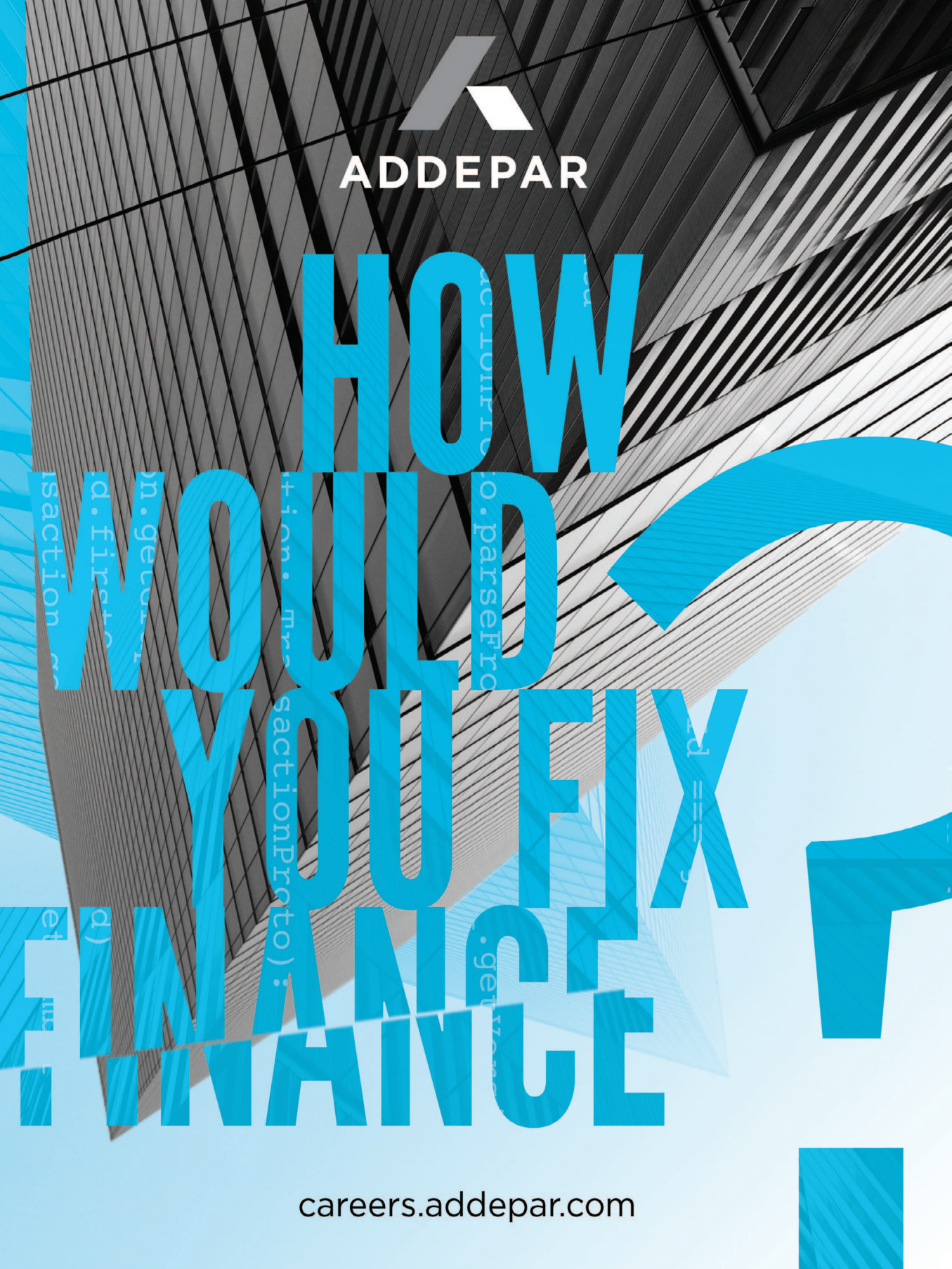
HACKERMONTHLY

Issue 30
November 2012

What If Hemingway Wrote JavaScript?

Angus Croll


ADDEPAR
HOW WOULD YOU FIX FINANCE?
careers.addepar.com

Now you can hack on DuckDuckGo

DuckDuckHack

Create instant answer plugins for DuckDuckGo

duckduckhack.com

Curator
Lim Cheng Soon

Contributors
Sau Sheong Chang
Angus Croll
Dave McClure
Matt Swanson
Boris Wertz
Joshua Gross
Noah Sussman
Eli Bendersky
Chris Eppstein
Alan O'Donnell
Nicholas C. Zakas
Dan Schultz
Alex Hillman

Proofreaders
Emily Griffin
Sigmarie Soto

Printer
MagCloud

HACKER MONTHLY is the print magazine version of Hacker News — *news.ycombinator.com*, a social news website wildly popular among programmers and startup founders. The submission guidelines state that content can be "anything that gratifies one's intellectual curiosity." Every month, we select from the top voted articles on Hacker News and print them in magazine format.
For more, visit *hackermonthly.com*

Advertising
ads@hackermonthly.com

Contact
contact@hackermonthly.com

Published by
Netizens Media
46, Taylor Road,
11600 Penang,
Malaysia.

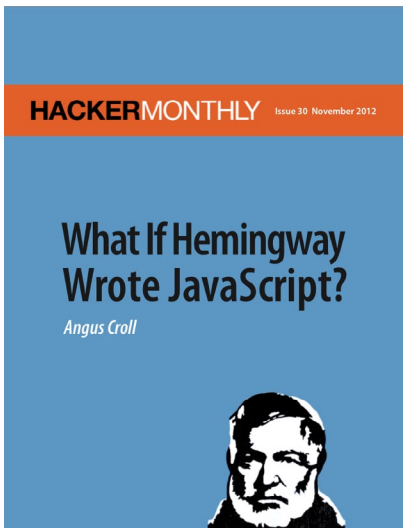


Illustration: Kevin O'Brien

Contents

For links to Hacker News dicussions, visit *hackermonthly.com/issue-30*

Getting Your Heart Rate Using R and Ruby

By SAU SHEONG CHANG

The heart rate, or the rate at which your heart beats, is one of the measurements you've probably heard most about in relation to exercise. It's also often a good indication of your health, because a heart rate that is too high or low could indicate an underlying health issue. The heart rate is usually measured in beats per minute (bpm) and varies from 40 to 220 bpm. An average healthy person at rest has a heart rate of 60–90 bpm, while conditioned athletes have a resting heart rate of 40–60 bpm.

A popular and fast way to effectively get the heart rate is pulse oximetry. A pulse oximeter is a device placed on a thin part of a person's body, often a fingertip or earlobe. Light of different wavelengths (usually red and infrared) is then passed through that part of the body to a photodetector. The oximeter works by measuring the amounts of red and infrared light absorbed by the hemoglobin and oxyhemoglobin in the blood to determine how oxygenated the blood is. Because this absorption happens in pulses as the heart pumps oxygenated blood throughout the body, the heart rate can also be determined.

We are not going to build an oximeter, but in this post we'll use the same concepts used in oximetry to determine the heart rate. We will record a video as we pass light through our finger for a short duration of time. With each beat of the heart, more or less blood flows through our body,

including our finger. The blood flowing through our finger will block different amounts of the light accordingly. If we calculate the light intensity of each frame of the video we captured, we can chart the amount of blood flowing through our finger at different points in time, therefore getting the heart rate.

Homemade Pulse Oximeter

Creating a homemade oximeter is really simple. You can use any of the following techniques, or even try your own methods. It doesn't really matter, as long as you can capture the video. Record for about 30 seconds. (Recording for a longer time can be more accurate, but not significantly so.)

Finger on a webcam

Place your finger directly on your computer's webcam (I used the iSight on my Mac). Shine a small light (penlight or table lamp; it doesn't matter much) through your finger. Then use any video recording software to record what's on the webcam (I used QuickTime video recording).

Finger on the phone camera

Place your finger directly on your phone camera. Turn on the flash or use a small light and shine it through your finger. Then use your phone's video recording software to record what's on the phone camera.

Finger on a digital video camera

This is slightly harder because the camera lens is normally larger than your finger. The parts that aren't covered don't really matter, but you need to position your finger so that the image captured is consistent throughout your recording. A trick is to use a lamp as the background, so you can have the light shining through your finger and maintain a consistent background at the same time.

In the following example, I used the phone camera method with my iPhone. That's the easiest for me, because the flash on the phone is very effective. If you did things right, you'll end up with a video filled with a red blotch that's your finger.

Extracting Data from Video

Assuming that you have a nice video file now (it doesn't really matter what format it is in; you'll see why soon), let's dig in a bit deeper to see how we can extract information from it. For the sake of convenience, I'll assume the file is called `heartbeat.mov`. Next we'll be using `FFmpeg`, a popular free video library and utility, to convert the video into a series of individual image files.

Let's take a look at some Ruby code.

```
require 'csv'
  require 'rmagick'
  require 'active_support/all'
  require 'rvideo'
  vid = RVideo::Inspector.new(:file => "heartbeat.mov")
  width, height = vid.width, vid.height
  fps = vid.fps.to_i
  duration = vid.duration/1000
  if system("/opt/local/bin/ffmpeg -i heartbeat.mov -f image2
'frames/frame%03d.png'")
    CSV.open("data.csv","w") do |file|
      file << %w(frame intensity)
      (fps*duration).times do |n|
        img = Magick::ImageList.new("frames/
frame#{sprintf("%03d",n+1)}.png")
          ch = img.channel(Magick::RedChannel) i= 0
  ch.each_pixel {|pix| i += pix.intensity} file << [n+1, i/
(height*width)]
  end end
  end
```

It doesn't look complicated, does it? The most complex part you'll probably have to tackle is installing the necessary Ruby libraries. In the case of both `RMagick` and `RVideo`, described next, you need native developer tools support in order to compile the native components of the gem for your platform.

We start off the code by inspecting the video and getting some attributes from it. These will be useful later on in the code. Specifically, we will need the number of frames per second, the duration of the video, and the height and width of the video. You can obtain these through `RVideo`, but if you didn't succeed in getting it installed, you can still find the information by simply opening up the video with any player and viewing its properties.

Next, we use the system method to issue a command to the underlying shell, and return either true or false depending on whether it succeeds or not:

```
system("/opt/local/bin/ffmpeg -i
heartbeat.mov -f image2 'frames/
frame%03d.png'")
```

This runs `ffmpeg`, taking in the input file `heartbeat.mov` and converting it frame by frame into a set of images ordered by number. This is the reason why the video format is unimportant. As long as `FFmpeg` has the correct library to support the codecs, it will convert the video file to a series of PNG image files, numbered sequentially.

In this example, we specify that there are three digits to this series of numbers. How do we know this? In my case, I have a 30-second video with a frame rate of 30 frames per second, so the number of still frames that will be created by `FFmpeg` is 30×30, or 900 frames. Slightly more frames could be created — some video players round off the duration — but the total would not be more than 999 frames. If the command runs successfully, we will get a set of frames in the frames folder, each named `framennn.png`, where `nnn` runs from 001 to 900 or so.

Next, we create a CSV file to store the data and enter the column names, which are the frame number and the average frame intensity:

```
file << %w(frame intensity)
```

Then, for every frame image, we create the `RMagick` Image object that represents that frame and extract the red channel (the file uses the RGB colorspace):

```
ch = img.
channel(Magick::RedChannel)
```

We iterate through each pixel in the red channel and add up their intensities, then divide the sum of pixel intensities by the total number of pixels:

```
i= 0
ch.each_pixel {|pix| i += pix.
intensity}
file << [n+1, i/(height*width)]
```

This is the value we consider to be the average frame intensity. Finally, we store the frame number and intensity in the CSV file.

