Chapter 4. Subroutines

You've already seen and used some of the built-in system functions, such as `chomp` , `reverse` , `print` , and so on. But, as other languages do, Perl has the ability to make *subroutines*, which are user-defined functions. These let you recycle one chunk of code many times in one program. The name of a subroutine is another Perl identifier (letters, digits, and underscores, but they can't start with a digit) with a sometimes-optional ampersand (`&`) in front. There's a rule about when you can omit the ampersand and when you cannot; you'll see that rule by the end of the chapter. For now, just use it every time it's not forbidden, which is always a safe rule. We'll tell you every place where it's forbidden, of course.

The subroutine name comes from a separate namespace, so Perl won't be confused if you have a subroutine called `&fred` and a scalar called `$fred` in the same program—although there's no reason to do that under normal circumstances.

Defining a Subroutine

To define your own subroutine, use the keyword `sub` , the name of the subroutine (without the ampersand), then the block of code in curly braces, which makes up the *body* of the subroutine. Something like this:

```
sub marine {
  $n += 1;  # Global variable $n
  print "Hello, sailor number $n!\n";
}
```

You may put your subroutine definitions anywhere in your program text, but programmers who come from a background of languages like C or Pascal like to put them at the start of the file. Others may prefer to put them at the end of the file so that the main part of the program appears at the beginning. It's up to you. In any case, you don't normally need any kind of forward declaration. If you have two subroutine definitions with the same name, the later one overwrites the earlier one. Although, if you have warnings enabled, Perl will tell you when you do that. It's generally considered bad form, or the sign of a confused maintenance programmer.

NOTE

We don't talk about subroutines of the same name in different packages until *Intermediate Perl*.

As you may have noticed in the previous example, you may use any global variables within the subroutine body. In fact, all of the variables you've seen so far are global; that is, they are accessible from every part of your program. This horrifies linguistic purists, but the Perl development team formed an angry mob with torches and ran them out of town years ago. You'll see how to make private variables in the section "Private Variables in Subroutines".

Invoking a Subroutine

You invoke a subroutine from within an expression by using the subroutine name (with the ampersand):

```
&marine;  # says Hello, sailor number 1!
&marine;  # says Hello, sailor number 2!
&marine;  # says Hello, sailor number 3!
&marine;  # says Hello, sailor number 4!
```

Most often, you refer to the invocation as simply *calling* the subroutine. You'll also see other ways that you may call the subroutine as you read through this chapter.

Return Values

You always invoke a subroutine as part of an expression, even if you don't use the result of the expression. When you invoked `&marine` earlier, you were calculating the value of the expression containing the invocation but then throwing away the result.

Many times, you call a subroutine and actually do something with the result. This means that you do something with the *return value* of the subroutine. All Perl subroutines have a return value—there's no distinction between those that return values and those that don't. Not all Perl subroutines have a *useful* return value, however.

Since you can call Perl subroutines in a way that needs a return value, it'd be a bit wasteful to have to declare special syntax to "return" a particular value for the majority of the cases. So Larry made it simple. As Perl chugs along in a subroutine, it calculates values as part of its series of actions. Whatever calculation is *last* performed in a subroutine is *automatically* also the return value.

For example, this subroutine has an addition as the last expression:

```
sub sum_of_fred_and_barney {
  print "Hey, you called the sum_of_fred_and_barney subroutine!\n";
  $fred + $barney;  # That's the return value
}
```

The last evaluated expression in the body of this subroutine is the sum of `$fred` and `$barney`, so the sum of `$fred` and `$barney` is the return value. Here's that in action:

```
$fred   = 3;
$barney = 4;
$wilma  = &sum_of_fred_and_barney;      # $wilma gets 7
print "\$wilma is $wilma.\n";

$betty  = 3 * &sum_of_fred_and_barney;  # $betty gets 21
print "\$betty is $betty.\n";
```

That code produces this output:

```
Hey, you called the sum_of_fred_and_barney subroutine!
$wilma is 7.
Hey, you called the sum_of_fred_and_barney subroutine!
$betty is 21.
```

That `print` statement is just a debugging aid, so you can see that you called the subroutine. You normally take out those sorts of statements when you're ready to deploy your program. But suppose you added another `print` to the end of the subroutine, like this:

```
sub sum_of_fred_and_barney {
  print "Hey, you called the sum_of_fred_and_barney subroutine!\n";
  $fred + $barney;  # That's not really the return value!
  print "Hey, I'm returning a value now!\n";      # Oops!
}
```

The last expression evaluated is not the addition anymore; it's now the `print` statement, whose return value is normally `1`, meaning "printing was successful," but that's not the return value you actually wanted. So be careful when adding additional code to a subroutine, since the last expression *evaluated* will be the return value.

