Tim Morgan

How Fighter Jets Lock On

HACKERMONTHLY

Issue 55 Dec 2014

HACK ON YOUR SEARCH ENGINE

and help change the future of search

duckduckhack.com

Curator
Lim Cheng Soon

Contributors
Tim Morgan
Eric Lippert
Justin Kan
Steve Klabnik
James Hague
Matthew Plant
Julia Evans
Jonathan Katz
Philip Guo
Bryan Kennedy

Proofreader
Emily Griffin

Printer
Blurb

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.

Cover: Milan Nykodym [flickr.com/photos/milannykodym]

Contents

Photo: Milan Nykodym
[flickr.com/photos/milannykodym]

For links to Hacker News dicussions, visit *hackermonthly.com/issue-55*

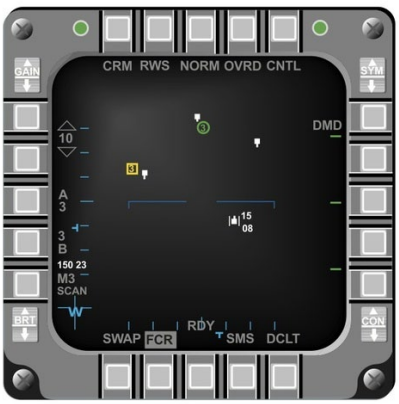
How Fighter Jets Lock On

By TIM MORGAN

Photo: *flickr.com/photos/milannykodym*

THE PRIMARY TECHNOLOGY that a military aircraft uses to lock and track an enemy aircraft is its onboard radar. Aircraft radars typically have two modes: search and track. In search mode, the radar sweeps a radio beam across the sky in a zig-zag pattern. When the radio beam is reflected by a target aircraft, an indication is shown on the radar display. In search mode, no single aircraft is being tracked, but the pilot can usually tell generally what a particular radar return is doing because with each successive sweep, the radar return moves slightly.

This is an example of the fire control radar display for an F-16 Fighting Falcon when the radar is in a search mode:



Each white brick is a radar return. Because the radar is only scanning, not tracking, no other information is available about the radar targets. (There is one exception: The Doppler shift of the radar return can be measured to estimate how fast the aircraft is traveling towards or away from you,

much like the pitch of an oncoming train's whistle can tell you how fast it's coming at you. This is displayed as the small white trend line originating from each brick.)

Note that the cursors are over the bottom-most brick (closest to our aircraft). The pilot is ready to lock up this target. This will put the radar into a track mode. In track mode, the radar focuses its energy on a particular target. Because the radar is actually tracking a target and not just displaying bricks when it gets a reflection back, it can tell the pilot a lot more about the target. This is what the F-16's fire control radar display looks like when a target is locked:

Along the top we have a lot of information about what our radar target is doing:

- Its aspect angle (angle between its nose position and our nose position) is 160° to the left.
- Its heading is 190°.
- Its airspeed is 450 knots.
- Our closure rate is 828 knots.

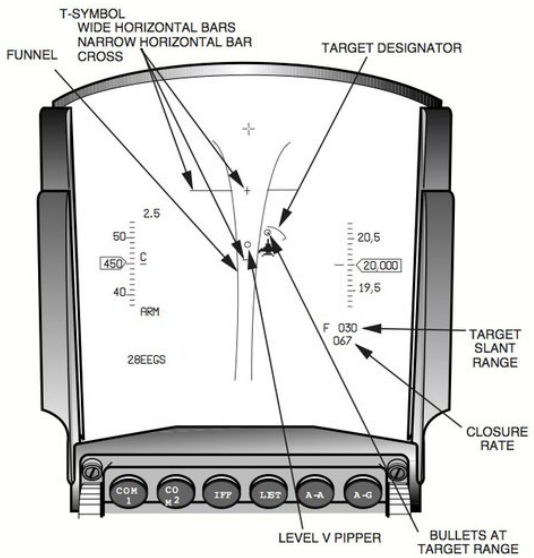
With this information, the pilot gets a much better idea of what the aircraft is doing, but at the expense of information about other aircraft in the area.

Note that in the above picture, the bottom-most (closest) target is locked (circle around it), the two targets further away are tracked (yellow squares), and there are two radar returns even further away (white bricks). This is demonstrating an advanced feature of modern radars, situational awareness modes. A radar in SAM combines both tracking and scanning to allow a pilot to track one or a small number of "interesting" targets while not losing the big picture of what other targets are doing. In this mode, the radar beam sweeps the sky, while briefly and regularly pausing its scan to check up on a locked target.

All of this comes with tradeoffs. In the end, a radar is only as powerful as it is, and you can put a lot of radar energy on one target, or spread it out weakly throughout the sky, or some compromise in between. In the above photo you can see two vertical bars spanning the height of the display; these are the azimuth scan limits. It's the aircraft's way of telling you, "OK, I can both track this target, and scan for other targets, but in return, I'm only going to scan a 40° wide cone in front of the aircraft, instead of the usual 60°. Radar, like life, is full of tradeoffs.

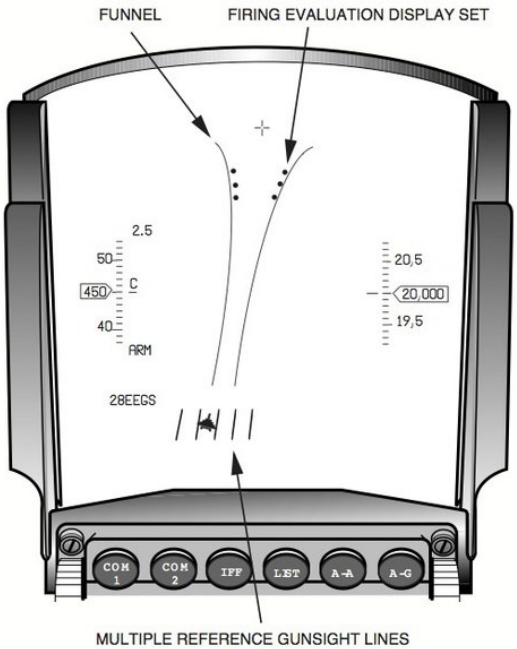
An important thing to note is that a radar lock is not always required to launch weapons at a target. For guns kills, if the aircraft has a radar lock on a target, it can accurately gauge range to the target and provide the pilot with the appropriate corrections for lead and gravity drop to get an accurate guns kill. Without the radar, the pilot simply has to rely on his or her own judgment.

As an example of that, let's take a look at the F-16's HUD (heads-up display) when in the process of employing guns at a radar-locked target:



It becomes incredibly simple: that small circle labeled "bullets at target range" is called the "death dot" by F-16 pilots. Basically, it represents where the cannon rounds would land if you fired right now, and the rounds traveled the distance between you and the locked target. In other words, if you want a solid guns kill, simply fly the death dot onto the airplane. Super simple.

But what if there's no radar lock? Well now the HUD looks like this:



No death dot, but you still have the funnel. The funnel represents the path the cannon rounds would travel out in front of you if you fired right now. The width of the funnel is equal to the apparent width of a predetermined wingspan at that particular range. So if you didn't have a lock on your target, but you knew it had a wingspan of 35 feet, you could dial in 35 feet, then fly the funnel until the width exactly lined up with the width of the enemy aircraft's wings, then squeeze the trigger.

And what about missiles? Again, a radar lock is not required. For

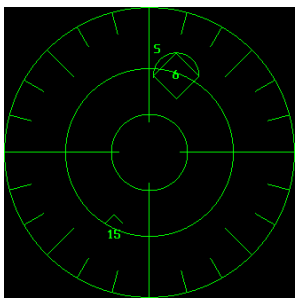
Photo: *flickr.com/photos/milannykodym*

heat-seeking missiles, a radar lock is only used to train the seeker head onto the target. Without a radar lock, the seeker head scans the sky looking for "bright" (hot) objects, and when it finds one, it plays a distinctive whining tone to the pilot. The pilot does not need radar in this case, he just needs to maneuver his aircraft until he has "good tone," and then fire the missile. The radar only speeds up the process.

Now, radar-guided missiles come in two varieties: passive and active. Passive radar missiles do require a radar lock, because these missiles use the aircraft's reflected radar energy to track the target.

Active radar missiles, however, have their own onboard radar, which locks and tracks a target. But this radar is on a one-way trip, so it's considerably less expensive (and less powerful) than the aircraft's radar. So, these missiles normally get some guidance help from the launching aircraft until they fly close enough to the target where they can turn on their own radar and "go active." (This allows the launching aircraft to turn away and defend itself.) It is possible to fire an active radar missile with no radar lock (so-called "maddog"); in this case, the missile will fly until it's nearly out of fuel, and then it will turn on its radar and pursue the first target it sees. This is not a recommended strategy if there are friendly aircraft in close proximity to the enemy.

Finally, an aircraft can tell if a radar is painting it or locked onto it. Radar is just radio waves, and just as your FM radio converts radio waves into sound, so can an aircraft analyze incoming radio signals to figure out who's doing what. This is called an RWR, or radar warning receiver, and has both a video and audio component. This is a typical RWR display:



Although an aircraft's radar can only scan out in front of the aircraft, an aircraft can listen for incoming radar signals in any direction, so the scope is 360°. A digital signal processor looks for recognizable radio "chirps" that correspond to known radars, and displays their azimuth on the scope. A chirp is a distinctive waveform that a radio uses. See, if two radios use the same waveform simultaneously, they'll confuse each other, because each radio won't know which radar returns are from its own transmitter. To prevent this, different radios tend to use distinct waveforms. This can also be used by the target aircraft to identify the type of radar being used, and therefore possibly, the type of aircraft.

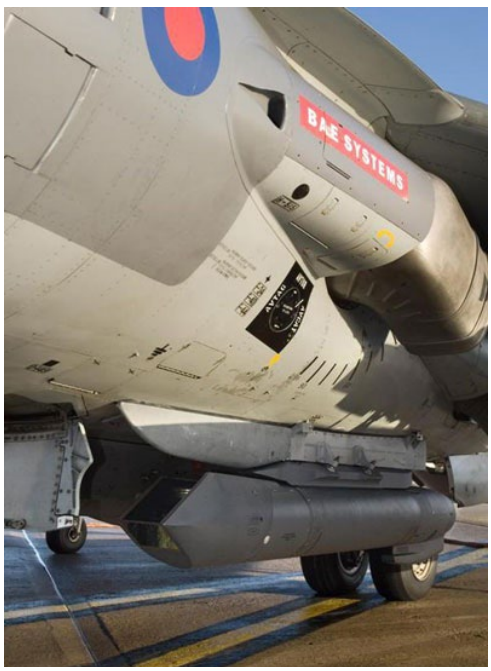
In this display, the RWR has detected an F-15 (15 with a hat on it indicating aircraft) at the 7-o'clock position. The strength of the radar is plotted as distance from the center — the closer to the center, the stronger the detected radar signal, and therefore possibly the closer the transmitting aircraft.