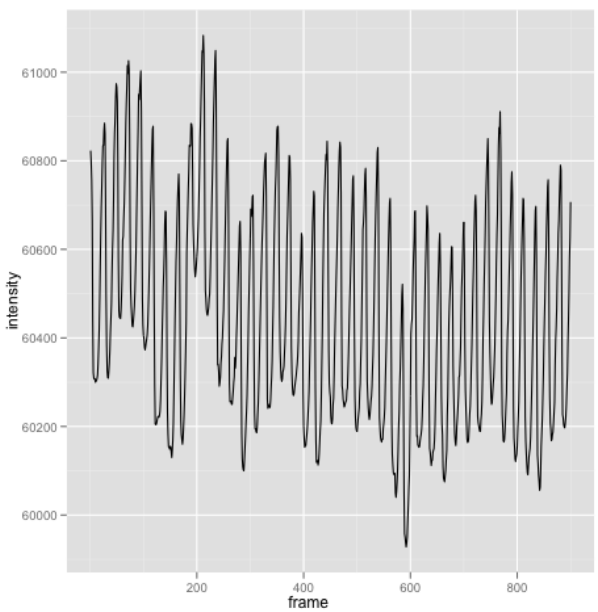
Once we have done this, we will end up with a data file with two columns. The first is the frame number, and the second is the corresponding frame's average intensity.

Generating the Heartbeat Waveform and Calculating the Heart Rate

Generating the heartbeat waveform is trivial, so we'll combine both creating the waveform and calculating the heart rate into a single R script.

```
library(PROcess)
  library(ggplot2)
  data <- read.csv(file='data.csv', header=T)
  png("heartbeat.png")
  qplot(data=data, frame, intensity, geom="line")
  dev.off()
  peaks <- peaks(data$intensity,span=10)
  peak_times <- which(peaks==T, arr.in=T)
  intervals <- c()
  i <- 1
  while (i < length(peak_times)) {
    intervals <- append(intervals, peak_times[i+1] - peak_times[i])
    i <- i + 1
  }
  average <- round(mean(intervals))
  print(paste("Average interval between peak intensities is", average))
  heartbeat_rate <- round(60 * (30/average))
  print(paste("Heartbeat rate is",heartbeat_rate))
```

All it takes to generate the waveform is a single line that calls `qplot` with the frame and the intensity and uses the line `geom`.



As you can see from the chart, the light intensity changes over time. Each pulse corresponds with a heartbeat. To find the heart rate, we need to find the number of frames between two peaks of the wave. We know that there are 30 frames in one second. Once we know the number of frames between the two peaks, we'll know how much time it takes to go from peak to peak, and therefore can calculate the number of beats per minute.

To calculate the distance from peak to peak, we need to first determine where the peaks are in the chart. For this, we will be using an R package that was originally designed to process protein mass spectrometry data, found in the `Bioconductor` library.

The `Bioconductor` library is a free/open source project that provides tools for analyzing genomic data. It's based primarily on R, and most of the `Bioconductor` components are R packages. The package we will be using is called `PROcess`. Once we include the library in our script, we can start using the `peaks()` function, which, true to its name, determines which values are peaks in data.

The input parameter to the `peaks()` function is the intensity data and a span value. This span value determines how many of its neighboring values it must exceed before it can be considered a peak. This is useful to filter off noise, though not perfectly.

The returned result is a logical vector that is the same length as the data. This means we have a vector of TRUEs and FALSEs, where the TRUEs indicate a peak:

```
[1] FALSE FALSE FALSE FALSE FALSE
FALSE FALSE FALSE FALSE FALSE
FALSE FALSE
     [13] FALSE FALSE FALSE FALSE
FALSE FALSE FALSE FALSE FALSE
FALSE FALSE FALSE
     [25] FALSE FALSE FALSE  TRUE
FALSE FALSE FALSE FALSE FALSE
FALSE FALSE FALSE
```

While this vector is informative, it's not really the answer we want, so we pass it through the `which()` function, and it returns a vector of the indices where the element is TRUE:

```
     [1]  28  50  73  96 119 142
167 190 213 236 259 282 306 330
353 374 397 420 445
    [20] 469 494 517 540 563 586
610 632 656 678 701 723 746 769
791 812 836 859 882
```

As before, we want to find the distance between the two peaks, so we take two consecutive elements and subtract the first from the second. This gives us a new vector that contains the differences:

```
[1] 22 23 23 23 23 25 23 23 23 23
23 24 24 23 21 23 23 25 24 25 23
23 23 23 24
    [26] 22 24 22 23 22 23 23 22
21 24 23 23
```

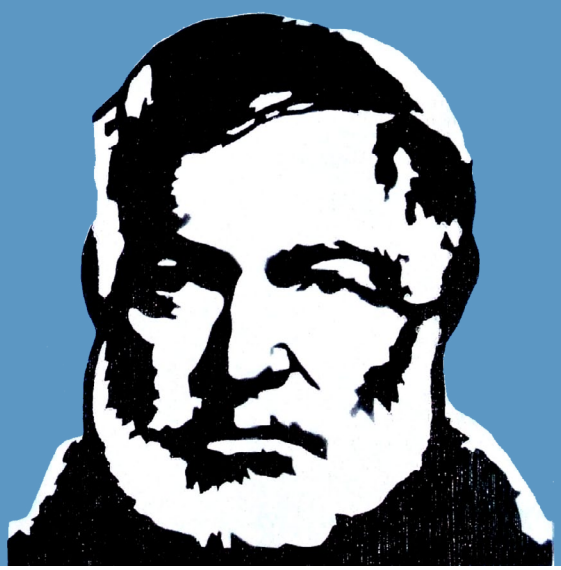
The final two steps are the same as in the previous section. First, we find the average distance using the `mean()` function. Then, from that, we know that there are 23 frames between two peaks, meaning each heartbeat takes 23 frames or 23/30 seconds (since each second has 30 frames). From that, we calculate that the heart rate is 78 bpm. ■

Sau Sheong is the Director of HP Labs in Singapore, and manages a team of engineers and research scientists focusing on research on how people use cloud computing. He is also an active programmer, technology enthusiast and a frequent speaker at technology-related conferences. Sau Sheong has written 3 books on Ruby and the latest is "Exploring Everyday Things with R and Ruby" published by O'Reilly Media.

Reprinted with permission of the original author.
First appeared in *hn.my/everydayruby* (airbrake.io)

If Hemingway Wrote JavaScript

By ANGUS CROLL



I LOVED LITERATURE LONG before I ever wrote a line of code. Now I write JavaScript — lots of it — and I'm writing a book about it.

What is it about JavaScript that attracts so many literature devotees? I have a few half-baked theories relating to the expressive potential of a limited syntax, but that's for another time. What about the great writers? What would they have made of JavaScript?

Even as a long-time Hemingway nut, I'd be the first to admit that Papa would probably have loathed programming (and programmers). Yet I'm betting that amongst all that general contempt there would have lurked a soft spot for JavaScript, because it's his kind of language, am I right? A spare and deceptively plain surface, masking substance and drama beneath.

The Mother of All Code Reviews

Recently, I had a dream in which I asked Hemingway and four other literary luminaries to write some JavaScript for me; specifically a function that returned a fibonacci series of a given length. Interestingly each author chose to solve the problem in a different way. They did pretty well actually — as far as I can tell, every solution works as advertised (yes, even Andre Breton's). Here's what I got:

Ernest Hemingway

```
function fibonacci(size) {

  var first = 0, second = 1, next, count = 2, result = [first, second];

  if(size < 2)
    return "the request was made but it was not good"

  while(count++ < size) {
    next = first + second;
    first = second;
    second = next;
    result.push(next);
  }
  return result;
}
```

No surprises here. Code reduced to its essentials with no word or variable wasted. It's not fancy; maybe it's even a little pedantic, but that's the beauty of Hemingway's writing. No need for elaborate logic or clever variable names. It's plain and it's clear and it does what it has to — and nothing more.

Hemingway didn't suffer fools gladly, so if you ask for a series with less than two numbers he'll just ignore you. "I'm tired and this question is idiotic."

William Shakespeare

```
function theSeriesOfFIBONACCI(theSize) {

  //a CALCKULATION in two acts.
  //employ'ng the humourous logick of JAVA-SCRIPTE

  //Dramatis Personae
  var theResult; //an ARRAY to contain THE NUMBERS
  var theCounter; //a NUMBER, serv'nt to the FOR LOOP

  //ACT I: in which a ZERO is added for INITIATION
  //[ENTER: theResult]
  //Upon the noble list bestow a zero
  var theResult = [0];
  //ACT II: a LOOP in which the final TWO NUMBERS
  //are QUEREED and SUMM'D
  //[ENTER: theCounter]
  //Commence at one and venture o'er the numbers
  for (theCounter = 1; theCounter < theSize; theCounter++) {
    //By divination set adjoining members
    theResult[theCounter] = (theResult[theCounter-1]||1) +
theResult[Math.max(0, theCounter-2)];
  }
  //'Tis done, and here's the answer.
  return theResult;
  //[Exeunt]
}
```

The Bard gets a little wordy here but we wouldn't have it any other way. Notice how the comments (other than titular captions and stage directions) are written in iambic pentameter — a meter of ten paired syllables with the stress falling on the second syllable of each pair (or *foot*). In his plays, Shakespeare often adds dramatic emphasis by deviating from strict iambic pentameter — he might add an extra syllable or use an alternate stress. It looks as though he's using the same trick in this coding exercise. Way to go, Will.

Andre Breton

```
function Colette(umbrella) {
  var staircase = 0, galleons = 0,
brigantines = 1, armada = [galleons, brigantines], bassoon;
  Array.prototype.embrace = [].push;
  while(2 + staircase++ < umbrella) {
    bassoon = galleons + brigantines;
    armada.embrace(brigantines = (galleons = brigantines, bassoon));
  }
  return armada;
}
```

As a founding member of the surrealist movement, Breton believed dreams were more interesting than reality and should form the basis of our creative endeavors. Nouns are chosen accordingly. Although it's easy to knock Breton, his work has aged well and is invariably heartfelt and beautiful — an unconscious upwelling of imagery folded into his own conscious expression. Here's a translation of the gorgeous poem Facteur Cheval [hn.my/cheval].

Breton has most likely named his fibonacci exercise after an old flame, while he imagines the resulting collection as a fleet of ancient vessels. The solution is underscored by characteristically elegant logic — he's using a comma operator to simultaneously shift elements between galleons, brigantines and bassoons. Hats off, Andre!