NOTE

The return value of `print` is true for a successful operation and false for a failure. You'll see how to determine the kind of failure in Chapter 5.

So, what happened to the sum of `$fred` and `$barney` in that second (faulty) subroutine? You didn't put it anywhere, so Perl discarded it. If you had requested warnings, Perl (noticing that there's nothing useful about adding two variables and discarding the result) would likely warn you about something like "a useless use of addition in a void context." The term *void context* is just a fancy way of saying that you aren't using the answer, whether that means storing it in a variable or using it any other way.

“The last evaluated expression” really means the last expression that Perl evaluates rather than the last statement in the subroutine. For example, this subroutine returns the larger value of `$fred` or `$barney`:

```
sub larger_of_fred_or_barney {
  if ($fred > $barney) {
    $fred;
  } else {
    $barney;
  }
}
```

The last evaluated expression is either `$fred` or `$barney`, so the value of one of those variables becomes the return value. You don’t know if the return value will be `$fred` or `$barney` until you see what those variables hold at runtime.

These are all rather trivial examples. It gets better when you can pass values that are different for each invocation into a subroutine instead of relying on global variables. In fact, that’s coming right up.

Arguments

That subroutine called `larger_of_fred_or_barney` would be much more useful if it didn’t force you to use the global variables `$fred` and `$barney`. If you wanted to get the larger value from `$wilma` and `$betty`, you currently have to copy those into `$fred` and `$barney` before you can use `larger_of_fred_or_barney`. And if you had something useful in those variables, you’d have to first copy those to other variables—say, `$save_fred` and `$save_barney`. And then, when you’re done with the subroutine, you’d have to copy those back to `$fred` and `$barney` again.

Luckily, Perl has subroutine *arguments*. To pass an argument list to the subroutine, simply place the list expression, in parentheses, after the subroutine invocation, like this:

```
$n = &max(10, 15);  # This sub call has two parameters
```

Perl *passes* the list to the subroutine; that is, Perl makes the list available for the subroutine to use however it needs to. Of course, you have to store this list somewhere, so Perl automatically stores the parameter list (another name for the argument list) in the special array variable named `@_` for the duration of the subroutine. You can access this array to determine both the number of arguments and the value of those arguments.

This means that the first subroutine parameter is in `$_[0]`, the second one is stored in `$_[1]`, and so on. But—and here's an important note—these variables have nothing whatsoever to do with the `$_` variable, any more than `$dino[3]` (an element of the `@dino` array) has to do with `$dino` (a completely distinct scalar variable). It's just that the parameter list must be in some array variable for your subroutine to use it, and Perl uses the array `@_` for this purpose.

Now, you *could* write the subroutine `&max` to look a little like the subroutine `&larger_of_fred_or_barney`, but instead of using `$fred`, you *could* use the first subroutine parameter (`$_[0]`), and instead of using `$barney`, you *could* use the second subroutine parameter (`$_[1]`). And so you *could* end up with something like this:

```
sub max {
  # Compare this to &larger_of_fred_or_barney
  if ($_[0] > $_[1]) {
    $_[0];
  } else {
    $_[1];
  }
}
```

Well, as we said, you *could* do that. But it's pretty ugly with all of those subscripts, and it's hard to read, write, check, and debug too. You'll see a better way in a moment.

There's another problem with this subroutine. The name `&max` is nice and short, but it doesn't remind us that this subroutine works properly only if called with exactly two parameters:

```
  $n = &max(10, 15, 27);  # Oops!
```

`max` ignores the extra parameters since it never looks at `$_[2]`. Perl doesn't care whether there's something in there or not. Perl doesn't care about insufficient parameters either—you simply get `undef` if you look beyond the end of the `@_` array, as with any other array. You'll see how to make a better `&max`, which works with any number of parameters, later in this chapter.