Detected at the 12- to 1-o'clock position are two surface-to-air missile (SAM) sites, an SA-5 "Gammon," and an SA-6 "Gainful." These are Russian SAM launching radars and represent a serious threat. The RWR computer has determined the SA-6 to be the highest priority threat in the area, and thus has enclosed it with a diamond.

RWR also has an audio component. Each time a new radar signal is detected, it is converted into an audio wave and played for the pilot. Because different radars "sound" different, pilots learn to recognize different airborne or surface threats by their distinctive tones. The sound is also an important cue to tell the pilot what the radar is doing: If the sound plays once, or intermittently, it means the radar is only painting our aircraft (in search mode). If a sound plays continuously, the radar has locked onto our aircraft and is in track mode, and thus the pilot's immediate attention is demanded. In some cases, the RWR can tell if the radar is in launch mode (sending radar data to a passive radar-guided missile), or if the radar is that of an active radar-guided

missile. In either of these cases, a distinctive missile launch tone is played and the pilot is advised to immediately act to counter the threat. Note that the RWR has no way of knowing if a heat-seeking missile is on its way to our aircraft.

Aside from radar, there are other technologies that are used to lock onto enemy aircraft and ground targets. A targeting pod is a very powerful camera mounted on an articulating swivel that allows it to look in nearly every direction. This camera is connected to an image processor that is able to tell apart vehicles and buildings from surrounding terrain and track moving targets. This is the SNIPER XR targeting pod:



And this is what the pilot sees when he operates it:

The pod is able to track vehicles day and night, using visual or infrared cameras. Heat-seeking missiles obviously use this same technology to home in on aircraft, and electro-optical missiles use this technology to track ground targets.

Lastly, there are laser-guided missiles as well. These "beam riders" follow a laser beam emanating from the aircraft to the target. Many ground vehicles use laser rangefinders as well, and some aircraft include a laser warning system (LWS) that works similarly to an RWR, but displays incoming laser signals instead. ■

Tim Morgan is a Ruby on Rails developer at Square and holder of a commercial pilot certificate. He has worked at a variety of startups as a web developer over the past eight years. Tim got his pilot certificate in 2007, and has been enjoying it as a hobby ever since.

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

Melting Aluminum

By ERIC LIPPERT

Here's another installment of my ongoing, seldom-updated series of posts about building my own backyard foundry. Today I'll describe how the final step works: actually melting and pouring the metal.

Start by assembling all the equipment you'll need in one place on a day with no chance of rain.

Going from left to right:

- I'm wearing safety goggles, a leather apron, cotton shirt and trousers, and leather boots without laces with the trouser legs around them. You want the boots to protect you from accidentally splashed molten metal, but also want to be able to kick them off in a hurry if they're on fire. I wear natural fibers for the rest of my clothing; synthetic fibers can melt onto you when exposed to high heat and produce a nasty burn.
- The mold is in a deep iron tray — an old baking pan. If the mold separates then the aluminum will spill into the tray, rather than onto my concrete driveway. Concrete can explode when heated rapidly.

- My ingot tray, a rusty old muffin tin, is also on an iron slab, this one the former top of a table saw that I found on the side of the road.
- The white can contains broken up bits of scrap aluminum, old sprues, and ingots to re-melt.
- The furnace and lid; note that the furnace is strapped to a hand truck for ease of moving it around, since it weighs 100 pounds.
- A bag of charcoal. The grocery store was out of wood charcoal, which is preferable because it leaves less "clinker" — tiny bits of stone in the ash — than briquettes, which are compressed wood and coal dust.
- A bucket half full of sand. In the event of a fire caused by spilling liquid metal, you always want to put it out with sand, not water. Water will turn to steam and launch the molten metal off in a random direction.
- Crucible lifters (in the bucket) — two iron rods with the ends bent into hooks that fit into the loops on the top of the crucible.
- Crucible pourer (in the bucket) — a thick iron rod with a bolt through it, so that I can lift the crucible by its loops and then pour it.
- Stir stick (in the bucket) — an iron rod to move things around inside the furnace.
- A long-handled spoon, to skim the dross off the top of the melt.
- A pair of pliers. In an emergency if I need to move the crucible in a hurry I can pick it up with the pliers.
- The crucible — a metal pipe with loops bolted to the opening — is in the bucket right now as well.
- Paper, lighter fluid, and matches to get the furnace going.
- A tub containing aluminum-foil-wrapped packets of KC^{l} and $Na^{2}CO^{3}$, used to flux the melt and reduce dissolved hydrogen. Today I'm melting very clean aluminum so I'm not anticipating that either will have much of an effect, but it doesn't hurt.
- Leather gloves and a face shield, for later.
- In the foreground, a thrift store hair dryer. In retrospect I realize that I've created a tripping hazard by running the hose between the furnace tools and the mold; next time I'll remember to run the hose the long way around. But I am never going to be walking backwards carrying the crucible, so it won't be too bad.

I put a layer of charcoal with some lighter fluid and paper in the furnace, get it lit, and a few minutes later, turn the air on low:

Note that I have removed the lighter fluid from the area. Let's keep the flammable liquids away from the liquid metal, eh?

A few minutes later the layer of charcoal at the bottom of the furnace is burning nicely, so I turn off the air and put the crucible in the furnace.

I then surround the crucible with additional charcoal, put the lid on the furnace, turn the air back on, and put an aluminum tube into the crucible through the hole in the lid. There's also a small amount of scrap aluminum in the crucible already. You don't want to fill the crucible with big pieces of aluminum as it will expand as it heats, and possibly damage the crucible. It's less of a concern with an iron crucible like mine; ceramic crucibles are inelastic and can easily crack if they're full of expanding metal.

The fire will be smoky for some time as it is still low-temperature and there's a lot of new charcoal to catch fire. Note that I have removed the charcoal bag from the area. I'm planning on melting a lot of aluminum today — part of the point of today's melt is to find out just how much I can fit into the crucible — so I will need to add more charcoal halfway through. But I don't want the bag cluttering up the area.

Ten minutes later the exhaust is much cleaner and hotter. Aluminum is not like iron, which gradually gets softer and more malleable as it heats up. Rather, aluminum is more like water ice: as it approaches the melting point it suddenly starts fracturing

easily, then it gets into a slushy state, and then it becomes a liquid. Ten minutes in, the aluminum in the furnace is not liquid but I could tear it with my stir stick more easily than I could tear paper. The tube has started to collapse into the furnace under its own weight.

I put on my gloves and face shield and add more bits of scrap. Less than three minutes later the tubes have melted completely. The liquid aluminum has a huge surface area on the red-hot crucible bottom, and transfers heat very quickly to the solid aluminum tubes sinking into it. Eight minutes later I have melted an entire lawn chair and am out of tubes, so I start on sprues and ingots, which take longer because they have much less surface area per unit mass. At this point the charcoal has burned down quite a bit, so I add some more around the crucible.

Aluminum tubing is easy to find as scrap, and in fact for today's melt, I didn't even have to leave my property; someone left a broken lawn chair on my front lawn. Normally I am irritated when people leave trash on my lawn, but I will gladly accept broken lawn chairs! The alloy is designed for easy extrusion, not for metalworking, so it is not ideal, but you can't beat the price, and it usually is not painted. Paint (or vinyl coating) makes toxic fumes and causes the melt to take on hydrogen gas, which then forms bubbles in the casting. The Na^2CO^3 will help remove the hydrogen, but I'd rather not go there in the first place if I can avoid it.

Twenty minutes later I have melted enough sprues and ingots that the crucible is almost completely full. I take the lid off, skim the dross — the bits of aluminum oxide and impurities that float — off the top and put it in the ingot tray to cool; this will be trash. (Trying to recover aluminum from aluminum oxide is not really worth it when aluminum is so plentiful.) I add some flux and degassing powder, and then turn the air blast on to maximum to get the last bit of heat out of the charcoal. I remove the cover from the mold. A minute or so later, I'm ready to pour; I remove the lid, take off the last bits of dross, and get ready to take the very full crucible out. Here's what the inside of the furnace looks like:

This isn't quite as hot as I would like — I'd prefer that the crucible be more obviously red-hot — but this was the largest melt I've done, and it's taken longer than usual. As you can see, almost all the fuel is used up. It's hot enough to pour, and so I'm going for it. I very carefully lift the crucible out of the furnace and put it down in the sand bucket:

And then lift it right back up again with the pourer:

And then pour into the mold; the remainder goes in the muffin tin.

Then wait for it to cool, dump the sand back into the sand bin and extract the casting from the sand. The result was 1632g of aluminum, so now I know the maximum mass of aluminum I can melt with this crucible and furnace. ■

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

Eric Lippert designs C# analyzers at Coverity; previously he was a member of the C# and JavaScript language design teams at Microsoft. When not working on computers he enjoys playing the piano, building things out of wood and metal, and trying to keep his tiny sailboat upright.

The Founder's Guide To Selling Your Company

By JUSTIN KAN

For most founders, selling a company is a life-changing event that they have had no training for. At Y Combinator, one big thing we help our startups with is navigating questions around the acquisition process. Originally, I wrote this guide for YC startups outlining what I've learned about selling startups in my last ten years as an entrepreneur. If you are going through an acquisition, hopefully this will be useful to you.

When to Sell

Similar to raising money, the best time to sell your startup is when you don't need to or want to. Paradoxically, you are probably thinking about selling your startup as you are experiencing a lack of traction, tough competition, or difficult time fundraising. However, this is a bad time to sell your startup: you will have few bidders and be more likely to acquiesce to the demands of anyone who does show up.

The best time to sell your startup is when you have many options. These options don't all have to be acquisition offers, they can also be venture term sheets for your next round. You might even be operating profitably and find yourself in the enviable position of confidently being able to turn down an offer. Usually, you will have these options because your startup is actually experiencing great traction; counterintuitively, the best way to "build to flip" is actually the same as building a successful company.

The following is a brief overview of the steps that go into valuing your company, garnering interest in it and navigating through the acquisition process.

Starting Acquisition Talks

Do not enter acquisition talks unless you are ready to sell your company. Negotiating an acquisition is the most distracting thing you can do in a startup: going through M&A is an order of magnitude more distracting than raising money. All of your ability to run the day-to-day operations of your company will grind to a halt. You should only enter an acquisition process if you are certain 1) you want to sell the company and 2) you are likely to get a price you will accept. Don't talk to potential acquirers "just to see what price you can get."