Roberto Bolano

```
function LeonardoPisanoBigollo(l) {
  if(l < 0) {
    return "I'd prefer not to respond. (Although several replies occur
to me)"
  }
  /**/

  //Everything is getting complicated.
  for (var i=2,r=[0,1].slice(0,l);i<l;r.push(r[i-1]+r[i-2]),i++)
  /**/
```

```
  //Here are some other mathematicians. Mostly
  //it's just nonsense.

  rationalTheorists = ["Archimedes of Syracuse", "Pierre de Fermat
(such margins, boys!)", "Srinivasa Ramanujan", "Rene Descartes",
"Leonhard Euler", "Carl Gauss", "Johann Bernoulli", "Jacob Bernoulli",
"Aryabhata", "Brahmagupta", "Bhaskara II", "Nilakantha Somayaji",
"Omar Khayyám", "Muhammad ibn Mūsā al-Khwārizmī", "Bernhard Riemann",
"Gottfried Leibniz", "Andrey Kolmogorov", "Euclid of Alexandria",
"Jules Henri Poincaré", "Srinivasa Ramanujan", "Alexander Grothendieck
(who could forget?)", "David Hilbert", "Alan Turing", "von Neumann",
"Kurt Gödel", "Joseph-Louis Lagrange", "Georg Cantor", "William Rowan
Hamilton", "Carl Jacobi", "Évariste Galois", "Nikolay Lobachevsky",
"Rene Descartes", "Joseph Fourier", "Pierre-Simon Laplace", "Alonzo
Church", "Nikolay Bogolyubov"]
  /**/
  //I didn't understand any of this, but here it
  //is anyway.
  return r
  /**/
  //Nothing happens here and if it does I'd
  //rather not talk about it.
}
```

If you don't read at least one Bolano book before you die then you've wasted your life. Bolano's writing is remarkable; at once effortlessly sophisticated and charmingly naive, his narrative style is characterized by a disarmingly winsome honesty. No aspect of human frailty is off limits, but the warmth and humor with which every foible is conveyed is both engaging and uplifting.

True to form, Roberto's exam paper is peppered with admissions of insecurity, embarrassment and ignorance. The solution, though rather brilliant, is presented as something of an afterthought. Always the obsessive, always tangential, he's much happier offering us a mildly interesting but ultimately useless list of mathematical genii.

There are other Bolano traits here — the juxtaposition of long and short paragraphs, the absence of semicolons (mirroring the absence of quotation marks in his novels), and the use of implicit globals that suggest each variable is destined to make further appearances in subsequent chapters.

Charles Dickens

```
function mrFibbowicksNumbers(enormity) {
  var assortment = [0,1,1], tally = 3, artfulRatio = 1.61803;

  while(tally++ < enormity) {
    //here is an exceedingly clever device
    assortment.push(Math.round(assortment[tally-2] * artfulRatio));
  }
  //should there be an overabundance of
  //elements, a remedy need be applied
  return assortment.slice(0, enormity);
}
```

I'm not a fan of Dickens. Mostly I agree with Henry James' damning assessment:

> *"If we might hazard a definition of his literary character, we should, accordingly, call him the greatest of superficial novelists. We are aware that this definition confines him to an inferior rank in the department of letters which he adorns; but we accept this consequence of our proposition. It were, in our opinion, an offense against humanity to place Mr. Dickens among the greatest novelists. For, to repeat what we have already intimated, he has created nothing but figure. He has added nothing to our understanding of human character."*
> *– Henry James on Charles Dickens, in a review of Our Mutual Friend, in The Nation (December 21, 1865).*

Boz's superficiality is borne out by his fibonacci solution. Yes, there are some mildly amusing names, but a complete lack of substance and understanding at its heart. He has failed to appreciate the underlying philosophy of the fibonacci series and has instead resorted to bludgeoning his way through the problem with multiplication. Sigh.

Closing Thoughts

Whether it's Crockford's protective albumen or the dry and narrow minded confines of computer science classes, doctrine and dogma are the enemies of good JavaScript. Some developers like rulebooks and boilerplate, which is why we have Java. The joy of JavaScript is rooted in its lack of rigidity and the infinite possibilities that this allows for. Natural languages hold the same promise. The best authors and the best JavaScript developers are those who obsess about language, who explore and experiment with language every day, and in doing so, develop their own style, their own idioms, and their own expression.

That's all. Hope you enjoyed it. It's mostly nonsense. ■

Angus Croll is a literature junkie and front end developer on the twitter web core team and author of the JavaScript JavaScript blog [javascriptweblog.wordpress.com]. He's writing an advanced JavaScript book for No Starch Press (for release in 2013) and is a regular conference speaker.

Reprinted with permission of the original author.
First appeared in *hn.my/hemingway* (byfat.xxx)

Illustration by Kevin O'Brien [poeticoddity.deviantart.com]

Are you a publisher?

Do you want to publish your Magazines, Comics or Kids books on iPad or Android tablets?

We have the right solutions for you.

kiurma

kiurma.com/products_magazine

Late Bloomer, Not A Loser

By DAVE MCCLURE

Most of the time I think of myself as a failure. When I'm optimistic, I think maybe I'm just a late bloomer.

I know a lot of folks won't understand this perspective, but when I was growing up, I was always the smartest kid around. It was expected that I would do great things, by my mom, by my teachers, and most importantly, by me. I don't know whether that's a good thing or bad thing, but high expectations were always around me, and for the first 10–15 years, the results would seem to indicate that I likely would do great things.

But after lots of good grades and academic achievements (I skipped 8th grade and another in high school), that kind of stopped happening. I went to college early, and found out that performing well wasn't always based on being smart. Hard work and regular, consistent effort was also required… and I wasn't really very good at those things. I also had a lot of trouble in college with too many fun things to do, many of which didn't involve school. I got really good at playing foosball, pool, frisbee, going to lots of parties, and making friends, but I kind of barely made it to graduation. Although I did make Dean's list later in college, I was also on probation a few times, and I spent a lot of time doing "recreational activities" (ahem) which caused a lot of pain and hassle for me, and probably even more for my family. I got through those times, but I started to think about all the things I was supposed to be, and the reality was that I wasn't quite getting to the goals that had been

expected. I didn't become an astronaut, or an astrophysicist, or a great singer or dancer or pianist, I didn't end up in politics, I didn't join the peace corps, I didn't get a PhD or even a masters degree. By my mid-twenties, I had headed west to California in search of myself. barely managed to become a decent programmer who bounced around a few jobs, and wasn't really sure where I was going next.

By my late twenties, I stumbled into running my own consulting firm, which sort of became my first startup. We had a lot of ups and downs, and although we won a few awards and did some interesting and innovative work, after five-to-six years of trials and tribulations and serious questioning of my own ability as an entrepreneur and leader, I barely escaped bankruptcy multiple times and ended up with only a very small and desperate acquisition that was hardly anything to brag about. I didn't take the job with Microsoft or Intel in the early '90s, and I didn't join Yahoo or Netscape in the late '90s. I had applied to business school at Stanford, but didn't get in. I was fortunate to get a job at PayPal in 2001 after the first dot-com blowup, but it wasn't with any fanfare, and I was struggling to adjust to a new career in marketing, working with people ten years younger than me from Stanford and MIT who seemed to have their shit together a lot more than I did. After three years' hard work at PayPal, I made some progress, but didn't get any promotions and mostly got shuffled around working with three different bosses who really didn't know what to do with me. In fact, I felt lucky I didn't get fired during my time there, and as I walked out the door I was relieved no one had figured out I was a lame duck who didn't know where the hell I was going.

Don't get me wrong: PayPal was a great place and I made some wonderful friendships and learned a hell of a lot. My own startup had been a comedy of errors, but I did learn a lot about running a business (mostly what not to do) and learned a lot about myself in the process. I also ran a lot of user groups and events, and realized I was pretty good at marketing, and I really loved technology and the Silicon Valley culture. But I still felt like an unfocused underachiever, and at forty I hadn't accomplished much other than finding a good woman foolish enough to marry me, and somehow managing to father two wonderful children I was vastly unqualified to raise. I joined Simply Hired for a few years and did some work I was proud of there, but then continued bouncing around at consulting gigs with oDesk, Mint.com, O'Reilly Media, and others where I still felt like I didn't quite fit in and wasn't making the impact I had hoped. At Mint, I was again fortunate to work with some amazing people, but Aaron

correctly assessed I wasn't really the right guy for the job, and I felt lucky to just play a small part in a decent success story. (Aaron did let me invest some money in the company, which worked out pretty well for me; thanks Aaron!)

So after twenty years in the valley, I had made only a little bit of money, had some modest accomplishments as a programmer, an entrepreneur, and a marketer. Meanwhile my peers at PayPal had gone on to create incredible businesses like LinkedIn, YouTube, Yelp, and Yammer, and other kids half my age were seemingly even more ambitious. Most folks thought I was a decent fellow, but over the hill with my best days behind me…and I guess I thought so, too. I watched as other friends helped make companies like Google and Facebook and Twitter into juggernauts, but mostly I was on the sidelines, only peripherally involved in their big ideas. But I had started doing some angel investing when I left PayPal in 2004, and after finding Mint.com, SlideShare, and Mashery, I figured maybe I had some talent as an investor — since it seemed like I was only a half-assed entrepreneur.

So after some small notoriety in 2007 teaching a class on Facebook at Stanford (strangely, a school where I wasn't good enough to get accepted as a student, somehow let me become a visiting lecturer), I decided I'd try to become a venture capitalist. My timing was of course impeccable, and as I was attempting to raise a small fund in the summer of 2008, the next huge financial crisis hit and the bottom fell out of the market. Again I was fortunate, and my plan B was to humbly say yes to a job offer by Sean Parker to help do some marketing and investing at Founders Fund. I was likely the only person hired in the entire venture industry in Q4 of 2008 (thanks, Sean, I owe you one). I threw myself into the job, and after a year and a half had made some decent picks investing in Twilio, SendGrid, Wildfire, and TaskRabbit among others. Along the way, I also got the opportunity to run the Facebook fbFund for a short time, and made some friends at Accel, Redpoint, and BlueRun. These folks, along with Founders Fund, Mitch Kapor, Michael Birch, Fred Wilson, Brad Feld, Marc Andreessen, and several other generous souls helped me to finally and barely raise a small fund in 2010 that I brazenly named 500 Startups. [500.co]

It would have been easy at any point in this journey to rationalize my limited success, and accept being a small cog in a bigger wheel, at likely much better pay and much less stress. But I was still hoping I had a little fire in the belly, and maybe some gas left in the tank to make something more of myself, before I ended up with just a broken spirit and a comfortable life.

> **“Nowhere near a great success story, yet fighting the good fight and perhaps helping others to achieve greatness as I attempt a bit of my own.”**

And so here I am: still standing in the arena, in hand-to-hand combat with demons mostly of my own making, aiming to make a small dent in the universe. Nowhere near a great success story, yet fighting the good fight and perhaps helping others to achieve greatness as I attempt a bit of my own. I'll be forty-six in a month, well past the age when most folks have already shown what they're made of. But I'm still grasping for that brass ring.

I don't mean to whine or bemoan my lot in life — I've been far more than lucky, and I've had a great time on this planet. I have nothing to complain about, nor will it be the end of the world if all I get to do in the next thirty-to-forty years is to breathe in the air. All things said, it's been a wonderful life.

But I'm not giving up yet.

I'm still betting my epitaph will read "late bloomer," and not "failure."

Wish me luck! ■

Dave McClure is a geek, startup investor, former software developer & entrepreneur, occasional tech blogger and internet marketing nerd. He's lived in Silicon Valley for over 20 years and loved every minute. Dave is the founding partner of 500 Startups, an Internet seed fund and startup incubator that has invested in ~400 companies all over the world.

Reprinted with permission of the original author.
First appeared in *hn.my/bloom* (500hats.com)

Move Your Feet

By MATT SWANSON

- Four books. 28 hours of screencasts. Two online courses.
  - Result: 0 specs in my Rails project.
- Old pair of shoes. Treadmill. One mile in 20 minutes, 23 seconds.
  - Result: 866 miles traveled by foot this year.

Why did I fail so hard at one activity and succeed at the other? With a bit of hindsight, I am starting to figure out the answer.

The first activity (doing TDD in a Rails project of mine) suffered from extreme analysis paralysis. After working as a professional developer for two years, it is so hard for me to just dive in and start sucking at something. I want to learn the best practices so I don't "waste" time doing it incorrectly.

