter, pointing at a hash whose keys are the people aboard the Minnow, and whose corresponding values are array references of the things they intend to bring on board.

For example, the hash reference might be constructed like so:

```
my @gilligan = ... gilligan items ...;
my @skipper = ... skipper items ...;
my @professor = ... professor items ...;
```

```
my %all = (
  "Gilligan" => \@gilligan,
  "Skipper" => \@skipper,
  "Professor" => \@professor,
);
check_items_for_all(\%all);
```

The newly constructed subroutine should call `check_required_items` for each person in the hash, updating their provisions list to include the required items.