The `@_` variable is private to the subroutine; if there's a global value in `@_`, Perl saves it before it invokes the next subroutine and restores its previous value upon return from that subroutine. This also means that a subroutine can pass arguments to another subroutine without fear of losing its own `@_` variable—the nested subroutine invocation gets its own `@_` in the same way. Even if the subroutine calls itself recursively, each invocation gets a new `@_`, so `@_` is always the parameter list for the *current* subroutine invocation.

NOTE

You might recognize that this is the same mechanism as used with the control variable of the `foreach` loop, as seen in Chapter 3. In either case, the variable's value is saved and automatically restored by Perl.

Private Variables in Subroutines

But if Perl can give you a new `@_` for every invocation, can't it give you variables for your own use as well? Of course it can.

By default, all variables in Perl are global variables; that is, they are accessible from every part of the program. But you can create private variables called *lexical variables* at any time with the `my` operator:

```
  sub max {
    my($m, $n);       # new, private variables for this block
    ($m, $n) = @_;    # give names to the parameters
    if ($m > $n) { $m } else { $n }
  }
```

These variables are private (or *scoped*) to the enclosing block; any other `$m` or `$n` is totally unaffected by these two. And that goes the other way too—no other code can access or modify these private variables, by accident or design. So you could drop this subroutine into any Perl program in the world and know that you wouldn't mess up that program's `$m` and `$n` (if any). Of course, if that program already had a subroutine called `&max`, you'd mess *that* up.

It's also worth pointing out that, inside those `if` blocks, you don't need a semicolon after the return value expression. The semicolon is really a statement separator, not a statement terminator. Although Perl allows you to omit the last semicolon in a block, in practice you omit it only when the code is so simple that you can write the block in a single line.

You can make the subroutine in the previous example even simpler. Did you notice that the list `($m, $n)` shows up twice? You can apply the `my` operator to a list of variables enclosed in parentheses that you use in a list assignment, so it's customary to combine those first two statements in the subroutine:

```
  my($m, $n) = @_;  # Name the subroutine parameters
```

That one statement creates the private variables and sets their values, so the first parameter now has the easier-to-use name `$m` and the second has `$n`. Nearly every subroutine starts with a line much like that one, naming its parameters. When you see that line, you'll know that the subroutine expects two scalar parameters, which you'll call `$m` and `$n` inside the subroutine.

Variable-Length Parameter Lists

In real-world Perl code, subroutines often have parameter lists of arbitrary length. That's because of Perl's "no unnecessary limits" philosophy that you've already seen. Of course, this is unlike many traditional programming languages, which require every subroutine to be strictly typed; that is, to permit only a certain predefined number of parameters of predefined types. It's nice that Perl is so flexible, but (as you saw with the `&max` routine earlier) that may cause problems when you call a subroutine with a different number of arguments than it expects.

Of course, you can easily check that the subroutine has the right number of arguments by examining the `@_` array. For example, you could have written `&max` to check its argument list like this:

```
sub max {
  if (@_ != 2) {
    print "WARNING! &max should get exactly two arguments!\n";
  }
  # continue as before...
}
```

That `if` test uses the "name" of the array in a scalar context to find out the number of array elements, as you saw in Chapter 3.

But in real-world Perl programming, virtually no one really uses this sort of check; it's better to make your subroutines adapt to the parameters.

A Better &max Routine

Rewrite `&max` to allow for any number of arguments, so you can call it like this:

```
$maximum = &max(3, 5, 10, 4, 6);

sub max {
  my($max_so_far) = shift @_;  # the first one is the largest yet seen
  foreach (@_) {               # look at the remaining arguments
    if ($_ > $max_so_far) {  # could this one be bigger yet?
      $max_so_far = $_;
    }
  }
  $max_so_far;
}
```

This code uses what has often been called the *high-water mark* algorithm; after a flood, when the waters have surged and receded for the last time, the high-water mark shows where the highest water was seen. In this routine, `$max_so_far` keeps track of our high-water mark, the largest number yet seen, in the `$max_so_far` variable.

The first line sets `$max_so_far` to `3` (the first parameter in the example code) by shifting that parameter from the parameter array, `@_`. So `@_` now holds `(5, 10, 4, 6)`, since you removed the `3`. And the largest number yet seen is the *only* one yet seen: `3`, the first parameter.

