Chapter 14

When Trouble Strikes

It's sad to say, but it is possible to write buggy programs using Ruby. Sorry about that.

But not to worry! Ruby has several features that will help debug your programs. We'll look at these features, and then we'll show some common mistakes you can make in Ruby and how to fix them.

Ruby Debugger

Ruby comes with a debugger, which is conveniently built into the base system. You can run the debugger by invoking the interpreter with the -r debug option, along with any other Ruby options and the name of your script:

```
ruby -r debug [ debug-options ] [ programfile ] [ program-arguments ]
```

The debugger supports the usual range of features you'd expect, including the ability to set breakpoints, to step into and step over method calls, and to display stack frames and variables. It can also list the instance methods defined for a particular object or class, and it allows you to list and control separate threads within Ruby. Table 14.1 on page 231 lists all the commands that are available under the debugger.

If your Ruby installation has readline support enabled, you can use cursor keys to move back and forth in command history and use line-editing commands to amend previous input.

To give you an idea of what the Ruby debugger is like, here is a sample session (with user input in bold type):

```
% ruby -r debug t.rb
Debug.rb
Emacs support available.
t.rb:1:def fact(n)
(rdb:1) list 1-9
[1, 9] in t.rb
=> 1  def fact(n)
   2    if n <= 0
   3      1
   4    else
   5      n * fact(n-1)
   6    end
   7  end
   8
   9  p fact(5)
(rdb:1) b 2
Set breakpoint 1 at t.rb:2
(rdb:1) c
breakpoint 1, fact at t.rb:2
t.rb:2:  if n <= 0
(rdb:1) disp n
  1: n = 5
(rdb:1) del 1
(rdb:1) watch n==1
Set watchpoint 2
(rdb:1) c
watchpoint 2, fact at t.rb:fact
t.rb:1:def fact(n)
1: n = 1
(rdb:1) where
--> #1  t.rb:1:in `fact'
    #2  t.rb:5:in `fact'
    #3  t.rb:5:in `fact'
    #4  t.rb:5:in `fact'
    #5  t.rb:5:in `fact'
    #6  t.rb:9
(rdb:1) del 2
(rdb:1) c
120
```

Interactive Ruby

If you want to play with Ruby, we recommend Interactive Ruby—irb, for short. irb is essentially a Ruby "shell" similar in concept to an operating system shell (complete with job control). It provides an environment where you can "play around" with the language in real time. You launch irb at the command prompt:

```
irb [ irb-options ] [ ruby_script ] [ program-arguments ]
```

irb will display the value of each expression as you complete it. For instance:

```
% irb
irb(main):001:0> a = 1 +
irb(main):002:0* 2 * 3 /
irb(main):003:0* 4 % 5
=> 2
irb(main):004:0> 2+2
=> 4
irb(main):005:0> def test
irb(main):006:1> puts "Hello, world!"
irb(main):007:1> end
=> nil
```

```
irb(main):008:0> test
Hello, world!
=> nil
irb(main):009:0>
```

irb also allows you to create subsessions, each one of which may have its own context. For example, you can create a subsession with the same (top-level) context as the original session or create a subsession in the context of a particular class or instance. The sample session shown in Figure 14.1 on the following page is a bit longer but shows how you can create subsessions and switch between them.

For a full description of all the commands that irb supports, see the reference beginning on page 278.

As with the debugger, if your version of Ruby was built with GNU readline support, you can use Emacs- or vi-style key bindings to edit individual lines or to go back and reexecute or edit a previous line—just like a command shell.

irb is a great learning tool. It's very handy if you want to try an idea quickly and see whether it works.

Editor Support

The Ruby interpreter is designed to read a program in one pass; this means you can pipe an entire program to the interpreter's standard input, and it will work just fine.

We can take advantage of this feature to run Ruby code from inside an editor. In Emacs, for instance, you can select a region of Ruby text and use the command Meta-| to execute Ruby. The Ruby interpreter will use the selected region as standard input, and output will go to a buffer named *Shell Command Output*. This feature has come in quite handy for us while writing this book—just select a few lines of Ruby in the middle of a paragraph, and try it!