But in this case, best practices are a poison; a hindrance that prevents me from even writing the first spec in my project until I have a perfect vision and roadmap for achieving some mystical TDD nirvana.

In contrast, I was able to ignore this mental roadblock in the second activity. Like many before me, I started the New Year wanting to get into better shape. But instead of finding a book or reading posts on reddit.com/r/running for 2 months, I did something different.

I found an old pair of shoes, got on the treadmill, and just started running. And man, did I really suck at running.

But I didn't care. I could see my improvement every week — the time to run a mile went down, the speed and distance went up (slowly!).

In the software domain, I struggled to convince myself that it was okay to regress in an area as I learned and improved. Instead of starting from the beginning, I tried to skip straight to mastery. With running, my activities were directly related to practicing and improving. Instead of reading guides or spending hours on Amazon trying to find the perfect shoes, I was actually running.

A few weeks ago, I finally went to get some proper running shoes. Once I got to the store, I reverted back to full "Engineer Mode" — trying to determine which brand of shoe was optimal, how many pairs of wicking socks I would need, etc. — when the trainer looked over and made a comment that really resonated with me:

> *"Want to know the secret to improving your running? Move your feet."* ■

Matt Swanson is a software engineer from Indiana and he ships code at SEP. When he's not hacking on side projects, Matt writes about his thoughts on software and personal development at *swanson.github.com* and tries to make jokes on Twitter (@_swanson).

Reprinted with permission of the original author. First appeared in *hn.my/feet* (swanson.github.com)

MEET MANDRILL

By MailChimp

Mandrill is a new way to send transactional, triggered, and personalized emails.
It's also the world's largest species of monkey.

MANDRILL.COM

The Only Two Ways to Build a $100 Million Business

By BORIS WERTZ

With tens of thousands of new startups being created every year, the potential of a company to truly scale and become a large, standalone business is more crucial than ever before. A great product is always the foundation, but a clear distribution strategy becomes essential to cut through the noise. So most early-stage VCs have started to evaluate investment opportunities with an imaginary benchmark in mind: can this company become a $100 million opportunity?

Generally speaking, there are two ways (and only two ways) to scale a business to hit that $100 million threshold:

- Your business has a high Life Time Value (LTV) per user, giving you the freedom to spend a significant amount of money in customer acquisition. High LTV can usually be found in transactional or subscription businesses.
- Your business has a high viral coefficient (or perhaps even a network effect) that lets you amass users cheaply without worrying too much about the monetization per user or spending money on paid acquisition.

Route ❶ High LTV Per User

The exact definition of a "high" user LTV depends on the specific vertical, so it's typically better to analyze the ratio between Customer Acquisition Costs (CAC) and the LTV of the customer. In my experience, having an LTV that's three to four times greater than CAC makes a business interesting.

The biggest driver for high LTV is repeat purchase behavior (in an e-commerce business) and a respectively low churn rate (in a SaaS company). Companies that score highest in this area are typically: e-commerce businesses that fulfill regular needs and offer a differentiated experience or SaaS businesses that help businesses or individuals manage core activities.

As a VC, the biggest challenge in evaluating LTV models is that metrics can dramatically change at scale. For example, CACs often increase once the more efficient marketing channels are maxed out and the company needs to find new users through less efficient means. In addition, churn tends to rise as a company grows. Early users of a product are often strong advocates and company ambassadors, while those users acquired through paid marketing channels down the road show far less loyalty.

Route ❷ The Viral Effect

The other way to scale a business is through a strong viral and/or network effect that lets businesses grow to tens or even hundreds of millions of users. With this model, user acquisition is generally close to free and monetization per user is often low (advertising-based or freemium businesses).

Many businesses built in the early days of the Facebook platform (like Zynga) benefitted from a huge viral coefficient and scaled very rapidly. (As we all know, this is no longer the case as Facebook has essentially removed most of the free viral channels and businesses must now pay for most of their user acquisition via Facebook.)

Even more interesting are businesses that create network effects like marketplaces or social networks. Not only do they acquire lots of users for free due to viral effects but also create important barriers to entry and lock-in effects as the network grows over time.

Startup Purgatory: No Man's Land

Unfortunately, many consumer internet startups find themselves stuck in the middle of these two strategies: they have a low monetization per user and limited viral effects. That unfortunate combination makes it rather difficult to reach the $100 million mark.

As the consumer internet space becomes more and more crowded, every startup founder needs to think about these two ways to scale a business. Too often I have seen entrepreneurs believe that customers will automatically flock to their cool new service, completely underestimating how tough it is to cut through the noise and build an audience.

To build a standalone company and capture the attention of investors, you need a viable way to scale your business. The earlier you figure this out the better, since it may require you to build your product differently. While the $100 million mark may seem far away in those early days, it's important to begin thinking about paths to reach this threshold from the start. ■

Boris Wertz is one of the top tech early-stage investors in North-America and the founding partner of version one ventures. His portfolio encompasses over 35 early-stage consumer internet and mobile companies. Boris is a venture partner of Munich-based Acton Capital Partners, a consumer Internet fund that is focused on later stage companies with an established track record of revenues and profitability. He is also one of the founders of GrowLab, a Vancouver-based start-up accelerator.

Reprinted with permission of the original author.
First appeared in *hn.my/100mil* (versiononeventures.com)

The "Work" Trap

By JOSHUA GROSS

I FIND THAT I — as well as many people I know — fall into a very dangerous trap. I call it The "Work" Trap. What is it? It's both a procrastination technique and a way of staying in your comfort zone while feeling or seeming productive.

You fall into the trap when you forgo other, perhaps beneficial, activities because you have "too much work to do." You turn down a coffee with someone new, avoid going to an interesting meetup, or put off replying to (or initiating) important emails.

How many times have you made that excuse? How many more times have you made that excuse, then failed to even actually do any work?

For many people, "doing work" is easy in comparison to these activities: it's known, familiar, expected, and the best part is that it's also time consuming and "productive." In reality, doing these things could be equally — if not more — beneficial than just attempting to get more work done.

I'm not going to make a bulleted five-point list of things you can do to avoid this trap — everyone justifies it differently. Just recognize when you're falling into the trap out of comfort, as opposed to a true, driving need to get something done. ■

Joshua Gross is a freelance web developer and designer based out if Brooklyn, NY. Beyond creating fun stuff for the web (Kerning.js & more), he spends way too much time playing with Polaroid cameras.

Reprinted with permission of the original author.
First appeared in *hn.my/trap* (unwieldy.net)

Falsehoods Programmers Believe About Time

By NOAH SUSSMAN

Over the past couple of years I have spent a lot of time debugging other engineers' test code. This was interesting work, occasionally frustrating but always informative. One might not immediately think that test code would have bugs, but of course all code has bugs and tests are no exception.

I have repeatedly been confounded to discover just how many mistakes in both test and application code stem from misunderstandings or misconceptions about time. By this I mean both the interesting way in which computers handle time, and the fundamental "gotchas" inherent in how we humans have constructed our calendar — daylight savings being just the tip of the iceberg.

In fact I have seen so many of these misconceptions crop up in other people's (and my own) programs that I thought it would be worthwhile to collect a list of the more common problems here.

All of these assumptions are wrong

- There are always 24 hours in a day.
- Months have either 30 or 31 days.
- Years have 365 days.
- February is always 28 days long.
- Any 24-hour period will always begin and end in the same day (or week, or month).
- A week always begins and ends in the same month.
- A week (or a month) always begins and ends in the same year.
- The machine that a program runs on will always be in the GMT time zone.
- Ok, that's not true. But at least the time zone in which a program has to run will never change.
- Well, surely there will never be a change to the time zone in which a program has to run *in production*.
- The system clock will always be set to the correct local time.
- The system clock will always be set to a time that is not wildly different from the correct local time.
- If the system clock is incorrect, it will at least always be off by a consistent number of seconds.
- The server clock and the client clock will always be set to the same time.
- The server clock and the client clock will always be set to around the same time.
- Ok, but the time on the server clock and time on the client clock would never be different by a matter of decades.
- If the server clock and the client clock are not in synch, they will at least always be out of synch by a consistent number of seconds.
- The server clock and the client clock will use the same time zone.
- The system clock will never be set to a time that is in the distant past or the far future.
- Time has no beginning and no end. [hn.my/2038]
- One minute on the system clock has exactly the same duration as one minute on any other clock. [hn.my/atomic]
- Ok, but the duration of one minute on the system clock will be pretty close to the duration of one minute on most other clocks.
- Fine, but the duration of one minute on the system clock would never be more than an hour.
- You can't be serious.
- The smallest unit of time is one second.
- Ok, one millisecond.

- It will never be necessary to set the system time to any value other than the correct local time.
- Ok, testing might require setting the system time to a value other than the correct local time, but it will never be necessary to do so in production.
- Time stamps will always be specified in a commonly understood format like 1339972628 or 133997262837.
- Time stamps will always be specified in the same format.
- Time stamps will always have the same level of precision.
- 3A time stamp of sufficient precision can safely be considered unique.
- A timestamp represents the time that an event actually occurred.
- Human-readable dates can be specified in universally understood formats such as 05/07/11.

Wait, There's More!

That thing about a minute being longer than an hour was a joke, right?

No.

There was a fascinating bug in older versions of KVM [hn.my/kvm] on CentOS. Specifically, a KVM virtual machine had no awareness that it was not running on physical hardware. This meant that if the host OS put the VM into a suspended state, the virtualized system clock would retain the time that it had had when it was suspended. For example, if the VM was suspended at 13:00 and then brought back to an active state two hours later (at 15:00), the system clock on the VM would still reflect a local time of 13:00. The result was that every time a KVM VM went idle, the host OS would put it into a suspended state and the VM's system clock would start to drift away from reality, sometimes by a large margin depending on how long the VM had remained idle.

There was a cron job that could be installed to keep the virtualized system clock in line with the host OS's hardware clock. But it was easy to forget to do this on new VMs and failure to do so led to much hilarity. The bug has been fixed in more recent versions. ■

Noah Sussman has been helping bricks-and-mortar businesses to leverage the Web since 1999. Thus he has had had ample opportunity to think about the discrepancies between how computers and people see the world. He lives in New York with his wife and two cats.

Reprinted with permission of the original author.
First appeared in *hn.my/falsetime* (infiniteundo.com)

How Statically Linked Programs Run on Linux

By ELI BENDERSKY

In this article I want to explore what happens when a statically linked program gets executed on Linux. By statically linked I mean a program that does not require any shared objects to run, even the ubiquitous `libc`. In reality, most programs encountered on Linux aren't statically linked and do require one or more shared objects to run. However, the running sequence of such programs is more involved, which is why I want to present statically linked programs first. It will serve as a good basis for understanding, allowing me to explore most of the mechanisms involved with less details getting in the way.

The Linux kernel

Program execution begins in the Linux kernel. To run a program, a process will call a function from the `exec` family. The functions in this family are all very similar, differing only in small details regarding the manner of passing arguments and environment variables to the invoked program. What they all end up doing is issuing the `sys_execve` system call to the Linux kernel.