How Your Startup Will be Valued

Investors value companies based on either their financial value or their strategic value. A company's financial value hinges on its profits and model of its future cash flows. For the vast majority of startups in tech, this will be zero. It is much more likely is that your startup will be valued based on where it fits in with the acquiring company's short or long term strategy. Here are some surprisingly common reasons your startup has strategic value to an acquirer:

- The CEO finds it interesting, or wants to keep it away from another large tech company.
- The executive that runs the relevant division of the acquirer needs to demonstrate "big moves."
- A competitor to an acquirer is outexecuting it in a business and you can help the acquirer become better.
- The acquirer doesn't have or can't retain talent in an area where you have employees.
- Your businesses actually have some synergies and combining them is theoretically value-accretive.
- The acquirer is running a similar business but you are executing much better. They are afraid of you.

If some of these reasons seem ridiculous and arbitrary, it is because sometimes they are. Remember that companies are bought, not sold: in order for a company to want to buy you, an internal champion will have to internalize one of these reasons. This isn't something that can be forced.

When it comes to setting price, there is no "right" price for a company, there is only the price that you can negotiate. Management teams, investment bankers, and corporate development people will concoct an array of metrics — like cost per user — to justify a number. Ultimately, though, the clearing price for a startup depends on what the big company can justify to the market (previous comparable acquisitions are a good benchmark) and the sale price agreed to by you and your investors.

A potential acquirer's first offer is rarely its best offer. Don't be afraid to say "no" — the potential acquirer isn't going anywhere. There are many negotiation strategies, but in order to extract the most value you need to (1) be willing to walk away and (2) initiate a competitive bidding process.

Getting Offers

The best way to solicit acquisition offers is to have ongoing conversations with potential acquirers about ways you can work together. These conversations usually involve many different people within a big company; you'll only get an offer once a sufficiently high-up decision maker is convinced that buying you is a better idea than partnering with you.

When you have an offer, the next thing to do is determine your alternative options. You can do this by letting other potential acquirers (including larger companies you have partnered with and/or competitors to the acquirer) know that you have received a term sheet from an acquirer and you are considering selling yourself, but would prefer a longer term future with their company (find a reason). Now would also be a good time to call up VCs you have been talking to and ask for a term sheet for your next round.

Sometimes, a company you are doing a critical business development partnership with will insist on a "cool down period" in an agreement. This is a period of time during which you have to wait once you've received an acquisition offer before you can sign (for your partner to theoretically prepare a better counter offer). Cool down clauses can actually work in your favor as a way to acquire additional offers once you've received a first term sheet.

Bullshit Offers

Most of the offers you will receive to buy your startup will be bullshit offers. It costs a company exactly zero dollars to tell you: "we want to buy your startup." In fact, companies employ teams of people, under the euphemism Corporate Development, to go around the Valley and repeat this phrase to founders.

Bullshit offers are dangerous because they can lull you into a false sense that you are being successful. Like TechCrunch articles, bullshit offers are a vanity metric, not an actual measure of success.

You can tell if an offer is bullshit because it will not be accompanied by an expiration date and/or a promise of a term sheet delivery within a very short period of time (24-48 hours). When a sufficiently high-up decision maker decides he/she wants to buy your startup, he/she will attempt to meet with you constantly and put time pressure on you, so as to prevent you from shopping the deal and getting a

better offer. The absence of this behavior indicates the other company is not serious about acquiring your business.

Often the expectations of the founders and corporate development people are very divergent. Before proceeding far into conversations with big acquirers, you should try to clarify valuation expectations (and other important consideration details, like retention packages) as quickly as possible. This strategy should help prevent you from having half a dozen meetings, only to find out the potential acquirer expects to pay $10 million for your rapidly growing startup that already has a term sheet for a $15 million A round.

Hiring a Banker

Like the world of venture capital, investment banking is a field filled with a very small number of extremely well-connected, analytical, and experienced people and a much larger number people pretending to be those things. It is unlikely you will be able to find someone in the former category unless your startup's realistic selling price is in the mid-hundreds of millions or above. But the good news is that you probably don't need an investment banker at all unless your selling price is that high, and maybe not even then.

Investment bankers are expensive (1 to 2% of the total deal value). However, the good ones can help you get a thorough understanding of the competitive landscape, who the individual decision makers are at every potential acquirer, and what buttons to push to maximize your deal value. Also, the people you are negotiating with in corporate development are professional negotiators: they spend all day every day trying to pay less for target companies. You have probably spent a lot of time doing things that are not negotiating. Having a professional negotiator on your side is often extremely valuable, if you can get someone good.

Pre-Term Sheet Diligence

If you are committed to going through an acquisition process, you should be fairly free with your company data (under an NDA), as it is better that the acquirer uncover any red flags before you've signed a term sheet and entered the closing process. However, if the potential acquirer asks to interview your team you should absolutely refuse (unless you are going through a talent acquisition and have no other options). Letting a potential acquirer interview your team is extremely distracting for them, and signals to the acquirer that you are willing to bend over.

Signing a Term Sheet

Once you have collected all your term sheets, you can sign one. Before you do, you should try to negotiate the business and legal points in as much detail as possible. Feel free to push back on

exploding offer deadlines and other pressure to sign immediately. After you sign, you can expect any points that weren't previously negotiated will end up with language in favor of the acquirer.

As the startup, you have all the leverage before you sign a term sheet. Once you sign, you have almost no leverage at all. This is mental: before you sign a term sheet, you haven't yet decided to sell. Once you do, you have committed to selling the deal to yourself, your employees and investors. Once you get negotiation fatigue — and maybe even before — you will start to agree quickly to things you wouldn't have considered at the term sheet stage. Also, once you've signed a term sheet you can no longer shop your company to other acquirers. If your deal falls apart, other acquirers may have cooled off or think that the deal fell through because your company is damaged goods. It may be impossible to resuscitate your other options.

When negotiating a term sheet, push for a shortest possible closing period (target 30 days) to avoid getting deal fatigue and to put pressure on the acquirer (although, be aware that sometimes a regulatory issue will dictate the timing of the closing and that is outside of everyone's control). A short closing period will also help you somewhat limit the distraction from your main job: running your startup.

Before you sign a term sheet that commits you to either working at the acquirer or accepting the acquirer's stock as consideration for the acquisition, ask yourself if you actually believe in the company. The Valley is replete with cautionary tales of startups that sold themselves in exchange for the stock of ultimately worthless acquirers. Don't let yourself become another one.

Just because you signed a term sheet does not mean your deal is done; in fact, it is very possible that it will still fall apart. Despite a commitment to trying to close, companies change their minds all the time during the diligence process. If your deal fails and the result is that your startup is collateral damage because you were distracted, remember that this is probably an acceptable outcome for the would-have-been acquirer.

Closing

The most dangerous stretch during the acquisition process is the time between when you decide you are going to sell, and when the sale actually occurs. You've decided to exchange stress for riches, and you can already see that new house on the horizon (and maybe one for your parents).

What happens when the acquirer comes back and changes the total deal value?

What happens when the acquirer changes its mind, and you have to go back to the grind?

What happens when the acquirer has talked to your senior management, and then decided your team isn't good enough?

If you are running low on cash, what happens when you run out of cash before the deal closes, because negotiating the deal documents took twice as long as you expected?

These things happen. You should be prepared to walk away from any deal up until the point where you are watching your bank account, waiting for the wire transfer from the acquirer to hit.

Getting to closing is a process largely driven by lawyers. The items to be negotiated are typically divided into legal points and business points — the lawyers will resolve the legal points, but you are expected to figure out the business issues. Remember that you are ultimately responsible for the outcome, and that the lawyers work for you (and usually get paid if the deal happens or not). You have to stay on top of the lawyers to make sure that they aren't slowing down the deal by getting bogged down in details that don't actually make any difference to you or your stakeholders.

In Summary

Entering the acquisition process is one of the most dangerous things an early stage startup can do, because the process is distracting, demoralizing, and usually involves giving your competition most of your proprietary business data. Founders who have been through the process have said it is ten times as distracting as fundraising. It often cripples your ability to oversee the business operations. Do not enter into an acquisition process lightly. ■

Justin Kan is a partner at Y Combinator. Previously he founded Justin.tv, TwitchTV and Socialcam.

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

TDD Your API

By STEVE KLABNIK

How do you TDD an API?

If you're not familiar, here's the basic outline of Test Driven Development:

1. Write a test for some behavior you'd like to introduce into your system.
2. Run your test suite, and make sure that test fails.
3. Write the simplest code that implements the behavior.
4. Run your test suite, and make sure that test passes.
5. Refactor, because the simplest code often has undesirable properties.
6. Commit, and GOTO 1.

To say that there's a large amount of literature on the benefits of this approach would be an understatement. I just want to focus on one of the properties of this approach: verifying behavior. TDD verifies behavior in a few ways. The first is in step two, with the failing test. This failing test is important because it verifies that our test does test for a new behavior. If the test was passing at this point, we probably have a bug in our test. The second way that behavior is verified in TDD is step four, when the test passes. This verifies that our new behavior satisfies the contract we've laid out in the test, which is incredibly important! Finally, when software is changed, we use TDD to verify that only the behavior we were interested in changing has changed. We don't want our good change to break anything else.

So how is this different when applied to the API context? Here's how you to TDD for an API:

1. Create a separate project for your API tests.
2. Write a test for some behavior you'd like to introduce into your system. Because this test suite is in a different project, this will be an

integration-style test, which makes HTTP calls against your customer-facing test environment.

3. Run your test suite, and make sure that test fails.
4. Implement this behavior in your system, and push that code to your test environment.
5. Run your test suite, and make sure that test passes.
6. Refactor any of your test code that may have gotten messy.
7. Commit, and GOTO 1.

There's another aspect which is important, though. This project should be at least partially public and included with all of your other documentation. Because Balanced is an open company, ours is 100% public, but if you aren't doing new feature development in the open the way we are, just have the latest passing suite public. This aspect is important because, in this model, these API tests become the canonical source of truth for what your API provides to customers. Your customers can trust them because they're automatic and open. Does the API work? Check the build status.

Push to card [balancedpayments.com/push-to-card] was the first real feature we've built in this way. It's been a long time coming, though. Let's take a journey.

Origins

From the start, there was an idea that something needed to be built to handle some API-related things. With this dummy commit, the first question kicked off: Markdown or reStructured Text.

There were three big questions to tackle:

1. How can we validate that our API is working as intended?
2. Can we generate documentation from this?
3. How would all this be written, tool-wise?

