

Because No One Writes Perfect Code



Perl Debugger

Pocket Reference

O'REILLY®

Richard Foley

Perl Debugger Pocket Reference

What Is the Perl Debugger?

In the ideal world, every program would be perfect the first time it is written. In reality, however, even the best programmers make mistakes or forget to provide for all situations.

The Perl debugger is an application you can use to follow the logic of a Perl program while it is being executed, saving you time and frustration in tracking down bugs in your programs. With the Perl debugger, you can stop the program at selected points, print and modify the contents of variables, and print stack traces out to see what has been called from where.

This book describes every command, option, and variable of the Perl debugger. Any intermediate to advanced Perl programmer should be familiar with the debugger to save time and frustration while debugging their programs.

Why Use the Debugger?

There are many ways to find out what is going wrong in a Perl program. Reading through the code with a co-programmer, for example, often helps bring revelations to light. The humble `print` statement itself has probably saved more programs than there are programmers; simply inserting `print` statements at strategic points in the program can reveal surprising behavior that often leads to the discovery of logical lapses.

Sometimes, however, there comes a time when the simple print statement is not enough. You might need to diagnose a problem on a program already in production that you are unable to modify. In this case, it is invaluable to be able to run the program, stop it wherever you wish, and check or perhaps modify the internal state of certain data structures without modifying the program itself.

Even when you can modify the program, scattering print statements all over the place is generally unhealthy. For example, you might insert an extra next or last statement into a loop that is accidentally forgotten, changing the program logic. The harder the problem is, the more risk of disruptions to the code, as more and more allegedly harmless code modifications are inserted.

The beauty of the debugger is that the running behavior of the application can be inspected and modified without changing any of the code. The Perl debugger gives you this control; the print statement does not.

Unfamiliar Territory

People shy away from using the debugger for several reasons. Many are awed by the slightly heavy looking documentation, which is both scattered and, in places, vague or incomplete. This book remedies this situation by bringing everything into a single, small, and easy-to-use reference volume.

Also, in today's multiskilled world, programmers are very often people who have cross-trained into the sector without having formally studied the subject. Many programmers working today have never been taught the basic functional use of a debugger in a computer-science class. Many programmers are therefore unfamiliar with actually stepping through their program, one executable statement at a time, to investigate the actual value of variables at runtime, and to change the behavior of the program without modifying the code.

Once you have discovered this ability, it is a marvelous thing to use. Even when a program is working perfectly, stepping through it in the debugger is oftentimes the best way to get to know the working behavior of an inherited program.

Finally, an often quoted declaration is that “Larry favors the print statement.” I can only reply (in addition to the comments above) that if I were Larry I’m sure I wouldn’t need to debug nearly so much in the first place. However, because I’m not Larry, I’ll use any tool at my disposal to make debugging my program easier and faster, including the debugger, which he also wrote.

In my experience, the debugger is a greatly underused tool. When I ask even hard-core Perl developers, only about 30% of them have actually used it. If this reference helps to redress the balance a bit, and more people put the debugger in their troubleshooting toolkit, this book will have done its job.

About This Book

The debugger commands in this book are grouped by the type of behavior, increasing in level of complexity. I show the syntax for every command or option, and an example of each argument variation. At the end of this book is a quick reference with simple summaries of each command.

The code I use to demonstrate the commands is fundamentally clean working code, which at first sight might seem a bit odd in a book describing a tool that can help in solving problem code. The rationale is that rather than find a new problem for each command to demonstrate how clever I am in finding new problems, I have decided to concentrate on explaining the commands and their operation in a fully operational environment. Engineers study working models to discover how materials and design solutions might work

* Larry Wall, inventor of Perl, in case you picked up the wrong book by accident.

together to prevent problems. They do not make a habit of studying only broken bridges when it is too late to fix them.

You are encouraged to try the examples in this book at any point throughout this book. Completing a task is a far better way of learning than just reading about it.

When you need to get into the debugger without a program, use the following syntax:

```
perl@monkey> perl -d -e 0
```

which supplies 0 to the debugger as the (bogus) command-line program to execute.