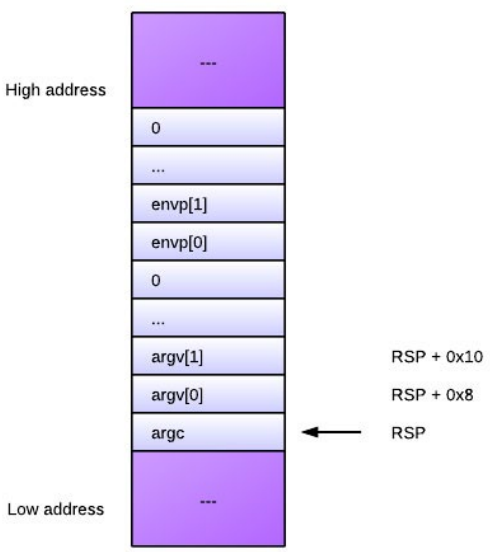
`sys_execve` does a lot of work to prepare the new program for execution. Explaining it all is far beyond the scope of this article — a good book on kernel internals can be helpful to understand the details. I'll just focus on the stuff useful for our current discussion.

As part of its job, the kernel must read the program's executable file from disk into memory and prepare it for execution. The kernel knows how to handle a lot of binary file formats and tries to open the file with different handlers until it succeeds (this happens in the function `search_binary_handler` in `fs/exec.c`). We're only interested in ELF here, however. For this format the action

happens in function `load_elf_binary` (in `fs/binfmt_elf.c`).

The kernel reads the ELF header of the program and looks for a `PT_INTERP` segment to see if an interpreter was specified. Here the statically linked vs. dynamically linked distinction kicks in. For statically linked programs, there is no `PT_INTERP` segment. This is the scenario this article covers.

The kernel then maps the program's segments into memory, according to the information contained in the ELF program headers. Finally, it passes the execution, by directly modifying the IP register, to the entry address read from the ELF header of the program (`e_entry`). Arguments are passed to the program on the stack (the code responsible for this is in `create_elf_tables`). Here's the stack layout when the program is called, for x64:



At the top of the stack is `argc`, the amount of command-line arguments. It is followed by all the arguments themselves (each a `char*`), terminated by a zero pointer. Then, the environment variables are listed (also a `char*` each), terminated by a zero pointer. The observant reader will notice that this argument layout is not what one usually expects in `main`. This is because `main` is not really the entry point of the program, as the rest of the article shows.

Program entry point

So, the Linux kernel reads the program's entry address from the ELF header. Let's now explore how this address gets there.

Unless you're doing something very funky, the final program binary image is probably being created by the system linker — `ld`. By default, `ld` looks for a special symbol called `_start` in one of the object files linked into the program and sets the entry point to the address of that symbol. This will be simplest to demonstrate with an example written in assembly (the following is NASM syntax):

```
section    .text
    ; The _start symbol must be declared for the linker (ld)
    global _start

_start:
    ; Execute sys_exit call. Argument: status -> ebx
    mov     eax, 1
    mov     ebx, 42
    int     0x80
```

This is a very basic program that simply returns 42. Note that it has the `_start` symbol defined. Let's build it, examine the ELF header and its disassembly:

```
$ nasm -f elf64 nasm_rc.asm -o nasm_rc.o
$ ld -o nasm_rc64 nasm_rc.o
$ readelf -h nasm_rc64
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  ...
  Entry point address:               0x400080
  ...
$ objdump -d nasm_rc64

nasm_rc64:     file format elf64-x86-64

Disassembly of section .text:

0000000000400080 <_start>:
  400080:     b8 01 00 00 00          mov    $0x1,%eax
  400085:     bb 2a 00 00 00          mov    $0x2a,%ebx
  40008a:     cd 80                   int    $0x80
```

As you can see, the entry point address in the ELF header was set to `0x400080`, which also happens to be the address of `_start`.

`ld` looks for `_start` by default, but this behavior can be modified by either the `--entry` command-line flag or by providing an `ENTRY` command in a custom linker script.

The entry point in C code

We don't, however, usually write our code in assembly. The situation is different for C/C++ because the entry point familiar to users is the main function and not the `_start` symbol. Now it's time to explain how these two are related.

Let's start with this simple C program, which is functionally equivalent to the assembly shown above:

```
int main() {
    return 42;
}
```

I will compile this code into an object file and then attempt to link it with `ld`, like I did with the assembly:

```
$ gcc -c c_rc.c
$ ld -o c_rc c_rc.o
ld: warning: cannot find entry
symbol _start; defaulting to
00000000004000b0
```

Whoops, `ld` can't find the entry point. It tries to guess using a default, but it won't work — the program will segfault when run. `ld` obviously needs some additional object files where it will find the entry point. But which object files are these? Luckily, we can use `gcc` to find out. `gcc` can act as a full compilation driver, invoking `ld` as needed. Let's now use `gcc` to link our object file into a program. Note that the `-static` flag is passed to force static linking of the C library and the `gcc` runtime library:

```
$ gcc -o c_rc -static c_rc.o
$ c_rc; echo $?
42
```

It works. So how does `gcc` manage to do the linking correctly? We can pass the `-Wl,-verbose` flag to `gcc`, which will spill the list of objects and libraries it passed to the linker. By doing this, we'll see additional object files like `crt1.o` and the whole `libc.a` static library (which has objects with telling names like `libc-start.o`). C code does not live in a vacuum. To run, it requires some support libraries, such as the `gcc` runtime and `libc`.

Since it obviously linked and ran correctly, the program we built with `gcc` should have a `_start` symbol at the right place. Let's check:

```
$ readelf -h c_rc
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 03 00 00 00 00 00 00 00 00
  Class:                             ELF64
  ...
  Entry point address:               0x4003c0
  ...

$ objdump -d c_rc | grep -A15 "<_start"
00000000004003c0 <_start>:
  4003c0:     31 ed                   xor    %ebp,%ebp
  4003c2:     49 89 d1                mov    %rdx,%r9
  4003c5:     5e                      pop    %rsi
  4003c6:     48 89 e2                mov    %rsp,%rdx
  4003c9:     48 83 e4 f0             and    $0xfffffffffffffff0,%rsp
  4003cd:     50                      push   %rax
  4003ce:     54                      push   %rsp
  4003cf:     49 c7 c0 20 0f 40 00    mov    $0x400f20,%r8
  4003d6:     48 c7 c1 90 0e 40 00    mov    $0x400e90,%rcx
  4003dd:     48 c7 c7 d4 04 40 00    mov    $0x4004d4,%rdi
  4003e4:     e8 f7 00 00 00          callq  4004e0 <__libc_start_
main>
  4003e9:     f4                      hlt
  4003ea:     90                      nop
  4003eb:     90                      nop
```

Indeed, `0x4003c0` is the address of `_start` and it's the program entry point. However, what is all that code at `_start`? Where does it come from, and what does it mean?

Decoding the start sequence of C code

The startup code shown above comes from `glibc` — the GNU C library, where for x64 ELF it lives in the file `sysdeps/x86_64/start.S`. Its goal is to prepare the arguments for a function named `__libc_start_main` and call it. This function is also part of `glibc` and lives in `csu/libc-start.c`. Here is its signature, formatted for clarity and with added comments to explain what each argument means:

```
int __libc_start_main(
  /* Pointer to the program's main function */
  (int (*main) (int, char**, char**),
  /* argc and argv */
  int argc, char **argv,
  /* Pointers to initialization and finalization functions*/
  __typeof (main) init, void (*fini) (void),
  /* Finalization function for the dynamic linker */
  void (*rtld_fini) (void),
  /* End of stack */
  void* stack_end)
```

Anyway, with this signature and the AMD64 ABI in hand, we can map the arguments passed to `__libc_start_main` from `_start`:

```
main:       rdi <-- $0x4004d4
argc:       rsi <-- [RSP]
argv:       rdx <-- [RSP + 0x8]
init:       rcx <-- $0x400e90
fini:      r8  <-- $0x400f20
rdld_fini: r9  <-- rdx on entry
stack_end: on stack <-- RSP
```

You'll also notice that the stack is aligned to 16 bytes and some garbage is pushed on top of it (`rax`) before pushing `rsp` itself. This is to conform to the AMD64 ABI. Also note the `hlt` instruction at address `0x4003e9`. It's a safeguard in case `__libc_start_main` did not exit (as we'll see, it should). `hlt` can't be executed in user mode, so this will raise an exception and crash the process.

Examining the disassembly, it's easy to verify that `0x4004d4` is indeed main, `0x400e90` is `__libc_csu_init` and `0x400f20` is `__libc_csu_fini`. The kernel also passes another argument to `_start` — a finish function for shared libraries to use (in `rdx`).

The C library start function

Now that we understand how it's being called, what does `__libc_start_main` actually do? Ignoring some details that are probably too specialized to be interesting in the scope of this article, here's a list of things that it does for a statically linked program:

1. Figure out where the environment variables are on the stack.

2. Prepare the auxiliary vector, if required.
3. Initialize thread-specific functionality (pthreads, TLS, etc.).
4. Perform some security-related bookkeeping (this is not really a separate step, but it is trickled all through the function).
5. Initialize `libc` itself.
6. Call the program initialization function through the passed pointer (`init`).
7. Register the program finalization function (`fini`) for execution on exit.
8. Call `main(argc, argv, envp)`.
9. Call `exit` with the result of `main` as the exit code.

Digression: init and fini

Some programming environments (most notably C++, to construct and destruct static and global objects) require running custom code before and after `main`. This is implemented by means of cooperation between the compiler/linker and the C library. For example, the `__libc_csu_init` (which, as you can see above, is called before the user's `main`) calls into special code that's inserted by the linker. The same goes for `__libc_csu_fini` and finalization.

You can also ask the compiler to register your function to be executed as one of the constructors or destructors. For example:

```
#include <stdio.h>

int main() {
    return 43;
}

__attribute__((constructor))
void myconstructor() {
    printf("myconstructor\n");
}
```

`myconstructor` will run before `main`. The linker places its address in a special array of constructors located in the `.ctors` section. `__libc_csu_init` goes over this array and calls all functions listed in it.

Conclusion

This article demonstrates how a statically linked program is set up to actually run on Linux. In my opinion, this is a very interesting topic to study because it demonstrates how several large components of the Linux eco-system cooperate to enable the program execution process. In this case, the Linux kernel, the compiler and linker, and the C library are involved. ■

Eli Bendersky is an Israeli programmer currently living and working in the Silicon Valley. He likes to disassemble software systems, understanding how they work deep down.

Reprinted with permission of the original author.
First appeared in *hn.my/slinked* (eli.thegreenplace.net)

Learning C with GDB

By ALAN O'DONNELL

Coming from a background in higher-level languages like Ruby, Scheme, and Haskell, learning C can be challenging. In addition to having to wrestle with C's lower-level features like manual memory management and pointers, you have to make do without a REPL. Once you get used to exploratory programming in a REPL, having to deal with the write-compile-run loop is a bit of a bummer.

It occurred to me recently that I could use GDB as a pseudo-REPL for C. I've been experimenting with using GDB as a tool for learning C, rather than merely debugging C, and it's a lot of fun.

My goal in this post is to show you that GDB is a great tool for learning C. I'll introduce you to a few of my favorite GDB commands, and then I'll demonstrate how you can use it to understand a notoriously tricky part of C: the difference between arrays and pointers.

An introduction to GDB

Start by creating the following little C program, minimal.c:

```
int main()
{
    int i = 1337;
    return 0;
}
```

Note that the program does nothing and has not a single printf statement.1 Behold the brave new world of learning C with GDB! Compile it with the `-g` flag so that GDB has debug information to work with, and then feed it to GDB:

```
$ gcc -g minimal.c -o minimal
$ gdb minimal
```

You should now find yourself at a rather stark GDB prompt. I promised you a REPL, so here goes:

```
(gdb) print 1 + 2
$1 = 3
```

Amazing! `print` is a built-in GDB command that prints the evaluation of a C expression. If you're unsure of what a GDB command does, try running `help name-of-the-command` at the GDB prompt.