Next, the `foreach` loop steps through the remaining values in the parameter list, from `@_`. The control variable of the loop is, by default, `$_`. (But remember, there's no automatic connection between `@_` and `$_`; it's just a coincidence that they have such similar names.) The first time through the loop, `$_` is `5`. The `if` test sees that it is larger than `$max_so_far`, so it sets `$max_so_far` to `5`—the new high-water mark.

The next time through the loop, `$_` is `10`. That's a new record high, so you store it in `$max_so_far` as well.

The next time, `$_` is `4`. The `if` test fails, since that's not larger than `$max_so_far`, which is `10`, so you skip the body of the `if`.

Finally, `$_` is `6`, and you skip the body of the `if` again. And that was the last time through the loop, so the loop is done.

Now, `$max_so_far` becomes the return value. It's the largest number you've seen, and you've seen them all, so it must be the largest from the list: `10`.

Empty Parameter Lists

That improved `&max` algorithm works fine now, even if there are more than two parameters. But what happens if there are none?

At first, it may seem too esoteric to worry about. After all, why would someone call `&max` without giving it any parameters? But maybe someone wrote a line like this one:

```
$maximum = &max(@numbers);
```

And the array `@numbers` might sometimes be an empty list; perhaps it was read in from a file that turned out to be empty, for example. So you need to know: what does `&max` do in that case?

The first line of the subroutine sets `$max_so_far` by using `shift` on `@_`, the (now empty) parameter array. That's harmless; the array is left empty, and `shift` returns `undef` to `$max_so_far`.

Now the `foreach` loop wants to iterate over `@_`, but since that's empty, you execute the loop body zero times.

So in short order, Perl returns the value of `$max_so_far`—`undef`—as the return value of the subroutine. In some sense, that's the right answer because there is no largest (non)value in an empty list.

Of course, whoever called this subroutine should be aware that the return value may be `undef`—or they could simply ensure that the parameter list is never empty.

Notes on Lexical (my) Variables

Those lexical variables can actually be used in any block, not merely in a subroutine's block. For example, they can be used in the block of an `if`, `while`, or `foreach`:

```
foreach (1..10) {
  my($square) = $_ * $_;  # private variable in this loop
```

```
    print "$_ squared is $square.\n";
  }
```

The variable `$square` is private to the enclosing block; in this case, that's the block of the `foreach` loop. If there's no enclosing block, the variable is private to the entire source file.

For now, your programs aren't going to use more than one source file, so this isn't an issue. But the important concept is that the `scope` of a lexical variable's name is limited to the smallest enclosing block or file. The *only* code that can say `$square` and mean that variable is the code inside that textual scope.

This is a big win for maintainability—if you find a wrong value in `$square`, you should also find the culprit within a limited amount of source code. As experienced programmers have learned (often the hard way), limiting the scope of a variable to a page of code, or even to a few lines of code, really speeds along the development and testing cycle.

A file is a scope too, so a lexical variable in one file isn't visible in another. However, we put off covering reusable libraries and modules until *Intermediate Perl*.

Note also that the `my` operator doesn't change the context of an assignment:

```
  my($num) = @_;  # list context, same as ($num) = @_;
  my $num  = @_;  # scalar context, same as $num = @_;
```

In the first one, `$num` gets the first parameter, as a list-context assignment; in the second, it gets the number of parameters, in a scalar context. Either line of code *could* be what the programmer wanted; you can't tell from that one line alone, and so Perl can't warn you if you use the wrong one. (Of course, you wouldn't have *both* of those lines in the same subroutine, since you can't have two lexical variables with the same name declared in the same scope; this is just an example.) So, when reading code like this, you can always tell the context of the assignment by seeing what the context would be without the word `my`.

Remember that without the parentheses, `my` only declares a *single* lexical variable:

```
my $fred, $barney;      # WRONG! Fails to declare $barney
my($fred, $barney);     # declares both
```

Of course, you can use `my` to create new, private arrays as well:

```
my @phone_number;
```

Any new variable will start out empty—`undef` for scalars, or the empty list for arrays.

In regular Perl programming, you'll probably use `my` to introduce any new variable in a scope. In Chapter 3, you saw that you could define your own control variable with the `foreach` structure. You can make that a lexical variable too:

```
foreach my $rock (qw/ bedrock slate lava /) {
  print "One rock is $rock.\n";  # Prints names of three rocks
}
```

This is important in the next section, where you start using a feature that makes you declare all your variables.