You can do something similar in the vi editor using :%!ruby, which *replaces* the program text with its output, or :w␣!ruby, which displays the output without affecting the buffer. Other editors have similar features.[1]

Some Ruby developers look for IDE support. Several decent alternatives came to the fore during 2007 and 2008. Arachno Ruby, NetBeans, Ruby in Steel, Idea, and so on, all have their devotees. It's a rapidly changing field, so I'd recommend a quick web search rather than rely on my advice here.

While we are on the subject, this would probably be a good place to mention that a Ruby mode for Emacs is included in the Ruby source distribution as ruby-mode.el in the misc/ subdirectory. Many other editors now include support for Ruby; check your documentation for details.

1. If you use a Mac, take a look at Textmate (http://macromates.com). It isn't free, but it is a great Ruby environment.

Figure 14.1. Sample irb Session

```
% irb
irb(main):001:0> irb
irb#1(main):001:0> jobs
#0->irb on main (#<Thread:0x401bd654>: stop)
#1->irb#1 on main (#<Thread:0x401d5a28>: running)
irb#1(main):002:0> fg 0
#<IRB::Irb:@scanner=#<RubyLex:0x401ca7>,@signal_status=:IN_EVAL,
           @context=#<IRB::Context:0x401ca86c>>
irb(main):002:0> class VolumeKnob
irb(main):003:1> end
=> nil
irb(main):004:0> irb VolumeKnob
irb#2(VolumeKnob):001:0> def initialize
irb#2(VolumeKnob):002:1> @vol=50
irb#2(VolumeKnob):003:1> end
=> nil
irb#2(VolumeKnob):004:0> def up
irb#2(VolumeKnob):005:1> @vol += 10
irb#2(VolumeKnob):006:1> end
=> nil
irb#2(VolumeKnob):007:0> fg 0
#<IRB::Irb:@scanner=#<RubyLex:0x401ca7>,@signal_status=:IN_EVAL,
           @context=#<IRB::Context:0x401ca86c>>
irb(main):005:0> jobs
#0->irb on main (#<Thread:0x401bd654>: running)
#1->irb#1 on main (#<Thread:0x401d5a28>: stop)
#2->irb#2 on VolumeKnob (#<Thread:0x401c400c>: stop)
irb(main):006:0> VolumeKnob.instance_methods
=> ["up"]
irb(main):007:0> v = VolumeKnob.new
#<VolumeKnob: @vol=50>
irb(main):008:0> irb v
irb#3(#<VolumeKnob:0x401e7d40>):001:0> up
=> 60
irb#3(#<VolumeKnob:0x401e7d40>):002:0> up
=> 70
irb#3(#<VolumeKnob:0x401e7d40>):003:0> up
=> 80
irb#3(VolumeKnob):004:0> fg 0
#<IRB::Irb:@scanner=#<RubyLex:0x401ca7>,@signal_status=:IN_EVAL,
           @context=#<IRB::Context:0x401ca86c>>
irb(main):009:0> kill 1,2,3
=> [1, 2, 3]
irb(main):010:0> jobs
#0->irb on main (#<Thread:0x401bd654>: running)
irb(main):011:0> exit
```

In this same irb session, we'll create a new subsession in the context of class VolumeKnob.

We can use fg 0 to switch back to the main session, take a look at all current jobs, and see what instance methods VolumeKnob defines.

Make a new VolumeKnob object, and create a new subsession with that object as the context.

Switch back to the main session, kill the subsessions, and exit.

But It Doesn't Work!

