Debugging

Debugging is really at the core of what we do as programmers. While the glory and marketing goes to shipping a finished application, the fact is that day-to-day life of a programmer is being stuck on some problem. Programmers are able to get through the trivial code rather quickly, so the majority of the time is spent on analyzing and understanding a problem, experimenting or coming up with an approach, or debugging bugs in the code.

Note that we'll talk about the general act of debugging, and not working with a specific *debugger* tool here.

Temperament

When thinking about what attributes make a good programmer, one thing comes to mind: temperament. If the key to programming is debugging, then the key to debugging is having a patient and logical temperament. Think Spock from Star Trek. Some people naturally possess this temperament, and that will be an advantage when learning to program. Some people do not, and they will need to learn to develop this temperament. Being naturally intelligent is very helpful, but from our experience, most people have enough intellectual ability to become an application developer, but what they need to improve is developing a systematic, patient temperament when faced with a problem.

Whenever debugging is required, that implies something is broken. When things break, it doesn't make you feel good. Programming is dealing with a constant stream of broken things and learning to deal with those ill feelings. Therefore, it's important to gauge how you normally respond to problems. For example, if you're walking to the bus stop and the bus leaves right before you get there, what's your natural reaction? Someone who has the programmer temperament should figure out 1) when the next one comes, or 2) if there's an alternative path of transfers you can take, or 3) other alternative forms of transportation. If your reaction is a sinking feeling and frustration, you'll have to learn to deal with that, then transition to a systematic search for a solution. Fortunately, this can be learned, but remember to keep even-keeled and be systematic.

Dealing with feelings of frustration is the first critical aspect of learning to program.

Reading Error Messages

When you run into an error, you'll likely be hit with a wall of text that looks like gibberish. This wall of gibberish is called the **stack trace** and is critical in helping you figure out where to start your debugging. One of the first things you'll have to get used to is learning how to *carefully* read the error stack trace. Embedded within the stack trace is the error message. The error message will include the first hint at where to start looking.

For example, click the button to see an example stack trace. Stack Trace

That's a real stack trace from a real project. Can you spot the error message? It's near the top: `NoMethodError: undefined method 'name' for nil:NilClass`. The trick is to train your eye to look for the relevant parts of the stack trace, and over time, you'll be able to spot the error faster and faster. Every language and library has a certain pattern to their stack trace, so the longer you work with a language or library, the easier it will become. When you see a large stack trace for the first time, don't despair. Study it carefully, and try to extract the meaningful bits.

Online Resources

1. Search Engine

   Now, once you've identified the error and have extracted the error message, it's time to take action. Study the error message, and try to walk backwards through the code to understand how the flow of the program arrived at the error condition. Think about the various data that is being used at the point of the error, and how missing or incorrect data could have caused the problem. Finally, use a search engine to look up the error message.

   The whole error message is `NoMethodError: undefined method 'name' for nil:NilClass` and it's probably ok to search for that entire phrase. Make sure to not include terms that are specific to your computer or program when searching. For example, don't include `/Users/jim/.rvm` in your search term, because that's specific to your machine. You could also eliminate the `'name'` part of the error message because that's your program's method name. Search results will likely pick up generic search terms, so make sure you don't simply copy and paste the entire stack trace.

   Finally, if you see a lot of results in a different programming language, you may want to preface "Ruby" in front of the search phrase. Sometimes, different languages will use similar names for common errors, so it may be necessary to filter out languages you're not interested in.

2. Stack Overflow

   Stack Overflow is a rich treasure trove of answers to common problems. Many of their answers rank highly in search engines, but sometimes it's worth searching on SO directly.

3. Documentation

   Finally, don't hesitate to consult the official Ruby documentation site: Ruby Docs. Make sure you're looking at the documentation for pure Ruby, and not a library or framework specific documentation. That's very important, as some frameworks (i.e., Rails) add a lot of functionality to core classes that's not available in pure Ruby.

Steps to Debugging

The debugging process varies tremendously from person to person, but below are some steps you can follow early on until you've started to hone your own habits.

1. Reproduce the Error

   The first step in debugging any problem is usually reproducing the problem. Programmers need a deterministic way to consistently reproduce the problem, and only then can we start to isolate the root cause. There's an old joke where programmers will say "works on my machine" because they can't reproduce an error that occurs in the production environment. This will become more important as you build more sophisticated applications with various external dependencies and moving parts. Reproducing the exact error will often end up being more than half the battle in many tricky situations.