The use strict Pragma

Perl tends to be a pretty permissive language. But maybe you want Perl to impose a little discipline; that can be arranged with the `use strict` pragma.

A *pragma* is a hint to a compiler, telling it something about the code. In this case, the `use strict` pragma tells Perl's internal compiler that it should enforce some good programming rules for the rest of this block or source file.

Why would this be important? Well, imagine that you're composing your program and you type a line like this one:

```
$bamm_bamm = 3;  # Perl creates that variable automatically
```

Now, you keep typing for a while. After that line has scrolled off the top of the screen, you type this line to increment the variable:

```
$bammbamm += 1;  # Oops!
```

Since Perl sees a new variable name (the underscore *is* significant in a variable name), it creates a new variable and increments that one. If you're lucky and smart, you've turned on warnings, and Perl can tell you that you used one or both of those global variable names only a single time in your program. But if you're merely smart, you used each name more than once, and Perl won't be able to warn you.

To tell Perl that you're ready to be more restrictive, put the `use strict` pragma at the top of your program (or in any block or file where you want to enforce these rules):

```
use strict;  # Enforce some good programming rules
```

Starting with Perl 5.12, you implicitly use this pragma when you declare a minimum Perl version:

```
use v5.12; # loads strict for you
```

Now, among other restrictions, Perl will insist that you declare every new variable, usually done with `my`:

```
my $bamm_bamm = 3;  # New lexical variable
```

Now if you try to spell it the other way, Perl recognizes the problems and complains that you haven't declared any variable called `$bammbamm`, so your mistake is automatically caught at compile time:

```
$bammbamm += 1;  # No such variable: Compile time fatal error
```

Of course, this applies only to new variables; you don't need to declare Perl's built-in variables, such as `$_` and `@_`. If you add `use strict` to an already written program, you'll generally get a flood of warning messages, so it's better to use it from the start, when it's needed.

NOTE

`use strict` doesn't check variables named `$a` and `$b` because `sort` uses those global variables. They aren't very good variable names anyway.

Most people recommend that programs that are longer than a screenful of text generally need `use strict`. And we agree.

From here on, we'll write most (but not all) of our examples as if `use strict` is in effect even where we don't show it. That is, we'll generally declare variables with `my` where it's appropriate. Although we don't always do so here, we encourage you to include `use strict` in your programs as often as possible. You'll thank us in the long run.

The return Operator

What if you want to stop your subroutine right away? The `return` operator immediately returns a value from a subroutine:

```
my @names = qw/ fred barney betty dino wilma pebbles bamm-bamm /;
my $result = &which_element_is("dino", @names);

sub which_element_is {
  my($what, @array) = @_;
  foreach (0..$#array) {  # indices of @array's elements
```

```
      if ($what eq $array[$_]) {
        return $_;          # return early once found
      }
    }
    -1;                     # element not found (return is optional here)
  }
```

You're asking this subroutine to find the index of `dino` in the array `@names`. First, the `my` declaration names the parameters: there's `$what`, which is what you're searching for, and `@array`, an array of values to search within. That's a copy of the array `@names`, in this case. The `foreach` loop steps through the indices of `@array` (the first index is `0`, and the last one is `$#array`, as you saw in Chapter 3).

Each time through the `foreach` loop, you check to see whether the string in `$what` is equal to the element from `@array` at the current index. If it's equal, you return that index at once. This is the most common use of the keyword `return` in Perl—to return a value immediately, without executing the rest of the subroutine.

But what if you never found that element? In that case, the author of this subroutine has chosen to return `-1` as a "value not found" code. It would be more Perlish, perhaps, to return `undef` in that case, but this programmer used `-1`. Saying `return -1` on that last line would be correct, but the word `return` isn't really needed.

Some programmers like to use `return` every time there's a return value, as a means of documenting that it *is* a return value. For example, you might use `return` when the return value is not the last line of the subroutine, such as in the subroutine `&larger_of_fred_or_barney`, earlier in this chapter. You don't really need it, but it doesn't hurt anything either. However, many Perl programmers believe it's just an extra seven characters of typing.