While this was being decided, at least Issues provided a good forum to discuss things. From the first version of the README:

> *The primary goal of this repo is to create more openness behind the decisions driving the designs and functionality of the Balanced API. We reached out to existing and potential customers when designing the API, but that was a limited set of people we already knew. We've received tremendous growth in the last few months, and our new customers have great feedback or at least want to understand the reasoning behind the original decisions.*

An initial solution

It was settled that reStructured Text was the answer. Here's what they looked like, roughly.

```
===============
accounts.create
===============

:uri: /v1/marketplaces/
(marketplace:marketplace)/accounts
:methods: POST

Fields
******

.. list-table::
   :widths: 20 80
   :header-rows: 1

   * - Name
     - Description
   * - ``email_address``
     - The email address of the
account, unique constraint.
   * - ``name``
     - The display name of the
account `optional`.
```

Some bits removed for brevity. And then creating a card:

```
====================
cards.create
====================

:uri: /v1/marketplaces/
(marketplace:marketplace)/cards
:methods: POST
```

This was okay, but it wasn't great. For example, I now have an account, and I have a card, but I can't add that card to that account, because they're two isolated tests, not one larger test. These tests assume that each URL/HTTP method combination can or should be tested in isolation, and that's just not true. Because of this, checking if tests passed or failed often involved humans. The diff would show that a timestamp was a fraction of a second off, or that an autogenerated ID would appear which couldn't be predicted ahead of time.

reStructured Text wasn't really designed to do this, anyway. A new approach was sought.

The YAML year

The next iteration of this idea used YAML instead of rST. The YAML tests looked like this:

```
require:
  - ../card_fixtures.yml
  - ../customer_fixtures.yml
scenarios:
  - name: add_card_to_customer
    request:
      method: PATCH
      href: "{card,cards.href}"
      schema:
        "$ref": "requests/_patch.
json"
      body: [{
        "op": "replace",
        "path": "/cards/0/links/
```

```
customer",
        "value":
"{customer,customers.id}"
      }]
    response:
      schema:

"$ref": "responses/cards.json"
      matches: { "cards":
[ { "links": { "customer":
"{customer,customers.id}" } } ] }
```

It's almost a serialized HTTP request. Since they have names, they can refer to each other, and you can do things like interpolate variables, use a regular expression to only match the parts you care about, and make scenarios depend on each other for re-usability.

Speaking of which, how does that work? Well, each scenario has a name, and if you refer to a previous scenario, it will run that one first. So, in the href line there, it refers to `{card,cards.href}`. This says "run the `card` scenario, and then get the value of cards.href in the resulting JSON. We later do the same, twice, with `{customer,customers.id}`.

Where are the `card` and `customer` scenarios? The first line of the file requires two other YAML files, those are probably good candidates. Here they are in their full glory. Can you find the dependent scenarios?

```
customer_fixtures.yml:
scenarios:
  - name: customer
    request:
      method: POST
      href: /customers
      schema:
        "$ref": "../requests/cus-
tomer.json"
      body: {
        "name": "Balanced testing"
      }
    response:
      status_code: 201
      schema:
        "$ref": "../responses/cus-
tomers.json"

  - name: underwritten_merchant
    request:
      method: POST
      href: /customers
      schema:
        "$ref": "requests/cus-
tomer.json"
      body: {
        "name": "Henry Ford",
        "dob_month": 7,
        "dob_year": 1963,
        "address": {
          "postal_code": "48120"
        }
      }
    response:
      status_code: 201
      schema:
        "$ref": "responses/custom-
ers.json"
      matches: { "customers": [ {
"merchant_status": "underwritten"
} ] }
```

```
  - name: customer_with_card
    request:
      method: POST
      href: /customers
      schema:
        "$ref": "requests/cus-
tomer.json"
      body: {
        "name": "Darius the
Great",
        "email": "darius.great@
gmail.com",
        "source": {
          "name": "Darius the
Great",
          "number":
"4111111111111111",
          "expiration_month": 12,
          "expiration_year": 2016,
          "cvv": "123",
          "address": {
            "line1": "965 Mission
St",
            "line2": "Suite 425",
            "city": "San Fran-
cisco",
            "state": "CA",
            "postal_code": "94103"
          }
        },
        "meta": {
          "ip_address":
"174.240.15.249"
        }
      }
    response:
      status_code: 201
      schema:
        "$ref": "responses/custom-
ers.json"

  - name: merchant_with_bank_
account
    request:
      method: POST
      href: /customers
      schema:
        "$ref": "requests/cus-
tomer.json"
      body: {
        "name": "Henry Ford",
        "dob_month": 7,
        "dob_year": 1963,
        "address": {
          "postal_code": "48120"
        },
        "destination": {
          "name": "Kareem Abdul-
Jabbar",
          "account_number":
"9900000000",
          "routing_number":
"021000021",
          "account_type": "check-
ing"
        }
      }
    response:
      status_code: 201
      schema:
        "$ref": "responses/custom-
ers.json"
      matches: { "customers": [ {
```

```
"merchant_status": "underwritten"
} ] }
```

Whew! That's a ton of stuff. grep can help you a bit here, but it's still not very fun. The same words repeat quite a bit.

Here's `card_fixtures.yml`:

```
require:
  - ./customer_fixtures.yml
scenarios:
  - name: card
    request:
      method: POST
      href: /cards
      schema:
        "$ref": "requests/card.
json"
      body: {
        "number": "4111 1111 1111
1111",
        "expiration_month": 12,
        "expiration_year": 2016,
        "cvv": "123",
        "address": {
          "line1": "965 Mission
St",
          "postal_code": "94103"
        }
      }
    response:
      status_code: 201
      schema:
        "$ref": "responses/cards.
json"

  - name: customer_card
    request:
      method: POST
      href: /cards
      schema:
        "$ref": "requests/card.
json"
      body: {
        "number": "4111 1111 1111
1111",
        "expiration_month": 12,
        "expiration_year": 2016,
      }
    response:
      status_code: 201
      schema:
        "$ref": "responses/cards.
json"

  - name: associate_customer_card_
with_customer
    request:
      method: PUT
      href: "{customer,customers.
href}"
      schema:
        "$ref": "requests/cus-
tomer.json"
      body: {
        "card_uri": "{customer_
card,cards.href}"
      }
    response:
      schema:
        "$ref": "responses/custom-
ers.json"

  - name: card_processor_failure
```

```
    request:
      method: POST
      href: /cards
      schema:
        "$ref": "requests/card.
json"
      body: {
        "number":
"4444444444444448",
        "expiration_month": 12,
        "expiration_year": 2018
      }
    response:
      status_code: 201
      schema:
        "$ref": "responses/cards.
json"
```

Did that make your eyes glaze over? Exactly. Nobody actually wants to write YAML. Nobody actually wants to read YAML. Tests that refer to each other help reusability but harm understanding. Since this was entirely homegrown, people didn't really know how it worked. This also meant that there was a large amount of work to get up and running.

Time to throw it all out and do something again!

At this point, it was clear that building something totally custom wasn't the right answer. The maintenance costs are just too high. And other people build APIs, so they must do something like this, right? Well, while there are some tools for doing things like this, none of them are particularly great. They were all missing at least one thing that we considered vital. One of the bigger issues is that many of these tools assume that you're simply doing "Rails style REST," and not using hypermedia. Or they require you to write a WSDL or WADL or use RAML.

Eat your cucumbers!

Replacing this system was my first task here. I decided to use Cucumber. The Cucumber tests look like this:

```
Feature: Credit cards

  Scenario: Add a card to a cus-
tomer
    Given I have tokenized a card
    And I have created a customer
    When I make a PATCH request to
/cards/:card_id with the body:
      """
        [{
          "op": "replace",
          "path": "/cards/0/links/
customer",
          "value": ":customer_id"
        }]
      """
    Then I should get a 200 OK
status code
    And I make a GET request to /
cards/:card_id
    Then the response is valid
according to the "cards" schema
    And the fields on this card
match:
```

```
      """
        {
          "links": { "customer":
":customer_id" }
        }
      """
```

I'm often skeptical of Cucumber, and it's often misused, but I think this is a pretty great use case. It's language agnostic, which is important to us, and it's got lots of room for explanations and clarifications. It's clear that this needs a card and a customer to work, and I can just find the step definitions (or ask Cucumber to tell me where they are) to see the details.

Here are the specs [hn.my/p2cspec] for push-to-card. By using this approach, we caught problems before the feature was even finished. The initial implementation contained a bug with a typo in one of the response keys. There was some confusion in one corner of the spec, and while some of the discussion happened in person, the ways in which the spec failed while the feature was being built out helped make sure that everyone was on the same page.

Going forward

Now that we have a testable validation that our API is working as intended, we can do all kinds of things. We'd like to integrate this suite into our more general continuous integration suite. We'd like to run this test every hour, or every three hours, which can be a form of alerting against regressions.

In order to do all that, we need to fix our build. That's what happens when you don't run the tests automatically: regressions can creep in and you won't realize it. These failures are all related to a bug which only happens with a brand new marketplace, and doesn't strictly affect customer behavior. I wanted the fix to roll out before this post, but you don't always get what you want. This is a good reminder that tests are not perfect, and that they're a tool to alert you that something may have gone wrong, not proof that there's an error. Test code can be fragile or have bugs, too. But without these tests, we wouldn't have realized that there was a regression, no matter how small.

It's hard to get any complex software system behavior correct. So far, we've found that these external tests give us a really nice forum for working out these kinds of issues. They also give us something to share with our customers, and a way to ask for advice from experts outside of the company. ■

Steve Klabnik is a Rails committer, Rust enthusiast, and works for Balanced Payments in San Francisco. He has authored "Designing Hypermedia APIs," "Rust for Rubyists," and "Rails 4 in Action." When he's not programming, he reads philosophy books and plays Android: Netrunner.

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

Lost Lessons from 8-Bit BASIC

By JAMES HAGUE

Unstructured programming with GOTO is the stuff of legend, as are calling subroutines by line number — GOSUB 1000 — and setting global variables as a mechanism for passing parameters.

The little language that fueled the home computer revolution has been long buried beneath an avalanche of derision, or at least disregarded as a relic from primitive times. That's too bad, because while the language itself has serious shortcomings, the overall 8-bit BASIC experience has high points that are worth remembering.

It's hard to separate the language and the computers it ran it on; flipping the power switch, even without a disk drive attached, resulted in a BASIC prompt. If nothing else, it could be treated as a calculator:

```
PRINT "seconds in a
week: ",60*60*24*7
```

or

```
PRINT COS(2)/2
```

Notice how the cosine function is always available for use. No importing a library. No qualifying it with MATH.TRIG.

Or take advantage of this being a full programming language:

```
T = 0
FOR I=1 TO
10:T=T+I*I:NEXT I
PRINT T
```

It wasn't just math. I remember seeing the Atari 800 on display in Sears, the distinctive blue background and READY prompt visible across the department. I'd switch to a bitmapped graphics mode with a command window at the bottom and dash off a program that looped across screen coordinates displaying a multicolored pattern. It would run as an in-store demo for the rest of the day or until some other know-it-all pressed the BREAK key.

There's a small detail that I skipped over: entering a multi-line program on a computer in a department store. Without starting an external editor. Without creating a file to be later loaded into the BASIC interpreter (which wasn't possible without a floppy drive).

Here's the secret. Take any line of statements that would normally get executed after pressing return:

```
PLOT 0,0:DRAWTO 39,0
```

and prefix it with a number:

```
10 PLOT 0,0:DRAWTO 39,0
```

The same commands, the same editing keys, and yet it's entirely different. It adds the line to the current program as line number 10. Or if line 10 already exists, it replaces it.