The `linecounter.pl` Script

When we need to step through an existing program in this book, I refer mostly to a program I call `linecounter.pl`, which reports which lines in a file match a given pattern. It is a reasonable demonstration script, because it has command-line arguments, uses a module, includes a loop, calls a subroutine, etc. For the purposes of this book, I usually call it something like this:

```
perl@monkey> perl -d linecounter.pl pattern inputfile
```

This and any other code used in this book can be found on the O'Reilly website at:

<http://www.oreilly.com/the-perl-debugger/code>

If you're reading this book on top of a mountain and don't have online access, here is the source code with comments removed for brevity:

```
use strict;
use FileHandle;

my @args = @ARGV;

my $help = grep(/^-[elp]*$/i, @args);
if ($help) {
    print logg(help());
    exit 0;
}
```

```

my $verbose = grep(/^-v[erbose]*$/i, @args);

my $REGEX = shift @args || '';

my @files = grep(!/^-(h[elp]*|v[erbose]*)$/i, @args);
unless (@files) {
    push(@files, $0);
    logg("using default $0 while no files given");
}

foreach my $file (@files) {
    if (-f $file && -r _) {
        my $FH = FileHandle->new("< $file");
        if ($FH) {
            logg("file($file)");
            my %report = %{report($FH, $REGEX)}; # -> subroutine
            if (keys %report) {
                logg("the LINENO and CHARACTERS matching the
pattern($REGEX) for '$file': ");
                foreach my $len (sort { $a <=> $b } keys %report) {
                    print "    ".sprintf("%-10.d", $len).
$report{$len}\n";
                }
            } else {
                logg("no matching pattern($REGEX) found in '$file'");
            }
        } else {
            error("failed to open file($file) $!");
        }
    } else {
        error("no such or unreadable file($file) $!");
    }
}

sub report {
    my $FH = shift;
    my $regex = shift;
    my %report = ();
    my $i_cnt = 0;
    while (<$FH>) {
        $i_cnt++;
        my $i_match = 0;
        my $line = $_;
        if ($line =~ /($regex)/) {
            $report{$i_cnt} = $1; #
            $i_match++;
        } else {
            $i_match = 0;
        }
        logg("\t[$i_cnt] regex($regex) matched($i_match)
<-$line");
    }
}

```

```

    }
    $FH->close;
    return \%report;
}

exit 0;

sub help {
    return qq|Usage: $0 pattern file [file]+ [-help]
        [-verbose]|.\n".
        qq|Example: perl $0 \\.*\s*mat input_file|
    ;
}

sub logg {
    my $msg = join("\n", @_)."\n";
    print STDOUT $msg if $verbose;
    return $msg;
}

sub error {
    my $error = shift;
    logg("Error: $error", @_);
}

```

Shell

Throughout this book I use the bash shell on Linux. Certain commands may need to be adapted if you are using a different shell on a different platform. For example, when exporting a variable into the environment, I typically do it in the bash shell like this:

```
perl@monkey> VAR_NAME=var_value perl -d program args
```

which exports the `VAR_NAME` variable into my environment with the value `var_value` for the duration of that program call. Your mileage may vary with your shell and operating system.

Truncated Output

Note that the debugger is somewhat verbose in its output, so I have truncated most of its output in order to save space. Empty lines have unceremoniously been removed. Snipped text is indicated with the `<...truncated output...>` marker,

used for verbose code or variable listings where the amount of peripheral noise would dilute the demonstration.

Also, where the output of some commands runs off the right-hand side of the page, and where this output is not deemed relevant to the example, I have trimmed the offending text and indicated this with the ... marker.

Versions of Perl

This book uses the Perl 5.8.0 distribution, spearheaded by Jarkko Hietenami and the *perl5-porters* (running on Linux). While there have been changes to certain commands from earlier versions, and those changes are mentioned where relevant, the command set described in this book applies to most historical versions of *perl5db.pl* with minimal changes. Before you do anything else, you should browse the existing *perldocs*; the most important are mentioned in the text as needed.

