Modeling with Data: the blog

Ben Klemens

December 19, 2011

Dear Reader,

There are some places in the world that typically have limited internet connectivity: your bed, the bus, the bathroom, the beach. Yet you still want to read this blog in those places. In fact, blogs are really best left for those down-time situations when you didn't just sit down in front of a computer to work.

So, here is the PDF version of the blog, with everything in one place, so you can follow the full thread of the story in whatever manner you prefer.

I could have changed everything around to read better on paper, but chose not to. Instead, you're getting the raw, exciting feel of the Internet, in a paper format.

0.1 Today I am a blog

6 March 2009

Hi.

This is the blog for my book, Modeling with Data. That means that I'll be discussing statistics, scientific inquiry, computing, academia, publishing, and what I had for lunch before sitting down to write.

Let me tell you why I'm setting up another blog: because *statistics is amazing*. Seriously enthralling. Pure mathematics is internally consistent, but comfortably ignores what we like to call reality; statistics is a field that does the dirty work of linking mathematical models with observed phenomena. This is a *hard problem*, not in the sense that with enough elbow grease we can solve it, but in the sense that we mortals are fundamentally incapable of a definitive solution.

Of course, we do our best anyway. What are the odds that it will rain tomorrow? Your weatherman will be happy to give you a number, even though just explaining what that question means is about impossible.

Modeling with Data grazed the surface of these questions, but it's a textbook, so I focused on specific techniques and details rather than the social, technological, and even philosophical issues underlying the techniques. I will not be talking about the politics of information, because I already have a blog about that. Also, despite being the author of a statistics textbook, I'm human, and can perhaps share with you some interesting ideas about writing, publishing, and doing research in the present day.

1 Description versus inference

1.1 Too many tests

16 March 2009

This is allegedly a blog to accompany *Modeling with Data*, so I don't feel too bad repeating its opening paragraph:

> Statistical analysis has two goals, which directly conflict. The first is to find patterns in static: given the infinite number of variables that one could observe, how can one discover the relations and patterns that make human sense? The second goal is a fight against *apophenia*, the human tendency to invent patterns in random static. Given that someone has found a pattern regarding a handful of variables, how can one verify that it is not just the product of a lucky draw or an overactive imagination?

It's the first paragraph in the book because this conflict is just that important. If you're in an inferential mindset while working on a descriptive technique, or vice versa, you'll be hopelessly confused. If you design a study so that it is as descriptive as possible, you can easily lose inferential power, and vice versa. The conflict is also a social conflict, and you'll find a lot of examples of yelling between a descriptively-oriented person on one side and an inferentially-oriented person on the other.

I'll be giving you many, many examples of how the descriptive-inferential conflict plays out, and why it's important for reading the newspaper, gathering data, and teaching. But for now, let me just clarify the point a little by giving you the most common example and the most common point of conflict: selecting the number of hypothesis tests to run.

First, here are two questions:

• Randomly draw a person from the U.S. population. What are the odds that that person makes more than $1m/year?

• Randomly draw a million people from the U.S. population. What are the odds that that wealthiest person in your list makes more than $1m/year?

The odds in the second case will be much higher, because we took pains in that one to pick the wealthiest person we could. [That is, the first is a hypothesis about just data, the second is a hypothesis about an order statistic of data.]

Now say that you have a list of variables before you.

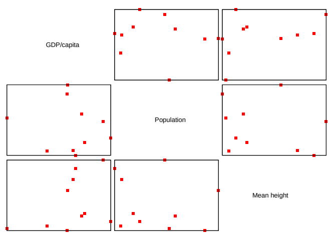

Figure 1.1: A lattice plot, relating three variables to each other

• Claim that A is correlated to B_1. What are the odds that your claim will pass the appropriate test with more than 95% confidence?

• Write down the best correlation between A and B_1, B_2, …, $B_{1,000,000}$. What are the odds that your best result will pass the appropriate test with more than 95% confidence?

You can sit down at your computer and run the same correlation test in the first example with B_1 and in the second example with B_{best}, and both tests will have the same name and produce the same format of output from your software, but you've just run two entirely different tests. Just as with the case of our wealthiest individual, the best result from a million results is very likely much better than any single result. Even a million hypothesis tests over noise are likely to find results that are very significant. [There are disciplines, techniques and tricks to mitigate the problem, but I won't get into those now. E.g., see the Bonferroni correction, on pp 318–319 of *Modeling with Data*.]

So the context of a test matters, in sometimes subtle ways. This creates friction between the descriptives and the inferentials, because the descriptives are building tools to search for the best relationships among as much data as possible, while the inferentials realize that those same tools can diminish our power to have confidence in those best relationships.

Next time, I'll talk about how this relates to some currently trendy aspects of dataviz. After that I'll reapply this abstract point to academia at large, and Freakonomics-type journalism in particular.

1.2 The two sides of the statistical war

6 August 2008

This is a continuation of last episode (p 3).

That said, let's start with a little exercise.

The first figure is a Trellis$^{\text{TM}}$ or lattice plot, giving a 2-D dot plot of each of three variables against each other variable. I didn't try too hard in producing the plot, and just pulled out three variables at random from a random data set.

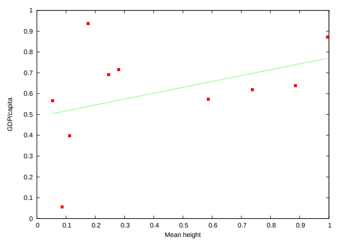

Figure 1.2: A close-up of the upper left plot in the lattice, with the line of best fit

But we can already see some patterns: GDP/capita and height have the positive correlation you'd expect, as per the blow up in the next figure. In this figure, I fit a linear regression to the data, and it looks pretty good, but for a few outliers at lower right. Maybe an exponential-family model may fit better.

So that's DataViz at work. We took a lot of data, displayed many relations at once, and zeroed in on one that matters.

Except, uh, for all that I said about this being a random data set. I just made up some pleasant-sounding variable names, generated a random data set, and plotted it. And yet we were able to find a plausible pattern in there.

And so we see another way of casting the descriptive versus inferential war—the problem of too many hypothesis tests. The descriptivists are working to produce methods like the lattice plot that let you see more relationships at once; the inferentialists are asking: if you fed complete noise to this method, what are the odds that some sort of pattern would turn up? As our methods get better at putting more data on the screen at once, they get worse at testing whether the patterns we see are real or just beautiful noise.

DataViz Thanks to a number of technological advances, dataViz is trendy right now. There are a few icons of the field who are working hard on self-promotion, such as Edward Tufte, whose books show how graphs can be cleaned up, chartjunk eliminated, and grainy black and white fliers from the 1970s cleaned up through the use of finely detailed illustrations in full color. John Tukey's Exploratory Data Analysis (cited above) is aggressively quirky, and encourages disdain for the inferential school.

These guys, and their followers, are right that we could do a whole lot better with our data visualizations, and that the stuff based on facilitating fitting the line with a straightedge should have been purged at least twenty years ago. Strunk and White gave us standards for writing clearly in 1959; it's about time we developed guidelines for exposition via graphics.

But we're talking not just about presenting a known relationship, but exploratory data analysis via graphics. In this context, the underlying philosophy is humanist to a fault. The claim is that the human brain is the best data-processor out there, and our computers still can't *see* a relationship among a blob of dots as quickly as our eye/brain combo can. This is true, and a fine justification for better graphical data presentation. And hey, we humans would all rather look at plots than at tables of numbers.

But apophenia is a powerful force. We look at clouds and see bunnies, or read the horoscope and think that it's talking directly to us, or listen to a Beatles song about playground equipment and think it's telling us to kill people. Given a handful of scatterplots like the lattice plot above, you *will* find a pattern—in fact, if a psychologist were to show you a series of ten seemingly random inkblots[1] and you didn't see a reasonable number of patterns in them, the psychologist might consider you to be mentally unhealthy in any of a number of ways.

The moral here is that our data visualization technology is getting really good really fast—I'll have even slicker examples next time. You'd be silly to ignore these recommendations and novel display methods. But the same power that makes patterns clear is the power that invents random patterns in static.

Next time: even more dataviz tools, which touch on an even bigger problem.

[1] `http://ar.geocities.com/test_de_rorschach/index.htm`

Figure 1.3: If you don't see faces, you're crazy. Oh, and there's a penis and vagina in every inkblot too.

1.3 Crowdsourcing data mining

24 March 2008

In the last episode (p 4), I wrote about how dataviz tools take an extreme position in the descriptive-versus-inferential balance, by giving you the option of eyeballing every possible test, and then picking the one that works best. But you can't run *every* test, because your time is limited. The solution: crowdsource! Put your data up on iCharts[2], iFree3d[3], Many Eyes[4], Swivel[5], Timetric[6], Track-n-Graph[7], Trendrr[8], or Widgenie[9], and then let others try every test that seems sensible to them.

The dicta of the dataviz gurus offer plenty of good advice for presenting a known result. When writing a good essay, you want every sentence to help the reader understand the essay's conclusion; when producing a good plot, every inkblot should help the reader understand the plot's conclusion. In that context, having easily accessible tools where you can just drop in your data and get many types of well-designed plots is fabulous.

But the promotional copy for these sites—and even the name *Many Eyes*—suggests that these aren't just tools for effectively plotting the results of research, but allow the crowdsourcing of the search for patterns. This is where problems arise, because as per last episode, this is a recipe for finding both patterns that are and are not present.

Now, to be fair, if you really wanted to just snoop around for the best fit, the time constraint I'd alluded to above is not a particularly serious problem. It's not very hard to write a loop to try every possible combination of a set of variables and report which set provides the best fit. [I'd actually written up the try-every-regression loop as an exercise in *Modeling with Data*, but cut it because I thought it was too cynical.]

Here's my favorite interview[10] regarding this issue. The testing-oriented interviewer came as close as politely possible to asking the people who first developed the lattice display, Rick Becker and Bill Cleveland, whether this display method treads too close to data snooping:

> *Interviewer:* OK, but there is another way to approach the study of a large database: develop a statistical model and see if it fits the data. If it does fit, use the model to learn about the structure of the data.
>
> *Becker & Cleveland:* Yes, and Trellis display is a big help in doing this because it allows you to make a good guess about an initial model to fit and then to diagnose how well it fits the data. [. . .]
>
> *Interviewer:* But instead of agonizing over all those panels I could do a bunch of chi-squared tests for goodness of fit.

[2]http://icharts.net
[3]http://ifree3d.com
[4]http://services.alphaworks.ibm.com/manyeyes/
[5]http://www.swivel.com/
[6]http://timetric.com/
[7]http://www.trackngraph.com/
[8]http://www.trendrr.com/
[9]http://widgenie.com/
[10]http://stat.bell-labs.com/project/trellis/interview.html

Becker & Cleveland: You're joking, right? If not, we're leaving.

Interviewer: OK, I guess I'm joking.

To a descriptive person, looking at a Trellis^{TM} plot is nothing like looking at a matrix of goodness-of-fit statistics—that's certainly not how it feels. But the two activities are in the end closely correlated, and if a regression line looks good on the plot, it has good odds of passing any goodness-of-fit tests.

Many eyes But let me get back to the problem of crowdsourcing the process of trying every combination of variables.

Say that one researcher finds the middle ground in the descriptive/inferential range. She comes in with some idea of what the data will say, rather than waiting for the scatterplot of Delphi to reveal it, and then refines the original idea in dialog with the data (and good plots of the data). The researcher is not on a pure fishing expedition, but she is not wearing blinders to what the data has to say.

So one researcher could be reasonable—but what happens when there are thousands of reasonable researchers? When a relevant and expensive data set has been released, a large number of people will interrogate it, each with his or her own prior expectations. I've been to an annual conference attended by about a hundred people built entirely around a single data set, and who knows how many weren't able to fly out. With so many researchers looking at the same set of numbers, *every reasonable hypothesis will be tested.* Even if every person maintains the discipline of balancing data exploration against testing, we as a collective do not.

Every person was careful to not test every option, so none would seem to be mining the data for the highest statistical significance. But collectively, a thousand hypothesis tests were run, and journals are heavily inclined to publish only those that scored highly on the tests. So it's the multiple testing problem all over again, but the context is the hundreds or thousands of researchers around the planet studying the same topic. Try putting *that* into a cookbook description of a test's environment.

So our tests just aren't as powerful as we think they are, because we're not taking into account the true, collective context. Both halves of the descriptive/inferential balance are essential, but the inferential side is increasingly diluted and weakened by the scaling-up of our descriptive powers. There's no short-term solution to this one, though in an episode or two, I'll discuss some band-aids.

1.4 Freakophenia

6 August 2008

Let me start with an example. You may have read in the New York Times that obesity is contagious[11], in the sense that you're more likely to be obese if your friends are. The linked article is reporting a publication [Christakis and Fowler, 2007] from the New England Journal of Medicine (NEJM), one of the most well-regarded journals around, which retains is high regard via a press office that puts out press releases

[11]`http://www.nytimes.com/2007/07/25/health/25cnd-fat.html`

on notable articles in each issue (as do many other journals). I made a point of not closely reading the article, and not critiquing the methods; I'm fine with believing that they were good enough to pass peer review, made honest use of the data, and that the statistical significance claimed is a correct read of the data.

But from this subsequent rebuttal [E and Fletcher, 2008]: "We replicate the NEJM results using their specification and a complementary dataset. We find that point estimates of the 'social network effect' are reduced and become statistically indistinguishable from zero once standard econometric techniques are implemented." That is, the results were basically an artifact of the original authors' data and methods, and statistical significance disappears upon replication.

So it goes. Maybe another study will come by and re-replicate. But right now it seems that the initial proposal was a matter of what I'd been discussing in a prior episode (p 4): if you have enough researchers staring at one data set—and we know there's a critical mass of researchers working on obesity and on social networks—then eventually one researcher will verify any given hypothesis.

This isn't improper behavior of any sort on the part of the authors of the original study, the NEJM, the NYT, or the many people who re-reported the results after they appeared in newspapers. But the publication system is built around the new and exciting, which is by definition the stuff that hasn't been replicated or seriously verified. After all, *New study verifies results of study that's already been out for a year* just doesn't count as news. Because of the novelty premium, it's easy to publish—and publicize—a study that seems statistically significant but happened to work out only because of luck and the volume of researchers staring at the problem.

There are some ways by which non-results can get published. In the example above, we saw a null-result rebuttal to a paper that found a positive result. That is, once a positive result appears, null results become newsworthy (in the academic sense. I don't think the NYT published anything about the failure to replicate the obesity headline). There is finally a Journal of Articles in Support of the Null Hypothesis[12] aimed at dealing with this very problem (which they call the "file drawer problem," because a study that gets significant results gets published, and a study that fails to reject the null winds up in the file drawer).