Omitting the Ampersand

As promised, now we'll tell you the rule for when you can omit the ampersand on a subroutine call. If the compiler sees the subroutine definition before invocation, or if Perl can tell from the syntax that it's a subroutine call, the subroutine can be called without an ampersand, just like a built-in function. (But there's a catch hidden in that rule, as you'll see in a moment.)

This means that if Perl can see that it's a subroutine call without the ampersand, from the syntax alone, that's generally fine. That is, if you've got the parameter list in parentheses, it's got to be a function call:

```
my @cards = shuffle(@deck_of_cards);  # No & necessary on &shuffle
```

In this case, the function is the subroutine `&shuffle`. But it may be a built-in function, as you'll see in a moment.

Or, if Perl's internal compiler has already seen the subroutine definition, that's generally OK too. In that case, you can even omit the parentheses around the argument list:

```
sub division {
  $_[0] / $_[1];                   # Divide first param by second
}

my $quotient = division 355, 113;  # Uses &division
```

This works because of the rule that you may always omit parentheses when they don't change the meaning of the code. You can't omit those parentheses if you use the `&`, though.

However, don't put that subroutine declaration after the invocation; if you do, the compiler won't know what the attempted invocation of `division` is all about. The compiler has to see the definition before the invocation in order to use the subroutine call as if it were a built-in. Otherwise, the compiler doesn't know what to do with that expression because it doesn't know what `division` is yet.

NOTE

As with many things in this book, our use of the `&` is expedient for teaching you the Perl language rather than teaching you to be an experienced Perl programmer. Some people disagree. We say more about this in the "Why we teach the subroutine ampersand" blog post.

That's not the catch, though. The catch is this: if the subroutine has the same name as a Perl built-in, you *must* use the ampersand to call your version. With an ampersand, you're sure to call the subroutine; without it, you can get the subroutine *only* if there's no built-in with the same name:

```
sub chomp {
  print "Munch, munch!\n";
}

&chomp;  # That ampersand is not optional!
```

Without the ampersand, you'd be calling the built-in `chomp`, even though you've defined the subroutine `&chomp`. So the real rule to use is this one: until you know the names of all of Perl's built-in functions, use the ampersand on function calls. That means you will use it for your first one hundred or so programs. But when you see someone else has omitted the ampersand in their own code, it's not a mistake; perhaps they simply know that Perl has no built-in with that name.

Nonscalar Return Values

A scalar isn't the only kind of return value a subroutine may have. If you call your subroutine in a list context, it can return a list of values.

NOTE

You can detect whether a subroutine is being evaluated in a scalar or list context using the `wantarray` function, which lets you easily write subroutines with specific list or scalar context values.

Suppose you want to get a range of numbers (as from the range operator, `..`), except that you want to be able to count down as well as up. The range operator only counts upward, but that's easily fixed:

```
sub list_from_fred_to_barney {
  if ($fred < $barney) {
    # Count upward from $fred to $barney
    $fred..$barney;
  } else {
    # Count downward from $fred to $barney
    reverse $barney..$fred;
  }
}

$fred = 11;
$barney = 6;
@c = &list_from_fred_to_barney; # @c gets (11, 10, 9, 8, 7, 6)
```

In this case, the range operator gives you the list from `6` to `11`, then `reverse` reverses the list so that it goes from `$fred` (`11`) to `$barney` (`6`), just as we wanted.

The least you can return is nothing at all. A `return` with no arguments will return `undef` in a scalar context or an empty list in a list context. This can be useful for an error return from a subroutine, signaling to the caller that a more meaningful return value is unavailable.

Persistent, Private Variables

With `my`, you were able to make variables private to a subroutine, although each time you called the subroutine you had to define them again.

With `state`, you can still have private variables scoped to the subroutine, but Perl will keep their values between calls.

Going back to the first example in this chapter, you had a subroutine named `marine` that incremented a variable:

```
sub marine {
  $n += 1;  # Global variable $n
  print "Hello, sailor number $n!\n";
}
```

Now that you know about `strict`, you add that to your program and realize that your use of the global variable `$n` is now a compilation error. You can't make `$n` a lexical variable with `my` because it wouldn't retain its value between calls.