Delving Deeper

This book concentrates on debugging programs written in Perl using the debugger as supplied in a standard Perl distribution. Getting at the internals of Perl (written in C) would require the use of a C source-level debugger such as *gdb* and is outside the scope of this book. If you are interested in debugging Perl itself, you should familiarize yourself with the *perlguts*, *perlapi*, *perlcalls*, and *perlhack* manpages.

Conventions

Constant width is used to indicate code, function names, and anything to be typed literally.

Constant bold is used to indicate user input.

Italics are used for program names.

Constant italics are used to show comments in the code listings.

Acknowledgments and Disclaimers

Thanks to my family—Joy, Catherine, Jennifer, and Spot the dog—for being so patient while this book was being written. Thanks also to the reviewers—Ronald Kimball, Joe McMahon, Nicholas Clark, and Gabor Szabo—who through their feedback have produced a much better book. Thanks to my editor at O’Reilly, Linda Mui, without whom this book would surely never have seen the light of day.

Although I have tried to be thorough and the reviewers did a great job, it is always possible that I may have missed something. I take full responsibility for any errors which may remain.

Before You Debug

Before running the debugger, there are a number of things you can do that will make debugging more straightforward. If you restrict the number of poor programming habits that are currently reflected in the code, you may find that you don’t need the debugger at all. Sometimes just talking through the execution of your application with a fellow programmer may be all it takes to clarify what’s wrong with the logic.

Some of the following tools and suggestions may also prove useful. If these aren’t already part of your programming arsenal, they should be.

Check Your Syntax

Perhaps the first thing to do with any program is to check it for syntax flaws without actually running it. The `-c` switch loads the Perl program, compiles it, and checks it for syntax errors:

```
perl@monkey> perl -c prog
program syntax OK
```

Note that Perl executes any BEGIN and CHECK blocks, as well as any use statements, as these are considered compile-time code.

Perl does not run any require statements or INIT blocks. require statements are runtime code and are not executed during the syntax check.

perl -c is especially useful when editing a file in a programmer's text editor, where it is possible to syntax check the current program being edited. For example, in vi's command-line mode, you can check that your code is Perl compliant with the following line:

```
:!perl -cWT %
```

Use strict

The strict pragma disallows soft references, ensures that all variables are declared before usage, and disallows barewords except for subroutines.

Once a program has been written, making it strict-compliant is notoriously awkward, particularly when your program (written by a predecessor; not you, of course!) may have reused certain variables in a different scope and possibly made use of global variables with abandon. I have seen some installations where a single subroutine spanning several hundred lines reused the same variable name in multiple nested loops, which is a disaster waiting to happen.

One of the powerful things about Perl is that strict mode is optional. Nevertheless, most people should enable it. It is much simpler to put use strict in *before* you start writing code, if only for a quieter life afterwards.

Consider the following code in a file called *variable.pl*, which has strict enabled and includes a compile-time error:

```
$variable = 'auto-declared'; # ok
use strict;
$variable = 'second use';    # this is a compile time error
my $variable = 'now ok';    # ok
print "$variable\n";
```

Now run the code:

```
perl@monkey> perl ./variable.pl
Variable "$variable" is not imported at ./variable.pl line 3.
Global symbol "$variable" requires explicit package name at ./
variable.pl line 3.
Execution of ./variable.pl aborted due to compilation errors.
```

Because `strict` is in effect, the program is aborted. If `use strict` had been omitted, the program would have run without a complaint.

Note that `strict` can be turned off selectively for a block of code:

```
use strict;
{
    no strict;
    $variable = 'never seen before'; # no strict effect
}
# $variable = 'strict in this scope'; # compile time error
my $variable = 'now ok';           # ok
print "$variable\n";
```

Now run the code:

```
perl@monkey> perl ./variable.pl
now ok
```

For more information, see the documentation for `strict`.

Establish a Good Development Environment

One frequent problem is the lack of a sensible developer environment to help programmers work together on concurrent versions of an application. When a number of people are developing or fixing each other's code, it is imperative to use a source code versioning system, like `cvs` , and to have an automatic testing system in place. By running the test suite automatically after any changes have been applied, you ensure that nothing has been broken, reducing the amount of debugging required in the first place.