Here's a somewhat more interesting example:

```
(gbd) print (int) 2147483648
$2 = -2147483648
```

I'm going to ignore why `2147483648 == -2147483648`; the point is that even arithmetic can be tricky in C, and GDB understands C arithmetic.

Let's now set a breakpoint in the `main` function and start the program:

```
(gdb) break main
(gdb) run
```

The program is now paused on line 3, just before `i` gets initialized. Interestingly, even though `i` hasn't been initialized yet, we can still look at its value using the `print` command:

```
(gdb) print i
$3 = 32767
```

In C, the value of an uninitialized local variable is undefined, so GDB might print something different for you!

We can execute the current line with the `next` command:

```
(gdb) next
(gdb) print i
$4 = 1337
```

Examining memory with x

Variables in C label contiguous chunks of memory. A variable's chunk is characterized by two numbers:

1. The numerical address of the first byte in the chunk.
2. The size of the chunk, measured in bytes. The size of a variable's chunk is determined by the variable's type.

One of the distinctive features of C is that you have direct access to a variable's chunk of memory. The `&` operator computes a variable's address, and the `sizeof` operator computes a variable's size in memory.

You can play around with both concepts in GDB:

```
(gdb) print &i
$5 = (int *) 0x7fff5fbff584
(gdb) print sizeof(i)
$6 = 4
```

In words, this says that `i`'s chunk of memory starts at address `0x7fff5fbff5b4` and takes up four bytes of memory.

I mentioned above that a variable's size in memory is determined by its type, and indeed, the `sizeof` operator can operate directly on types:

```
(gdb) print sizeof(int)
$7 = 4
(gdb) print sizeof(double)
$8 = 8
```

This means that, on my machine at least, `int` variables take up four bytes of space and `double` variables take up eight.

GDB comes with a powerful tool for directly examining memory: the `x` command. The `x` command examines memory, starting at a particular address. It comes with a number of formatting commands that provide precise control over how many bytes you'd like to examine and how you'd like to print them; when in doubt, try running `help x` at the GDB prompt.

The `&` operator computes a variable's address, so that means we can feed `&i` to `x` and thereby take a look at the raw bytes underlying `i`'s value:

```
(gdb) x/4xb &i
0x7fff5fbff584: 0x39    0x05
0x00    0x00
```

The flags indicate that I want to examine 4 values, formatted as hex numerals, one byte at a time. I've chosen to examine four bytes because `i`'s size in memory is four bytes; the printout shows `i`'s raw byte-by-byte representation in memory.

One subtlety to bear in mind with raw byte-by-byte examinations is that on Intel machines, bytes are stored in "little-endian" order: unlike human notation, the least significant bytes of a number come first in memory.

One way to clarify the issue would be to give `i` a more interesting value and then re-examine its chunk of memory:

```
(gdb) set var i = 0x12345678
(gdb) x/4xb &i
0x7fff5fbff584: 0x78    0x56
0x34    0x12
```

Examining types with ptype

The `ptype` command might be my favorite command. It tells you the type of a C expression:

```
(gdb) ptype i
type = int
(gdb) ptype &i
type = int *
(gdb) ptype main
type = int (void)
```

Types in C can get complex, but `ptype` allows you to explore them interactively.

Pointers and arrays

Arrays are a surprisingly subtle concept in C. The plan for this section is to write a simple program and then poke it in GDB until arrays start to make sense.

Code up the following `arrays.c` program:

```
int main()
{
    int a[] = {1,2,3};
    return 0;
}
```

Compile it with the `-g` flag, run it in GDB, and then next over the initialization line:

```
$ gcc -g arrays.c -o arrays
$ gdb arrays
(gdb) break main
(gdb) run
(gdb) next
```

At this point you should be able to `print` the contents of a and examine its type:

```
(gdb) print a
$1 = {1, 2, 3}
(gdb) ptype a
type = int [3]
```

Now that our program is set up correctly in GDB, the first thing we should do is use `x` to see what a looks like under the hood:

```
(gdb) x/12xb &a
0x7fff5fbff56c: 0x01  0x00  0x00
0x00  0x02  0x00  0x00  0x00
0x7fff5fbff574: 0x03  0x00  0x00
0x00
```

This means that `a`'s chunk of memory starts at address `0x7fff5fbff5dc`. The first four bytes store `a[0]`, the next four store `a[1]`, and the final four store `a[2]`. Indeed, you can check that `sizeof` knows that `a`'s size in memory is twelve bytes:

```
(gdb) print sizeof(a)
$2 = 12
```

At this point, arrays seem to be quite array-like. They have their own array-like types and store their members in a contiguous chunk of memory. However, in certain situations, arrays act a lot like pointers! For instance, we can do pointer arithmetic on `a`:

```
(gdb) print a + 1
$3 = (int *) 0x7fff5fbff570
```

In words, this says that `a + 1` is a pointer to an int and holds the address `0x7fff5fbff570`. At this point you should be reflexively passing pointers to the `x` command, so let's see what happens:

```
(gdb) x/4xb a + 1
0x7fff5fbff570: 0x02  0x00  0x00
0x00
```

Note that `0x7fff5fbff570` is four more than `0x7fff5fbff56c`, the address of `a`'s first byte in memory. Given that int values take up four bytes, this means that `a + 1` points to `a[1]`.

In fact, array indexing in C is syntactic sugar for pointer arithmetic: `a[i]` is equivalent to `*(a + i)`. You can try this in GDB:

```
(gdb) print a[0]
$4 = 1
(gdb) print *(a + 0)
$5 = 1
(gdb) print a[1]
$6 = 2
(gdb) print *(a + 1)
$7 = 2
(gdb) print a[2]
$8 = 3
(gdb) print *(a + 2)
$9 = 3
```

We've seen that in some situations `a` acts like an array and in others it acts like a pointer to its first element. What's going on?

The answer is that when an array name is used in a C expression, it "decays" to a pointer to the array's first element. There are only two exceptions to this rule: when the array name is passed to `sizeof` and when the array name is passed to the `&` operator.

The fact that `a` doesn't decay to a pointer when passed to the `&` operator brings up an interesting question: is there a difference between the pointer that `a` decays to and `&a`?

Numerically, they both represent the same address:

```
(gdb) x/4xb a
0x7fff5fbff56c: 0x01  0x00  0x00
0x00
(gdb) x/4xb &a
0x7fff5fbff56c: 0x01  0x00  0x00
0x00
```

However, their types are different. We've already seen that the decayed value of `a` is a pointer to `a`'s first element; this must have `type int *`. As for the type of `&a`, we can ask GDB directly:

```
(gdb) ptype &a
type = int (*)[3]
```

In words, `&a` is a pointer to an array of three integers. This makes sense: `a` doesn't decay when passed to `&`, and `a` has type `int [3]`.

You can observe the distinction between `a`'s decayed value and `&a` by checking how they behave with respect to pointer arithmetic:

```
(gdb) print a + 1
$10 = (int *) 0x7fff5fbff570
(gdb) print &a + 1
$11 = (int (*)[3]) 0x7fff5fbff578
```

Note that adding `1` to `a` adds four to `a`'s address, whereas adding `1` to `&a` adds twelve!

The pointer that `a` actually decays to is `&a[0]`:

```
(gdb) print &a[0]
$11 = (int *) 0x7fff5fbff56c
```

Conclusion

Hopefully I've convinced you that GDB is a neat exploratory environment for learning C. You can print the evaluation of expressions, examine raw bytes in memory, and tinker with the type system using `ptype`.

If you'd like to experiment further with using GDB to learn C, I have a few suggestions:

1. Use GDB to work through the Ksplice pointer challenge. [hn.my/ksplice]
2. Investigate how a struct is stored in memory. How does this compare to arrays?
3. Use GDB's `disassemble` command to learn assembly programming! A particularly fun exercise is to investigate how the function call stack works.
4. Check out GDB's "tui" mode, which provides a graphical ncurses layer on top of regular GDB. On OS X, you'll likely need to install GDB from source. ■

Alan is a self-taught programmer who works at Hacker School, where he helps people (including himself) get better at programming. His interests include math, concurrency, programming languages, and the art of learning. He lives in Brooklyn and enjoys Crossfit and playing fetch with his cat.

Reprinted with permission of the original author.
First appeared in *hn.my/gdb* (hackerschool.com)

The Innovations of Internet Explorer

By NICHOLAS C. ZAKAS

Long before Internet Explorer became the browser everyone loves to hate, it was the driving force of innovation on the Internet. Sometimes it's hard to remember all of the good Internet Explorer did before Internet Explorer 6 became the scourge of web developers everywhere. Believe it or not, Internet Explorer 4-6 is heavily responsible for web development as we know it today. A number of proprietary features became de facto standards and then official standards, with some ending up in the HTML5 specification. It may be hard to believe that Internet Explorer is actually to thank for a lot of the features that we take for granted today, but a quick walk through history shows that it's true.

DOM

If Internet Explorer is a browser that everyone loves to hate, the Document Object Model (DOM) is the API that everyone loves to hate. You can call the DOM overly verbose, ill-suited for JavaScript, and somewhat nonsensical, and you would be correct on all counts. However, the DOM gives developers access to every part of a webpage through JavaScript. There was a time when you could only access certain elements on the page through JavaScript. Internet Explorer 3 and Netscape 3 only allowed programmatic access to form elements, images, and links. Netscape 4 improved the situation by expanding programmatic access to the proprietary `<layer>` element via `document.layers`. Internet Explorer 4 improved the situation even further by allowing programmatic access of every element on the page via `document.all`.

In many regards, `document.all` was the very first version of `document.getElementById()`. You still used an element's ID to access it through `document.all`, such as `document.all.myDiv` or `document.all["myDiv"]`. The primary difference was that Internet Explorer used a collection instead of the function, which matched all other access methods at the time, such as `document.images` and `document.forms`.

Internet Explorer 4 was also the first browser to introduce the ability to get a list of elements by tag name via `document.all.tags()`. For all intents and purposes, this was the first version of `document.getElementsByTagName()` and worked the exact same way. If you want to get all `<div>` elements, you would use `document.all.tags("div")`. Even in Internet Explorer 9, this method still exists and is just an alias for `document.getElementsByTagName()`.

Internet Explorer 4 also introduced us to perhaps the most popular proprietary DOM extension of all time: `innerHTML`. It seems that the folks at Microsoft realized what a pain it would be to build up a DOM programmatically and afforded us this shortcut, along with `outerHTML`, both of which proved to be so useful that they were standardized in HTML5. The companion APIs dealing with plain text, `innerText`, and `outerText`, also proved influential enough that DOM Level 3 introduced `textContent`, which acts in a similar manner to `innerText`.

Along the same lines, Internet Explorer 4 introduced `insertAdjacentHTML()`, yet another way of inserting HTML text into a document. This one took a little longer, but it was also codified in HTML5 and is now widely supported by browsers.

Events

In the beginning, there was no event system for JavaScript. Both Netscape and Microsoft took a stab at it and each came up with different models. Netscape brought us event capturing, the idea that an event is first delivered to the window, then the document, and so on until finally reaching the intended target. Netscape browsers prior to version 6 supported only event capturing.

Microsoft took the opposite approach and came up with event bubbling. They believed that the event should begin at the actual target and then fire on the parents and so on up to the document. Internet Explorer prior to version 9 only supported event bubbling. Although the official DOM events specification evolved to include both event capturing and event bubbling, most web developers use event bubbling exclusively, with event capturing being saved for a few workarounds and tricks buried deep down inside of JavaScript libraries.