So, you've read through enough of the book, you start to write your very own Ruby program, and it doesn't work. Here's a list of common gotchas and other tips:

- First and foremost, run your scripts with warnings enabled (the -w command-line option).
- If you happen to forget a comma (,) in an argument list—especially to print—you can produce some very odd error messages.
- A parse error at the last line of the source often indicates a missing end keyword, sometimes quite a bit earlier.
- This ugly message:

  ```
  syntax error, unexpected $end, expecting keyword_end
  ```

  means that you have an end missing somewhere in your code. (The $end in the message means end-of-file, so the message simply means that Ruby hit the end of your code before finding all the end keywords it was expecting.) Try running with -w, which will warn when it finds ends that aren't aligned with their opening if/while/class....
- An attribute setter is not being called. Within a class definition, Ruby will parse setter= as an assignment to a local variable, not as a method call. Use the form self.setter= to indicate the method call:

  ```
  class Incorrect
    attr_accessor :one, :two
    def initialize
      one = 1          # incorrect - sets local variable
      self.two = 2
    end
  end

  obj = Incorrect.new
  obj.one   # =>   nil
  obj.two   # =>   2
  ```
- Objects that don't appear to be properly set up may have been victims of an incorrectly spelled initialize method:

  ```
  class Incorrect
    attr_reader :answer
    def initialise      # <-- spelling error
      @answer = 42
    end
  end

  ultimate = Incorrect.new
  ultimate.answer   # =>   nil
  ```

The same kind of thing can happen if you misspell the instance variable name:

```
class Incorrect
  attr_reader :answer
  def initialize
    @anwser = 42      #<-- spelling error
  end
end

ultimate = Incorrect.new
ultimate.answer   # =>   nil
```

1.9 • As of Ruby 1.9, block parameters are no longer in the same scope as local variables. This may cause compatibility problems with older code. Run with the -w flag to spot these issues:

```
entry = "wibble"
[1, 2, 3].each do |entry|
  # do something with entry
end
puts "Last entry = #{entry}"
```

produces:

```
/tmp/prog.rb:2: warning: shadowing outer local variable - entry
Last entry = wibble
```

• Watch out for precedence issues, especially when using {} instead of do/end:

```
def one(arg)
  if block_given?
    "block given to 'one' returns #{yield}"
  else
    arg
  end
end
def two
  if block_given?
    "block given to 'two' returns #{yield}"
  end
end
result1 = one two {
  "three"
}
result2 = one two do
  "three"
end
puts "With braces, result = #{result1}"
puts "With do/end, result = #{result2}"
```

produces:

```
With braces, result = block given to 'two' returns three
With do/end, result = block given to 'one' returns three
```

- Output written to a terminal may be buffered. This means you may not see a message you write immediately. In addition, if you write messages to both STDOUT and STDERR, the output may not appear in the order you were expecting. Always use nonbuffered I/O (set sync=true) for debug messages.

- If numbers don't come out right, perhaps they're strings. Text read from a file will be a String and will not be automatically converted to a number by Ruby. A call to Integer will work wonders (and will throw an exception if the input isn't a well-formed integer). The following is a common mistake Perl programmers make:

  ```
  while line = gets
    num1, num2 = line.split(/,/)
    # ...
  end
  ```

  You can rewrite this as follows:

  ```
  while line = gets
    num1, num2 = line.split(/,/)
    num1 = Integer(num1)
    num2 = Integer(num2)
    # ...
  end
  ```

  Or, you could convert all the strings using map:

  ```
  while line = gets
    num1, num2 = line.split(/,/).map {|val| Integer(val) }
    # ...
  end
  ```

- Unintended aliasing—if you are using an object as the key of a hash, make sure it doesn't change its hash value (or arrange to call Hash#rehash if it does):

  Download **samples/trouble_10.rb**

  ```
  arr = [1, 2]
  hash = { arr => "value" }
  hash[arr]     # =>   "value"
  arr[0] = 99
  hash[arr]     # =>   nil
  hash.rehash   # =>   {[99, 2]=>"value"}
  hash[arr]     # =>   "value"
  ```

- Make sure the class of the object you are using is what you think it is. If in doubt, use puts my_obj.class.

- Make sure your method names start with a lowercase letter and class and constant names start with an uppercase letter.

- If method calls aren't doing what you'd expect, make sure you've put parentheses around the arguments.

- Make sure the open parenthesis of a method's parameter list butts up against the end of the method name with no intervening spaces.
- Use irb and the debugger.
- Use Object#freeze. If you suspect that some unknown portion of code is setting a variable to a bogus value, try freezing the variable. The culprit will then be caught during the attempt to modify the variable.