Lines are syntax checked as entered. Well, each line is parsed and tokenized so that previous example turns into this:

```
Line #: 10
Bytes in line: 6
PLOT command
   X: 0
   Y: 0
DRAWTO command
   X: 39
   Y: 0
```

That's how the line is stored in memory, provided there aren't any errors. The displayed version is an interpretation of those bytes. Code formatting is entirely handled by the system and not something you think about.

All of this, from the always-available functions, to being able to develop programs without external tools, to code stored as pre-parsed tokens, made BASIC not just a language but a development system. Compare that to most of today's compilers which feed on self-contained files of code. Sometimes there's a run-eval-print loop so there's interactivity, but editing real programs happens elsewhere. And then there are what have come to be known as Integrated Development Environments which tie together file-oriented compilers with text editors and sometimes interactive command lines, but now they get derided for reasons that BASIC didn't: for being bulky and cumbersome.

Did I mention that Atari BASIC was contained in an 8kb ROM cartridge?

How did IDEs go so wrong? ■

James Hague has been Design Director for Red Faction: Guerrilla, editor of "Halcyon Days: Interviews with Classic Computer and Video Game Programmers," co-founder of an indie game studio, and a published photographer. He started his blog "Programming in the 21st Century," in 2007.

Reprinted with permission of the original author.
First appeared in *hn.my/8bitbasic* (dadgum.com)

Writing a Simple Garbage Collector in C

By MATTHEW PLANT

People seem to think that writing a garbage collector is really hard, a deep magic understood by a few great sages and Hans Boehm (et al) [hn.my/boehm]. Well, it's not. In fact, it's rather straight forward. I claim that the hardest part of writing a GC is writing the memory allocator, which is as hard to write as it is to look up the malloc example in K&R.

A few important things to note before we begin. First, our code will be dependent on the Linux kernel. Not GNU/Linux, but the Linux kernel. Second, our code will be 32-bit and not one bit more. Third, please don't use this code. I did not intend for it to be wholly correct, and there may be subtle bugs I did not catch. Regardless, the ideas themselves are still correct. Now, let's get started.

Making the Malloc

To begin, we need to write a memory allocator, or as we will be calling it, a malloc function. The simplest malloc implementations maintain a linked-list of free blocks of memory that can be partitioned and given out as needed. When a user requests a chunk of memory, a block of the right size is removed from the free list and returned. If no blocks of the right size exist, either a block of a larger size is partitioned into smaller blocks or more memory is requested from the kernel. Freeing a chunk of memory simply adds it back to the free list.

Each chunk of memory in the free list begins with a header describing the block. Our header will contain two fields, one indicating the size of the chunk and the second pointing to the next free block of memory:

```
typedef struct header {
    unsigned int    size;
    struct block    *next;
} header_t;
```

Using headers that are embedded in the memory we allocate is really the only sensible way of doing this, but it has the added benefit of automatically word-aligning the chunks, which is important.

Because we will need to keep track of the blocks of memory currently in use as well as the blocks that are not, we will have a used list in addition to a free list. Items will be added to the used list when they are removed from the free list, and vice-versa.

We are almost ready to complete the first step and write our malloc implementation. Before we do that, we first need to understand how to request memory from the kernel.

Dynamically allocated memory resides in the so-called heap, a section of memory between the stack and the BSS (uninitialized data segment — all your global variables that have the default value of zero). The heap starts at a low address bordering the BSS and ends at the program break, which resides somewhere between the BSS and the stack. Attempting to access any memory between the stack and the break will cause an access violation (unless you access within the amount the stack can be extended by, but that's a whole separate conversation). In order to obtain more memory from the kernel, we simply extend the break, thus allowing us to access more memory. To do this, we call the Unix sbrk system call, which extends the break by its argument and returns the address of the previous break on success, thus giving the program more memory. On failure, sbrk returns -1 casted to a void pointer, which is a terrible convention that no one likes.

We can use this knowledge to create two functions: morecore and add_to_free_list. In the case that we are out of blocks in the free list, we will call morecore to request more memory. Since requesting the kernel for more memory is expensive, we will do it in page-size chunks. Knowing what a page is isn't important right now, but a terse explanation is that it is the smallest unit of virtual memory that can be mapped to any particular location in physical memory. We will use the function add_to_free_list to do exactly what it sounds like.

```
/*
 * Scan the free list and look for
 * a place to put the block.
 * Basically, we're
 * looking for any block to be
 * freed block might have been
 * partitioned from.
 */
static void
add_to_free_list(header_t *bp)
{
    header_t *p;

    for (p = freep; !(bp > p && bp
< p->next); p = p->next)
        if (p >= p->next && (bp >
p || bp < p->next))
            break;

    if (bp + bp->size == p->next)
{
        bp->size += p->next->size;
        bp->next = p->next->next;
    } else
        bp->next = p->next;

    if (p + p->size == bp) {
        p->size += bp->size;
        p->next = bp->next;
    } else
        p->next = bp;

    freep = p;
}

#define MIN_ALLOC_SIZE 4096

/* We allocate blocks in page
sized chunks. */

/*
 * Request more memory from the
 * kernel.
 */
static header_t *
morecore(size_t num_units)
{
    void *vp;
    header_t *up;

    if (num_units < MIN_ALLOC_
SIZE)
        num_units = MIN_ALLOC_SIZE
/ sizeof(header_t);

    if ((vp = sbrk(num_units *
sizeof(header_t))) == (void *) -1)
        return NULL;

    up = (header_t *) vp;
    up->size = num_units;
    add_to_free_list (up);
    return freep;
}
```

Now that we have our two helper functions, writing our malloc function is pretty straightforward. We simply scan the free list and use the first block that is at least as big as the chunk we're trying to find. Because we use the first block we find instead of trying to find a "better" block, this algorithm is known as first fit.

A quick note to clarify: the size field in the header struct is measured in header-sized blocks, and not bytes.

```
static header_t base;
/* Zero sized block to get us
started. */
static header_t *usedp, *freep;

/*
 * Find a chunk from the free list
 * and put it in the used list.
 */
void *
GC_malloc(size_t alloc_size)
{
    size_t num_units;
    header_t *p, *prevp;

    num_units = (alloc_size
+ sizeof(header_t) - 1) /
sizeof(header_t) + 1;
    prevp = freep;

    for (p = prevp->next;; prevp
= p, p = p->next) {
        if (p->size >= num_units)
{
/* Big enough. */
            if (p->size == num_
units)
/* Exact size. */
                prevp->next =
p->next;
            else {
                p->size -= num_
units;
                p += p->size;
                p->size = num_
units;
            }

            freep = prevp;

            /* Add to p to the
used list. */
            if (usedp == NULL)

                usedp = p->next =
p;
            else {
                p->next = usedp-
>next;
                usedp->next = p;
            }

            return (void *) (p +
1);
        }
        if (p == freep) {
/* Not enough memory. */
            p = morecore(num_
units);
            if (p == NULL)
/* Request for more memory failed.
*/
                return NULL;
        }
    }
}
```

Note that the success of this function depends on when we first use that freep = &base. We'll make sure this is the case in our init function.

Although this code isn't going to win any awards for low fragmentation, it'll work. And if it works, that means we can finally get to the fun part: the garbage collection!

Mark and Sweep

We did say that the garbage collector was going to be simple, so we will be using the simplest algorithm possible: stop the world naive mark and sweep. This algorithm works in two parts:

First, we scan all the blocks of memory that could possibly point to heap data and see if any do. To do this, for each word-size chunk in the memory we're looking at, we look at each block in the used list. If the word-sized chunk's value is within the range of a used block, we mark the block.

Next, after all possible memory locations have been searched, we go through the used list and add to the free list all blocks that haven't been marked.

Many people (or at least I did) get tripped up into thinking that garbage collection is impossible in C, because by writing a simple function like malloc, there is no way of knowing many things about the outside world. For example, there is no function in C that returns a hash map to all the variables that have been stack-allocated. But we can get by without this by realizing two important facts:

Firstly (gosh I say that a lot), in C, you can attempt to access any memory location you want. There is no chunk of memory that for some reason the compiler can access but has an address that cannot be expressed as an integer and then cast to a pointer. It isn't possible. If memory is used in a C program, it can be accessed by the program. This is a confusing notion for programmers unfamiliar to C, as many languages provide restricted access to virtual memory addresses. C does not.

Secondly, all variables are stored somewhere in memory. Well, duh. But what that means is that if we know generally where the variables are stored, we can look through that memory and find all the possible values of every variable. Additionally, because memory access is generally only word-aligned, we only need to look through every word in the memory regions.

Local variables can also be stored in registers, but we won't worry about this because registers are usually dedicated to local variables, and by the time our function is called they'll probably be saved on the stack anyway.

Now we have a strategy for the marking phase of our collector: look through a bunch of memory regions and see if there is any memory that looks like it references something in

the used list. Writing a function to do that is pretty clear cut:

```
#define UNTAG(p) (((unsigned int)
(p)) & 0xfffffffc)

/*
 * Scan a region of memory and
mark any items in
 * the used list appropriately.
 * Both arguments should be word
aligned.
 */
static void
mark_from_region(unsigned int *sp,
unsigned int *end)
{
    header_t *bp;

    for (; sp < end; sp++) {
        unsigned int v = *sp;
        bp = usedp;
        do {
            if (bp + 1 <= v &&
                bp + 1 + bp->size
> v) {
                    bp->next =
((unsigned int) bp->next) | 1;
                    break;
            }
        } while ((bp = UNTAG(bp-
>next)) != usedp);
    }
}
```

To ensure we only use two words in the header we use a technique here called tagged pointers. Since our next pointers will be word aligned, a few of the least significant bits will always be zero. Thus, we mark the least significant bit of the next pointer to indicate that the current block (not the one pointed to by next!) has been marked.

Now we can scan memory regions, but which memory regions should we look through? There are several:

1. The BSS (uninitialized data segment) and the initialized data segment. These contain all the global and static variables in the program. Thus, they could reference something in our heap.
2. The used chunks. Of course, if the user allocates a pointer to another allocated chunk, we don't want to free the pointed-to chunk.
3. The stack. Since the stack contains all the local variables, this is arguably the most important place to look.

But now there's the problem of knowing where these memory regions start and end. We already know where the used chunks are, but the other regions are trickier.