In medicine, there is the funnel plot, which plots all of the p-values for a hypothesis from several studies, and then draws a theoretical symmetric funnel around the points; the gaps in the ideal funnel are assumed to be missing (i.e., unpublished) papers. This is done in medicine and not other fields because medicine has enough studies on a single question that you could do this sort of thing.

But Freakonoscience doesn't have that luxury: it's all about quirky one-off studies that have zero attempts at replication. So we don't have funnel plots, or any other easy tools to tell us what confidence to place in the results trumpeted in the headlines. As in prior episodes, even though the reported confidence levels are correct for the researcher's context, they are not correct for the reader's larger context, which should include both this one study and all those others that may or may not have been published. In the larger context, we basically have nothing.

In an episode or two, some notes on how we can respond to this problem.

[12] `http://www.jasnh.com/`

2 EUGENICS AND GENETICS

2.1 Your genetic data

6 April 2009

[Or, *The ethical implications of SQL.*]

The figure explains how my work in statistical genetics is all possible. It is what a genetics lab looks like. That's a work bench, like the ones upon which thousands of pipettes have squirted millions of liters of fluid in the past. But you can see that it is now taken up by a big blue box, which hooks up to a PC. Some of these big boxes use a parallel port (like an old printer) and some run via USB (like your ventilator or toothbrush). The researcher puts processed genetic material in on the side facing you in the photo, onto a tray that was clearly a CD-ROM drive in a past life. Then the internal LASER scans the material and outputs about half a million genetic markers to a plain text file on the PC.

I know I'm not the first to point this out, but the study of human health is increasingly a data processing problem. My complete ignorance regarding all things biological isn't an issue in doing analysis, as long as I know how to read a text file into a database and run statistical tests therefrom.

Implication one: Research methods Historically, the problem has been to find enough data to say something. One guy had to sail to the Galapagos Islands, others used to wait for somebody to die so they could do dissections, and endless clinical researchers today post ads on bulletin boards offering a few bucks if you'll swallow the blue pill.

But now we have exactly the opposite problem: I've got 18 million data points, and the research consists of paring that down to one confident statement. In a decade or so, we went from grasping at straws to having a haystack to sift through.

I've got tools printed in textbooks from the 1970s that will eke out a relationship from a minimum of data, and 5GB of genetic data regarding people with bipolar disorder over on the other screen. Applying one to the other will give me literally thousands of ways of diagnosing bipolar disorder, none of which are in any way trustworthy.

So the analytic technology is not quite there yet. There's a specific protocol for drawing blood that every nurse practitioner knows by heart, and another protocol for

Figure 2.1: The tools of the data processing field known as Biology

breaking that blood down to every little subpart. We have protocols for gathering genetic data, but don't yet have reliable and standardized schemes for extracting information from it.

When we do have such a protocol—and it's plausible that we soon will—that's when the party starts.

Implication two: Pathways If you remember as much high school biology as I do, then you know that a gene is translated in human cells into a set of proteins that then go off and do some specific something (sometimes several specific somethings).

If you know that a certain gene is linked to a certain disorder, then you know that there is an entire pathway linked to that disorder, and you now have several points where you could potentially break the chain. [Or at least, that's how it'd work in theory. Again, there's no set protocol.] There are many ways to discover the mechanism of a disorder, but the genetic root is the big fat hint that can make it all come together right quick. So the scientists still working with squishy biological organisms are also keeping their eye on the database-crunchers for clues about what to poke at.

Then, once there's consensus on a pathway, the drug companies go off and develop a chemical that breaks the destructive chain, and perhaps make a few million per year in the process.

Implication three: Free will versus determinism One person I talked to about the search for genetic causes thought it was all a conspiracy. If there's a genetic cause for mental illness, then that means that it's not the sufferer's fault or responsibility. Instead of striving to improve themselves, they should just take a drug. And so, these genetic studies are elaborate drug-company advertising.

From my casual experience talking to folks about it, I find that this sort of attitude is especially common regarding psychological disorders. See, every organ in the human body is susceptible to misfiring and defects—*except* the brain, which is created in the image of '', and is always perfect.

Annoyed sarcasm aside, psychological disorders are hard to diagnose, and there's a history of truly appalling abuse, such as lobotomies for ill behavior, giving women hysterectomies to cure their hysteria, the sort of stories that made *One Flew over the Cuckoo's Nest* plausible, &c. Further, there are often people who have no physiological defect in their brains, but still suffer depression or other mood disorders. They get some sun, do some yoga, and everything works out for them.

But none of that means that the brain can not have defects, and that those defects can not be treated.

The problem is that our ability to diagnose is falling behind our ability to cure. We know that certain depressives respond positively to lithium, Prozac, Lexapro, Wellbutrin, Ritalin, Synthroid, and I don't know today's chemical of the month. But we still don't have a system to determine which are the need-of-drugs depressives and which are the get-some-sun depressives.

Or to give a physical example, we don't know which obese individuals have problems because of genetic barriers and which just need to eat less and exercise. It's only harder because, like the brain, the metabolism is an adaptive system that can be conditioned for the better or for the worse, confounding diagnosis. Frequently, it's both behavior and genetics, albeit sometimes 90% behavior and other times 90% genes.

A genetic cause provides genetic tests. If we have a drug based on a genetic pathway, as opposed to a drug like Prozac that just seemed to perk people up, we can look for the presence or absence of that genetic configuration in a given individual. This ain't a silver bullet that will sort people perfectly (if that's possible at all), but having a partial test corresponding to each treatment is already well beyond the DSM checklists we're stuck with now.

From here, there are ethical implications, which I'll save for next time.

2.2 Your genetic data—ethics

8 April 2009

In the last episode (p 11), I wrote about how biology is slowly moving from the gathering of limited data about organisms to relying on mass genotyping. Even the biologists working entirely with squishy organisms are looking to the genetic databases for clues and cues. Here are a few more implications to that transition.

Implication four: Eugenics We can test for genetics not only among adults and children, but even fetuses. On one small survey, five out of 76 British ethics committee members (6.6%) "thought that screening for red hair and freckles (with a view to termination) was acceptable."[1]

Fœtal gene screens to determine Down syndrome or other life-changing conditions are common, and 92% of fetuses that return positive for the test for Down Syndrome

[1] `http://adc.bmj.com/cgi/content/full/88/7/607`

are aborted [Mansfield et al.].

Biology has an embarrassing past in eugenics. And we're not just talking about the Nazis—the USA has a proud history of eugenics to go along with its proud history of hating immigrants (I mean recent immigrants, not the ones from fifty years ago, who are all swell). [The lead author of my last paper refers me to this article on eugenics[2], and having read it I too recommend the first 80%.]

If I may resort to a dictionary definition, the OED tells us that eugenics is the science "pertaining or adapted to the production of fine offspring, esp. in the human race." In the past, that meant killing parents who turned out badly in life or had big noses, but hi-tech now allows us to go straight to getting rid of the offspring before anybody has put in too heavy an investment.

Anyway, I won't go further with this, but to point out that what we'll do with all this fœtal genetic info is an open question—and a loaded one, since the only choices with a fœtus are basically carry to term or abort. The consensus seems to be that aborting due to Down syndrome is OK and aborting due to red hair is not, but there's a whole range in between. If you know your child has a near-certain chance of getting Alzheimer's 80 years after birth, would you abort? [This Congressional testimony[3] approximately asks this question.]

Implication five: the ethics of information aggregation This is also well-trodden turf, so I'll be brief:

• It is annoying and stupid that every time you show up at the doctor's office, the full-time paperwork person hands you a clipboard with eight papers, each of which asks your name, full address, and Social Security Number. By the seventh page, I sometimes write my address as "See previous pp" but they don't take kindly to that, because each page goes in a different filing cabinet.

You may recall Sebadoh's song on data and database management: "You can never be too pure/ or too connected." If all of your information is in one place, either on your magical RF-enabled telephone or somewhere in the amorphousness of the web, then that's less time everybody wastes filling in papers and then re-filling them in when the bureaucrat mis-keys everything. I have a FOAF whose immigration paperwork was delayed for a week or two because somebody spelled her name wrong on a form.

• Having all of your information in one place makes it easier for people to violate your privacy and security. As advertisers put it, it makes it easier to offer you goods and services better attuned to your lifestyle, which is the nice way of saying 'violate your privacy'. It means more things they can do to you on routine traffic stops.

The data consolidation=efficiency side is directly opposed to the data disaggregation=privacy side. There is no solution to this one, and both sides have their arguments. The current compromise is to consolidate more and put more locks on the data, but that doesn't work very well in practice, as one breach anywhere can ruin the privacy side of the system.

Back to genetics, when we have a few more snips of information about what all those genes do, your genetic info will certainly be in your medical records. This is a

[2]`http://www.logosjournal.com/issue_6.1-2/jacobsen.htm`

[3]`http://www.hhs.gov/asl/testify/t960917c.html`

good thing because it means that those who need to will be able to diagnose you more quickly and efficiently; it is a bad thing because those who don't need to know may also find a way to find out personal information about you.

At the moment, you can rely on the anonymity of being a needle in a haystack, the way that some people who live at the top of high rise buildings are comfortable walking around naked and with the curtains open—who's gonna bother to look? But as the tools and filters and databases become more sophisticated, the haystack may provide less and less cover.

So we're going to have a haystack of data about you (and your fœtus) right soon. Unfortunately, we don't quite yet know how to analyze, protect, or act on that haystack.

3 Why Word is a terrible program

First of all, it is time to speak some truth to power in this country: *Microsoft Word is a terrible program.*

[...For example,] there is the moment when you realize that your notes are starting to appear in 12-pt. Courier New. Word, it seems, has, at some arbitrary point in the proceedings, decided that although you have been typing happily away in Times New Roman, you really want to be in the default font of the original document. You are confident that you can lick this thing: you painstakingly position your cursor in the Endnotes window (not the text!, where irreparable damage may occur) and click Edit, then the powerful Select All; you drag the arrow to Normal (praying that your finger doesn't lose contact with the mouse, in which case the window will disappear, and trying not to wonder what the difference between Normal and Clear Formatting might be) and then, in the little window to the right, to Times New Roman. You triumphantly click, and find that you are indeed back in Times New Roman but that all your italics have been removed. What about any of this can be considered high-speed?

From *The end matter* by Louis Menand, The New Yorker, issue of 2003-10-06.

If you are a casual user, then Word is probably fine for you—maybe even ideal. It's great that there is a product out there that will help complete novices and your proverbial Aunt Myrtle to produce beautiful documents.

But for many office workers, regardless of their job title, their actual occupation is "Word user". They come in at nine-ish in the morning, edit documents in Word for eight hours, and go home. If that describes you, if half of your waking life is spent staring at that program, then you no doubt have a strong interest in working out how to use your most-used tool efficiently. Maybe the right tool for Aunt Myrtle is not the right tool for you, and your life would be better if you could jump ship for something better.

I realize that Word is a standard that many of you are forced to use by your employers or colleagues. After you send your boss or colleague a copy of this paper, you may want to have a look at Section 3.6 for practical suggestions on what you can do to remain compatible with your coworkers while not letting them drag you down into inefficiency.

I recently wrote a book entitled *Math You Can't Use: Patents, Copyright, and Software*, so I should clarify that that book and this document are 100% unrelated. That book was about the law and politics of software, but this paper isn't: this paper is about efficiency and usability, and I promise you minimal discussion of the politics of Microsoft in the pages that follow.

By the way, my publisher asked me to write that book in Word, so I have ironic first-hand experience with using Word to write a complex document.

3.1 Semantic editing

The key failing of Word is the difficulty of semantically-oriented editing.

The way most of us format a document in a word processor is to change the formatting of individual elements as we need them. Titles need to be marked in boldface; there needs to be this much space put between paragraphs; the margins should be just so on the cover page. I will call this *literal markup*, where you make changes on the screen until the text looks the way you want it to look.

The alternative is to specify what each element actually *means*, and then worry about how the formatting happens later. Mark the titles as <title>, mark the paragraphs as <text>, and mark the cover page as <cover>. Then, write a style sheet that lists rules that titles should be bold, that text has this much space between paragraphs, and that the cover page's margins are extra-wide. Then, the computer knows to apply the formatting described in the style sheet to your document.

The benefits to semantic markup are immense. First, your boss's boss is going to tell you to change your titles to italics instead of bold as soon as she sees the document. In the semantic system, you change the definition of a title element in one place and you're done; in the literal markup system, you need to go through the entire document and change every title individually—and then repeat when your boss's boss decides that no, you were right, it does look better in bold.

And did you catch the title on page sixty-eight down at the bottom? With semantic markup, because you didn't change fifty points in the document, you don't have to worry about whether you are still consistent or not. More generally, it is by construction impossible to have inconsistent style with a semantic markup scheme, because you define each style exactly once. With literal markup, you need to be on guard for consistency all the time.

In the literal markup world, you wear two hats at the same time: author and typesetter. In the semantic world, you wear the hats one at a time. When working on content, you are not distracted by stylistic junk. Some would describe working only on content and putting off stylistic issues until the very end to be "no fun", but it is certainly more efficient. Of course, you are welcome to play around with the style sheet between writing every other sentence if you so desire.

The document you are working on now is probably not the only document you are writing during your career. Once you have a visual style that you like, you can save those definitions of titles, text, and cover sheet to use on every future document. In the literal markup world, you have to redo the margins on every cover page every time, duplicating a few minutes' effort with every document.[1]

Along a similar vein, your company has a standard letterhead, and may have a graphics department that would prefer all documents to have a consistent style. In semantic-land, your company can distribute a single style sheet and ask that everyone apply it (even if they think it looks ugly; there are always dissenters in this system). In literal markup-land, the graphics department sends out a list of fifty rules everyone must follow when setting up their cover page, wasting everyone's time and guaranteeing that half of the rules won't get followed.

Finally, it is increasingly important that your document be available as a PDF, a web page, and in your company's legacy TPS format. The semantic system, done right, is output-independent. You send the same document to one program that marks up titles appropriate for the printed page, and another that marks it up for the web. If you had to do literal markup appropriate for both the web and paper, it will look terrible in one or the other, and you'd just wind up writing and maintaining two documents.

Semantic markup is hands-down the way to format a document. It doesn't take any more cognitive effort or ability to mark up your title with \title{Intro} or <title>Intro</title> than it does to mark it up to read as **Intro**, so semantic markup provides all of the above benefits over literal markup for basically no cost.