In addition to creating event bubbling, Microsoft also created a bunch of additional events that eventually became standardized:

- `contextmenu` – Fires when you use the secondary mouse button on an element. First appeared in Internet Explorer 5 and later codified as part of HTML5. Now supported in all major desktop browsers.
- `beforeunload` – Fires before the `unload` event and allows you to block unloading of the page. Originally introduced in Internet Explorer 4 and now part of HTML5. Also supported in all major desktop browsers.
- `mousewheel` – Fires when the mouse wheel (or similar device) is used. The first browser to support this event was Internet Explorer 6. Just like the others, it's now part of HTML5. The only major desktop browser to not support this event is Firefox (which does support an alternative `DOMMouseScroll` event).
- `mouseenter` – A non-bubbling version of `mouseover`, introduced by Microsoft in Internet Explorer 5 to help combat the troubles with using `mouseover`. This event became formalized in DOM Level 3 Events. Also supported in Firefox and Opera, but not in Safari or Chrome (yet?).
- `mouseleave` – A non-bubbling version of `mouseout` to match `mouseenter`. Introduced in Internet Explorer 5 and also now standardized in DOM Level 3 Events. Same support level as `mouseenter`.
- `focusin` – A bubbling version of `focus` to help more easily manage focus on a page. Originally introduced in Internet Explorer 6 and now part of DOM Level 3 Events. Not currently well supported, though Firefox has a bug opened for its implementation.
- `focusout` – A bubbling version of `blur` to help more easily manage focus on a page. Originally introduced in Internet Explorer 6 and now part of DOM Level 3 Events. As with `focusin`, not well supported yet, but Firefox is close.

`<iframe>`

Frames were initially introduced by Netscape Navigator 2 as a proprietary feature. This included `<frameset>`, `<frame>`, and `<noframes>`. The idea behind this feature was pretty simple: at the time, everyone was using modems and roundtrips to the server were quite expensive. The main use case was to provide one frame with navigational elements that would only be loaded once and another frame that could be controlled by the navigation and changed separately. Saving server render time and data transfer by having navigation as a separate page was a huge win at the time.

Internet Explorer 3 supported frames as well, since they were becoming quite popular on the web. However, Microsoft added its own proprietary tag to

that functionality: `<iframe>`. The basic idea behind this element was to embed a page within another page. Whereas Netscape's implementation required you to create three pages to have static navigation (the navigation page, the content page, and the frameset page), you could create the same functionality in Internet Explorer using only two pages (the primary page including navigation, and the content page within the `<iframe>`). Initially, this was one of the major battlegrounds between Internet Explorer and Netscape Navigator.

The `<iframe>` started to become more popular because it was less work than creating framesets. Netscape countered by introducing `<ilayer>` in version 4, which had very similar features to `<iframe>`. Of course, the `<iframe>` won out and is now an important part of web development. Both Netscape's frames and Microsoft's `<iframe>` were standardized in HTML4, but Netscape's frames were later obsoleted (deprecated) in HTML5.

XML and Ajax

Although XML isn't used nearly as much in the web today as many thought it would be, Internet Explorer also led the way with XML support. It was the first browser to support client-side XML parsing and XSLT transformation in JavaScript. Unfortunately, it did so through ActiveX objects representing XML documents and XSLT processors. The folks at Mozilla clearly thought there was something there because they invented similar functionality in the form of `DOMParser`, `XMLSerializer`, and `XSLTProcessor`. The first two are now part of HTML5. Although the standards-based JavaScript XML handling is quite different than Internet Explorer's version, it was undoubtedly influenced by Internet Explorer.

The client-side XML handling was all part of Internet Explorer's implementation of `XMLHttpRequest`, first introduced as an ActiveX object in Internet Explorer 5. The idea was to enable retrieval of XML documents from the server in a webpage and allow JavaScript to manipulate that XML as a DOM. Internet Explorer's version requires you to use `new ActiveXObject("MSXML2.XMLHttp")`, also making it reliant upon version strings and making developers jump through hoops to test and use the most recent version. Once again, Firefox came along and cleaned up the mess up by creating a then-proprietary `XMLHttpRequest` object that duplicated the interface of Internet Explorer's version exactly. Other browsers then copied Firefox's implementation, ultimately leading to Internet Explorer 7 creating an ActiveX-free version as well. Of course, `XMLHttpRequest` was the driving force behind the Ajax revolution that got everybody excited about JavaScript.

CSS

When you think of CSS, you probably don't think much about Internet Explorer. After all, it's the one that tends to lag behind in CSS support (at least up to Internet Explorer 10). However, Internet Explorer 3 was the first browser to implement CSS. At the time, Netscape was pursuing an alternate proposal, JavaScript Style Sheets (JSSS). As the name suggested, this proposal used JavaScript to define stylistic information about the page. Netscape 4 introduced JSSS and CSS, a full version behind Internet Explorer. The CSS implementation was less than stellar, often translating styles into JSSS in order to apply them properly. That also meant that if JavaScript was disabled, CSS didn't work in Netscape 4.

While Internet Explorer's implementation of CSS was limited to font family, font size, colors, backgrounds, and margins, the implementation was solid and usable. Meanwhile, Netscape 4's implementation was buggy and hard to work with. Yes, in some small way, Internet Explorer led to the success of CSS.

The box model, an important foundation of CSS, was heavily influenced by Internet Explorer. Their first implementation in Internet Explorer 5 interpreted `width` and `height` to mean that the element should be that size in total, including padding and border. This came to be known as border-box sizing. The W3C decided that the appropriate box sizing method was `content-box`, where `width` and `height` specified only the size of the box in which the content lived so that padding and border added size to the element. While Internet Explorer switched its standards mode to use the `content-box` approach to match the standard, Internet Explorer 8 introduced the `box-sizing` property as a way for developers to switch back to the `border-box` model. Of course, `box-sizing` was standardized in CSS3 and some, most notably Paul Irish, recommend that you should change your default `box-sizing` to `border-box`.

Internet Explorer also brought us other CSS innovations that ended up being standardized:

- `text-overflow` – Used to show ellipses when text is larger than its container. First appeared in Internet Explorer 6 and standardized in CSS3. Now supported in all major browsers.
- `overflow-x` and `overflow-y` – Allows you to control overflow in two separate directions of the container. This property first appeared in Internet Explorer 5 and later was formalized in CSS3. Now supported in all major browsers.
- `word-break` – Used to specify line-breaking rules between words. Originally in Internet Explorer 5.5 and now standardized in CSS3. Supported in all major browsers except Opera.

- `word-wrap` – Specifies whether or not the browser should break lines in the middle of words. First created for Internet Explorer 5.5 and now standardized in CSS3 as `overflow-wrap`, although all major browsers support it as `word-wrap`.

Additionally, many of the new CSS3 visual effects have Internet Explorer to thank for laying the groundwork. Internet Explorer 4 introduced the proprietary filter property making it the first browser capable of:

- Generating gradients from CSS instructions (CSS3: gradients).
- Creating semitransparent elements with an alpha filter (CSS3: `opacity` and RGBA).
- Rotating an element an arbitrary number of degrees (CSS3: `transform` with `rotate()`).
- Applying a drop shadow to an element (CSS3: `box-shadow`).
- Applying a matrix transform to an element (CSS3: `transform` with `matrix()`).

Additionally, Internet Explorer 4 had a feature called transitions, which allowed you to create some basic animation on the page using filters. The transitions were mostly based on the transitions commonly available in PowerPoint at the time, such as fading in or out, checkerboard, and so on.

All of these capabilities are featured in CSS3 in one way or another. It's pretty amazing that Internet Explorer 4, released in 1997, had all of these capabilities and we are now just starting to get the same capabilities in other browsers.

Other HTML5 contributions

There is a lot of HTML5 that comes directly out of Internet Explorer and the APIs introduced. Here are some that have not yet been mentioned in this post:

- **Drag and Drop** – One of the coolest parts of HTML5 is the definition of native drag-and-drop. This API originated in Internet Explorer 5 and has been described, with very few changes, in HTML5. The main difference is the addition of the `draggable` attribute to mark arbitrary elements as draggable (Internet Explorer used a JavaScript call, `element.dragDrop()` to do this). Other than that, the API closely mirrors the original and is now supported in all major desktop browsers.
- **Clipboard Access** – Now split out from HTML5 into its own spec. It grants the browser access to the clipboard in certain situations. This API originally appeared in Internet Explorer 6 and was then copied by Safari, who moved `clipboardData` off of the `window` object and onto the `event` object for clipboard events.

Safari's change was kept as part of the HTML5 version and clipboard access is now available in all major desktop browsers except for Opera.

- **Rich Text Editing** – Rich text editing using `designMode` was introduced in Internet Explorer 4 because Microsoft wanted a better text editing experience for Hotmail users. Later, Internet Explorer 5.5 introduced `contentEditable` as a lighter weight way of doing rich text editing. Along with both of these came the dreaded `execCommand()` method and its associated methods. For better or worse, this API for rich text editing was standardized in HTML5 and is currently supported in all major desktop browsers as well as Mobile Safari and the Android browser.

Conclusion

While it's easy and popular to poke at Internet Explorer, in reality, we wouldn't have the web as we know it today if not for its contributions. Where would the web be without `XMLHttpRequest` and `innerHTML`? Those were the very catalysts for the Ajax revolution of web applications, upon which a lot of the new capabilities have been built. It seems funny to look back at the browser that has become a "bad guy" of the Internet and see that we wouldn't be where we are today without it.

Yes, Internet Explorer ha its flaws, but for most of the history of the Internet it was the browser that was pushing technology forward. Now that were in a period with massive browser competition and innovation, it's easy to forget where we all came from. So, the next time you run into people who work on Internet Explorer, instead of hurling insults and tomatoes, say thanks for helping to make the Internet what it is today and for making web developers one of the most important jobs in the world. ■

Nicholas C. Zakas is a web technologist, consultant, author, and speaker. He worked at Yahoo! for almost five years, where he was front-end tech lead for the Yahoo! homepage and a contributor to the YUI library. He blogs regularly at *nczonline.net* and can be found on Twitter via @slicknet

Reprinted with permission of the original author.
First appeared in *hn.my/ie* (nczonline.net)

A Software Architect

By CHRIS EPPSTEIN

A SOFTWARE ARCHITECT LIVES to serve the engineering team — not the other way around.

A software architect is a mentor.

A software architect is a student.

A software architect is the code janitor, happily sweeping up after the big party is over.

A software architect helps bring order where there is chaos, guidance where there is ambiguity, and decisions where there is disagreement.

A software architect codes the parts of the system that are the most precious and understands them through and through.

A software architect creates a vocabulary to enable efficient communication across an entire company.

A software architect reads far more code than he or she writes, catching bugs before they manifest as systems change.

A software architect provides technological and product vision without losing sight of the present needs.

A software architect admits when he or she is wrong and never gloats when right.

A software architect gives credit where it is due and takes pride simply in a job well done. ■

Husband & Father, Software Architect for @Caring, Rubyist, Creator of the Compass stylesheet framework, Sass Core Developer, Beer Drinker, Alumnus of Caltech.

Reprinted with permission of the original author.
First appeared in *hn.my/architect* (coderwall.com)

How I Learned to Defrag My Brain

By ALEX HILLMAN

Steven Johnson is one of my favorite authors. I wish I could remember who introduced me to him so I could thank them. The first book of his I read was The Invention of Air, and his most recent Where Good Ideas Come From.