We already know everything about the heap, so writing a mark_from_heap function is trivial:

```
/*
 * Scan the marked blocks for
 * references to
 * other unmarked blocks.
 */
static void
mark_from_heap(void)
{
    unsigned int *vp;
    header_t *bp, *up;

    for (bp = UNTAG(usedp->next);
bp != usedp; bp = UNTAG(bp->next))
{
        if (!((unsigned int)bp-
>next & 1))
            continue;
        for (vp = (unsigned int *)
(bp + 1);
             vp < (bp + bp->size
+ 1);
             vp++) {
            unsigned int v = *vp;
            up = UNTAG(bp->next);
            do {
                if (up != bp &&
                    up + 1 <= v &&
                    up + 1 + up-
>size > v) {
                    up->next =
((unsigned int) up->next) | 1;
                    break;
                }
            } while ((up =
UNTAG(up->next)) != bp);
        }
    }
}
```

Fortunately for the BSS and initialized data segments, most modern Unix linkers export the etext and end symbols. The address of the etext symbol is the start of the initialized data segment (the last address past the text segment, which contains the program's machine code), and the address of the end symbol is the start of the heap. Thus, the BSS and initialized data segment are located in between &etext and &end. This is simple enough, but not platform independent.

The stack is a little harder. The top of the stack is super simple to find using a little bit of inline assembly, as it is stored in the %esp register. However, we'll be using the %ebp register as it ignores a few local variables.

Finding the very bottom of the stack (where the stack began) involves some trickery. Kernels tend to randomize the starting point of the stack for security reasons, so we can't hard code an address. To be honest, I'm not an expert on finding the bottom of the stack, but I have a few rather poor ideas on how you can make an accurate attempt. One possible way is you could scan the call stack for the env pointer, which would be passed as an argument to main. Another way would be to start at the top of the stack and read every subsequent address greater and handling the inexorable SIGSEGV. But we're not going to do it either way. Instead, we're going to exploit the fact that linux puts the bottom of the stack

in a string in a file in the process's entry in the proc directory (phew!). This sounds silly and terribly indirect. Fortunately, I don't feel ridiculous for doing it because it's literally the *exact same thing Boehm GC does to find the bottom of the stack!*

Now we can make ourselves a little init function. In it, we open the proc file on ourselves and find the bottom of the stack. This is the 28th value printed so we discard the first 27. Boehm GC differs from us in that they only use sys calls to do the file reading in order to avoid the stdlib from using the heap, but we don't really care.

```
/*
 * Find the absolute bottom of the
 * stack and set stuff up.
 */
void
GC_init(void)
{
    static int initted;
    FILE *statfp;

    if (initted)
        return;

    initted = 1;

    statfp = fopen("/proc/self/
stat", "r");
    assert(statfp != NULL);
    fscanf(statfp,
           "%*d %*s %*c %*d %*d
%*d %*d %*d %*u "
           "%*lu %*lu %*lu %*lu
%*lu %*lu %*ld %*ld "
           "%*ld %*ld %*ld %*ld
%*llu %*lu %*ld "
           "%*lu %*lu %*lu %lu",
&stack_bottom);
    fclose(statfp);

    usedp = NULL;
    base.next = freep = &base;
    base.size = 0;
}
```

Now we know the location of every memory region we would need to scan, and thus, we can finally write our explicitly-called collection function:

```
/*
 * Mark blocks of memory in use
 * and free the ones not in use.
 */
void
GC_collect(void)
{
    header_t *p, *prevp, *tp;
    unsigned long stack_top;
    extern char end, etext;
/* Provided by the linker. */

    if (usedp == NULL)
        return;

/* Scan the BSS & initialized data
segments. */
    mark_from_region(&etext,
&end);
```

```
/* Scan the stack. */
    asm volatile ("movl %%ebp, %0"
: "=r" (stack_top));
    mark_from_region(stack_top,
stack_bottom);

    /* Mark from the heap. */
    mark_from_heap();

    /* And now we collect! */
    for (prevp = usedp, p =
UNTAG(usedp->next);; prevp = p, p
= UNTAG(p->next)) {
    next_chunk:
        if (!((unsigned int)
p->next & 1)) {
            /* The chunk hasn't
been marked.
               Thus, it must be
set free. */
            tp = p;
            p = UNTAG(p->next);
            add_to_free_list(tp);

            if (usedp == tp) {
                usedp = NULL;
                break;
            }

            prevp->next =
(unsigned int)p | ((unsigned int)
prevp->next & 1);
            goto next_chunk;
        }
        p->next = ((unsigned int)
p->next) & ~1;
        if (p == usedp)
                break;
    }
}
```

And that, my friends, is all there is. One simple garbage collector written in C and for C. This code isn't complete in itself, it needs a few little tweaks here in there to get it working, but most of the code mostly holds up on its own.

Conclusion

From elementary school through half of high school, I played drums. Every Wednesday at around 4:30pm I had a drum lesson from a teacher who was quite good.

Whenever I was having trouble learning a new groove or beat or whatever, he would always give me the same diagnosis: I was trying to do everything at once. I looked at the sheet of music, and I simply tried to play with all my hands. But I couldn't. And the reason why is because I didn't know how to play the groove yet, and simply trying to play the groove wasn't how I was going to learn.

So my teacher would enlighten me as to how I could learn: don't try playing everything at once. Learn to play the high-hat part with your right hand. Once you've got that down, learn to play the snare with your left. Do the same with the bass, the tom-toms, and whatever other parts there are. When

you have all the individual parts down, slowly begin to add them together. Add them together in pairs, then in threes, and eventually you'll be able to play the entire thing.

I never got good at drums, but I did take these lessons to heart in my programming. It's really hard to just start to type out an entire program. The only algorithm you need to write code is divide and conquer. Write the function to allocate memory. Then, write the function to look through memory. Then, write the function that cleans up memory. Finally, add them all together.

As soon as you get past this barrier as a programmer, nothing practical becomes "hard." You may not understand an algorithm, but anyone can understand an algorithm with enough time, paper, and the right book. If a project seems daunting, break it up into its individual parts. You may not know how to write an interpreter, but you sure as hell can write a parser. Find out what else you need to add, and do it. ■

Matthew Plant is a sophomore undergraduate computer science student at the University of Illinois at Urbana-Champaign. He was born in San Francisco in 1995.

Reprinted with permission of the original author.
First appeared in *hn.my/gcinc* (web.engr.illinois.edu)

How to Read an Executable

How is a Binary Executable Organized?

By JULIA EVANS

I USED TO THINK that executables were totally impenetrable. I'd compile a C program, and then that was it! I had a Magical Binary Executable that I could no longer read.

It is not so! Executable file formats are regular file formats that you can understand. I'll explain some simple tools to start! We'll be working on Linux, with ELF binaries. (Binaries are kind of the definition of platform-specific, so this is all platform-specific.) We'll be using C, but you could just as easily look at output from any compiled language.

Let's write a simple C program, `hello.c`:

```
#include <stdio.h>

int main() {
    printf("Penguin!\n");
}
```

Then we compile it (`gcc -o hello hello.c`), and we have a binary called `hello`. This originally seems impenetrable (how do we even binary?!), but let's see how we can investigate it! We're going to learn what symbols, sections, and segments are. At a high level:

- Symbols are like function names, and are used to answer "If I call `printf` and it's defined somewhere else, how do I find it?"
- Symbols are organized into sections — code lives in one section (`.text`), and data in another (`.data`, `.rodata`)
- Sections are organized into segments

Throughout, we'll use a tool called `readelf` to look at these.

So, let's dive into our binary!

Step 1: open it in a text editor!

This is most naive possible way to view a binary. If I run `cat hello`, I get something like this:

```
ELF>@@H@8
@@@@@@□□88@@@@□□ ((`(`□
PP`P`□□P□td@,,Q□tdR□td((`(`□□/
lib64/ld-linux-x86-64.
so.2GNUGNU□□n□□8□w□j7*oL□h□□
__gmon_start__libc.so.6puts__libc_
start_mainGLIBC_2.2.5ui
1```H□□k□□□□H□□□5 H□[]□fff.□H□=p
UH□□t□H□□]□H`□□]Ð□UH□□□□@□□□□□]
Ð□□□□□□□□□□□□□H□l$□L□d$□H□- L□%
L□l$□L□t$□L□|$□H□\$□H□□8L
)□A□□I□□H□□□I□□□s□□□H□□t1@
L□□L□□D□□A□□H□□H9□u□H□\
H□l$L□d$L□l$
L□t$(L□|$0H□□8□□Ð□□□□□□□□□□□□□□UH□
□SH□H□
H□□□t□(`DH□□□H□H□□□u□H□[]
Ð□H□□o□□□H□□Penguin!;,□□□□H
```

There's text here, though! This was not a total failure. In particular it says "Penguin!" and "ELF." ELF is the name of the binary format. So that's something! Then there are a bunch of unprintable symbols, which isn't a huge surprise because this is a binary.

Step 2: use readelf to see the symbol table

Throughout we're going to use a tool called `readelf` to explore our binary. Let's start by running `readelf --symbols` on it. (Another popular tool to do this is `nm`.)

```
$ readelf --symbols hello
   Num:    Value          Size
Type    Bind   Vis      Ndx Name
    48: 0000000000000000     0
FUNC    GLOBAL DEFAULT  UND puts@@
GLIBC_2.2.5
    59: 0000000000400410     0
FUNC    GLOBAL DEFAULT   13 _start
    61: 00000000004004f4    16
FUNC    GLOBAL DEFAULT   13 main
```

Here we see three symbols: `main` is the address of my `main()` function. `puts` looks like a reference to the `printf` function I called in it (which I guess the compiler changed to `puts` as an optimization?). `_start` is pretty important.

When the program starts running, you might think it starts at `main`. It doesn't! It actually goes to `_start`. This does a bunch of Very Important Things that I don't understand very well, including calling `main`. So I won't explain them.

Symbols

When you write a program, you might write a function called `hello`. When you compile the program, the binary for that function is labelled with a symbol called `hello`. If I call a function (like `printf`) from a library, we need a way to look up the code for that function! The process of looking up functions from libraries is called linking. It can happen either just after we compile the program ("static linking") or when we run the program ("dynamic linking").

So symbols are what allow linking to work! Let's find the symbol for `printf`! It'll be in `libc`, where all the C standard library functions are.

If I run `nm` on my copy of `libc`, it tells me "no symbols." But the internet tells me I can use `objdump -tT` instead! This works! `objdump -tT /lib/x86_64-linux-gnu/libc-2.15.so` gives me this output. [hn.my/objdump]

If you look at it, you'll see `sprintf`, `strlen`, `fork`, `exec`, and everything you might expect `libc` to have. From here we can start to imagine how dynamic linking works — we see that `hello` calls `puts`, and then we can look up the location of `puts` in `libc`'s symbol table.

Step 3: use objdump to see the binary, and learn about sections!

