# 3 Data types

Programming, in any language, involves creating named entities within the machine and manipulating them – using their values to calculate the value for a new entity, changing the values of existing entities, and so forth. Some languages recognize many different kinds of entity, and require the programmer to be very explicit and meticulous about "declaring" what entities he will use and what kind each one will be before anything is actually done with them.[4] In C, for instance, if a variable represents a number, one must say what *kind* of number – whether an integer (a whole number) or a "floating-point number" (what in everyday life we call a decimal), and if the latter then to what degree of precision it is recorded. (Mathematically, a decimal may have any number of digits after the decimal point, but computers have to use approximations which round numbers off after some specific number of digits.)

Perl is very free and easy about these things. It recognizes essentially just three types of entity: individual items, and two kinds of sets of items – *arrays*, and *hashes*. Individual entities are called *scalars* (for mathematical reasons which we can afford to ignore here – just think of "scalar" as Perl-ese for an individual data item); a scalar can have any kind of value – it can be a whole number, a decimal, a single character, a string of characters (for instance, an English word or sentence) ... We have already seen that variable names representing scalars (the only variables we shall be considering for the time being) begin with the `$` symbol; for arrays and hashes, which we shall discuss in chapters 12 and 17, the corresponding symbols are `@` and `%` respectively.

Furthermore, Perl does not require us to declare entity names before using them. In the mini-program (1), the scalars `$a` and `$b` came into existence when they were assigned values; we gave no prior notice that these variable names were going to be used.

In program (1), the variable `$b` ended up with the value 4. But, if we had added a further line:

```
$b = "pomegranate";
```

then `$b` would have ceased to stand for a number and begun to stand for a character-string – both are scalars, so Perl is perfectly willing to switch between these different kinds of value. That does not mean that it is a good idea to do this in practice; as a programmer you will need to bear in mind what your different variable names are intended to represent, which might be hard to do if some of them switch between numerical and alphabetic values. But the fact that one *can* do this makes the point that Perl does not force us to be finicky about housekeeping details.

Indeed, it is even legal to use a variable's value before we have given it a value. If line 1.2 of (1) were changed to `$b = $a + $c`, then `$b` would be given the sum of 2 plus the previously-unmentioned scalar `$c`. Because `$c` has not been given a value by the programmer, its value will be taken as zero (so `$b` will end up with the value 2). Relying on Perl to initialize our variables in this way is definitely a bad idea – even if we need a particular variable to have the initial value zero, it is much less confusing in the long run to get into the habit of always saying so explicitly. But Perl will not force us to give our variables values before we use them.

Because this free-and-easy programming ethos makes it tempting to fall into bad habits, Perl gives us a way of reminding ourselves to avoid them. We ran program (1) with the command:

```
perl twoandtwo.pl
```

The `perl` command can be modified by various options beginning with hyphens, one of which is `-w` for "give warnings". If we ran the program using the command:

```
perl -w twoandtwo.pl
```

then, when Perl encounters the line `$b = $a + $c` in which `$c` is used without having been assigned a value, it will obey the instruction but will also print out a warning:

```
Use of uninitialized value in addition (+) at twoandtwo.pl line 2.
```

If a skilled programmer gets that warning, it is very likely to be because he thinks he has given `$c` a value but in fact has omitted to do so. And `perl -w` gives other warnings about things in our code which, while legal, might well be symptoms of programming errors. It is a good idea routinely to use `perl -w` to run your programs, and to modify the programs in response to warning messages until the warnings no longer appear – even if the programs seem to be giving the right results.