For more information on `diff` , `patch` and `cvs` , see their respective manpages. For a test module, see `Test::More` .

Warnings

Turning warnings on causes Perl to emit warning messages about code constructs it considers dubious under certain conditions. You can turn warnings on for the entire program with the `-w` command-line switch. To turn on warnings locally (that is, only within a discrete block of code), you can enable use `warnings` or use the special variable `W`.

For example, consider *print.pl*, a program containing only the following line:

```
print $ARGV[0];
```

If you use the `-w` switch, expect the following output:

```
perl@monkey> perl -w print.pl
Use of uninitialized value in print at program line 1 (#1)
```

If you want to turn warnings off for the entire program, use the `-X` command-line switch. To turn off warnings locally, use `no warnings` or set `W=0`, which overrides all locally set flags.

If you want to turn all warnings on, use `-W`. Warnings are set regardless of locally-set `W=0` and `no warnings` declarations.

For more information, see the *perllexwarn* and *warnings* manpages.

Diagnostics

If Perl's error and warning messages do not provide enough information, use the `diagnostics` pragma. With the *print.pl* program shown previously, add `diagnostics` output by hard-wiring the pragma within the program:

```
use diagnostics;
print $ARGV[0];
```

Now the program emits verbose diagnostic messages:

```
perl@monkey> perl ./print.pl
Use of uninitialized value in print at warning line 1 (#1)
```

(W uninitialized) An undefined value was used as if it were already defined. It was interpreted as a "" or a 0, but maybe it was a mistake.
To suppress this warning assign a defined value to your variables.

To help you figure out what was undefined, Perl tells you what operation you used the undefined value in. Note, however, that Perl optimizes your program and the operation displayed in the warning may not necessarily appear literally in your program. For text, "that \$foo" is usually optimized into "that " . \$foo, and the warning will refer to the concatenation (.) operator, even though there is no . in your program.

If you don't want to hardwire the use diagnostics statement directly into your code, you can get the same behavior by using the `-M` command-line switch:

```
perl@monkey> perl -Mdiagnostics -w print.pl
Use of uninitialized value in print at warning line 1 (#1)
(W uninitialized) An undefined value was used as if it were
<...truncated output...>
```

Note that this diagnostic information is generated via *perldiag*. It is equally valid to look up the messages directly in the Perl manpages.

If you think you have discovered a bug in Perl, it's a good idea to run your program with use diagnostics enabled before you report the bug. Sometimes this helps to clarify otherwise inexplicable behavior and saves the trouble of generating an erroneous bug report. You might also check for known bugs at the Perl bug database.

For more information on diagnostics, see the *perldiag* and *splain* manpages.

Taint Mode

Tainting is a kind of paranoid mechanism in which Perl treats every variable from outside your program as "tainted," or dangerous, and refuses to run external programs from unacknowledged locations. You enable taint mode from the command line with the `-T` command-line option.

The *splain* Program

The *splain* program accepts warning messages and converts them to verbose messages, precisely as `use diagnostics` does, but again with the advantage of not having to hardwire anything into the code. Any time you have a message you don't understand, give *splain* a chance to decipher it on your behalf:

```
perl@monkey> perl -w print.pl | splain
Use of uninitialized value in print at warning line 1
(W uninitialized) An undefined value was used as if it
<...truncated output...>
```

For more information, see the manpage for *splain*.

When you pass the `-T` flag to Perl, it enables certain checks to prevent the careless programmer from shooting herself in the foot. It forces her to check, or *untaint*, each external chunk of data before being allowed to use it. Any process running on behalf of other users, such as CGI programs that normally run under a special web server account, should have taint enabled.