Recently, Steven started a series called "The Writers Room." [hn.my/wroom] Truth be told, his last post is nearly a month old but has moved me so hard for the last month that I wanted to share.

Enter the Spark File

The Spark File, Steven describes, is a process/tool that he uses to collect "half-baked ideas" and then revisit them. For 8 years, he has maintained a single document with notes and ideas with zero organization or taxonomy; simply a chronology of thoughts. He calls this document his Spark File.

Once a month, he reviews the ENTIRE Spark File from top to bottom, revisiting old ideas and potentially combing them with newer ideas.

I've adopted this process for the last 30 days and it's had a remarkable effect. The most astounding part is how often I find myself writing the same thing in different ways. I've taken that pattern as a clue to explore a concept further and see if it merits more investigation.

Your Crippling Compulsion, and the Solution

I was sharing this process with one of my co-conspirators, Tony Bacigalupo, while working with him last week and he said "this process is amazing, it sounds like a defragmentation for your brain."

And it is.

This is particularly important because, as Tony pointed out, we don't have ideas all at once and we certainly don't have them in any particular order. Perhaps more importantly, we tend to either have a compulsion to act on our ideas immediately or not at all.

This compulsion is blocking your greatest work.

By using a Spark File, I'm able to "act" on an idea simply by writing it down at the bottom of the document. Compulsion fulfilled. But unlike the process without Spark File assistance, the idea's destiny isn't written yet. It has the potential to become something greater than an idea, and I'd argue something greater than most 99.9% of all execution.

Any of your half-baked ideas can contribute to the development of **better answers**.

Where Better Answers Come From

Once a month (or any time I wish), I revisit my Spark File notes and look for patterns and clues. I can find inspiration and most importantly, I can find answers — sometimes answers to questions I didn't even know how to ask while I was jotting down my half-baked ideas.

I've found that the inspiration and answers I'm gleaning from my Spark File tend to be more complete, overall deeper and more thorough than if I sit down to work on a single idea "in the moment" that I'm having that idea.

Homework

Your homework, should you choose to accept it:

- Read The Invention of Air. [hn.my/invention]
- Read Where Good Ideas Come From. [hn.my/goodideas]
- Read Steven's post on his Spark File. [hn.my/sparkfile]
- Start a Spark File of your own. Write in it every day.
- Read through your entire Spark File every few weeks (but not every day) looking for links and patterns.

You can defrag your brain too. ■

Alex Hillman is the co-founder of Indy Hall [indyhall.org], one of the world's most respected coworking communities with hundreds of active members and thousands of participants annually from around the world. He publishes the Coworking Weekly email newsletter [coworkingweekly.com] every Thursday. And he teaches people how to build amazing communities in the Community Builder Masterclass [masterclass.indyhall.org].

Reprinted with permission of the original author.
First appeared in *hn.my/defrag* (alexhillman.com)

A Tor of the Dark Web

By DAN SCHULTZ

Tell me if you've been in this situation: you're chatting about online anonymity with your wife and the other Knight-Mozilla Fellows over a pizza in Florence. A quiet-spoken stranger sitting across the room walks up to your table and asks, "Are you all here for the Tor hackathon?" You respond, "Why yes, yes we are!"

He goes on to explain that he is a journalist writing about Tor. He also tells us that he bets that the CIA and the Italian Secret Service are going to have moles there. What he obviously meant to say was, "I work for the CIA and I've been watching you for quite some time now."

It's possible that he didn't actually work for the CIA. His name and photo checked out under the website he claimed to write for. It was probably just a one-time job. Even if this isn't true, even if a network of government spies didn't track my position across Europe just to meet us in a restaurant, his comment set the tone for my weekend in Florence.

Tor is serious business.

What the hell is Tor?

Tor [torproject.org] is a program that makes you anonymous. This means that, for better or for worse, the big brothers, neighborhood hackers, and ad agencies of the world can't tell what you are doing on the Internet without going through a lot of effort and expense.

Is that too abstract? Here are some illustrative statements. *Taps the microphone*

- A Tor user walks into a bar and the bartender asks, "Who are you?"
- How many Tor users does it take to screw in a light bulb? Only a few, but you'll never know who did it.

- I used Tor last night and now my wife says that she doesn't even know who I am any more.

I'll be here all night.

If you use Tor you become Spartacus. Tor takes everything you do, makes it look exactly like what everyone else is doing, and gets random computers on their network to do the talking for you. Ta-da! Now it is practically impossible to pin an action on you.

The Original Need

I bet you wouldn't have guessed that this idea was invented by the U.S. Navy. You would have? Oh.

Put on your paper sailor hat and I'll explain. Imagine you are the king of the Navy and you're going to war with your fleet of a thousand brand new Navy cars (I don't really know how the Navy works). Being king, you are in the most important car of all because you're calling the shots. You don't want the enemy to know which vehicle is yours. You also don't want them to know who is receiving orders because that could give away your tactics.

"I know," you say, "I'll encrypt everything so that they can't see the content. Then they won't be able to tell that my broadcasts are more important than others."

Unfortunately for you, the enemy has fancy technology. They can't decrypt messages, but they are able to track where everything comes from and where it is going. They can't tell what you're saying, but they have all they need.

After about 5 minutes you think you're doing well. Half of the enemy cars are already on fire! Yours explodes. "How did they do that?" you ask in the afterlife. "Easy," responds God, "they were able to see that your car was sending out the most messages. They knew exactly where you were." Then he slaps you with a piece of linguini and drifts away.

To prevent this from ever happening again, the Navy decided to invent the concept of an "Onion Network" (not to be confused with The Onion Network). Now instead of having packets go directly from point A to point B, each one randomly hops around the fleet first. Because of encryption, the enemy can't tell the difference between a new message and a "hop" message — they all look the same. It's like running an invisible sprinkler in a thunderstorm.

Suddenly, nobody but the sender and the recipient can figure out the end points of a message chain. Even the middle men (the ones doing the hops) don't know the path. Each piece of the hop — each "layer" of the message — is encrypted with a different key, so the only thing a relay knows is who gave them the package and where it should go next.

Trolls use the Internet, Ogres use Tor
(Illustration by Anne Buckwalter)

Onions have layers too, that's why this setup is called an Onion Network. Get it? It's like Shrek!

What's it Good For?

Tor has applications in the real world. You can buy drugs and guns, share illegal pictures, and hire assassins. Oh wait, I'm just describing Tor's reputation (more on that later). Seriously, there are a lot of important situations where people have moral and compelling reasons to want anonymity.

Here are a few:

- **Protecting witnesses and victims of domestic abuse.** Anyone who wants to be able to access the Internet without being discovered by a third party can use Tor to defend against their stalkers.
- **If you don't like being tracked** by your government, Internet Service Providers, or search engines.
- **Providing truly anonymous tips.** There are times when people need or want to share information against the wishes of powerful and potentially dangerous forces (e.g., mafias, governments, corporations).
- **Safely bypassing censorship.** If you live in Syria, China, or The United States of RIAA/MPAA, you might use Tor to access content from the outside world more safely.

These kinds of reasons explain why organizations with very good reputations, like the Knight Foundation, are devoting resources to Tor.

The Dark Web

What I've just described is a spin on the way people access normal information online. If you point Tor Browser to Google you will see the same old Google. It's just that now Google doesn't know who you are. That's powerful enough, but there's more: Tor also lets you see hidden content on the Internet.

Using Tor is like entering a cheat code into real life and playing the lost levels. It is the digital equivalent of platform 9 and 3/4 [hn.my/platform9]. This secret section of the Internet is possible because Tor users can serve content anonymously too.

If you don't know much about how the Internet works, believe me when I say that if a website's location is hidden it becomes essentially impossible to access. It would be like trying to visit someone's house without knowing anything about where they live — not even the country. Tor gives you a blindfold and leads you there. You still don't know where the house is, but at least you can visit.

Anonymous sites are accessed through something called an "onion address," which is made up of a series of random letters and numbers. For instance, this is a "clean" version of Tor's wikipedia: 3suaolltfj2xjksb.onion [hn.my/onion]. Feel free to try going to the link; it won't work (unless, of course, you are using the Tor browser [hn.my/torbrowser]).

That random looking string is used to find the server within the Tor network. Because the addresses don't point to a real address on the Internet, there is no way to fully access this content without Tor. There are services [onion.to] you can use to get there without using Tor, but you lose all benefits of anonymity and content is often censored.

Onion addresses are the most fascinating part of Tor, albeit the most potentially disturbing. Rest assured that they don't all lead to child porn, guns, and drugs. For example there is a secret version of Twitter [hn.my/twitteronion], a bunch of blogs [hn.my/blogonion], a search engine [hn.my/searchonion], and an email service [hn.my/emailonion]. There is even a secret version of 4chan (called Torchan), which I won't link to because that one does lead to child porn and drugs.

These types of content networks — ones that are served on top of the normal web so that you need special programs to reach them — are known as the Dark Web. Not necessarily because the content is darker (it is), but because it is hidden from view and can't really be searched and scraped as reliably.

Implications of the Dark Web

Most uses for Tor become more potent with onion addresses. Anonymous servers are just as protected from higher powers as anonymous users. If Amazon suddenly started selling illegal drugs they would get in trouble. If a Tor marketplace started selling illegal drugs, the law would have to figure out a way to find them first.

This power applies to legitimate uses as well. If a government official wanted to contact The Boston Globe with a corruption leak, he or she could use Tor to create a gmail account anonymously. The government could then subpoena Google, and Google might be willing to give away the information they have. They won't know much, but now things like account access patterns and full email logs would be fair game.

If the official had used Tormail, then even Google wouldn't know what happened. The government would have no course of action because there would be no service provider to ask. Every journalist in the world should be able to agree that there is no good reason for a watchdog to trust the organizations they are watching. Why should you trust corporations and governments to keep sources safe?

Tor has a reputation because it has a lot of criminal content, but the social good that it supports is just so important (criminals will always be criminals). I'm working on a game called Torwolf [hn.my/torwolf] to simulate a few situations where Tor would be effective (if you have played Werewolf or Mafia, you can start to imagine what the game will be like). In the mean time, read up on Tor if you're curious [hn.my/torfaq]. Better yet, go try it out. ■

Dan Schultz (@slifty) is a 2012 Knight-Mozilla Fellow at the Boston Globe developing open code and exploring innovation in journalism. He recently graduated from the MIT Media Lab, where he designed and prototyped Truth Goggles, an automated bullshit detector for the Internet. Before coming to the lab Dan was trained to think in terms of systems at Carnegie Mellon University, and was awarded a Knight News Challenge grant to write about "Connecting People, Content, and Community" on the PBS Idea Lab.

Reprinted with permission of the original author.
First appeared in *hn.my/tor* (slifty.com)

Accept payments online.

MEMSET® HOSTING

Rent your IT infrastructure from Memset and discover the incredible benefits of cloud computing.

MINISERVER™ CLOUD COMPUTE

From £0.015p/hour
to 4 x 2.9 GHz Xeon cores
31 GBytes RAM
2.5TB RAID(1) disk

MEMSTORE™ CLOUD STORAGE

£0.07p/GByte/month or less
99.999999% object durability
99.995% availability guarantee
RESTful API, FTP/SFTP and CDN Service

SCAN THE CODE FOR MORE INFORMATION

MEMSET® HOSTING

CarbonNeutral® hosting

ISO 9001: Quality
ISO 14001: Environmental
ISO 27001: Security

Find out more about us at www.memset.com

or chat to our sales team on 0800 634 9270.