Semantic markup v WYSIWYG Too bad Word is written from the ground up as a program for literal markup. If you read the manual, you will see that there exists a style editor, which claims to allow semantic markup. It lets you define paragraph types and character types, like a title style or a text style.

So the first tip, should you be using Word, is to use the style editor. Avoid hard-coding any sort of formatting; instead, define a style and apply that style.

But it's not entirely that easy, because Word will try its hardest to frustrate you. Word thinks it is smarter than you, so it will often guess the style you mean to be applying to a line, and sometimes revert styles back to where they were. Each element can have only one paragraph and one character style, but there are often reasons to apply multiple styles at once (blockquote on the cover page, italics in a title). Be careful to note where your styles are being saved. If they are being saved to your `normal.dot` template, then when you send your document to your colleagues, your styles won't go with.[2] Best of luck cutting and pasting between documents with different style sheets. There *is* a style organizer that allows you to move style sheet elements from one document to another; it is well-hidden (ask the paperclip for it) but it works. If you want one style for the web version of your document and one for print, you want too much.

The markup is always invisible: you need to click on the item while the style editor is up and then scroll through to see what is highlighted. This may seem trivial, but is frustrating if you have multiple, subtly different styles. And you will, because Word eagerly tries to manage the style list for you. If you italicize an item currently in the

[1]There are of course tricks, like cutting and pasting the cover sheet from past documents and hoping the formatting follows. Even when they work, you can see that they are still inefficient relative to applying a style sheet.

[2]Tip #2: write yourself a template. Depending on whether the Earth is in a Fire sign or a Water sign, you may need to send the template with your document.

title style, then it will autogenerate a title-1 style. Now, when you change all of your titles to non-bold, the lone item in title-1 may or may not follow along.

In short, you *can* use Word for semantic markup, but only with discipline and patience. Word is a literal markup system, and the style editor is your window on Word's internal means of organizing literal markup. It is not a full-blown semantic style sheet, as shown by its little failings above.

The bibliography Louis Menand's opening comments are from an essay entitled "The End Matter", and spends much time describing the futility, misery, and in the end, impossibleness of writing a bibliography.

> Annotation may seem a mindless and mechanical task. In fact, it calls both for superb fine-motor skills and for adherence to the most exiguous formal demands [...] and the combination is guaranteed to produce flawed page after flawed page. In the world of End Matter, there is no such thing as a flyspeck. Every error is an error of substance, a betrayal of ignorance and inexperience, the academic equivalent of the double dribble.

If you are writing bibliographies by hand, as Mr Menand is doing as a Word user, you are wasting your life. Remembering where to put the commas, what to italicize, when to use the full first name and when to use initials, is the sort of work that a computer does easily and that we humans have trouble doing perfectly. Word's demand that users need to hand-edit their bibliographies has no doubt cost the world literally millions of person-hours.

The correct way to do a bibliography is via a database. You provide one entry for each reference, typing out the author, the title, the publisher, et cetera. Then, the computer reads the database and puts the result in your document according to a style sheet such as that published by the University of Chicago or the APA. That is, the best means is via semantic mark-up: you tell the system that "Joe Guzman" is the author, and leave it to the computer to decide whether to print "Guzman, J", "Guzman, Joe", "J. Guzman" or what-have-you on the page.

OpenOffice.org wins points for including a bibliography editor. LaTeX includes bibtex. Word does not include one, although you can purchase one from a third party for a hundred dollars or so.[3]

3.2 Multiple views

A major contribution of the IT era has been to allow multiple views of the same work. Products are becoming less like a single book, which is a fixed, indivisible object, and more like a song, which is merely a brief view on something for which hundreds of other views exist.

Your favorite pop song was probably recorded onto 24 tracks, and then loaded onto a sound mixer such as Audacity, a program built around facilitating multiple views of

[3]Of course, if you use an add-on like this, you won't be able to email your document to a colleague (or yourself at another computer) for editing unless the recipient has also paid out the hundred dollars for the same program.

the music. There's a visual representation of the sound and of course the noise the thing makes. The data is multilayered, and the user can view/hear the final work with some layers processed, muted, inverted, et cetera.

Users of the GIMP or Photoshop are familiar with the same process: there are all these layers, and each can be viewed differently, separated, filtered. At some point, you set a fixed view and publish it as the final image, but with both the visual and audio systems, the base version holds much more information than the final view.

Databases have what is literally called a view, but even without them, users are encouraged to think in terms of the root data existing in the database and what's on the screen being a slice of it. The root data needs to be taken care of, but mangle the view all you want; it's disposable. HTML is the markup language used by web pages; it is plain text but then viewed via a cute renderer like Internet Explorer or Firefox. Your file browser will let you look at the pile of bits on your hard drive sorted by name, type, or size, or a number of other slices.

Almost all information processing in all media takes the views-of-base-data form. But there are three hard-and-fast exceptions to the paradigm: spreadsheets, word processors, and presentation software. There is a picture of the page on the screen, and that's the document. There are few ways to view the work differently when you're working on it than when the final output will be printed or displayed on somebody else's screen. With due creativity, you can find the outline view or other marginal shifts [by which I mean you can choose whether the page margins appear on the screen or not], but for the most part, these systems work by vehemently insisting that there be only one view. All of the document's information is present in all versions.

And that is one more reason why Word is a terrible program: it constrains the user in the classical paradigm of "one work, one view". Can I distribute drastically different views of the same work to different people? They would just be two different works, that you'll have to maintain separately. When I present to the world congress, is there a clean version that I can use? Nah, just hit <F9> to blow up your working copy to full-screen size.

This is clearly what many people wanted, and most are happy with it. Multiple views are overkill for simple documents. The two-bit philosophy questions of which is the true version of the work evaporate; the conceptual structure of a root object which is viewed in different ways flattens out; our documents are just like they were in the 70s, but backlit. But being stuck in the seventies means that there is a clear and evident ceiling in efficacy, because the ideal view for working on a project matches the output view in only the most simple and lucky of circumstances.

The view issue dovetails with the semantic markup issue from last time, because semantic markup by definition means working with markup and then producing beautiful output from that. For example, a bibliography editor or database gives a straight-information view of the bibliography, and then the system produces a different view of the data for the APA, Chicago, or Harvard stylebook. But if we have only a one work = one view paradigm, we can't work on content in one view and formatting in a separate view.

HTML gives us some hope for the future here, because the actual work is done in a markup language and the output has myriad possible views—and people have no problem with the concept. One person can read the work on his big-screen browser, one can read it on his telephone, and one can hear it on her text-to-speech reader, and all agree that they've read the same document.

Commenting A compounding failing of Word is that it won't let me insert the wealth of expletives that it inspires in me. With the one work = one view paradigm, there is nowhere for me to leave personal comments to myself that won't go into the public version. I often leave notes about sources, the full excerpt of a passage I edited down, or other side-notes about note-worthy bits. Without properly enforced private and public views, I'd need to either keep a side document or just throw out this info.

Computer programmers have a trick known as 'commenting out'. Because the computer ignores anything marked as a comment, a coder can mark functioning lines of code as a comment to see how the program would run if that line were eliminated. It's a sort of purgatory for code that should maybe be deleted, but the judgement hasn't yet been handed down. Similarly, I write more than enough prose that is maybe a bit too verbose to put in the final work. I am reluctant to delete it, because I spent ten minutes composing that paragraph, but commenting it out is painless and reversible.

In Word, I am unable to leave personal notes to to guide myself, and I am unable to comment out sections that should probably be deleted. That is, Word gives me fewer tools to write with so that it can enforce its intuitive paradigm.

3.3 Intuition and lying to the user

A book entitled *Design of Everyday Things*, by Donald A Norman [Norman, 1988] very clearly had an influence on the design of many of Microsoft's products. It in turn was influenced by what was trendy at the time (1988): the original Macintosh features prominently, there is a whole page on the promise of hypertext, and he complains about EMACS. In his section on Two Modes of Computer Usage, he explains that there's a third-person mode wherein you give commands to the computer, and then the computer executes them; and there's a first-person mode where you do things your own darn self, like telling the computer to multiply matrix A by matrix B versus entering numbers into the cells of a spreadsheet. At the ideal, you can't tell that you're using a computer; the intermediary dissolves away and it just feels like working on a problem. Of course, some tasks are too hard for first-person execution, as Mr. Norman explains: "I find that I often need first-person systems for which there is a backup intermediary, ready to take over when asked, available for advice when needed." This paragraph, I posit without a shred of proof, is the genesis of Clippy the Office Assistant.

Although Mr. Norman points out that we feel more human and less like computer users when we are in first-person mode, it is often a terribly inefficient way to work. A word-processor document is *not* like handwriting a letter, so pretending it is is sometimes folly. For example, you don't hard-code numbers: instead of writing Chapter 3, you'd write Chapter \ref{more_rambling} (LaTeX form; Word has a similar thing), and let the computer work out what number goes with the more_rambling reference.[4]

[4]I used the LaTeX markup for Chapter \ref{more_rambling} here because it saves me the trouble of having to explain the seven-step process it takes to do the same thing in Word. And by the way, if you change the referred-to chapter's title, all of the references will break and you'll have to repeat the seven-step

In the context above, first-person mode matches literal markup. Don't write a note to the computer that it should find all titles and boldface them; instead, go and boldface them all the way you would if you had a highlighter and paper in hand. Third-person commands are inhuman, unintuitive, and how we get computers to make our lives easier and more efficient.

Forcing the user DOET has much to say about saving the user from him, her, or itself. Make it impossible to make errors, he advises designers. His shining example of good design are car doors that can only be locked from the outside using the key. There's a trade-off of some inconvenience, but it is absolutely impossible to lock the car keys inside. Word clearly fails on this one: you want to hard-code your references? Feel free; in fact, we'll make it hard for you to do otherwise, since doing otherwise doesn't follow the metaphor of simply writing on paper.

More generally, a good design has restrictions: if you can only put your hand in one place on the door's surface, then that's where you'll put your hand, and the door will open on the first try. What about LaTeX? It gives you a blank page. You can type a basically infinite range of possibilities. This is where DOET leaves the command line: it isn't restrictive enough to guide the user, and therefore is a bad design.

I think he's got the interpretation entirely wrong: there is only one thing that you can do with the blank slate that you get in EMACS, LaTeX, or a command line: read the manual (RTFM). Just as your car won't let you lock yourself out, you can't write a crappy document in LaTeX until you've gotten a copy of the manual and at least had half a chance to expose yourself to the correct way to do things. Mr. Norman again: "Alas, even the best manuals cannot be counted on; many users do not read them. Obviously it is wrong to expect to operate complex devices without instruction of some sort, but the designers of complex devices have to deal with human nature as it is." True, people won't read manuals unless you force them to. So force them to.

Ease of initial use The benefit of the intuitive interface is that you don't have to read the manual.[5] You can jump in and go. Aunt Myrtle only writes one letter a month, so making her spend an hour reading the introduction manual—which she will entirely forget by next month—is inefficient and bad design.

But ease of initial use is only important for those items that we only use once or occasionally. Think of the things you use every day: your preferred means of transport may be an automobile, a bicycle, or your shoelaces. You spend all day typing with a QWERTY keyboard. Perhaps you play a musical instrument. The fact that you are reading this indicates that you are literate. None of these things are intuitive. You spent time (in some cases, years) learning how to do them, and now that you did, you enjoy driving, riding, playing, and reading without thinking about the time you spent practicing.

Simply put, not having to read the manual is massively overrated. If a person is going to use a device for several hours every day for the next year or even the next

process for each reference.

[5]By the way, I rarely find intuitive interfaces to *actually* be intuitive. They're designed around certain target users whom I'm evidently incapable of thinking like. More generally, the concept of having an intuitive interface assumes that the intuition of everybody on Earth is exactly the same.

decade, then for them to spend an hour, and maybe even weeks, learning to use the device efficiently makes complete sense. More on this important point later.

Metaphor shear Another problem is what Neal Stephenson calls *metaphor shear*. That's when you're happily working with a mental model in the back of your mind, and one day your metaphor breaks. Back to DOET: "Three different aspects of mental models must be distinguished: the *design model*, the *user's model*, and the *system image* [...]. The design model is the conceptualization that the designer had in mind. The user's model is what the user develops to explain the operation of the system. Ideally, the user's model and the design model are equivalent. However, the user and designer communicate only through the system itself: its physical appearance, its operation, the way it responds, and the manuals and instructions that accompany it. Thus, the *system image* is critical; the designer must ensure that everything about the product is consistent with and exemplifies the operation of the proper conceptual model."

This is where DOET overestimates computing. It's a book that's mostly about doors and faucets and other everyday objects. He's right that if you have to RTFM to work a door (even if the manual just says *Push*), the door's design is broken. He's right that for complex systems, like panels of airline instruments, they should not work *against* intuition (e.g., if two levers do different things, they should look different). But he combines them into a false conclusion: complex systems should work with intuition so well that you shouldn't have to read the manual.

First, this is absurd in any setting but desktop computers. Would you feel OK if your pilot told you the plane was so intuitive that she didn't bother learning how to use it before the flight?

But back to the main point, making a word processor which is so intuitive to the user that he or she doesn't have to RTFM is a *much* more complex task than making a manual-less faucet. If we needed to build a faucet such that it runs if the user presses it with his hand, bangs it with a pot, or bumps it with his elbow, that would be easy—put a button on the top. But to program a picture of a faucet such that the user can click on the thing, or double-click on the thing, or type R and all make the picture of a faucet run requires programming a call to the Run method for three separate events. If the user comes up with something that the programmer didn't think of, like holding down the alt key and clicking on the picture, then the user's metaphor shears. What your momma told you is true: it's easier to just present the truth than to weave a whole world around a lie.[6]

Mr. Norman's call for simple interfaces (he doesn't really say anything about metaphors to physical objects, but he does talk about simple mental models, and for most of us that means physical metaphors) therefore leads us down a supremely difficult path: first, the program designer must lie to the user by presenting a metaphor that is easy for the user to immediately guess at. Then, the designer must now design the program so that anything the user does, no matter how unpredictable, will cause the program to

[6]For those down with the lingo: every event has to have a method for every object, which is dozens of events times dozens of objects equals hundreds of things that could go wrong with the metaphor—assuming you got good rules about passing the right events to the right objects to begin with. Inheritance doesn't help because most of the time the inherited methods don't quite work as they should, leaving you with objects which almost fit the metaphor.

behave in the correct metaphorical manner. This is a very high bar, to the point that a program as complex as Word simply can not achieve it.