2. Determine the Boundaries of the Error

   Once you can consistently reproduce the problem, it's time to tweak the data that caused the error. For example, the stack trace earlier was generated by this code `post.categories << news` . Does calling `post.categories` cause issues? What about just calling `post` ? What happens if we try to append a different object, like this: `post.categories << sports` ? How does modifying the data affect the program behavior? Do we get expected errors, or does a new error occur that sheds light on the underlying problem?

   What we're trying to do is modify the data or code to get an idea of the scope of the error and determine the boundaries of the problem. This will lead to a deeper understanding of the problem, and allow us to implement a better solution. Most problems can be solved in many ways, and the deeper you understand the problem, the more holistic solution you can come up with.

3. Trace the Code

   Once you understand the boundaries of the problem, it's time to trace the code. Let's use a new example.

```ruby
def car(new_car)
  make = make(new_car)
  model = model(new_car)
  [make, model]
end

def make(new_car)
  new_car.split(" ")[0]
end

def model(new_car)
  new_car.split(" ")[2]
```

```ruby
end

make, model = car("Ford Mustang")
make == "Ford"          # => true
model.start_with?("M") # => NoMethodError: undefined
```

This code is fairly straightforward. One aspect of it that's a bit tricky is the return value of the `car` method and the assignment from that method to local variables `make` and `model`. When an array is assigned to two variables on the same line, each element of that array gets assigned to one of the variables. In the example above, the first array element gets assigned to `make` and the second array element gets assigned to `model`. This type of assignment, where we assign more than one value on the same line, is called "multiple assignment".

When we try to see if `model` starts with the character `"M"`, we get an error. After reproducing the problem consistently and testing various data inputs, you notice that `model` always returns `nil`. In this example, `make` is expected to be `"Ford"` and `model` is expected to be `"Mustang"`. It looks like we've got a bug here.

Let's trace the code backwards. When you first call `car`, a string is passed in as an argument. The string represented by the local variable `new_car` is passed into two helper method: `make` and `model`. Inside each of these methods, the intention is to split `new_car` into two new strings: `"Ford"` and `"Mustang"`. The `make` method should return `"Ford"` and the `model` method should return `"Mustang"`. In this case, the `make` method returns the correct value but the `model` method does not. Based on these observations, we know that the bug in this code originates from the `model` method. This is called *trapping the error*.

4. Understand the Problem Well

After narrowing the source of the bug to the `model` method, it's time to break down the code within the method. We know that the return value of this method is always `nil` , so let's inspect each return value in order to pinpoint the source of the unexpected return value.

```ruby
def model(new_car)
  new_car # => "Ford Mustang"
end
```

That's the expected return value of `new_car` . No issues so far.

```ruby
def model(new_car)
  new_car.split(" ") # => ["Ford", "Mustang"]
end
```

The return value here is an array, which is expected based on our knowledge of how `#split` works.

```ruby
def model(new_car)
  new_car.split(" ")[2] # => nil
end
```

Aha! It looks like the unexpected return value here is the result of calling `[2]` on the `["Ford", "Mustang"]` array. The return value is `nil` because there is no element at index `2` in this array. Since arrays have a zero-based index, we need to call `[1]` in order to return `"Mustang"` from the array.

```ruby
def model(new_car)
  new_car.split(" ")[1] # => "Mustang"
end
```

5. Implement a Fix

There are often multiple ways and multiple layers in which you can make the fix. For example, we could suppress the error from being thrown with this code:

```
model.start_with?("M") rescue false # => false
```

We'll still have the original error in the `model` method, though. In some cases, you'll be using a library or code that you can't modify. In those situations, you have no choice but to deal with edge cases in your own code. In this example, we should fix the offending code in the `model` method.

One very important note is to fix *one problem at a time*. It's common to notice additional edge cases or problems as you're implementing a fix. Resist the urge to fix multiple problems at once.

> You'll almost never want to use the trailing `rescue` like we did in the above example. It's usually a **code smell** that you haven't thought carefully about the possible problems that could go wrong, and therefore you haven't thought about how to handle the potential error conditions. Also, by not specifying any particular error to rescue, you're suppressing all possible errors, including potentially very destructive ones that may impact your program in unexpected ways.

6. Test the Fix

   Finally, after implementing a fix, make sure to verify that the code fixed the problem by using a similar set of tests from step #2. After you learn about automated testing, you'll want to add an additional automated test to prevent regression. For now, you can test manually.

Techniques for Debugging

1. Line by line

Your best debugging tool is your patience, which is why we mentioned temperament first in this article. Most bugs in your code will be from overlooking a detail, rather than a complete misunderstanding of a concept. Being careful, patient and developing a habit of reading code line-by-line, word-by-word, character-by-character is your most powerful ability as a programmer. If you are naturally impatient or like to gloss over details, you must train yourself to behave differently when programming. All other debugging tips and tools won't matter if you aren't detail oriented.