Declaring our variable with `state` tells Perl to retain the variable's value between calls to the subroutine and to make the variable private to the subroutine. This feature showed up in Perl 5.10:

```
use v5.10;

sub marine {
  state $n = 0;  # private, persistent variable $n
  $n += 1;
  print "Hello, sailor number $n!\n";
}
```

Now you can get the same output while being `strict`-clean and not using a global variable. The first time you call the subroutine, Perl declares and initializes `$n`. Perl ignores the statement on all subsequent calls. Between calls, Perl retains the value of `$n` for the next call to the subroutine.

You can make any variable type a `state` variable; it's not just for scalars. Here's a subroutine that remembers its arguments and provides a running sum by using a `state` array:

```
use v5.10;

running_sum( 5, 6 );
running_sum( 1..3 );
running_sum( 4 );

sub running_sum {
  state $sum = 0;
  state @numbers;

  foreach my $number ( @_ ) {
    push @numbers, $number;
    $sum += $number;
  }

  say "The sum of (@numbers) is $sum";
}
```

This outputs a new sum each time you call it, adding the new arguments to all of the previous ones:

```
The sum of (5 6) is 11
The sum of (5 6 1 2 3) is 17
The sum of (5 6 1 2 3 4) is 21
```

There's a slight restriction on arrays and hashes as `state` variables, though. You can't initialize them in list contexts as of Perl 5.10:

```
state @array = qw(a b c); # Error!
```

This gives you an error that hints that you might be able to do it in a future version of Perl, but as of Perl 5.24, you still can't:

```
Initialization of state variables in list context currently forbidden ...
```

This restriction is lifted in Perl 5.28, which allows you to initialize arrays and hashes in `state`. For example, a Fibonacci number generator needs the first two numbers to get started, so you seed the `@numbers` array with them:

```
use v5.28;

say next_fibonacci(); # 1
say next_fibonacci(); # 2
say next_fibonacci(); # 3
say next_fibonacci(); # 5

sub next_fibonacci {
  state @numbers = ( 0, 1 );
  push @numbers, $numbers[-2] + $numbers[-1];
  return $numbers[-1];
}
```

Prior to v5.28, you could have used an array reference instead since all references are scalars. However, we don’t tell you about those until *Intermediate Perl*.

Subroutine Signatures

Perl v5.20 added a long-awaited and exciting feature called *subroutine signatures*. So far it’s experimental (see Appendix D), but we’re hoping it’s stable soon. We think it deserves a section in this chapter, even if to merely tease you with it.

Subroutine signatures are different from prototypes, a much different feature that many people have tried to use for the same reason. If you don’t know what a prototype is, that’s good. You probably don’t need to know about them, at least not in this book.

So far, you have seen subroutines that take a list of arguments in `@_` and then assign them to variables. You saw that earlier in the `max` subroutine:

```
sub max {
  my($m, $n);
  ($m, $n) = @_;
  if ($m > $n) { $m } else { $n }
}
```

First, you have to enable this experimental feature (see Appendix D):

```
use v5.20;
use experimental qw(signatures);
```

After that, we can move the variable declarations outside the braces right after the subroutine names:

```
sub max ( $m, $n ) {
  if ($m > $n) { $m } else { $n }
}
```

That’s an exceedingly pleasant syntax. The variables are still private to the subroutine, but you type much less to declare and assign to them. Perl handles that for you. Otherwise, the subroutine is the same.

Well, it’s almost the same. Previously, you could pass any number of arguments to `&max` even if you only used the first two of them. That doesn’t work anymore:

```
&max( 137, 48, 7 );
```

You get an error if you try this:

```
Too many arguments for subroutine
```

The signature feature is also checking the number of arguments for you! But you want to take the maximum of a list of numbers where you don’t know the length of the list. You can modify the subroutine in the same way you modified it earlier. You can use an array in the signature:

```
sub max ( $max_so_far, @rest ) {
  foreach (@rest) {
    if ($_ > $max_so_far) {
      $max_so_far = $_;
    }
  }
  $max_so_far;
}
```

You don’t have to define an array to slurp up the rest of the arguments, though. If you use a plain `@`, Perl knows the subroutine can take a variable number of arguments. The argument list still shows up in `@_`:

```
  sub max ( $max_so_far, @ ) {
    foreach (@_) {
      if ($_ > $max_so_far) {
        $max_so_far = $_;
      }
    }
    $max_so_far;
  }
```

