20 The debugger

Anyone who writes programs to carry out realistic, non-trivial tasks is very familiar with the fact that they do not often work properly first time. Without noticing it, we make mistakes in translating our intentions into code. Quite a large fraction of a programmer's working life consists of finding and correcting bugs.

Various weapons are available to us in the never-ending war against bugs. Perhaps the most basic is summarized in the slogan "Keep your interpreter happy!" – meaning, in the case of Perl, always run it using the `-w` warning option, and when warnings are generated, investigate them and modify your program to eliminate them, even if the program seems to be performing correctly. A warning is a sign that something is odd in the code; even if it does not lead to errors with the data you have fed into the program so far, it may well lead to errors with other data, or after other parts of the code are modified – intermittent bugs are the hardest ones to cure.[36]

Plenty of bugs will never be caught by `-w`, though.[37] If studying your code on paper does not reveal where it is going wrong, there are all kinds of *ad hoc* ways to get the machine to tell you.

I mentioned in chapter 6 the technique of inserting diagnostic `print` statements at key points in one's code, to check whether some variable has the value which it ought to have at that point. Or, if an error seems to be related to the fact that, say, a variable `$x` ought to be positive at a given point and you suspect that it might not be, you can try inserting a diagnostic such as:

```
if (not ($x > 0))
  {
  die "$x not positive at line XXX\n";
  }
```

Inserting diagnostics like these into one's code is a recognized debugging technique, and it can work fine, particularly if there are not too many bugs.

This technique can be cumbersome, though, specially if bugs prove numerous. Each diagnostic statement is extra code to be written (and, once its work is done, it must be deleted with care to avoid disturbing the "real" code surrounding it). Perl incorporates a *debugger* facility, which eliminates the need to insert special diagnostic lines into the middle of the code which is doing the actual work of your program, and which provides great flexibility for the task of tracking bugs down.

The debugger is invoked by running Perl with the `-d` option on the command line (i.e. `perl -d`). Rather than (as normal) running the program, too rapidly for a human being to follow, until it runs out of code or hits an `exit` statement or the like, in response to `-d` the system will initiate a dialogue with the user: the user will be able to control line-by-line progress through the program code, and can ask questions about the state of play at any point.

By far the easiest way to give the reader a feel for how the debugger works is to show an extended example. Let's run the countyCalcs program (33) of chapter 18 in debug mode.

Figure 3 is a screenshot of a dialogue which a programmer might have with the debugger, initiated by the command

```
perl -d countyCalcs.pl
```

(the `%` symbol at the beginning of the first line of Figure 3 is the user's system prompt).

```
% perl -d countyCalcs.pl

Loading DB routines from perl5db.pl version 1.28
Editor support available.

Enter h or `h h' for help, or `man perldebug' for more help.

main::(countyCalcs.pl:1):     open(INFILE, "<countyDataPlus.txt") or die
("can\'t open countyDataPlus.txt");
  DB<1> s
main::(countyCalcs.pl:2):     $i = 0;   #initialize index for rows of
@counties
  DB<1> p $i

  DB<2> s
main::(countyCalcs.pl:3):     while ($line = <INFILE>)
main::(countyCalcs.pl:4):       {
  DB<2> p $i
0
  DB<3> b 11
  DB<4> c
main::(countyCalcs.pl:11):      $name = despace($name);
  DB<4> p $name
Bedfordshire
  DB<5> c
main::(countyCalcs.pl:11):      $name = despace($name);
  DB<5> c
main::(countyCalcs.pl:11):      $name = despace($name);
  DB<5> c
main::(countyCalcs.pl:11):      $name = despace($name);
  DB<5> p $name
Cambridgeshire and  Isle of Ely
  DB<6> b 12
  DB<7> c
main::(countyCalcs.pl:12):      $counties[$i]{name} = $name;
  DB<7> p $name
Cambridgeshire_and_Isle_of_Ely
  DB<8> q
%
```

Figure 3

The three lines following that command line are standard announcements by the debugger which it gives whenever it is invoked, and after that it displays the first line of the countyCalcs program.

After showing this first statement, the debugger gives its prompt, `DB<1>` – the debugger prompts are always underlined, and the user's inputs are in bold.[38]

Because the user is feeling his way in an unfamiliar system, his first input is a cautious **`s`** meaning "move a single step forward". The debugger responds by showing program line 2, which initializes the variable `$i`. At this point the user finds that he is unsure whether the line of code displayed by the debugger at any point is the statement which has just been executed, or the one which is about to be executed: to discover the answer, he prints the current value of `$i` (**`p`** means "print"). The system responds with a blank line: the program line displayed is always the statement which is *about to be* executed, and before line 2 is executed Perl does not yet know about `$i`.

To check, the user takes one further step forward (at which point the debugger displays the next code lines:

```
    while ($line = <INFILE>)
      {
```

– two lines which logically belong together as a unit), and again asks for the value of `$i` to be printed. This time it is shown as zero, because line 2 has been executed.

Now the user is ready to look at an aspect of the program where errors are more likely. Line 11 is the "despacing" line which is supposed to turn messy, multi-word county names into neat, all-black equivalents: there is plenty of room for programming errors there. The first few county names (Bedfordshire, Berkshire, Buckinghamshire) are not very challenging, but it would be interesting to see what happens to the next name.

Continuing to move through the `while` loop one step at a time with **`s`** would become excessively tedious, so instead the user inputs **`b 11`**, which makes line 11 (the line after `$name` has been assigned an "unreformed" string as its value) into a *breakpoint*. Then the user can input **`c`** ("continue"), meaning "run through the code until you reach a breakpoint". (The numbers within angle brackets in the debugger prompts rise with each user "action": printing a variable value, or setting a breakpoint, count as actions, but stepping or continuing do not count as actions by the user.)

After the first **`c`**, the user prints the value of `$name` and finds that, as expected, it is `Bedfordshire`. Three more **`c`** inputs cause the program to make further passes through the `while` loop, and then another **`p`** confirms that the last of these passes has reached the interesting county name. Now the user sets a second breakpoint at line 12, after `$name` should have been "despaced", and continues to that line. When **`p $name`** is entered again, the county name is shown in what turns out indeed to be its correct despaced form.

The user is now happy (for the time being, at least!), so enters **`q`** (quit) and returns to the system prompt.

On this occasion there was no bug to be found and corrected. But it is clear that when things do go wrong, the debugger offers a very powerful mechanism for seeking out the root of the error. The debugger commands we have looked at – step, set breakpoint, continue, print value – are only a handful (though probably the most useful handful) among dozens of different commands that the debugger recognizes. The command **`W`** sets a "watch" on a particular variable – rather than setting breakpoints at lines 11 and 12 and printing out the current values of `$name`, we could have begun with the command **`W $name`**, in which case the debugger would have reported on each occasion when `$name` was given a new value. We can override our program code by imposing a new value on a variable at a given point in code execution, and looking to see whether the consequences of that are what we expect them to be. And the debugger offers many other possibilities for monitoring the performance of our code and finding out precisely where it is going wrong.