2. Rubber Duck

   Rubber Duck Debugging sounds crazy, but its effectiveness is so well known that it has its own Wikipedia page. The process centers around using some object, like a rubber duck, as a sounding board when faced with a hard problem. The idea is that when you are forced to explain the problem to a rubber duck, you are actually forcing yourself to articulate the problem, detail by detail. This often leads to discovering the root of the problem. Of course, the object doesn't have to be a rubber duck -- it can be anything, including another human being. The point is to focus your mind on walking through the code, line by line, and in that process, noticing a small detail that may reveal a deeper problem. If you've never experienced this phenomenon, we encourage you to try this out yourself next time you're stuck on a bug.

3. Walking Away

   Another interesting debugging technique is to go for a walk. Some have claimed a swim, jog or even a shower helps too. Some claim this technique works because it activates a different part of your brain, and even when you walk away from the computer, your brain is still working on the problem in the background. Note that this is only effective *if you first spend time loading the problem in your brain*. You cannot just walk away when you first encounter a problem and expect this technique to work. Once you've loaded the problem in your brain, your brain may work on the problem even while you're sleeping! Also, our brains get tired too, so it's ok to step away and come back to it with fresh eyes and a refreshed brain.

4. Using Pry

Pry is a powerful Ruby REPL that can replace IRB. We don't recommend that you do that just yet, and just use Pry for simple debugging. To use it, first install it like any other gem.

```
$ gem install pry
```

In order to use Pry, we have to require it using `require "pry"` . Once we've required Pry, we can then insert `binding.pry` anywhere in our code, and when Ruby gets to that line, execution will stop and we'll be able to inspect the state of our program at that point.

count.rb

```ruby
require "pry" # add this to use Pry

counter = 0

loop do
  counter += 1
  break if counter == 1
  binding.pry
end
```

Suppose the above code is saved into a file called `count.rb` . If we run the file, we should see the following:

```
$ ruby count.rb
$
```

What happened? We expected Pry to stop execution on line 8, but that doesn't seem to be the case. Looking back at the code, we can see that the `break` condition evaluates to `true` if `counter` equal `1` . Since we increment `counter` before this condition, `counter` is indeed `1` by the time `break` is executed. This means that Ruby broke out of the loop before reaching line 8; `binding.pry` was never executed.

Let's change the condition to read `counter == 5` and move `binding.pry` to the line *before* the `break` condition.

count.rb

```ruby
require "pry" # add this to use Pry

counter = 0

loop do
  counter += 1
  binding.pry # execution will stop here
  break if counter == 5
end
```

If we run `count.rb` again, we should see the following:

```
$ ruby count.rb

From: /Users/temp/count.rb @ line 7 :

    2:
    3: counter = 0
    4:
    5: loop do
    6:   counter += 1
 => 7:   binding.pry # execution will stop here
    8:   break if counter == 5
    9: end

[1] pry(main)> counter
=> 1
[2] pry(main)>
```

This time, Pry stops execution at the line where `binding.pry` is declared and gives us a prompt where we can type in an expression, such as `counter` , and see what the return value is. We could also change variable values if we wanted to. This is an incredibly helpful way to systematically debug our program, without spraying our entire program full of "puts".

In order to continue execution of the program, press `Ctrl + D`. Since we're in a loop, the loop will continue to iterate until `counter` equals `5`. This means that `binding.pry` will execute on every iteration until we break out of the loop. We could press `Ctrl + D` every time Pry opens a session, but that could be tiresome if there are a lot of iterations. Alternatively, we can exit the program by entering `exit-program`.

```
$ ruby count.rb

From: /Users/temp/count.rb @ line 7 :

    2:
    3: counter = 0
    4:
    5: loop do
    6:   counter += 1
 => 7:   binding.pry # execution will stop here
    8:   break if counter == 5
    9: end

[1] pry(main)> exit-program
$
```

5. Using a Debugger

   Pry can also be used as a real debugger, which gives us the mechanism to step into functions and more systematically walk over code. We won't be covering that yet, so for now we just want to introduce the concept of a **debugger** program. It's a little more advanced, so we encourage you to just use Pry in the manner we showed above.

Summary

In summary, debugging is arguably the most important skill you need to learn as a programmer. Focus on developing a patient, systematic temperament; carefully read error messages; use all the wonderful resources at your disposal; approach debugging in sequential steps; and use the techniques we covered above -- especially Pry. If you haven't yet, go install Pry now and play around with it a little bit.