That handles the case of too many arguments, but what about too few arguments? Signatures can also specify defaults:

```
  sub list_from_fred_to_barney ( $fred = 0, $barney = 7 ) {
    if ($fred < $barney) { $fred..$barney }
    else                 { reverse $barney..$fred }
  }


  my @defaults    = list_from_fred_to_barney();
  my @default_end = list_from_fred_to_barney( 17 );

  say "defaults: @defaults";
  say "default_end: @default_end";
```

When you run this, you can see the default values at work:

```
  defaults: 0 1 2 3 4 5 6 7
  default_end: 17 16 15 14 13 12 11 10 9 8 7
```

Sometimes you want optional arguments that don't have defaults. You can use the `$=` placeholder to denote an optional argument:

```
sub one_or_two_args ( $first, $= ) { ... }
```

There's a Perl special variable for formats, `$=`, but that's not what's going on here.

And sometimes you want exactly zero arguments. You could create a constant value like this:

```
sub PI () { 3.1415926 }
```

You can read more about signatures in perlsub. We also write about them in the "Use v5.20 subroutine signatures" blog post.

Prototypes

Prototypes are an older Perl feature that allows you to tell the parser how to interpret your source; they are a primitive form of signatures that never evolved. It's not a feature that we recommend, but it does clash with signatures, so we want to mention them merely so you know they exist.

Suppose you want a subroutine that takes exactly two arguments. You could note that in the prototype. Since those are instructions to the parser, you need the prototype to show up before any call to the subroutine:

```
sub sum ($$) { $_[0] + $_[1] }

print sum( 1, 3, 7 );
```

This code gives you a compilation error because `sum` has one more argument than it should:

```
Too many arguments for main::sum
```

Prototypes and signatures both use parentheses after the name in a subroutine definition, and each has its own syntax. That's a problem if you want to use both. To get around this, v5.20 also adds the `:prototype` attribute, another feature we aren't going to explain other than to note how you preserve prototypes while also using signatures. Put `:prototype` before the parentheses:

```
sub sum :prototype($$) { $_[0] + $_[1] }

print sum( 1, 3, 7 );
```

We advise that you avoid prototypes altogether unless you understand what you are doing. Even then, you may want to think twice about it. The full details are in perlsub. If you have no idea what we're talking about, that's fine, because you won't see this again in this book!

Exercises

See "Answers to Chapter 4 Exercises" for answers to these exercises:

1. [12] Write a subroutine, named `total`, that returns the total of a list of numbers. Hint: the subroutine should *not* perform any I/O; it should simply process its parameters and return a value to its caller. Try it out in this sample program, which merely exercises the subroutine to see that it works. The first group of numbers should add up to 25.

   ```
   my @fred = qw{ 1 3 5 7 9 };
   my $fred_total = total(@fred);
   print "The total of \@fred is $fred_total.\n";
   print "Enter some numbers on separate lines: ";
   my $user_total = total(<STDIN>);
   print "The total of those numbers is $user_total.\n";
   ```

Note that using `<STDIN>` in list context like that will wait for you to end input in whatever way is appropriate for your system.

2. [5] Using the subroutine from the previous problem, make a program to calculate the sum of the numbers from 1 to 1,000.
3. [18] Extra credit exercise: write a subroutine, called `&above_average`, that takes a list of numbers and returns the ones above the average (mean). (Hint: make another subroutine that calculates the average by dividing the total by the number of items.) Try your subroutine in this test program:

```
my @fred = above_average(1..10);
print "\@fred is @fred\n";
print "(Should be 6 7 8 9 10)\n";
my @barney = above_average(100, 1..10);
print "\@barney is @barney\n";
print "(Should be just 100)\n";
```

4. [10] Write a subroutine named `greet` that welcomes the person you name by telling them the name of the last person it greeted:

```
greet( "Fred" );
greet( "Barney" );
```

This sequence of statements should print:

```
Hi Fred! You are the first one here!
Hi Barney! Fred is also here!
```

5. [10] Modify the previous program to tell each new person the names of all the people it has previously greeted:

```
greet( "Fred" );
greet( "Barney" );
greet( "Wilma" );
greet( "Betty" );
```

This sequence of statements should print:

```
Hi Fred! You are the first one here!
Hi Barney! I've seen: Fred
Hi Wilma! I've seen: Fred Barney
Hi Betty! I've seen: Fred Barney Wilma
```