Feature creep Mr. Norman is right that we shouldn't have to RTFM for simple, everyday tasks. Writing a letter or one-page paper is so common that his principle that it should be manual-less should probably apply. Further, we have the technology. However, as I've learned ever-so-painfully, writing a book is an order of magnitude more technically difficult. Programs like Word and Scientific Word imply that writing a letter and a book are are identical, just a matter of extent, when in the end they aren't: one has a valid paper metaphor attached, which programmers can easily implement, and one does not. A good word processor, then, would let you do basic things without effort, and then put its foot down at some point. You get all the tools you need to write a business letter, and then if you want more, you'll need to get a new tool with a manual. Clearly, nobody is ever going to write a program like this. To some extent, this is a good thing, since it pushes technology forward, but at the expense of annoying users who have to sit through half-appropriate metaphors badly implemented. Mr. Norman writes about creeping featurism as an evil which pervades all of design, and he's right: nobody ever says "I'm done."[7]

One good way to implement this would be a simple graphical front end to the basic features of a less metaphor-laden back-end program. When you've sapped the offerings of the graphical front end, you'll have a bearing when you RTFM on the less intuitive stuff. This is how a host of Unixy programs work, but the front ends also eventually succumb to featurism. Scientific Word takes it to the extreme, by trying to give you a button for every last feature and refusing to admit that it is a front-end—perhaps because it is an expensive front-end to free software.

Since no programmer will ever have the discipline to admit that their manual-less tool will work only for a limited range of tasks, the discipline falls upon the user to realize that it's OK to use simplifying metaphors for simple situations, but complex tasks require tools that don't lie to you.

Word is carefully built from the ground up to be intuitive, not to be efficient—and it lies to you every step of the way to give the impression that the system actually works the way you intuitively guess it does. The next section describes how even the smallest intuitive but inefficient detail can add up to immense time costs in a system you use all day, every day.

3.4 Ergonomics

[I wrote this in late 2005:] Google recently put out an RSS reader. It's pretty cute, and I personally have switched to it.

If you aren't familiar with RSS, then that is no matter here (it's a syndication system for web sites). The interesting feature of the reader for our purposes is that the J key will let you go down in the list of headlines. Yes, J, as in, uh, *jo down.* K, as in *kup*

[7]There is a stand-out exception to this: TeX was done in 1988, after nobody claimed the author's cash prize for finding bugs, and the code base has not changed since then. The add-on, LaTeX, was cemented in 1994. Authors who want to change something in the system must add a package to the base systems.

goes up in the list. There is absolutely nothing mnemonic about the J and K keys, but they *feel wonderful.* I assume you knows how to type properly, with hands on the home keys; I generally find my hands are on the home keys even when I'm just staring at the screen, and my hand doesn't need any help from my brain to find the little nubbin on the J key.

But that J key. It's the index finger of 90% of the world's dominant hand, and the keyboard is designed so that that index finger knows exactly where to rest. Moving down on the page is the most common operation, both in reading and even editing, so it makes complete ergonomic sense to attach this to the strongest finger of the strongest hand. Even the lefties will have no problem with it.

But it flies in the face of all mnemonics. Maybe you can come up with some word having to do with the process of scrolling down that begins with the letter J, but I've got nothin'. Nor could I think of a more efficient keymap.

I personally think the use of the J key is easy to learn because of its ergonomic delight. But it throws ease of initial use out the window—almost belligerently. You want to use the nifty hotkeys? Then RTFM.

An interface which works *against* intuition can be destructive, so if U went down and D went up, we'd have to write off the application as hopeless, but J doesn't work against anything. It's just a gesture.

Within a week of Google's RSS rollout, Bloglines, a competing RSS aggregation service, added a little header to its page: "You can now navigate through Bloglines with hotkeys[...]: j - next article k - previous article [...]"

Anybody familiar with the OpenOffice.org internals? bdamm (at) openoffice (dot) org will give you a hundred bucks to write code to have J move the cursor down a line (plus a handful of other keystrokes like K).

The war Lest you think this J thing is some sort of recent meme, it all comes from vi, a text editor written in 1976. I am using a version of vi (named vim) to write this right now. Let's pause for a second and let that sink in: most programs have a shelf life of about six months, and this guy wrote a program thirty years ago which is still in somewhat common use today. j goes down, k goes up, {`jfw` will go to the first instance of the letter w in your paragraph, and, since I can't stand seeing that unclosed open-bracket, I have to tell you that }`j%d%` will delete a parenthetical remark in the first line of the next paragraph. Which is all to show you that Mr. Joy, the author of vi, fell soundly on the efficiency side of the efficiency vs intuition scale—and that is why his text editor has survived for thirty years, and is being imitated by cutting-edge web services.

We sometimes like to write documents that actually have Js in them, and vi thus has modes: in editing mode, j goes down and d$ will delete the rest of the line; in insert mode, the j key puts a j on the screen, and typing d$ puts gibberish on the screen which quickly reminds you you're in the wrong mode.

There are two competitors to J. The first is the ctrl-D school, rooted in EMACS, written by a certain Mr. RM Stallman. EMACS's keymap is sort of like vi's, in that it's not particularly intuitive, but once you've learned it, you're done. However, it's a compromise along the efficiency vs intuition scale, because you don't need to deal with the unintuitive modes but reaching for the ctrl key all the time is not nearly as pleasant as twitching your index finger to hit the j key.[8] The EMACS vs vi war is a long-standing one, which is just silly, because they're of basically comparable efficiency. No, there are other schools that are a real drain on the economy, like the down-arrow school.

Let me take a paragraph or two to make this as clear as possible: the down-arrow school is a total failure when it comes to efficiency. On my screen right now, getting to the first w in the last paragraph via arrow keys is 27 keystrokes (using ctrl-arrow to go by word where possible). It's about three or four seconds for a single navigation. Do forty three-second navigations in a day and you're already up to nine hours in a work-year—a full work day a year just hitting the arrow key. You get to multiply by your wage to see what your company is spending per annum to facilitate ease of initial use. Even if it's one tap of the arrow key, your hands are already off the home keys; going off and on again is another half-second. If you do a hundred arrow-key navigations in a day (and if you're an office worker who does a lot of writing, you probably do closer to a thousand), that's another full work day a year just moving your right hand back and forth between the arrow keys and the home keys.

There is only one school that fails with such vehemence that it makes the down-arrow school look like Nirvana: the mouse school. In the mouse school, you take one hand—typically your dominant hand—off of the keyboard entirely, reaching to some part of the desk that is ergonomically suboptimal (because your keyboard is already in the optimal location). You position your hand on the mouse, and then move the cursor along the screen. It is an analog device, so aim and precision matter, meaning that some people simply do not have the eyesight and dexterity to use the mouse at all: try getting Aunt Myrtle to highlight the letter *i* in a font where that letter is one pixel wide. You guide the mouse to the pixels that are by the word you want to change, click, carefully drag, and return your hand to the keyboard. The entire process can easily take more than four or five seconds, just to position the cursor. And if you have to scroll through the document to find the point, that's easily ten seconds as prelude to a single edit.

The rabidness of the aforementioned text editor wars comes from the fact that text editing absorbs a huge amount of one's life. If you're like most office drones, most of your time at the computer is spent writing and editing plain text—and you're just one office drone; there are millions in the U.S.A. who are all operating computers basically identical to yours, using a down-arrow/mouse school text editor of some sort. Sure, there are people doing flashy data-slinging with big servers, but the bulk of computing is the literally billions of person hours per year spent editing text. Now multiply that half-second to move the right hand to the arrow key; at this scale, it adds up to millions of person-days per year spent on making that little twitch. With an entirely straight face, I can say that on the order of a billion dollars per year is spent on paying people to hit arrow keys.

When the programmer guys got together and wrote whatever it is you use to write your documents and navigate your web pages, they had all of the paradigms above at hand. Half of these guys are using EMACS or vi themselves. We get frustrated when we ask Mr. Computer Geek for help and he (always a boy, eh) comes back with over-everyone's-head exposition about just opening up regedt and doing a quick ctrl-f for

[8]The joke is that EMACS stands for escape meta alt control shift.

HKEY {343-f2ea53e}. Less blatant but just as insidious is when Mr. Geek assumes you are an idiot. He knows that he knows more about PCs than you do, therefore you are dumb and wholly incapable of learning the reams of knowledge that he has compiled. I have been at many a workplace with IT departments that are stocked with such people; it's only some vestige of courtesy that keeps them from installing drool-guards on all the company keyboards.

Of course, the IT department is thinking about the worst-case users. But when was it ever efficient to force everybody in a several-hundred person organization to work with exactly the tools that the least-able could work with? You may have a legally blind worker at your workplace, but that doesn't mean that every computer in the building needs to operate exclusively at super-magnified resolution. A reasonable approach would be a system where you could select between the various schools of navigation. Most versions of vi let you do this (and EMACS allows ctrl-D, down-arrow, and a limited j-mode), but few down-arrow school programs include the wealth of editing keystrokes that those programs provide.

And so I take Google's j and k keys as a slight victory in a long battle against the forces of condescension. It's just two keys, a far cry from a word processor with a full vi keymap, but it's a sign that the guys who designed and programmed the system felt that it was more important to make usage efficient than to make it drool-proof. As such, it gives me hope that maybe the software of the future might focus on long-term efficiency over the quick sell.

Formatting and ergonomics Beyond editing, all this applies to formatting in Word too, because you have to use the mouse or an absurd amount of tabbing and arrowing to navigate the menus and dialog boxes to get to the option you want to change. For almost every step of the way, Word eagerly picks intuition over efficiency.

Of course, the most commonly-used features, like boldface, have their own ctrl-key combination, to at least save the user mouse and arrow-key inefficiency for the dozen most commonly-used operations. Also, you can use alt-F to access the File menu, alt-E to use the Edit menu, et cetera.

But even having the few control-key combinations you do have creates problems, because there are only 26 control-letters to use. If they are taken up with the typesetting features of Word, then they can't be used for the plain old editing of text. EMACS and vi give the user fifty-odd keystrokes that edit text (I'm guessing because I couldn't possibly count them all); Word gives you cut, paste, copy, and that's about it. For every other editing task, you have to make do with the arrow keys. The majority of your time putting together a paper is spent writing and editing, so having so many keystrokes at your fingertips for formatting but almost none for editing is backward.

There is no place in Word's intuitive editing model for a key combination to delete a word at a time, to repeat the last edit, to jump to wherever you were last working, or to switch a lowercase letter to capital. But such keystrokes provide immense speed gains to users who have taken the time to learn them. But which do you do more often in a day: skip back to the beginning of a sentence, or switch to boldface?

One reason we have so many formatting commands is—once again—the lack of style sheets, which means that formatting is not produced by listing what you want the formatting to look like, but by applying it over and over again, which means that keystrokes to apply formatting are competing with editing keystrokes for frequency of use. It would be nice to have a dedicated editing program plus a separate dedicated formatting program, but Word's DOC format precludes this.

3.5 The DOC format and standards compliance

The World Wide Web consortium (the W3C) maintains the standards for what is a valid web page, and they provide a validator for web authors to use to check the validity of our own pages, at `http://validator.w3.org`.

Most authors could care less about validation. They figure that if it looks OK on the browser they're using, and maybe one other, then they're done. For example, try validating the home page of the World Bank (265 errors).[9]

Even as esteemed an organization as the Library of Congress (whose front page validates perfectly) has considered building web pages that violate standards to the point of only working in one brand of browser, but at least they were polite enough to float the possibility with a request for comments first. Tim Berners-Lee, the author of the original HTML standard and frequently credited as the founder of the Internet, submitted a comment that explained the importance of documents written around standards instead of programs:

> At the outset, we would like to stress that nothing in this letter should be construed as a criticism of Microsoft's Internet Explorer [...]. We would write the same letter if the choice was to offer support solely for Mozilla Firefox, Safari, or any other product. [...]
>
> While a large proportion of the marketplace uses the Microsoft Internet Explorer to browse the Web, certain classes of users will find it either impossible or extremely inconvenient to do so. [...] Users with disabilities often must augment their browsing software with special assistive software and/or hardware ("assistive technology"). [...] In addition, some individuals with disabilities rely on alternative browsers (for instance, "talking browsers") that are designed to meet their specific needs. Users with disabilities rely on a standards-based Web to ensure that services they access on the Web will be usable through the variety of mainstream software and specialized assistive technologies that they use.

He also points out that when a security flaw is found in a product, people or institutions will often switch to a competitor until the security flaw is patched. That is, even we of decent eyesight would do well to keep a variety of readers on our hard drives (I use three). This is obviously only possible if a variety of readers can all understand the same document format.

[9] Stats are from validation attempts in late 2005. I sincerely hope they do better if you try them today.

Extending the standards So standards are good. But despite the obviousness of that statement, folks still insist on not complying.

Surely, the most common reason for ignoring a standard is that it does not allow for some form of expression that the author eagerly wants to use. But the author needs to bear in mind that freer expression bears all the costs of broken standards. My favorite Thai restaurant near work, Thaiphoon,[10] has a website that I sometimes check so I can order ahead. When I open with my usual browser, I get a notice that I need to get the Flash plugin to view the site. Since I'm checking from a heavily restricted work computer, I can't install Flash, and often wind up eating at the Chinese place instead. But what happens when I visit the website in a Flash-enabled browser? I get a menu. A plain English text menu.

Or consider the sad state of email. Like a restaurant menu, about 100% of email is also plain text. You tell people things, using words. For about 40 years, there has been a standard (ASCII) that allows different programs to interpret text correctly. Ah, what Nirvana: all the information we need to get across can be gotten across with an easy and supremely well-supported standard. If the UN worked this well, we would have world peace. In fact, now that the computing world is increasingly international, there are more character sets than English-centric ASCII, but nearly every known language is supported by the Unicode standard (yes, Ogham, Ugaritic, Deseret, and Limbu are in there.) Yet people increasingly throw the standard out and encode the text into a word processor document in a proprietary format. If you're lucky, you have a word processor that can read the proprietary-format documents your colleague emailed. For example, if the sender has Word 2000 and the recipient has Word 95, communication won't happen. Putting plain text in a word processor document—even with a bit of extra formatting—is exactly on par with putting a plain old menu in a Flash plugin: yeah, there's a little more glitz, but it comes at the price of potentially excluding, imposing work upon, or alienating the reader.