Sometimes you have to jump through various hoops to get your program to work smoothly with taint enabled, but it is invariably worth it. In the following example (in a program called *taint.pl*), an external program (*/bin/echo*) is called, where the directory in which it resides (*/bin/*) is not in the environment `$PATH`:

```
#!/usr/bin/perl -T

# $ENV{PATH}='/bin';
system "/bin/echo Hello World";
```

Perl rightfully complains when this program is run:

```
perl@monkey> ./taint.pl
Insecure $ENV{PATH} while running with -T switch at taint.pl
line 1.
```

Uncomment the `$ENV{PATH}='/bin';` line, and Perl allows the program to call the executable (presuming the path is correct) and prints the expected line:

```
perl@monkey> ./taint.pl
Hello World
```

Perl doesn't like to take any chances with taint mode, so it insists that taint be enabled at the command line. There's no use `taint`, special variable, or any other mechanism for turning taint mode on and off in the middle of the program; taint mode must be turned at the command line or not at all. This means that if you need to call Perl directly on the command line (perhaps you want to use a particular version of Perl, or you are passing other command-line switches such as `-d` for the debugger), Perl will complain about it being too late for the taint switch, even if it appears on the `#!` line at the very top of the program. Even the first line of a program is considered "too late":

```
perl@monkey> perl ./taint.pl
Too late for "-T" option at ./taint.pl line 1.
```

The solution is simple: just place `-T` as a command-line option, so Perl can see it early enough:

```
perl@monkey> perl -T ./taint.pl
Hello World
```

If you need something a bit lighter, you can use `-t` (introduced in 5.8.0), which enables taint in a warnings-only mode. While the standard behavior with `-T` is to treat failing constructs as fatal errors, the `-t` option allows the program to still run, emitting only warnings:

```
perl@monkey> perl -t ./taint.pl
Insecure $ENV{PATH} while running with -t switch at ./program
line 3.
Insecure directory in $ENV{PATH} while running with -t switch
at ./program line 3.
Insecure dependency in system while running with -t switch at
./program line 3.
Hello World
```

The `-t` option overrides later `-T` switches.

For more information on taint mode, see the *Taint* and *perlsec* manpages.

A Debugger Tutorial

In this section I present a couple of very short sessions to demonstrate some simple debugger usage. I show a simple debugger session with no program and then a walkthrough of debugging a CGI program with a couple of problems.

Starting a Session

For an example of its simplest usage, start the debugger by giving it something innocuous as the evaluation argument to the debugger via the command line. I like to use a 0 (zero) as the evaluation argument as my entry to the debugger. Use the `-d` option to call the debugger and the `-e 0` option to specify 0 as the evaluation script:

```
perl5db@monkey> perl -d -e 0
Default die handler restored.

Loading DB routines from perl5db.pl version 1.19
Editor support available.

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

main::(-e:1): 0
DB<1>
```

Perl stops at the first executable statement, which in this example is 0, a perfectly valid (though minimal) Perl expression. If you continue now, you'll just fall off the end of the program.

Although this example may not immediately appear to be very useful, at this point you can in fact interact completely with the debugger. You can look at any environment variables (V) and you can also check out the help (h) page/s, and explore your surroundings.

To get the help page simply type `h` at the command prompt:

```
DB<1> h
List/search source lines:
l [ln|sub]      List source code
- or .         List previous/current line
w [line]       List around line
filename       View source in file
/pattern/ ?patt? Search forw/backw
v             Show versions of modules
Debugger controls:
O [...]       Set debugger options
<[<]{{[]}>[>]
  [cmd]       Do pre/post-prompt
! [N|pat]     Redo a previous command
H [-num]     Display last num commands
= [a val]    Define/list an alias
h [db_cmd]   Get help on command
[[]db_cmd    Send output to pager
q or ^D      Quit
Data Examination:
expr
x[m expr     Evals expr in list context, dumps the result or lists methods.
p expr       Print expression (uses script's current package).
S [{}pat]    List subroutine names [not] matching pattern
V [Pk [Vars]] List Variables in Package. Vars can be ~pattern or !pattern.
X [Vars]     Same as "V current_package [Vars]".
For more help, type h cmd_letter, or run man perldebug for all docs.
DB<2>
```

