19 Built-in variables

In chapter 16, we saw that the symbol `@_` is a special "built-in variable": whenever our program calls a user-defined function, say the function `despace()` which we defined in that chapter, `@_` stands for the list of arguments it is called with. Our program might say:

```
$stringA = "a string of letters";
print(despace($stringA), "\n");
```

```
a_string_of_letters
```

– in this case, while the code headed `sub despace` is being interpreted by the machine, the symbol `@_` (which is used within that code) stands for the one-element array whose single element is `$stringA`.

Perl has numerous built-in variables – a few others are also arrays, most are scalars. Let's look at some of the most useful of them.

Apart from `@_`, the most important built-in array variable is `@ARGV`, which does a job similar to `@_` at the level of the *command line* – the line addressed to your system prompt which tells it to run a Perl program. Our very first program, (1), was a program to add two and two and print out their sum. For a first program that was fine, but in real life it would obviously be more satisfying to have a program which added and printed the sum of *any* pair of numbers we choose to give it. Here is a program to do that; let's name it `printsum.pl`:

(34)

```
$a = $ARGV[0];
$b = $ARGV[1];
print $a + $b, "\n";
```

If we have created a file `printsum.pl` containing (34), we can use it by placing the arguments (that is, for this program, the numbers to be summed) after the program name (without commas):

```
perl -w printsum.pl 2.3 7.9
```

```
10.2
```

The array of arguments to `printsum.pl` is called `@ARGV`, so on this occasion `$ARGV[0]` is 2.3 and `$ARGV[1]` is 7.9.[35]

Better still, we can generalize the program by accepting *any number* of values to be summed – let's call the revised program `printsumm.pl` ("m" for "many"):

(35)

```
$t = 0;
foreach $item (@ARGV)
  {
  $t += $item;
  }
print "$t\n";
```

With `printsumm.pl` defined, we can write:

```
perl -w printsumm.pl 5 19 520 4

548
```

Turning to built-in scalar variables, in fact we have already seen some of these, in chapter 10 on pattern matching. `$1, $2, $3,` and so on stand for elements identified by round brackets in the pattern section of a pattern-match:

```
$word = "beautiful";
$word =~ /[^aeiou]([aeiou]+)[^aeiou]+([aeiou]+)[^aeiou]/;
# finds the 1st 2 vowel-sequences surrounded by non-vowels
print $1, "\n", $2, "\n";

eau
i
```

Related to these are the built-in variables `$&`, `` $` ``, and `$'` which, following a pattern-matching operation on a target string, stand for:

`$&`	the section of the target string which matched the pattern
`` $` ``	the preceding section of the target string
`$'`	the following section of the target string

Thus:

```
$word = "beautiful";
$word =~ /eau(..)/;
print $1, "\n";
print $&, "\n";
print $`, "\n";
print $', "\n";

ti
eauti
b
ful
```

The pattern between slashes covers the five characters `eauti` of `$word` (remember that `.` in a pattern stands for any single character); so `$&` stands for that five-character substring. The brackets round `..` mean that `$1` has the value `ti`; `` $` `` and `$'` stand for the portions of `$word` before and after the segment `eauti`.

Other built-in scalar variables have nothing to do with pattern matching. For instance, `$^T` gives an integer representing the time at which the current program began running (expressed in seconds since the beginning of the year 1970). This huge value may not sound much use in its own right, but for instance we can discover how long a system takes to execute some task by comparing `$^T` with the value returned by `time()`, which is a built-in function giving a count of seconds-since-1970 at the moment when the function call is reached in a program. How long does it take Perl to count to a hundred million? On my machine, six seconds, as measured by the following program `showtime.pl`:

(36)

```
for ($i = 0; $i < 100000000; ++$i)
  {;}
print $^T, "\n", time() - $^T, "\n";
```

```
perl -w showtime.pl

1277200878
6
```

Many built-in scalar variables represent fairly arcane systems-programming concepts, which at this introductory level we can afford to ignore. The most frequently-used built-in scalar variable of all, `$_`, will be passed over briefly here for a different reason. We encountered `$_` once, in chapter 12, in connexion with the `map()` function (where it is indispensable). But the commonest use of `$_` is to provide idiomatically brief alternatives to Perl constructions that take slightly longer to spell out explicitly. For seasoned programmers to whom brevity is important, this may be handy, but beginners are better advised to make their code fully explicit, and hence they should probably avoid using `$_`. (Actually, even professional programmers – not to speak of those who have to maintain their code after they have moved on – are probably better off in the long run making everything explicit at the cost of a few extra keystrokes. There is a geeky side to Perl which delights in terse obscurity for its own sake, and the symbol `$_` is arguably a symptom of that.)

After I have said that much, the reader will doubtless want me to say *something* specific about this use of `$_`, so I will give one example. We know that `foreach` is used to access each element of an array in turn:

(37)

```
@colours = ("blue","green","red","yellow");
foreach $colour (@colours)
  {
  $capColour = uc($colour);
  print "$capColour\n";
  }
```

```
BLUE
GREEN
RED
YELLOW
```

Alternatively, it is permissible to omit `$colour` after `foreach`, in which case `$_` is understood:

```
foreach (@colours)
  {
  $capColour = uc($_);
  print "$capColour\n";
  }
```

gives the same output as (37). The symbol `$_` here is like the word *it* in English: the first version of the `foreach` loop was saying something like "for each colour word in the array, change that colour word to upper case", the second version abbreviated that to something more like "for whatever is in the array, change *it* to upper case". In English, our speech would quickly become tedious if we spelled everything out rather than using the ambiguous word *it*. But then, in English we negotiate our meaning with one another constantly as we converse, so that ambiguities are eliminated as fast as they arise. In communicating with computers, real ambiguity and real misunderstandings are all too common and hard to avoid. Consequently I would recommend that beginners leave `$_` alone for a while.