Of course, word processor documents are nice because they do provide extensions on top of plain text. They let you control the font and layout that the recipient sees in ways that plain text can only approximate. Flash certainly does things that HTML will never even think of supporting. But there is a trade-off that many people ignore, under the presumption that everybody is just like them. "Well, *I* have a copy of Word 2000 and an email client that displays web pages, so everybody else must too. My eyesight and dexterity with mouse and keyboard is fine, so my recipient's must be too." In a social context, the presumption that everybody is like you is the source of a great deal of impoliteness, offense, and general unhappiness, and we teach people from early childhood to understand that others are not like them and that they should maintain standards of decorum until they know that the other party is OK with breaking them. Sure, we can wear the risquè t-shirt to work and maybe make some people smile, but we know that such free expression carries a trade-off in the form of a risk of offending some. We should do the same when writing documents: stick to the basic standards unless we have a reason to do otherwise and we know that the recipient is OK with our new-fangled alternative.

There do exist valid reasons to ignore standards or set out to establish new ones;

[10]CT & S, NW DC. Try the Panang tofu.

e.g., the correct response to a spoken "thank you" is "you're welcome", but it is accepted custom to send a "thank you" email but not a "you're welcome" email, because that sort of thing just sort of clutters up the in box [**?**]. But those who ignore the standards for no reason or for lousy reasons ("I don't have to say thanks—he owed me.") are just rude.

Bringing it back to the subject at hand, Word establishes its own standard, when it doesn't have to. First, users often write a Word document when a simple plain text file will do. An email with no text in the body but a Word attachment with a single paragraph of plain text is a waste in every sense.

Second, there are standards that do approximately everything a Word document does, such as HTML. You can probably think of a few things that you can do in Word that you can't do in HTML. You can also probably live your entire professional life not using them.

Alternative tools Microsoft goes out of its way to make its DOC format opaque, because users are better locked-in if they can only edit their colleagues' documents with Microsoft tools. But I promised you a paper that does not discuss Microsoft's business strategy, but how Word's design hurts your efficiency. The closed-format design means that, by definition, the only way to edit a Word document is in Word.

There are literally hundreds of editors for a LaTeX or HTML document. You can use anything that can read ASCII-formatted files—even including Word. That means that a market has sprung up that eagerly attempts to appease the needs and skills of different users. As above, EMACS and vi are specialized text editors and therefore have dozens of commands to just edit text, but there are hundreds of other text editors that I didn't mention; pick the one that most fits your lifestyle and run with it. For Word documents, you have no choice but to edit them in Word.

On the output end, there are a wide variety of programs that read LaTeX-formatted documents and display them via formats like HTML, PDF, or plain text. Because the file format is open, many people have implemented programs to process LaTeX-marked text to produce interesting new output. I'll have more such options in the sequel.

Meanwhile, the only thing you can do with a Word document is open it in Word. If Word is not to your liking for any reason, you are stuck. If you need to output something besides Word DOC format, you had better hope that Word allows you to do the conversion.[11]

XML The more tech-savvy readers know that the latest version of Word uses the extensible markup language (XML), which is a commonly-accepted standard for semantic markup. However, this is slightly misleading. First, there is not yet a mechanism to write your own style sheets as I described above. Markup like <b>this</b> is valid XML, but it's just an elaborate way to say boldface. That is, Word takes a system designed for semantic markup and uses it for literal markup.

[11]Yes, many people try eagerly to write Word-document compatible extensions, with varying success. But the market for such extensions is absolutely miniscule compared to the market around plain text.

OpenOffice.org will save DOC files as PDFs, by the way. Even if you are married to Word, you may want to download OpenOffice.org and keep it around exclusively as a PDF converter.

But more importantly, using the XML standard does not yet mean easy interoperability. XML is a format for writing down data in a tree structure using plain text, so that it can be easily parsed by readers in any system. XML parsers are common in most coding languages, your browser, Word, and many telephones. But once you've got the XML tree read in, what can you do with it?

An XML file depends on a companion document type definition (DTD) file that explains what headings and types and modifiers are available. There are many, depending on your purpose. An address book will define structures for people and organizations, while XHTML defines headers and tables. Two XML systems that read different DTDs are, in the end, incompatible. If one system marks paragraphs with <p> and another with <par>, then the two won't be able to do anything with each other's data, even though parsing the XML structure will be a non-issue.

Word's XML is Word-specific. Politics: although there exist open DTDs for text documents, including DocBook and OpenDoc, Microsoft is insisting on supporting one and only one XML schema: its own. It has applied for patents on that schema in the U.S. and Europe, and although it has stated that it will allow others to use its soon-to-be-patented technology for free, many are wary of whether the format will remain open.

So Word's XML is a near-miss: it solves the problem of parsing the bits on the hard drive in a standard manner, but it doesn't take advantage of the possibility of semantic markup, or of using any of the myriad existing formats that are well-supported by others.

I've argued for the value of decoupling the interface from the document, so that if you don't like how Word does its thing, you can use another tool to edit your document, and then send your finished product to a Word-using colleague who doesn't care what you used to produce the document. But for Word's format such options are limited. There are many tools that will do certain limited operations; there are a handful of competing word processors that try to look like Word, which get within spitting distance of fully supporting Word documents; and that's about it for handling Word's format. I am not aware of a single non-Microsoft product that claims 100% compatibility with Word's XML format.

So, as long as you're using Word's format, you're more-or-less stuck using Word.

3.6 Alternatives to Word

At this point, I hope I've demonstrated the efficiency gains in having a means of just writing content, a means of just formatting content, and a standard format linking the two.

In every case I can think of, the text writing part is in what you'd call a text editor. As above, there are many that you could choose from. Your OS provides a very basic one with zero learning curve and few features (Windows=Notepad, MacOS=TextPad, Unices=pico), but you can find others that are more comfortable to live in for large projects, like EMACS, vi, Ultraedit, Notepad++, and a whole lot more.

The rest of the story breaks down as to your preferred output and the closely-related question what standard you're going to lean on.

Plain text One option is to not use a formatting system at all. Just open up your preferred text editor and go to town.

Hemmingway:

- Brief.
- Did not use bullet points.
- Used un-formatted plain text.

But Hemmingway was fortunate enough to live before word processors. Today, an unadorned block of text is unacceptable, meaning that you will probably have to move your plain text to some sort of formatted system.

HTML The Web has a text-based standard that can be successfully read by dozens of web browsers on all types of computer. HTML documents from the birth of the web in the mid-80s can still be read today. Even Word can read HTML.

HTML stands for HyperText Markup Language, and although the HyperText part is probably not too relevant to the discussion here, the Markup Language part indicates that this is exactly the sort of semantic language discussed above. This is especially true with the advent of Cascading Style Sheets (CSS). CSS lets you define a class, and describe how that class is to be formatted on the screen. Then, you mark up your text with class delimiters: this is a header, this is a digression. That is, HTML with CSS is exactly the sort of semantic markup language that we're looking for.

Your colleagues will be able to read these documents with their web browser, and even edit them with software on their computer.

If you don't want to write the HTML markers yourself, there are a few systems that will turn easy-to-write plain text into proper HTML. Txt2tags[12], markdown[13], or textile[14] specify easy plain-text markers, like **boldface**, and then they'll filter that into the correct HTML.

LaTeX If you are in academia, use LaTeX. It was written for academic publishing, and universities are used to LaTeX users. It is designed around semantic markup of articles, books, and letters, and pegs them perfectly. This document is written in it, and as you can see, it looks beautiful. Any journal you want your papers to be seen in accept (and frequently prefer) LaTeX-formatted documents, and will provide you with a style sheet to apply to your document so that you can conform to their rules. Mathematics in Word looks amateurish, because only 0.02% of Word's buyers have equations in their papers; LaTeX's math typesetting makes you look smarter instantly.

It is not a strictly semantic markup, but a bit of a hybrid. I think it does a good job of combining the two, and if you want stricter semantics, then you are welcome to add `\def`s to the top of your documents to effect that.

One thing Word is good at, by the way, is deliberate inconsistency. If you want your first page in Helvetica, your second in Times, and your third page to be two-column

[12] `http://txt2tags.sourceforge.net/`
[13] `http://daringfireball.net/projects/markdown/`
[14] `http://www.textism.com/tools/textile/`

format, this will be a pain in most semantically-oriented systems. But because Word's literal markup has no mechanism to impose consistency on the document, inconsistent formatting is much easier than in LaTeX. So there's my token compliment to Word.

If you are not in academia, then you have a stronger compatibility-with-Word problem, but consider using LaTeX anyway. Because there are reasonably effective (but imperfect) LaTeX-to-HTML translators, you can think of the language as a document-oriented HTML-producing language, and can then send HTML to your trapped-in-Word colleagues. This method will especially benefit those who want to use bibtex or makeindex to autogenerate the end matter in larger works.

Now, the above methods require work and learning, but I hope by now you agree that spending time learning something that you will use every day for years is worth the effort. But, I'm not going to tell you how to go about learning HTML and CSS markup or which text editor to use. You know how to ask your favorite search engine for "efficient text editor", "HTML tutorial" or what have you. Many of these open standards and tools are entirely free, so there is at least no financial cost to downloading the tools and playing around. Better than the search engines is to ask your favorite guru for help; many are happy to take time to help a friend work more efficiently.

Also, because standard formats are so open, there is probably somebody who has already fixed every problem you have, but it might be a separate tool. Some text editors include a spell checker, some expect you to choose an external full-time external spell checker from the various available options. If you want to see the difference between your version of the document and the one your colleague edited, your editor may include a dedicated diff mode, or you may need a copy of the `diff` program.

Word Format You may have to use the Word document format at your workplace, though you can continue to use the structure above: use a plain text editor to write plain text, perhaps using format markers like those above, then, at the last minute, open the document in Word. That is, spend the bulk of your weeks of editing and revising working on content and worry about format and visual appeal only as a final step.

Because of Word's fundamentally first-person paradigm, you still need to change your format markers to real formatting yourself, but (1) Word's macro feature can help with this, and (2) you may still save time and effort, because the editing features of text editors can add that much more efficiency.

OpenOffice.org is a word processor initially from Sun Microsystems. [Due to trademark issues, they can't just call it OpenOffice.] Its key claim to fame is that it can read and write Microsoft's document formats very well, meaning that you can interoperate with your coworkers without their knowing that you aren't one of them.

Its stylist solves many of Word's style editor problems, so you may have better success with using it semantically. It has a built in bibliography database system. Maybe Mr. bdamm will get his wish for a basic vi keymap for efficient editing. Its own format is open, and you can save anything to PDF. So complaints about some details are alleviated, but it tries to imitate Word to the point of imitating Word's paradigmatic failings. The literal markup, intuitive-over-efficient, and one work = one view paradigms remain.

3.7 Conclusion

A great many people have spent a great deal of time thinking about how to best edit and format text, and most of them have come up with solutions that look very, very different from Word. Part of the reason for this is that the authors of Word were writing for Aunt Myrtle, while the author of LaTeX was writing a package for his own use; meaning that Word was built around ease of initial use, while LaTeX was built around efficiency. There is no metaphor that one could make between an HTML document with a cascading style sheet and a physical paper with text—but this is liberating and allows for new possibilities and an easier time with formatting.

Perhaps you are stuck with Word, and company policy dictates that you write and maintain long, complex business documents using the same tool Aunt Myrtle uses to write her thank-you notes. Hopefully this paper has given you some ideas for working more efficiently: use the style sheet, stick to plain text where possible, maybe get a copy of OpenOffice.org on the sly for saving to PDF. But hopefully you have the liberty to take the effort and time to learn some of the other paradigms. It will take you days or even weeks, your first documents will look amateurish, and over the next several years of your career you will thank yourself over and over again as you gracefully produce output with truly efficient tools.

4 YOU AND YOUR COMPUTER

4.1 Moore's law won't save you

9 March 2009

First, let's get Moore's law straight: According to Gordon Moore himself[1], the law is that "the complexity for minimum component costs has increased at a rate of roughly a factor of two per year." The press typically translates that to English to say that every year you can buy about twice as much memory for your PC for your dollar.

It's just about true: measures of the PC memory market's megabytes per dollar typically double about every two years.

But that's not what people want from Moore's law. They want to say that their PCs run faster. Memory is nice—more memory means more windows open in your browser—but it doesn't immediately translate to zippier computing in its various forms, such as more complex processes and on-the-fly behavior that used to be background behavior.

No, for that, you just need a faster processor, which partly depends on having cheaper components, and partly on better design.

Here's what I did for this little demonstration: first, I asked Wikipedia[2] for a table of processors and their posted speeds. Processor speeds are measured in millions of instructions per second—MIPS, though by an instruction we mean a computer instruction like 'shift that bit to the left', and it takes several thousand such instructions to execute a human instruction like 'divide 13 by 8'.

Then, I divided each processor's MIPS by the number of cores in the processor. [I'll talk about cores next time if you're not familiar with them.] Then I used the best score for the year as the data point to plot.

This is a logarithmic scale, meaning that what looks like a line here is actually exponential growth, from the Intel 286 in '82, at 2.6 MIPS, to the Intel Core i7 Extreme 965EE, which justifies its absurd name by running 19,095 MIPS.

That's certainly amazing progress. For much of the period, it's doubling even faster than every year. In 1994, the fastest retail chip (Motorola 68060, used in MacIntoshes) ran at 88 MIPS, the fastest retail chip two years later (Intel Pentium Pro) ran at 541 MIPS—more than six times faster in two years.

[1] `http://www.slate.com/id/2080097/`
[2] `http://en.wikipedia.org/wiki/Instructions_per_second`

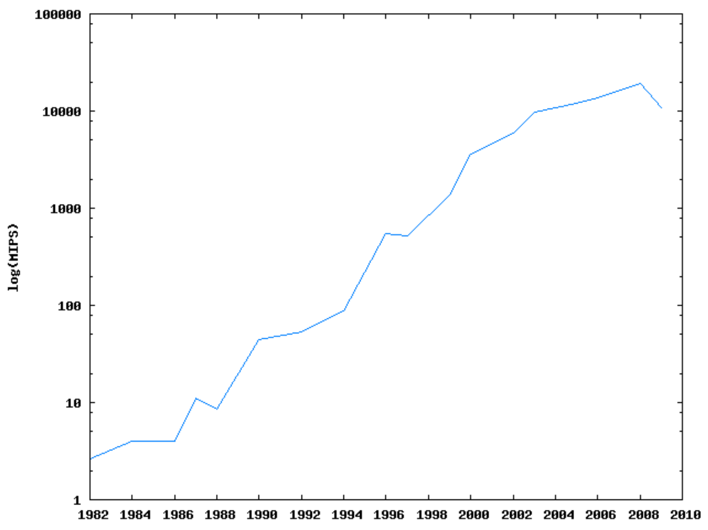

Figure 4.1: Processor speed continues to progress, but not at its former pace

But then things flatten out. The fastest in 2003 was 9,700 MIPS; the fastest in 2008 was 19,095 MIPS, so the doubling of speeds took five years instead of several months.