You can run any Perl code you like in this environment. You can think of it as a form of Perl shell. For example, suppose you can't remember whether the results of a regular expression are given in scalar or array context. Instead of looking it up in *perlre*, try it in the debugger. First create a string `$str`:

```
DB<2> $str = 'some string'
DB<3>
```

Then print `$str` with the debugger's `p` command, which is just Perl's print command in disguise:

```
DB<3> p $str
some string
DB<4>
```

Now capture the return value of a regex into a scalar named `$res` and print the result:

```
DB<4> $res = $str =~ /(\w+)\s+(\w+)/
DB<5> p $res
1
DB<6>
```

From this result, you should see that a regex doesn't assign a meaningful value to a scalar. Now capture the return value of the same regex into an array variable named `@res` and print that:

```
DB<7> @res = $str =~ /(\w+)\s+(\w+)/
DB<8> p @res
somestring
DB<9>
```

From this little experiment, you should have discovered that regexes return their results in an array context. Use `x` to dump the results to get a clearer picture:

```
DB<9> x \@res
0 ARRAY(0x844df5c)
0 'some'
1 'string'
DB<10>
```

In a similar fashion, see what happens when you increment a string, a strange but possibly useful thing to want to do. In any event, it is harmless enough to try it. First create and print a string variable:

```
DB<10> $foo = 'abc'
DB<11> p $foo
abc
DB<12>
```

Increment the string and print the result:

```
DB<12> ++$foo
DB<13> p $foo
abd
DB<14>
```

So incrementing a string ends up incrementing the last character in the string.

At this point, you haven't loaded any program to debug, but you should have a feel of how to move around inside the debugger. You can continue to play around in the debugger to your heart's content now. To exit a debugger session, use the `q` command:

```
DB<14> q
perl5@monkey>
```

A Simple CGI Debugger Session

For this session you will use an actual program, with actual problems. First copy this code, which can be downloaded from the O'Reilly website, into a file called *problem.cgi*:

```
use CGI;
my $o_cgi = new CGI();

my %data = (
    name    => $o_cgi->param('Name')    || 'UNKNOWN',
    company => $o_cgi->param('company') || 'UNKNOWN',
);

$data{name} =~ s/^(.)?(.+)$/uc($1);lc($2)/e;

print $o_cgi->header(-type=>'text/html'),
      $o_cgi->start_html(-title=>'Perl Debugger Tutorial'),
      qq|
      <hr>
      Hi <b>$data{name}</b> welcome to the <i>$data{company}</i>
      debugger home page!
      <hr>
      |, $o_cgi->end_html."\n";

exit 0;
```

Now call it on the command line with `name` and `company` arguments to view the output:

```
perl5@monkey> perl problem.cgi name=monkey company=Perl
Content-Type: text/html; charset=ISO-8859-1

<?xml version="1.0" encoding="iso-8859-1"?>
<!DOCTYPE html
  PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" lang="en-US"><head>
<title>Perl Debugger Tutorial</title>
```

```

</head><body>
  <hr>
  Hi <b>nknown</b> - welcome to the Perl debugger home page!
  <hr>
</body></html>
perl@monkey>

```

The program prints vast amounts of Web-centric information that is irrelevant to us. More importantly, a user with a name of monkey has been welcomed with the impressively wrong:

```
Hi <b>nknown</b> - welcome to the Perl debugger home page!
```

Instead of monkey, the program thinks the user is called nknown.

There are several ways to approach this problem. Running the program with warnings or diagnostics would certainly help locate some problems. You can also try to make sense of it via the debugger. Run the same command again, this time with the `-d` option:

```
perl@monkey> perl -d problem.cgi name=monkey company=Perl
<...truncated output...>
main::(problem.cgi:2): my $o_cgi = new CGI();
DB<1>

```

The program has stopped execution at the first executable line and is waiting for an instruction. Get a code listing to see where you are; use the `l` command for a code listing:

```

DB<1>l
2==>   my $o_cgi = new CGI();
3
4:     my %data = (
5:         name    => $o_cgi->param('Name')    || 'UNKNOWN',
6:         company => $o_cgi->param('company') || 'UNKNOWN',
7:     );
8
9:     $data{name} =~ s/^(.)(.+)$/uc($1);lc($2)/e;
10
11:    print $o_cgi->header(-type=>'text/html'),
DB<1>