Opening our binary in a text editor was a bad way to open it. `objdump` is a better way. Here's an excerpt:

```
$ objdump -s hello
Contents of section .text:
 400410 31ed4989 d15e4889 e24883e4
f0505449  1.I..^H..H...PTI
 400420 c7c0a005 400048c7 c1100540
0048c7c7  ....@.H....@.H..
 400430 f4044000 e8c7ffff fff49090
4883ec08  ..@.........H...
Contents of section .interp:
 400238 2f6c6962 36342f6c 642d6c69
6e75782d  /lib64/ld-linux-
 400248 7838362d 36342e73 6f2e3200
x86-64.so.2.
Contents of section .rodata:
 4005f8 01000200 50656e67 75696e21
00        ....Penguin!.
```

You can see that it shows us all the bytes in the file as hex on the left, and a translation into ASCII on the right.

There are a whole bunch of sections here. This shows you all the bytes in your binary! Some sections we care about:

- `.text` is the program's actual code (the assembly). `_start` and `main` are both part of the `.text` section.
- `.rodata` is where some read-only data is stored (in this case, our string "Penguin!")
- `.interp` is the filename of the dynamic linker!

The major difference between sections and segments is that sections are used at link time (by `ld`) and segments are used at execution time. `objdump` shows us the contents of the sections, which is nice, but doesn't give us as much metadata about the sections as I'd like. Let's try `readelf` instead:

```
$ readelf --sections hello
Section Headers:
  [Nr] Name              Type
Address           Offset
       Size              EntSize
Flags  Link  Info  Align
  [13] .text             PROGBITS
0000000000400410  00000410
       00000000000001d8
0000000000000000  AX       0     0
16
  [15] .rodata           PROGBITS
00000000004005f8  000005f8
       000000000000000b
0000000000000000   A       0     0
4
  [24] .data             PROGBITS
0000000000601010  00001010
       0000000000000010
0000000000000000  WA       0     0
8
  [25] .bss              NOBITS
0000000000601020  00001020
       0000000000000010
0000000000000000  WA       0     0
8
  [26] .comment          PROGBITS
0000000000000000  00001020
       000000000000002a
0000000000000001  MS       0     0
1
Key to Flags:
  W (write), A (alloc), X (exe-
cute), M (merge), S (strings), l
(large)
  I (info), L (link order), G
(group), T (TLS), E (exclude), x
(unknown)
  O (extra OS processing required)
o (OS specific), p (processor
specific)
```

Neat! We can see `.text` is executable and read-only, `.rodata` ("read only data") is read-only, and `.data` is read-write.

Step 4: Look at some assembly!

We mentioned briefly that `.text` contains assembly code. We can actually look at what it is really easily. If we were magicians, we would already be able to read and understand this:

```
Contents of section .text:
 400410 31ed4989 d15e4889 e24883e4
f0505449  1.I..^H..H...PTI
 400420 c7c0a005 400048c7 c1100540
0048c7c7  ....@.H....@.H..
 400430 f4044000 e8c7ffff fff49090
4883ec08  ..@.........H...
```

It starts with `31ed4989`. Those are bytes that our CPU interprets as code! And runs! However we are not magicians (I don't know what `31 ed` means!), and so we will use a disassembler instead.

```
$ objdump -d ./hello
Disassembly of section .text:

0000000000400410 <_start>:
  400410:       31 ed
xor    %ebp,%ebp
  400412:       49 89 d1
mov    %rdx,%r9
  400415:       5e
pop    %rsi
  400416:       48 89 e2
mov    %rsp,%rdx
  400419:       48 83 e4 f0
and    $0xfffffffffffffff0,%rsp
```

So we see that `31 ed` is xoring two things. Neat! That's all the assembly we'll do for now.

Step 5: Segments!

Finally, a program is organized into segments or program headers. Let's look at the segments for our program using `readelf --segments hello`.

```
Program Headers:
  [... removed ...]
  INTERP
0x0000000000000238
0x0000000000400238
0x0000000000400238

0x000000000000001c
0x000000000000001c  R      1
      [Requesting program inter-
preter: /lib64/ld-linux-x86-64.
so.2]
  LOAD
0x0000000000000000
0x0000000000400000
0x0000000000400000

0x00000000000006d4
0x00000000000006d4  R E    200000
  LOAD
0x0000000000000e28
0x0000000000600e28
0x0000000000600e28

0x00000000000001f8
0x0000000000000208  RW     200000
  [... removed ...]

 Section to Segment mapping:
  Segment Sections...
   00
```

```
   01     .interp
   02 .interp .note.ABI-tag .note.
gnu.build-id .gnu.hash .dynsym
       .dynstr .gnu.version .gnu.
version_r .rela.dyn .rela.plt
.init .plt
       .text .fini .rodata .eh_
frame_hdr .eh_frame
   03     .ctors .dtors .jcr
.dynamic .got .got.plt .data .bss
   04     .dynamic
   05     .note.ABI-tag .note.gnu.
build-id
   06     .eh_frame_hdr
   07
   08     .ctors .dtors .jcr
.dynamic .got
```

Segments are used to determine how to separate different parts of the program into memory. The first `LOAD` segment is marked R E (read / execute) and the second is `RW` (read/ write). `.text` is in the first segment (we want to read it but never write to it), and `.data`, `.bss` are in the second (we need to write to them, but not execute them).

Not magic!

Executables aren't magic. ELF is a file format like any other! You can use `readelf`, `nm`, and `objdump` to inspect your Linux binaries. Try it out! Have fun. ■

Julia Evans likes programming, playing with data, and finding out why things that seem scary actually aren't. She lives in Montreal and works on Stripe's machine learning team.

Reprinted with permission of the original author.
First appeared in *hn.my/exe* (jvns.ca)

Don't Become a Scientist!

By JONATHAN KATZ

Are you thinking of becoming a scientist? Do you want to uncover the mysteries of nature, perform experiments, or carry out calculations to learn how the world works? Forget it!

Science is fun and exciting. The thrill of discovery is unique. If you are smart, ambitious, and hardworking you should major in science as an undergraduate. But that is as far as you should take it. After graduation, you will have to deal with the real world. That means that you should not even consider going to graduate school in science. Do something else instead: medical school, law school, computer science, engineering, or something else which appeals to you.

Why am I (a tenured professor of physics) trying to discourage you from following a career path which was successful for me? Because times have changed (I received my Ph.D. in 1973, and tenure in 1976). American science no longer offers a reasonable career path. If you go to graduate school in science, it is in the expectation of spending your working life doing scientific research, using your ingenuity and curiosity to solve important and interesting problems. You will almost certainly be disappointed, probably when it is too late to choose another career.

American universities train roughly twice as many Ph.D.'s as there are jobs for them. When something, or someone, is a glut on the market, the price drops. In the case of Ph.D. scientists, the reduction in price takes the form of many years spent in holding pattern postdoctoral jobs. Permanent jobs don't pay much less than they used to,

but instead of obtaining a real job two years after the Ph.D. (as was typical 25 years ago) most young scientists spend five, ten, or more years as postdocs. They have no prospect of permanent employment and often must obtain a new postdoctoral position and move every two years.

For example, consider two of the leading candidates for a recent Assistant Professorship in my department. One was 37, ten years out of graduate school (he didn't get the job). The leading candidate, whom everyone thinks is brilliant, was 35, seven years out of graduate school. Only then was he offered his first permanent job (that's not tenure, just the possibility of it six years later, and a step off the treadmill of looking for a new job every two years). The latest example is a 39 year old candidate for another Assistant Professorship; he has published 35 papers. In contrast, a doctor typically enters private practice at 29, a lawyer at 25 and makes partner at 31, and a computer scientist with a Ph.D. has a very good job at 27 (computer science and engineering are the few fields in which industrial demand makes it sensible to get a Ph.D.). Anyone with the intelligence, ambition, and willingness to work hard to succeed in science can also succeed in any of these other professions.

Typical postdoctoral salaries begin at $27,000 annually in the biological sciences and about $35,000 in the physical sciences (graduate student stipends are less than half these figures). Can you support a family on that income? It suffices for a young couple in a small apartment, though I know of one physicist whose wife left him because she was tired of repeatedly moving with little prospect of settling down. When you are in your thirties you will need more: a house in a good school district and all the other necessities of ordinary middle class life. Science is a profession, not a religious vocation, and does not justify an oath of poverty or celibacy.

Of course, you don't go into science to get rich. So you choose not to go to medical or law school, even though a doctor or lawyer typically earns two to three times as much as a scientist (one lucky enough to have a good senior-level job). I made that choice, too. I became a scientist in order to have the freedom to work on problems which interest me. But you probably won't get that freedom. As a postdoc you will work on someone else's ideas, and may be treated as a technician rather than as an independent collaborator. Eventually, you will probably be squeezed out of science entirely. You can get a fine job as a computer programmer, but why not do this at 22, rather than putting up with a decade of misery in the

scientific job market first? The longer you spend in science, the harder you will find it to leave, and the less attractive you will be to prospective employers in other fields.

Perhaps you are so talented that you can beat the postdoc trap; some university (there are hardly any industrial jobs in the physical sciences) will be so impressed with you that you will be hired into a tenure track position two years out of graduate school. Maybe. But the general cheapening of scientific labor means that even the most talented stay on the postdoctoral treadmill for a very long time; consider the job candidates described above. And many who appear to be very talented, with grades and recommendations to match, later find that the competition of research is more difficult, or at least different, and that they must struggle with the rest.

Suppose you do eventually obtain a permanent job, perhaps a tenured professorship. The struggle for a job is now replaced by a struggle for grant support, and again there is a glut of scientists. Now you spend your time writing proposals rather than doing research. Worse, because your proposals are judged by your competitors you cannot follow your curiosity, but must spend your effort and talents on anticipating and deflecting criticism rather than on solving the important scientific problems. They're not the same thing: you cannot put your past successes in a proposal, because they are finished work, and your new ideas, however original and clever, are still unproven. It is proverbial that original ideas are the kiss of death for a proposal; because they have not yet been proved to work (after all, that is what you are proposing to do) they can be, and will be, rated poorly. Having achieved the promised land, you find that it is not what you wanted after all.

What can be done? The first thing for any young person (which means anyone who does not have a permanent job in science) to do is to pursue another career. This will spare you the misery of disappointed expectations. Young Americans have generally woken up to the bad prospects and absence of a reasonable middle class career path in science and are deserting it. If you haven't yet, then join them. Leave graduate school to people from India and China, for whom the prospects at home are even worse. I have known more people whose lives have been ruined by getting a Ph.D. in physics than by drugs.

If you are in a position of leadership in science then you should try to persuade the funding agencies to train fewer Ph.D.'s. The glut of scientists is entirely the consequence of funding policies (almost all graduate education is paid for by federal grants). The funding agencies are bemoaning the scarcity of young people interested in science when they themselves caused this scarcity by destroying science as a career. They could reverse this situation by matching the number trained to the demand, but they refuse to do so, or even to discuss the problem seriously (for many years the NSF propagated a dishonest prediction of a coming shortage of scientists, and most funding agencies still act as if this were true). The result is that the best young people, who should go into science, sensibly refuse to do so, and the graduate schools are filled with weak American students and with foreigners lured by the American student visa. ■

Jonathan Katz is a professor of physics at Washington University in St. Louis. He has worked in many branches of physics, including astrophysics, plasma physics and applied physics, both in academia and at National Laboratories. His present work is in high energy astrophysics and rheology.