Caveats I suppose I should mention the trouble with MIPS. First, how do you measure a processor's MIPS? Answer: using the processor's clock to time itself doing things. So you already have potential for shenanigans. If Motorola's definition of 'instruction' differs from Intel's, it may be hard to compare MIPS across them.

And there's more to zippy processing than just summing numbers. Much of the work is in retrieving data and instructions, which can take a lot of time. After all, the computer's memory and the processor are often separated by *several centimeters*. More importantly, there needs to be a component (a bus) doing traffic control and getting the right bits to the right place. All that takes time. One solution is to have a local space on the processor itself that keeps copies of what is expected to be the most useful data, which is why the typical modern PC processor has a hierarchy of increasingly fast Level 1, Level 2, and sometimes Level 3 data caches. According to the Slate article linked from Gordon Moore's name above, that's why Figure one shows a drop in 2009: Intel decided to spend more transistors on memory and power-saving, and fewer on fancy calculating tricks. They seem to expect that that'll make for a more pleasant computing experience for most users, and I'm willing to believe them.

So there are a lot of ways to make a computer go faster than just more math per second; I could list a dozen of them here, but that might be off topic. This is a blog about computational statistics, so all of those great advances don't matter for our purposes as much as the simple question of how many times the system can make a random draw from a Normal distribution.

I'll continue on this thread, and add even more caveats, next time. But all the caveats aside, I believe the story from the figure is basically true: the era of immense speed gains is over, or is at least currently in a lull. We're certainly progressing, and next year's PC will be faster than this year's, but it won't be ten times faster or even twice as fast. In the mid 1990s, if a program ran too slowly, we could just wave it off and say that next year's computer will run it just fine, but we can't do that anymore.

4.2 Parallel II

13 March 2009

In the last episode, I pointed out that every year we get more, smaller transistors, but it's harder and harder to invent new tricks to string them together to do math faster.

So what's the easy solution? Instead of ramping up the speed of a processor, just get two. Thus, the advertised speed gains in the last several years have been via including two semi-distinct processors in one package—the multi-core chip.

On a larger scale, it's the same story: much of large-scale computing is really parallel computing. Systems like the NIH's Biowulf cluster are a pile of a few hundred separate computers, interlinked with high-speed networking cable.

Unfortunately, having two processors does not magically make your programs run twice as fast. Some work is fundamentally sequential, and can't be split into simultaneously-running subparts. To give a somewhat time-of-year appropriate example, think about filling out federal tax forms. Some parts can be filled out without regard to others: if you were a multi-core processor, core one could be filling in the name/address part, core two could be gathering data for the itemized deduction form, and core three could be totalling up sources of income, all at the same time and independent of each other. But you can't calculate total taxes owed until you've worked out all your income and deductions. The tax owed process on core four will be sitting around on its little transistor hands until the others are done.

Then, once all that's done, you can do your state taxes.

So even for a process as easy and simple as doing your taxes, you've already got some parts that can run in parallel and some that have to be run in sequence (i.e., in serial). Without looking at the hairy details, it's hard to say whether having two cores will double the speed of a program or do absolutely nothing.

So that's the central problem for today's story. If we want next year's program to run faster than this year's, we can't just wait for Intel or AMD to ramp up their MIPS count, because, as per last episode (p 35), they aren't really playing that game anymore. The cheapest expansion is now adding capacity to run parallel processes.

So, how is this transition to parallel computing going to work?

From here, we can work at a few levels, some of which are more amenable to parallelization than others. At the overall operating system level, for example, things are smooth and easy: put the drawing program on one core, the spreadsheet on another core, the general OS on another, and you can expect that these separate programs can do most of their work independently.

Figure 4.2: Parallel threads let you do more, but there can still be bottlenecks.

So for those of you who tend to have a dozen programs up at once (that's me), or for those of you using a multi-user system where security dictates that programs generally shouldn't be talking to each other at all, more cores equals less gridlock.

Computing platforms But scientific computing tends to be about a single resource-intensive algorithm. This breaks down into two sub-threads: how well does the single program you're using thread itself, and how well does your specific algorithm thread?

First, the platform. There are many anecdotes like the one about Lotus and Microsoft at the top of the linked page, where a company succeeded by writing software that was bloated for its time, but two years later ran great. But that was the mid-90s, which is the steepest range in Figure 1 from last episode.

You're not writing raw assembly code, which means that you're using some sort of platform sitting between you and the cores: maybe a spreadsheet, or a stats package like Matlab, or a programming language like Python or C. The odds are good that the platform was originally written in the good ol' days that Joel the guru was talking about, back when parallelization wasn't a consideration, nor was code that worked a processor a little too hard.

How easy your life will be depends on the implementers of your platform, because it's up to them to correctly thread the internals where possible, and hand you tools to thread your own work. This one is a minefield, and some platforms have much better threading, both internal and user-side, than others. I won't name names.

How does the package that's my fault, Apophenia, fare on threadability? It does OK. The `apply` and `map` family of functions will look at the `apop_opts.thread_-`

`count` variable, and break the process down into the appropriate number of independent threads. So in a situation where you have a matrix or vector of a million rows, each of which can be independently processed, such as calculating a log likelihood, you can just set a single variable and go. If things get more complex than that, you may have to start rewriting your code. Given the tax code example above, this is not so surprising: I as the package author can't guess what sort of interdependencies your system may have. But I'll admit that there are more cases that could be handled on the back end.

Stats methods At the algorithmic level, some types of research do break into parallel threads more easily, the most obvious being agent-based or simulation-type methods. Here, you have a few million agents or particles, each of which has set rules for interacting with other agents. The core of the simulation, for each period, is a loop that updates the status of each agent. For most situations, with four cores, you can touch base with the first quarter of your agents on the first core, the second quarter on the second, and so on.

Also easy: drawing random numbers. If I want to create a posterior distribution via Bayesian updating, I can basically make independent draws from the prior and independently use them to grow the posterior. [But remember that each thread will need its own seed for the RNG. The easiest way to do this is to simply give thread zero a seed of 0, thread one a seed of 1, and so on.]

We thus have a specific way in which technology affects how we do research: agent-based models are going to be cheaper every year, as are other methods that feel out a model via random draws.

We assume that many data sets [but by no means all!] are produced via a method that makes each independent of the other. The word 'independent' means that massive data sets [e.g., over on the other screen I have a search using 1.48 billion data points] are easily parallelized, since processing on one point is independent of processing on the others.

Other methods write down a fixed model and search for the optimum. These usually involve stepping along and trying a sequence of new points. Is this candidate better than the current? If so, then we'll use that point until a better one comes along. This is inherently sequential, so the options for parallelization are much more limited.

Summary paragraph So the future is in parallelization, which is a somewhat different game from the game we had in the go-faster '90s, and that means the winners in the future may not be the winners today. Many programs (I'm still not naming names) have internals that don't thread well—lots of important global variables, lots of bottlenecks—and those will seem slower because they can't farm work out to multiple processors. Even statistical methods that were popular a decade ago may fall beind relative to methods like agent-based simulations and those that use random draws to feel out a distribution.

4.3 Programming your blog

20 July 2009

First, back up to the six-part series on Why Word is a Terrible Program, which you can read on this here blog, or at fluff.info/terrible[3]. There's a PDF version[4] linked from there.

One of the key themes of that series was the importance of having output and formatting independent of the content, and you can see by the many forms of the same text—episodic blog, long essay, paper—that I do practice what I rant.

Here, I'll mention some technical details of this site that may help you to shunt around your own writing. It's not just me talking about me, but using this site as an example of how the Web is assembled. The summary sentence: Web software was written by programmers, so it pays to think like a programmer when putting together beautiful Web sites.

Content and its management The *content management system* is sold to people as *blogging software* and to businesses via the value-add acronym *CMS*. It's all the same: these systems take the principles of structured code and apply it to human-oriented text.

In well-structured code, you first produce a set of small, modular functions. One function reads input, the next searches any text for a given word, another puts any blob of text on the screeen in blue, and given all that, it's trivial to string together these three functions to write a new function to display the results of a lookup of *Steven*.

Blogging software [I can't stand the managementspeak name] asks you to turn your webpage into small units, and then puts those units together for you. You provide a style sheet, some text for the sidebar, a series of blog entries, and the software produces pages accordingly. If it's sold to a business, then employees, product descriptions, and so on are also reduced to a single unit of content each.

Now that the computer has uniform, modular blocks of content, it can string them together via master templates (i.e., the parent functions).

In that respect, CMSes are not particularly high tech, at all. There are many important bells and whistles to be had (comment forms, search boxes, spam filtering), but the core of it is simply specifying a format for blocks of content and helping you paste them together via a template.

This here blog The needs of this blog are not great. For the most part, every page has three blocks of content: The header plus sidebar (which are a unit), the blog entry/entries, the comment form.

You'll notice that the header and sidebar change depending on the page. This is done by the web server, Apache, which has a very standard Server Side Include (SSI) plugin that could potentially serve as blogging software in its own right. Here's the actual text of `about_the_author.html` (minus the actual content), with discussion to follow:

[3] `http://fluff.info/terrible`
[4] `http://fluff.info/terrible/terrible.pdf`

```
<!--#set var="isabout" value="true" -->
<!--#set var="isauthor" value="true" -->
<!--#include file="head.html" -->

<div id="maintext">
<div id="content">

<h3>About the author</h3>
<p> The actual content goes here.  </p>

</div>
</div>
</body></html>
```

The interesting part here is the first three lines, which are directives to the SSI. I can think of no better evidence for my thesis that web software is written by traditional programmers in a relatively traditional mindset than the format of these SSI directives. Compare the web server's `#include file="head.html"` with the C preprocessor's `#include "head.h"`.

You can picture the thought process of the person who first wrote the SSI package: 'Gee, when I write code, I have this nice C preprocessor that lets me include files. I'd like to do that when I post stuff online, but HTML doesn't let me. Maybe I should write a preprocessor for the Web. After all, unlike those people who just post Web pages about ponies, I have the prerequisites to actually write new software, which the pony-posters will eventually be forced to use.'

`head.html` has the usual code which you can almost inspect via your browser's *view source* option for this page (The web page version, I mean). But you won't quite see what I wrote because the server did some editing using the above variables. Here's the code for a single button in the bar across the top of the screen:

```
<li><a href="http://modelingwithdata.org/about_the_book.html"
<!--#if expr="${isabout}" -->class="active"<!--#endif -->
>About the Book</a></li>
```

The variable set on the page itself advises the system to add `class="active"` if the variable `isabout` is true. Otherwise, that blob of text doesn't appear. In coder-speak, this is the most natural thing in the world: the `head.html` macro has some if-then statements that change output depending on the input variables.

Getting back to the structure of this site, there are still some blog-type jobs to be done, like producing archives, the previous/next links, the search box, the main page with the amalgam of several subpages. I'm using a lightly hacked version of Greymatter, which is one of the original blogging systems, and is so unsupported that it seems that the author has disappeared from the 'Net.[5]

But the demands are so simple that it keeps working anyway. Blocks of text are as just as easy to paste together almost a decade later.

[5] `http://compliticytheory.vox.com/library/post/noah-grey-is-leaving-the-internet.html`

I'd hacked it to use tags, but realized that I never use the tags on anybody else's blogs, and stats to which I have access indicate limited use of the tag pages. The search box and the index of titles seems to be preferred.

The content itself The other hack is about how I write the actual content. All blogging software seems to have some form of pidgin HTML that lets you set boldface and easily add links, but it's never quite sufficient out of the box, especially for technical prose like the stuff on this site.

Raw HTML is a pain to write by hand. (Almost) every tag needs an end tag, the syntax for many details like linking are verbose, you have to mark paragraph breaks, and so on. Worse are the XML variants, which are virtually impossible to get right when writing by hand.

'By hand', of course, is a relative term. After all, I'm using a text editor, not a quill, and it has various conveniences. There are HTML- or XML-oriented text editors that take care of all the above garbage for you. My own preference is to do things in a more stripped-down, no-need-for-tools manner.

So I use $\LaTeX$. It can be written without pain in any text editor, and as a bonus, can be compiled on any system to several types of output. That's how you get the PDF and the HTML version of every web page. If you're not a fan of $\LaTeX$, see the prior entry (p **??**) and its comments for other systems that produce HTML with less hassle.

All that said, the process of producing several versions of a block of content is a repetitive script. As you can imagine, every step but the first is basically automatic:

- Write actual content with $\LaTeX$ tags, but no head.
- Paste on a head (`cat htmlhead content.tex > newblog.tex`)
- Have sed do some little search/replaces to get some details down
- `latex2html newblog.tex`
- Copy the HTML to the Web; notify Greymatter that it needs to make a new page using the standard template and the new content
- Repeat with `texhead` instead of `htmlhead` to produce the PDF version
- Add a reference in the book version; recompile and reship

Once you've written the actual content, and have headers specifying formatting, the rest is just logistics.

What's with that last item? If you were reading this online, I'd be pointing you to the link to the compilation. There are several reasons for this: first, you've surely noticed that several of these entries are a coherent thread, serialized over several entries. It ain't Dickens, but you might want to have the whole thing in one place. On my side, thinking about how today's stupid little post fits in to a larger scheme forces me out of bad blogging habits like repetitive prose or stream-of-consciousness writing that doesn't go anywhere.

Also, a lot of the world is still not wired, so there's a place for paper. And darn it, the format of paper looks nice; given a choice between reading an article on screen and via PDF (a choice many journal web sites offer), I always go for the PDF. I would *love* to see other blogs offer PDF editions and compilations.

But why rationalize. It's there, and after the setup of treating my content like code, took very little extra effort to implement.

4.4 The schism, or why C and C++ are different

6 May 2006

Those of you who actually read my posts about efficient computing, rather than just going to read the comics at the first sight of the word 'computing', may by now have noticed a few patterns.

The most basic is that standards are important. I know this sounds obvious to you, but if it's so obvious, why do people get it wrong so darn often. Why are people constantly modifying and violating standards that work just fine?

I know many of you have suspected this for a while, but let me state it loud and clear: I am conservative. Rabidly conservative. I think that people need to have a really good reason for not conforming to technical standards, and I think most people don't—they just use the shiniest thing available. A large amount of my writing on technical matters is simply pointing out that well-thought-out technical standards tend to work better than the newest and shiniest, and that the value of stability often more than makes up for inevitable flaws in the standards. Even my work on patents is aimed at making sure that open standards remain open and free to implement.