```

Although you have not executed anything yet, print out the command line using `p` to confirm that the program has really been given the correct arguments:

```
DB<1> p "args: @ARGV"
args: name=monkey company=Perl
DB<2>
```

This looks good. Now use `n` to step over the first executable statement, which enables CGI to pick up the user input:

```
DB<2> n
main:(./problem.cgi:4):      my %data = (
main:(./problem.cgi:5):      name    => $o_cgi->
param('Name') || 'UNKNOWN',
main:(./problem.cgi:6):      company => $o_cgi->
param('company') || 'UNKNOWN',
main:(./problem.cgi:7):      );
DB<2>
```

This shows that Perl considers the next 4 lines of code as a single command to execute within the current scope. Notice that you are about to be given an invalid parameter by using `Name`, instead of the `name` that you used on the command line.

Check this by printing the value returned by `$o_cgi->param('Name')`:

```
DB<2> p $o_cgi->param('Name')
DB<3>
```

After making a note to modify line 5 with the appropriate name argument in the source code at a later stage, adjust the current value for the CGI object directly, printing it to make sure it is correct:

```
DB<3> p $o_cgi->param('Name', 'monkey')
monkey
DB<4>
```

Use `n` to step to the next executable statement:

```
DB<4> n
main:(./problem.cgi:9):      $data{name} =~ s/^(.)(.+)$/
uc($1);lc($2)/e;
DB<4>
```

Use `x` to print out the contents of the `%data` structure. Note the use of a reference, which tells the debugger to “pretty print” the dumped data:

```
DB<4> x \%data
0 HASH(0x817e758)
  'company' => 'Perl'
  'name' => 'monkey'
DB<5>
```

This explains why the parameter appears as `unknown`, but not why it appears as `nknown`. The first character of the user’s name has been lost. Use `c 11` to continue to line 11 and check the `$data{name}` again:

```
DB<5> c 11
main::(./problem.cgi:11): print $_cgi->header(-type=>'text/html'),
main::(./problem.cgi:12): $_cgi->start_html(-
title=>'Perl Debugger Tutorial'),
main::(./problem.cgi:13): qq|
main::(./problem.cgi:14): <hr>
main::(./problem.cgi:15): Hi <b>$data{name}</b>
welcome to the <i>$data{company}</i> debugger home page!
main::(./problem.cgi:16): <hr>
main::(./problem.cgi:17): |, $_cgi->end_html."\n";
DB<6> p $data{name}
onkey
DB<7>
```

Clearly, something is not right: the name `monkey` has its first character stripped and appears as `onkey`. If you look at line 9, you can see that `uc()` is used on the first and single character and `lc()` on the remainder of `monkey`, which looks suspiciously like the source of the problem:

```
DB<7> l 9
9=> $data{name} =~ s/^(.)?(.+)$/uc($1);lc($2)/e;
DB<8>
```

First, check the value of the current matches:

```
DB<8> p $1
m
DB<9> p $2
onkey
DB<10>
```

These look good, but a closer look at the expression reveals that `/uc($1);lc($2)/` produces two values when you are only assigning to one.

Try resetting the `$data{name}` and running a version of line 11 that concatenates the matches, by replacing the `;` with a `.`, and see what happens:

```
DB<10> $data{name} =~ s/^(.?)?(.+)$/uc($1).lc($2)/e
DB<11> p $data{name}
Monkey
DB<12>
```

That's it! Fix the source code, rerun the code to check that the output looks like the following line, and you're done:

```
Hi <b>Monkey</b> welcome to the <i>Perl</i> debugger home
page!
```

For an extensive tutorial introduction to the Perl debugger, see *perldebtut*, the debugger tutorial provided with Perl. For CGI-specific debugging see the *CGI*, *cgidebug*, and *perlfaq9* manpages.