Reprinted with permission of the original author.
First appeared in *hn.my/scientist* (physics.wustl.edu)

The 1,000-Hour Rule

By PHILIP GUO

I'm not yet qualified to give general life advice to kids, but I would like to share one simple piece of advice that I would've liked to hear when I was a kid:

Find something you genuinely enjoy doing for its own sake, stick with it, keep learning more about it, and after a decade or so, you can't help but get good at it and feel proud of yourself.

This is my own personal take on the popular 10,000-hour rule, which claims that it takes around 10,000 hours of intense practice to become an expert in a particular topic. For instance, top-notch musicians, artists, athletes, scientists, and other experts in their respective fields all share a common experience: They practiced consistently and with high intensity for over 10,000 hours, often starting at a young age. That amounts to practicing for 4 hours every weekday for a decade straight, which takes tremendous passion and perseverance.

I don't want to pressure you to become an expert at anything, but I do want you to become good at something, which I think requires far less time and dedication. So here is my 1,000-hour rule, a much gentler version of the 10,000-hour rule: I claim that it takes roughly 1,000 hours of practice to get good at something. 1,000 hours of practice over a decade amounts to roughly 2 hours each week, which is far more sustainable than 4 hours each day. Surely you can spare 2 hours each week for some hobby. So find something you genuinely enjoy doing for its own sake, stick with it, keep learning more about it, and after a decade or so, you can't help but get good at it and feel proud of yourself.

> **“A decade from now, you will think that it’s pretty cool that you’ve grown fairly good at something that you’ve consistently kept up over many years.”**

A Bit Of Narcissism

To give a personal example, the website you are now viewing is the primary hobby that I have maintained throughout the past decade. I now consider myself fairly good at creating personal websites and writing online articles.

I wasn’t always proud of my website design and writing. I’ve only begun receiving some praise for my website in the past few years, but I’ve been making websites for over a decade, starting back in 1997. My first few attempts looked pretty horrendous (e.g., MS Paint and stock clip art, 3-D naked men). Some people have told me that they think I have some sort of talent for writing and website design. I can guarantee that nobody told me that when I first started at age 13! I was a horrible writer back in middle school: I got C and D grades on some of my essays. And I had absolutely no sense of visual aesthetics.

However, I learned to improve my writing and visual design sense throughout the past decade of working on my website in my spare time. I never remembered spending a tremendous amount of time doing so, or forgoing my school or work obligations. It was a hobby that I could do whenever I felt in the right mood. Thus, if you pursue a hobby consistently over a decade, even if you only put in a few hours each week, I guarantee that you can’t help but get pretty good at it! It’s fairly easy to rack up your 1,000 hours when spread over an entire decade.

I think I’ve surpassed my 1,000 hours by now, but I’m not nearly at the 10,000 hours which are indicative of true expertise. I don’t consider myself an expert writer or website designer by any means. I am merely a hobbyist, albeit one who has pursued this hobby over many years. However, even as a hobbyist, I’ve managed to hone some

deep intuitions about this craft that simply can't be gleaned from perusing a book or how-to guide. There is simply no substitute for consistent practice over time, and no easy way to get good overnight. However, it is pretty easy to get good if you just stick with something you like doing over a long period of time.

Take-Home Message

But enough about me. You don't care about how my hobby has been personally fulfilling for me; you only care about what you can gain out of reading this article.

My advice is simple: Find something that you enjoy doing and then keep doing it. Maintain a constant curiosity to learn more about it, and do it purely as a hobby without any ulterior motives. A decade from now, you will think that it's pretty cool that you've grown fairly good at something that you've consistently kept up over many years. And the best part is that it won't ever be difficult or strenuous!

Becoming a world expert in something takes 10,000 hours of shedding blood, sweat, and tears, at times even driving you to the brink of insanity, but simply becoming good at a hobby takes much less time and emotional investment. Very few people can have both the potential and opportunity to develop into world experts, but simply becoming good at a skill is well within almost everyone's reach. ■

Philip Guo is an assistant professor of computer science at the University of Rochester. His main research interests are in human-computer interaction (HCI), with a focus on user interfaces for online learning. He created a free Web-based tool for learning programming called Online Python Tutor [pythontutor.com], which has been used by over 500,000 people in over 165 countries. He has previously researched online education at edX and Google.

Reprinted with permission of the original author.
First appeared in *hn.my/1000hr* (pgbovine.net)

It's Just Wood

By BRYAN KENNEDY

A few months back, I decided to start a vegetable garden. I have fond memories of gardening with my folks when I was a young boy, and I figured a garden would be a good outlet from my day to day life in tech.

Looking up and down the aisles at Lowes and not finding a raised garden bed that met my needs, I inquired with a staff member:

"Sorry, we don't carry anything like that."

"Darn. Ok, I'll try another store then," I sighed in resignation.

"You know," he said to me, "it's just wood."

It took me a moment to realize how liberating this statement was. To him, woodworking held no mystery at all — it was just a thing you did. If he wanted a garden bed, there'd be no question in his mind that he'd make it himself. Why spend more on something suboptimal when you can build exactly what you want?

So I did exactly that, and we now have a wonderful, custom-built gardening bed growing some very tasty kale.

Most people let mystery stop them

We resign to not knowing how to change the oil in our car, replace a busted light switch, or develop that app idea into a real app. "You need to be an expert to do that," we say. Thinking back to the times I've challenged mystery and learned to do something myself, I recall it giving me great satisfaction. And once demystified, those tasks that seemed impossible before now seem trivial.

Did you know that there are over 110K videos on YouTube about replacing a light switch? Or that Hack Reactor claims 99% job placement rate after graduating their 12-week developer program? The resources you need to overcome the mysteries around you are plentiful.

Challenge mystery

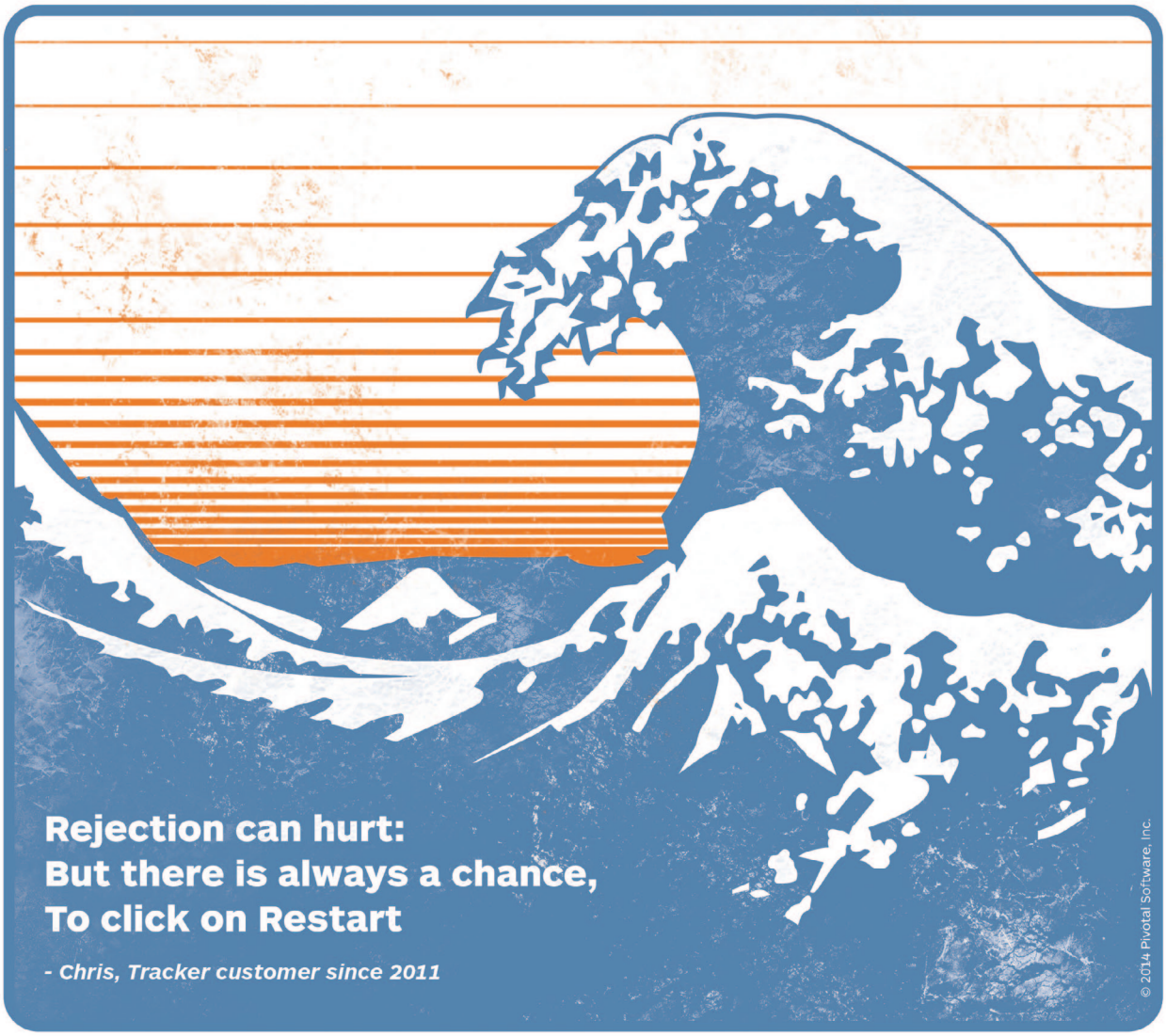
When you take it upon yourself to learn how to do something, you've not only become more self-reliant, but you've expanded your mind to new possibilities. If I had settled for a pre-built planter box, I wouldn't have been able to construct exactly the kind of box I wanted, nor improve the design along the way to fit my needs. Now the next time I want to construct something out of wood, I have the confidence to know that I can figure it out.

The best part? The more you challenge the mysteries in your life, the better you'll get at it.

What will you demystify? ■

Bryan Kennedy is the Co-Founder and CTO of Sincerely, helping to scale thoughtfulness across the world. Bryan is a YCombinator alum and an angel investor. On warm summer nights he runs *MobMov.org*, a worldwide collective of guerrilla drive-ins.

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



Discover the newly redesigned Pivotal Tracker

As our customers know too well, building software is challenging. That's why we created Pivotal Tracker, a pleasure-to-use project management tool, designed to facilitate constructive communication, keep teams focused and reflect the true status of all your software projects.

With a new UI, cross-project functionality, in-app notifications and more, staying zen in the face of looming business deadlines just got a little easier.

Sign up for a free trial, no credit card required, at pivotaltracker.com

PivotalTracker

Build better software, faster.