I originally tried to make this into an essay about both computing standards and general customs, but over the course of writing it, I came to realize that the two are fundamentally different. If somebody doesn't quite conform to your human customs—if they use the wrong fork or speak non-native English or wear ratty t-shirts to the office—then the person will be funny or diverse or annoying or just normal. Meanwhile, if computing standards aren't followed—if somebody gets sick of C's array notation, `array[i][j]`, and decides it looks nicer as `array[i, j]`—then their writing is 100% gibberish and they might as well be speaking Hindu to an English-speaker. Standards-breaking in social settings can be fun; standards-breaking in computing is just breaking things.

So although I usually try to put something in the technical essays that will be interesting to those who could care less about machinery, I don't think any of the below is truly applicable to social norms. Or you can read on and decide for yourself.

[Nor is this a comprehensive essay on standards drift and revolution, because that would take a volume or two. Just file this one as assorted notes on one question with an interesting proposed solution: what to do with all those people who keep trying to revise and update and modify the standards?]

Schisms Intuitively, there's the English-teacher approach to retaining a standard, where we force everybody to stay in line with the basic standard. When you go home to write your pals, your English teacher instructed you, be sure to use perfect grammar at all times.

But another approach is to let the whippersnappers fork. On the face of it, it may seem contradictory to think that splitting a standard in half would somehow make it purer, but under the right conditions, giving those who want to experiment room to do so can be the best approach.

For any technological realm, you've got one set of people who just want features—lots and lots of features, enough to wallow in like they're a bed of slightly moist hundred dollar bills—and you've got another team that wants fewer moving parts, and takes care to maintain discipline and stick to the existing norms. We can bind the two teams together, in which case they will constantly be fighting over little modifications to the system and neither team will be happy. That's what happens with English. Or you can have the schism.

Allow me to cut and paste from Amazon:

The C Programming Language by Brian W. Kernighan, Dennis M. Ritchie
274 pages
Publisher: Prentice Hall PTR; 2nd edition (March 22, 1988)
Amazon.com Sales Rank, paperback: #4,457
Amazon.com Sales Rank, hardcover: #445,546

First edition 228pp, 1978:
Amazon.com Sales Rank, paperback: #60,113

The C++ Programming Language by Bjarne Stroustrup
911 pages
Publisher: Addison-Wesley Professional; 3rd edition (February 15, 2000)
Amazon.com Sales Rank, paperback: #11,797
Amazon.com Sales Rank, hardcover: #6,215

First edition, 327pp.
Amazon.com Sales Rank, paperback: #1,243,918

Things we conclude: C++ is much more complex than C—274pp v 911pp. C++ keeps evolving: from 1986 to 2000, the book has had three editions, over which it has almost tripled in size. People are still buying the 1978 edition of K&R C because it's still correct; the first edition of Stroustrup is so incompatible with current C++ that people can't give it away. Finally, Prentice-Hall *really* needs to lower the price on the hardcover edition of K&R. I mean, *my book* is selling better than their hardcover, which ain't right.

Meanwhile, C is as stable as can be. Cyndi Lauper has put out seven albums since K&R C came out. The changes from first to 2nd ed. of K&R are pretty small—literally, they're a fine print appendix. And, I contend here, it owes its immense stability to Bjarne Stroustrup. With Bjarne putting out a new version of C++ every few years that frolics along with still more features, Prentice-Hall is free to reprint the same version of the C book without people whinging about how it's missing discussion of mutable virtual object templates. The guys who want simplicity and stability buy K&R and the guys who want niftiness and fun features buy Stroustrup and everybody's happy.

The other technical standard I use heavily is $\TeX$, and I'd been meaning, for the sake of full disclosure, to give a critique of $\TeX$ comparable to this here critique of

Word[6] Fortunately, Mr. Nelson Beebe already did it for me, in this (PDF) essay entitled 25 Years of TeX and Metafont[7]. The article alludes to exactly the sort of schism in typesetting as in general programming: you've got the people who are totally ignorant of standards and just want the shiniest new thing, and the people who built a standard system that has been stable for the better part of 25 years. Since he's on the standards-oriented team, he gives many examples of how such stability has led to large-scale projects that have significantly helped humanity.

His discussion of its limitations is interesting because there really are features that need to be added to TeX—notably, better support for non-European languages and easier extensibility. But "TeX is quite possibly the most stable and reliable software product of any substantial complexity that has every been written by a human programmer." (p 15) Changing a code base that hasn't seen a bug in fifteen years is not to be taken lightly, and may never happen. Instead, we can expect to see a schism.

Evolution In that 1986 edition of the C++ book, Bjarne wrote this: "since [two standards] will be used on the same systems by the same people for years, the differences should be either very large or very small to minimize mistakes and confusion." I'm going to call this Bjarne's principle.

When you read about the raging debate between Blu-ray and HD DVD (I'm rooting for the one that isn't an acronym), don't think 'now I have to worry about all my stuff being obsolete'. Thank those guys for distracting attention from DVD, which is a nice, stable format that hasn't changed in a decade, ensuring that your stuff has not become obsolete. People have made haphazard attempts to revise the CD format, but thanks to distractions like the MiniDisc and even DVD, your copy of Cyndi Lauper's first album is still the cutting-edge CD standard (specified in The Red Book, 1980). Attempts to incrementally tweak the CD standard never took off. Remember CD+G? If so, you're the only one.

So this is how conservatives evolve. Not from clean standards to floundering in pits of features, but revolutionary breaks from old clean standards to new clean standards. The feature pits are just distractions.

The process of evolution via incremental fixes directly breaks Bjarne's principle, because you get a stream of similar standards that are easily confused and comingled. Corporate-sponsored standards often suffer this failing (but not always), because setting standards that last for two decades and selling frequent updates are hard to reconcile. One company spent a while there naming its document standards with a year—standard '98, standard 2000, et cetera—which in my book means none of the formats are actually standard.

The only way to evolve while conforming to Bjarne's principle is to is to ride a system until it really doesn't do what you need anymore, and then revolt, building a new one that is clearly distinguished from the old, as we saw with DVD's overthrow of CD because CDs truly can not store movies, or Ω's eventual overthrow of TeX because TeX truly can not typeset Tamil.

The trick is to know when to revolt. When is a new feature so valuable that the old

[6]`http://fluff.info/terrible`

[7]`http://www.tug.org/TUGboat/Articles/tb25-1/beebe-2003keynote.pdf`

system should be abandoned? Many a dissertation has been written on this one, and I ain't gonna answer it here. But for well-thought-out technical standards, it's much later than you think, as demonstrated by the active 25-year old standards above.

Back to C vs C++ I copied Bjarne's principle from the first edition of his C++ book, so it comes as no surprise that in the mid-80s, C++ made an effort to conform to Bjarne's principle. In the present day, it just doesn't, and the confusion lies in thinking that it still does.

Even in the first edition, there are incompatibilities between C and the new C++, but just a page or so in the appendix. The author explicitly states (1st ed., p 5) that he's walking into a world of C programmers and C code everywhere, so retaining compatibility is sensible marketing and efficient.

But all those enthusiastically added features, that puffed the third edition up to nine hundred pages, each break a little something in raw C. To give a simple example, I use the variable name `template` a few times, and a user wrote me to tell me that his C++ compiler broke on that, because in C++ `template` is a reserved keyword. Bjarne's principle dies another little death.

On the other side, the ISO added a few features to C a decade ago. The most notable for me is designated initializers; I've written several entries here about how much you can get out of this syntactic tweak. However, C++ has no intention of supporting them. This author[8] feels the rationale paper for not using designated initializers gives "arguments that aren't very convincing", and I'd agree.

The `restrict` keyword, also added to C in 1999, does a lot to get code running faster. The authors of C++ have to date rejected the idea of supporting it. But because it's just optimization advice that can be taken or left, here is a valid rule for the parsing of this keyword: replace all instances of `restrict` with a blank space. With no serious technological reason to exclude `restrict`, we're left with just social and æsthetic reasons, and in the subjective balancing of issues, C compatibility and Bjarne's principle was clearly a low priority.

On a positive note, the last revision of C took a number of ideas from C++, after they'd been tested in C++'s feature pit for a few years, including the in-line comments with // [which I use constantly] and the `inline` keyword [which I never use because the compiler will inline functions for you where appropriate]. But in all cases, the rationale was because these features seemed useful and well-tested, not that adopting them would reduce the distance between the two languages.

All of these examples are to show you that modern C++ has basically thrown out Bjarne's principle. Many people still write "C/C++", thinking of them as the same language, comfortably presuming that a C program will compile in a C++ compiler. But that hasn't been really true for maybe fifteen years now. Better would be to just acknowledge the schism. Let them drift further, because things can only get better once the pair are past confusion-maximizing near-similarity, leaving one well-set in its stability and one free to pursue novelty.

[8] `http://www.informit.com/guides/content.aspx?g=cplusplus&seqNum=325`

4.5 The great packaging problem–the easy part

16 May 2011

A *library* is a collection of functions and data structures. Given a set of libraries installed in one place, functions in one library can readily call functions or structures in another library. For example, a textbook recipe organizing program could use a textbook XML library to save the recipes, so the author of the recipe program would not rewrite any XML parsing routines, but just call them from the other library. In such a setup, sets of functions will be organized into libraries, for the convenience of users, who can pick those that they need.

[I'll lean on *library* for now, though many systems call them packages, Ruby calls them gems, &c.]

The problem statement: how does a function in one library find a function in another? This remains one of the great unsolved problems of modern computing. It breaks down into two problems.

- Local: given a program installed on the hard drive, how does one find and load the requisite file?
- Global: given all the computers of the world, how does one find a needed library?

Next time, I'll be writing about the global problem, and discussing why it's so broken; as a warm-up, this time I'm starting with the local problem to show you just how solved the local problem is. If you've never put thought into it, this may also help you with debugging next time an installation fails.

It is solved in the same manner on every platform I've ever known: when a library is needed, a small set of directories are exhaustively checked for libraries. The implementation is even pretty similar in all cases, wherein an environment variable, like R_LIBS, LD_LIBRARY_PATH, PERLLIB, CLASSPATH, lists those directories that should be checked, and when a new library gets called in by a running program, the system checks each directory on the path in turn. If you're using a system with some sort of global registry (which is effectively a gigantic pile of environment variables), then the path may be listed there. [Since this is a web site for *Modeling with Data*, let me mention that Appendix A has more on getting and setting paths.]

The problem of installing a new library on an unknown system becomes the problem of knowing exactly where everything is. If a file needs to be generated, what compiler is available; if there are files that need to be modified, where are they; what is the right libpath to use for the given system?

A few libpaths are handed to you, like the ones that actully have an environment variable set. Or you could have the platform self-report its environment, like a `newlang --get_environment` command whose output the script could then use, or you could force the user to have an environment variable on hand, or you could use a local registry, or depend on the Linux Standard Base (an effort to define the right paths once and for all), or use Autoconf's voluminous hard-coded knowledge about system-specific details, or just install in /usr/local/share no matter what. Everybody has their custom, due to differing opinions about what is technically optimal and historical glitches. But once you've found the right way to query the local system for where everything is, you'll have no problem putting everything in its right place.

GNU autotools asks the dependent library author for the name of the dependent-upon library, and a sample function. It then produces a small program—basically just

```
int main(){your_function();}
```

—and then compiles it via `gcc -lyour_libname sample_program.c`. If that works, that we haven't just asserted that the library is somewhere on the libpath, but have actually tested that the library can be loaded and used, which is pretty cool. The compiler will search its own libpath, so Autotools is implicitly searching the libpath by free riding on another system that already does so.

So that's the whole local library use problem:

- Search the libpath.
- If you found a match, optionally test that it's valid, or just load it and hope it doesn't break.
- If it isn't found, then it's on to the global problem—search the planet Earth for the library, and install it on the right path.

For installing a new system, we need to preface this with an initial step where we work out what the local paths are. There are diverse solutions to that step, but just read the platform's manual and you'll be fine. Next time, I'll cover that last step of searching for and installing a library or package from somewhere else.

4.6 The great packaging problem—the hard part

22 May 2011

Last time I (p **??**) discussed the problem of how a package, library, or whatever set of files on a given system gets installed and used. The gist is that you need to know where everything goes, via a number of mechanisms, which is invariably expressed as a set of directories to search—a path. There are a few differences due to technical details and historical glitches, but no need to rehash those.

But if the library isn't on the system at all, then it's on to the global problem of finding a package or library from out in the world and installing it in the right place, which turns out to be much more difficult. Technically, it doesn't raise any issues more difficut than the local problem, but the political problems are more complex.

I'll start with the decentralized solution, which will naturally lead in to the centralized.

In a decentralized world, when something is missing, you tell the user, and the user goes out and gets the item and installs it. We consider this to be lacking. Modern users are allowed to be lazy, and cuss at the screen and mumble 'if you know I need a package, why don't you just get it for me instead of telling me to do the work.'

You could actually write into the install script that if a library is missing, run a quick script to download and install, like

```
wget http://sourceforge.net/libwhatever;
./configure;
```

`make;`
`make install`.
Consensus seems to be that this is a bit ad hoc, and potentially fragile, because the dependency is not just on the library, but on the unknown repository that holds the library. If the library to be downloaded releases a new version, will the makefile pull the right version? If the library to be downloaded has its own dependency issues, will users have a clue as to which step in the dependency chain they have to focus on to get things running again?

Which leads us to a system that can query a centralized repository holding a registry of packages, each of which knows exactly what other packages it depends upon.

By the way, I am referring to both package managers for individual languages or coding platforms, like Perl's CPAN, R's corresponding CRAN, Ruby's gem system, &c; and package managers for entire systems, like Linux distributions such as RPM, Debian, Pacman, Portage, or any of the other several dozen on Wikipedia's list[9]. The general package managers do not really care what is in the package. Each package is expected to include an install script that finds the right libpath or works whatever magic is needed to put things in the right place. Conversely, a language-specific library manager can get all the language-specific details right. R has the most stringent package management I've seen, requiring that package authors document every user-visible function, include tests, et cetera.

Say that I just wrote a module in Perl, and it calls out to a C library, so the Perl package will have to check for the library and install it if possible (which might mean installing sub-dependencies). Ideally, it would work out the right way to do this for a Debian system, a Red Hat system, OS X, and all the others. After all, the procedure in all cases is to find the right paths and programs, then put them all together in the same sequence, using a standard shell script. But what I describe here is absolutely impossible, several times over: your Perl package manager doesn't know how to install a C library, your C library probably knows how to work out the paths it needs but not how to resolve dependencies; you'll need to write a new and potentially painful new pakage manifest for every operating system variant.