One major technique makes writing Ruby code both easier and more fun. *Develop your applications incrementally.* Write a few lines of code, and then write tests (perhaps using Test::Unit). Write a few more lines of code, and then exercise them. One of the major benefits of a dynamically typed language is that things don't have to be complete before you use them.

But It's Too Slow!

Ruby is an interpreted, high-level language, and as such it may not perform as fast as a lower-level language such as C. In the following sections, we'll list some basic things you can do to improve performance; also take a look in the index under *Performance* for other pointers.

Typically, slow-running programs have one or two performance graveyards, places where execution time goes to die. Find and improve them, and suddenly your whole program springs back to life. The trick is finding them. The Benchmark module and the Ruby profilers can help.

Benchmark

You can use the Benchmark module, also described on page 731, to time sections of code. For example, we may wonder what the overhead of method invocation is. How to use Benchmark to find out is shown in Figure 14.2 on the next page.

You have to be careful when benchmarking, because oftentimes Ruby programs can run slowly because of the overhead of garbage collection. Because this garbage collection can happen any time during your program's execution, you may find that benchmarking gives misleading results, showing a section of code running slowly when in fact the slowdown was caused because garbage collection happened to trigger while that code was executing. The Benchmark module has the bmbm method that runs the tests twice, once as a rehearsal and once to measure performance, in an attempt to minimize the distortion introduced by garbage collection. The benchmarking process itself is relatively well mannered—it doesn't slow down your program much.

Figure 14.2. Determining Method Calling Costs Using Benchmark

Download samples/trouble_11.rb

```
require 'benchmark'
include Benchmark

LOOP_COUNT = 1_000_000

bmbm(12) do |test|
  test.report("inline:")    do
    LOOP_COUNT.times do |x|
      # nothing
    end
  end
  test.report("method:") do
    def method
      # nothing
    end
    LOOP_COUNT.times do |x|
      method
    end
  end
end
```

produces:

```
Rehearsal --------------------------------------------------
inline:       0.080000   0.000000   0.080000 (  0.083641)
method:       0.140000   0.000000   0.140000 (  0.137155)
--------------------------------------- total: 0.220000sec

                  user     system      total        real
inline:       0.080000   0.000000   0.080000 (  0.083544)
method:       0.140000   0.000000   0.140000 (  0.136044)
```

The Profiler

Ruby comes with a code profiler (documentation begins on page 792). The profiler shows you the number of times each method in the program is called and the average and cumulative time that Ruby spends in those methods.

You can add profiling to your code using the command-line option -r profile or from within the code using require 'profile'.

For example:

Download **samples/trouble_12.rb**

```
require 'profile'
count = 0
words = File.open("/usr/share/dict/words")
while word = words.gets
  word = word.chomp!
  if word.length == 12
    count += 1
  end
end
puts "#{count} twelve-character words"
```

The first time we ran this (without profiling) against a dictionary of almost 235,000 words, it took a noticeable time to complete. Wondering if we could improve on this, we added the -r profile command-line option and tried again. Eventually we saw output that looked like the following:

```
20460 twelve-character words
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
  0.00     0.00      0.00        1     0.00     0.00  File#initialize
  0.00     0.00      0.00        1     0.00     0.00  IO#open
  0.00     0.00      0.00        2     0.00     0.00  IO#write
  0.00     0.00      0.00        1     0.00     0.00  Fixnum#to_s
  0.00     0.00      0.00        1     0.00     0.00  Kernel.puts
 16.05     1.25      1.25   234936     0.01     0.01  String#chomp!
 20.67     2.86      1.61   234937     0.01     0.01  IO#gets
  0.00     7.79      0.00        1     0.00  7790.00  #toplevell
```

The first thing to notice is that the timings shown are a lot slower than when the program runs without the profiler. Profiling has a serious overhead, but the assumption is that it applies across the board, and therefore the relative numbers are still meaningful. This particular program clearly spends a lot of time in the loop, which executes almost 235,000 times. Each time, it invokes both gets and chomp!. We could probably improve performance if we could either make the stuff in the loop less expensive or eliminate the loop altogether. One way of doing the latter is to read the word list into one long string and then use a pattern to match and extract all twelve character words:

Download **samples/trouble_13.rb**

```
words = File.read("/usr/share/dict/words")
count = words.scan(/^............\n/).size
puts "#{count} twelve-character words"
```

Our profile numbers are now a lot better (and the program runs more than five times faster when we take the profiling back out):

```
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 95.45     0.21      0.21        1   210.00   210.00  String#scan
  4.55     0.22      0.01        1    10.00    10.00  IO#read
  0.00     0.22      0.00        1     0.00     0.00  Fixnum#to_s
  0.00     0.22      0.00        1     0.00     0.00  Array#size
  0.00     0.22      0.00        2     0.00     0.00  IO#write
  0.00     0.22      0.00        1     0.00     0.00  IO#puts
  0.00     0.22      0.00        1     0.00     0.00  Kernel.puts
  0.00     0.22      0.00        1     0.00   220.00  #toplevel
20460 twelve-character words
```

Remember to check the code without the profiler afterward, though—sometimes the slowdown the profiler introduces can mask other problems.

Ruby is a wonderfully transparent and expressive language, but it does not relieve the programmer of the need to apply common sense: creating unnecessary objects, performing unneeded work, and creating bloated code will slow down your programs regardless of the language.

Code Execution Coverage

Ruby 1.9.1 comes with experimental support for code coverage analysis—it will track which lines of code were executed in your code. However, the support is currently labeled as *experimental* and is by default disabled in the VM. By the time you read this, it may have settled down. Try doing this:

```
$ ri coverage
```

Of course, there's a good chance it may never make it to production. In that case, feel free to cut out this section of your book and use the words in your next writing project.

Table 14.1. Debugger Commands

b [reak] [file|class:]line	Sets breakpoint at given line in *file* (default current file) or *class*.
b [reak] [file|class:]name	Sets breakpoint at *method* in *file* or *class*.
b [reak]	Displays breakpoints and watchpoints.
wat [ch] expr	Breaks when expression becomes true.
del [ete] [nnn]	Deletes breakpoint *nnn* (default all).
cat [ch] exception	Stops when *exception* is raised.
cat [ch]	Lists current catches.
tr [ace] (on|off) [all]	Toggles execution trace of current or all threads.
disp [lay] expr	Displays value of *nnn* every time debugger gets control.
disp [lay]	Shows current displays.
undisp [lay] [nnn]	Removes display (default all).
c [ont]	Continues execution.
s [tep] nnn=1	Executes next *nnn* lines, stepping into methods.
n [ext] nnn=1	Executes next *nnn* lines, stepping over methods.
fin [ish]	Finishes execution of the current function.
q [uit]	Exits the debugger.
w [here]	Displays current stack frame.
f [rame]	Synonym for where.
l [ist] [start–end]	Lists source lines from start to end.
up nnn=1	Moves up *nnn* levels in the stack frame.
down nnn=1	Moves down *nnn* levels in the stack frame.
v [ar] g [lobal]	Displays global variables.
v [ar] l [ocal]	Displays local variables.
v [ar] i [stance] *obj*	Displays instance variables of *obj*.
v [ar] c [onst] Name	Displays constants in class or module name.
m [ethod] i [nstance] *obj*	Displays instance methods of *obj*.
m [ethod] Name	Displays instance methods of the class or module name.
th [read] l [ist]	Lists all threads.
th [read] [c[ur[rent]]]	Displays status of current thread.
th [read] [c[ur[rent]]] nnn	Makes thread *nnn* current and stops it.
th [read] stop nnn	Makes thread *nnn* current and stops it.
th [read] resume nnn	Resumes thread *nnn*.
th [read] [sw[itch]] nnn	Switches thread context to nnn.
[p] expr	Evaluates *expr* in the current context. *expr* may include assignment to variables and method invocations.
h[elp]	Shows summary of commands.
empty	A null command repeats the last command.