Let me waste a paragraph stressing how common cross-platform writing is (and should be). Every scripting language I've ever seen has a throwaway or two in the documentation of the form *if you get into trouble, just code your procedure in C.* So at the least, it makes sense to have a mechanism to depend on C libraries. In the communist world, C libraries have a standard means of installation on all systems (the miracle of modern science that is Autotools), so let's not pretend that there are no standards to rely upon. Or what if you want to use TCL/TK for a quick window front-end, or read in and clean data via Perl and then make pretty graphs with R?

For that Perl module, you could start with the general package manager, and then the general install script will call the language-specific install script. Conversely, you could start with the language-specific package manager, and then write dummy packages for the off-language dependencies, like a Perl module with zero lines of Perl code but a full C library hidden in a subdirectory.

[9] `http://en.wikipedia.org/wiki/List_of_software_package_management_systems`

But at this point, things are a mess. I've actually made some efforts to use many of the above package managers (Apt, RPM, whatever R's is called, Python's distutils), and prepping the metadata for three out of four of them them were a real pain. For the system to be robust, the author has to fill out a lot of forms in just the right way, and if there's a central repository, there's the social problem of convincing a gatekeeper that this isn't just a homebrew project and that the forms were filled out correctly. I made the blithe suggestion in the last paragraph that if you need to do cross-silo installation, just write two package manager specs, but I admit that that's like suggesting that if you're into two people, then you can simplify your life by just dating them both at the same time.

Calculating dependencies and knowing the URL to pull from, querying the system to get the right paths and commands, writing a shell script that uses the gathered info to do the install—these are all closer to technical annoyances than the Great Engineering Challenges of our time. So what makes things so darn difficult?

Politics At this point, you may have noticed just how not-Internet this system is. It's centralized. There's typically a gatekeeper at the central repository, who may impose lots of rules. Pretend you got a letter from Steve Jobs, referring to the iTunes Music Store, the archetype of the centrally-controlled system.

> Dear author:
>
> I have written a somewhat popular platform, which you have used and for which you have written useful software. That platform includes a distribution system, which is included by default with every copy of my platform. So if your program is in my registry, then every user will have easy access to your work.
>
> When I put your program in my registry, I will check for dependencies, but *all dependencies must be in my registry*. If you need an XML parser, and don't find one in my registry, you will have to fill in that hole in my registry before I distribute your code. It is important that I maintain party cohesion, and packages that depend upon commonly-installed libraries using other platforms will break the user's impression that mine is the One True Platform.
>
> I reserve the right to impose coding standards of my choice; for example, all programs loaded onto an iPhone must do garbage collection in the manner the Apple specifies, and may not depend on any of the still-running and still-debugged libraries originally written in FORTRAN. Of course, my automated tests can't really detect software quality, just adherence to our rules, so the quality bar is really sort of vacuous. Nonetheless, I'm going to bill my repository as *Comprehensive*, so if you don't conform to my standards, you'll look like an idiot.
>
> Yours,
>
> The Author of the One True Platform

I doubt that any one language's package manager or the maintainer of any one platform fully fits my fictional stereotype (maybe not even Apple). But the incentives are there for somebody promoting a distro or a language or other platform to build a silo, and offer access only to those who agree to stay within the silo's walls.

Silos bother me, whether they are for sinister purposes or just lazy writing that built obstacles without thinking. I don't need yet another reason for somebody to tell me that their language is so cool it's absolutely impossible for them to use any other.

Centralization isn't inevitable. To pull an example from a nearby issue, revision control systems are another great unsolved problem in computing, with dozens of competing systems. In the last few years, there's been a push toward decentralized RCSes that look more like the Internet at large and less like the iTunes Music Store. Like a peer-to-peer file sharing system, each repository of the code history in a network has met at least one other (to get the code to begin with), but effort is made to not allow one to proclaim itself central and canonical. If the CIA took down Linus Torvalds and his favorite server, the code base and version history of Linux would chug along via all the other repositories with an equal claim to centrality. Similarly, a repository system can do more or less to promote decentralized package distribution.

There are programs like alien[10] that will do an OK job of translating the metadata from an RPM-formtted package into metadata for a Debian package, or vice versa.[11] There's nothing keeping the users of R or Perl from using a general package manager and writing install hooks to run tests and documentation checks. If the R and Perl people were using the same general package management system, there'd be nothing keeping the CRAN from depending on CPAN packages or Portage source packages.

The signs of a technical problem are different from the signs of a political problem. The technical problem might involve data structures that are difficult to translate or procedures that undo each other, but we've got none of that here. The reasons why we have so many entirely incompatible systems, the reasons we're collectively in package manager purgatory and can't cross platforms easily, are political: developers of a platform want to dictate what is an acceptable extension, competitors have decided that it's more beneficial to build silos around their own systems than to build bridges, and for the minor technial glitches—¿libc-devel or dev-libc?—everybody is just waiting for somebody else to do the grunt work of making things compatible.

The quickest way to bolt on a package manager to a new programming language would be to instruct the package author to use an existing general build system, with a few language-specific hooks provided by the author if need be. But that would give up any hope of siloing authors of new packages, and some level of (mostly political, to a small extent technical) independence. The language maintainers don't get to be gatekeepers any more and can't reject packages that don't live up to their technical, æsthetic, or political ideals. Package authors and users would benefit from easier and more connected package management systems, but the users aren't the ones who design

[10] `http://kitenet.net/~joey/code/alien/`

[11] As I understand it—and I would love somebody to correct me on this—alien has trouble translating dependencies. Part of this is that package names change across systems: the GSL's development package may be libgsl0-dev, libgsl-devel, dev-libgsl, or if the system weren't so pedantic, the dev package would be just a part of libgsl. Differences in naming are really the perfect example of a social problem blocking technical solutions.

the system.

5 Statistics, sort of

BLAIRBLAIR

5.1 Data is typically not a plural

26 June 2008

BLAIRBLAIR [Today's episode is a guest blog by Mr. BK of Baltimore, MD] MODELMODEL

5.2 Data is typically not a plural

10 June 2009

When we learned all those darn grammatical exceptions, we were usually told that they came about in some distant past, due to some arcane relic of old Dutch or something. But here in the new millennium, we have the chance to witness the development of a new grammatical exception.

If this sounds boring, bear with me: by the end of the column, about 360,000 people will die over this corner of grammar.

See, English has the concept of a collective singular, wherein a group of elements is treated as a unit: e.g., *that clump of birds is moving pretty fast.* The new exception is that this concept can apply to any group of anything *except data*. *The data shows a steep slope* is considered incorrect by some, who prefer *the data show a steep slope.*

If you are one of the people who think that *the data is* is wrong, please stop.

Some examples First, let us imagine a world where English grammar would require all groups to remain plural:
1. The agenda are on the table.
2. The trivia in this book are silly.
3. Steely Dan are playing at the pavillion.
4. The NIH owe me $12,000.
5. The U.S.A. are in a recession.

Notes:
1. Agendum/agenda has the same Latin-based form as datum/data. Yet I have never heard a person who uses *the data are* use *the agenda are.*

rems that this framework gives superior models relative to linear projection, but it does make better use of computing technology.

Hemlines The second thread of statistical fashion is whim-driven like any other sort of fashion. Golly, the population collectively thinks, everybody wore hideously bright clothing for so long that it'd be a nice change to have some understated tones for a change. Or: now that music engineers all have ProTools, everything is a wall of sound; it'd be great to just hear a guy with a guitar for a while. Then, a few years later, we collectively agree that we need more fun colors and big bands. Repeat the cycle until civilization ends.

Statistical modeling sees the same cycles, and the fluctuation here is between the parsimony of having models that have few moving parts and the descriptiveness of models that throw in parameters describing the kitchen sink. In the past, parsimony won out on statistical models because we had the technological constraint.

If you pick up a stats textbook from the 1950s, you'll see a huge number of methods for dissecting covariance. The modern textbook will have a few pages describing a Standard ANOVA (analysis of variance) Table, as if there's only one. This is a full cycle from simplicity to complexity and back again. Everybody was just too overwhelmed by all those methods, and lost interest in them when linear regression became cheap.

Along the linear projection thread, there's a new method introduced every year to handle another variant of the standard model. E.g., last season, all the cool kids were using the Arellano-Bond method on their time series so they could assume away endogeneity problems. The list of variants and tricks has filled many volumes. If somebody used every applicable trick on a data set, the final work would be supremely accurate—and a terrible model. The list of tricks balloons, while the list of tricks used remains small or constant. Maximum likelihood tricks are still legion, but I expect that the working list will soon find itself pared down to a small set as optimum finding becomes standardized.

In the search-for-optima world, the latest trend has been in 'non-parametric' models. First, there has never been a term that deserved air-quotes more than this. A 'non-parametric' model searches for a probability density that describes a data set. The set of densities is of infinite dimension. If all you've got a hundred data points, you ain't gonna find a unique element of $\Re^\infty$ with that. So instead, you specify a certain set of densities, like sums of Normal distributions, and then search for that subset that leads to a nice fit to the data. You'll wind up with a set of what we call *parameters* that describe that derived distribution, such as the weights, means, and variances of the Normal distributions being summed.

But 'non-parametric' models allow you to have an arbitrary number of parameters. Your best fit to a 100-point data set is a sum of 100 Normal distributions. If you fit 100 points with 100 parameters, everybody would laugh at you, but it's possible. In that respect, the 'non-parametric' setup falls on the descriptive end of the descriptive-to-parsimonious scale. In my opinion.

I don't want to sound mean about 'non-parametric' methods, by the way. It's entirely valid to want to closely fit data, and I have used the method myself. But I really think the name is false advertising. How about *distribution-fitting methods* or *methods* *with open parameter counts*?

Bayesian methods are increasingly cool. If you want to assume something more interesting than Normal priors and likelihoods, then you need a computer of a certain power, and we beat that hurdle in the 90s as well, leaving us with the philosophical issues. In the context here, those boil down to parsimony. Your posterior distribution may be even weirder than a multi-humped sum of Normals, and the only way to describe it may just be to draw the darn graph. Thus, Bayesian methods are also a shift to the description-over-parsimony side.

[Method of Moments estimators have also been hip lately. I frankly don't know where that's going, because I don't know them very well.

Also, this guy really wants multilevel modeling to be the Next Big Thing in the linear model world, and makes a decent argument for that. He likes it because it lets you have a million parameters, but in a structured manner such that we can at least focus on only a few. I like him for being forthright (on the blog) that the computational tools he advocates (in his books) will choke on large data sets or especially computationally difficult problems.]

Increasing computational ability invites a shift away from parsimony. Since PCs really hit the world of day-to-day stats recently, we're in the midst of a swing toward description. We can expect an eventual downtick toward simpler models, which will be helped by the people who write stats packages—as opposed to the researchers who caused the drift toward complexity—because they write simple routines that implement these methods in the simplest way possible.

So is your stats textbook obsolete? It's probably less obsolete than people will make it out to be. The basics of probability have not moved since the Central Limit Theorems were solidified. In the end, once you've picked your paradigm, not much changes; most novelties are just about doing detailed work regarding a certain type of data or set of assumptions. Further, those linear projection methods or correlation tables from the 1900s work pretty well for a lot of purposes.

But the fashionable models that are getting buzz shift every year, and last year's model is often considered to be naïve or too parsimonious or too cluttered or otherwise an indication that the author is not down with the cool kids—and this can affect peer review outcomes. A textbook that focuses on the sort of details that were pressing five years ago, instead of just summarizing them in a few pages, will have to pass up on the detailed tricks the cool kids are coming up with this season—which will in turn affect peer reviews for papers written based on the textbook's advice.

A model more than a few years old has had a chance to be critiqued while a new model has not. So using an old technique gives peer reviewers the opportunity to use their favorite phrase: *the author seems to be unaware*, in this case that somebody has had the time to find flaws in the older technique and propose a new alternative that fixes those flaws—while the new technique is still sufficiently novel that nobody has had time to publish papers on why it has even bigger flaws.

All this is entirely frustrating, because we like to think that our science is searching for some sort of true reflection of constant reality, yet the methods that are acceptable for seeking out constant reality depend on the whim of the crowd.

5.6 Supreme Court rules against overreliance on p-values

8 April 2011

This is the case of Matrixx Initiatives Inc. et al. vs. James Siracusano et al. (PDF Opinion[5])

The question in the case is whether Matrixx responded correctly to a doctor's publishe findings regarding ten cases of people out in the public who used their flagship product, Zicam, and then permanently lost their sense of smell. If we were running a controlled experiment, ten cases out of tens of thousands is not statistically significant. Matrixx is a publicly traded company, so it is their obligation to reveal to shareholders all pertinent info, but Matrixx didn't disclose the news about this study, because the results were not statistically significant.

Initially, Matrixx did a ham-fisted job of responding: they sent a cease-and-desist letter to the author of the paper telling him that he did not have permission to use the brand name Zicam in his paper, which just made them look like bullies, created a paper trail that they had seen the study, and which was irrelevant anyway, because tradmark $\neq$ copyright, and you don't need any permission from anybody to make true and above-the-board statements about a product by name. You think the Chicago Tribune[6] or Forbes[7] asked for permission before repeatedly using the word Zicam in their coverage? But enough about what looks like a solid botch of intellectual property law.

Let's get back to the botching of statistics. The key claim that Justice Sotomayor spent the ruling tearing apart was that "reports that do not reveal a statistically significant increased risk of adverse events from product use are not material information." That, is Matrixx claimed a bright-line rule that if a study turns up $p > 0.05$, then it is immaterial.

I won't go into great detail on the Court's argument, because I'm writing on a statistics and computing blog, and I do not believe that any of you reading this blog would take a bright-line p-value rule at all seriously in your own work. You can maybe find some stats textbooks that suggest something like this to undergrads, but I'd guess that the authors feel terrible about oversimplifying so much. You may believe that a journal has a bright-line editorial custom of only publishing studies that eke out a $p < 0.05$, but at the same time make nasty comments about how the system is broken. Like neckties, it's one of those self-perpetuating customs that we all know we'd be better off without.

The Court's discussion begins at *A* on page nine of the PDF linked at the top of this column, and I give you the page number because it is recommende reading. I worked in tech law (until it got boring), and the Supreme Court rulings were always the funnest part of the work. First, the ruling is about a specific question, which may not be what the press yammers about; you may be surprised that the case is really about a legal technicality, and that the Court really wants to say something else but instead winds up writing a ruling that just keeps some detail of the legal machinery clean. The

[5]`http://www.supremecourt.gov/opinions/10pdf/09-1156.pdf`

[6]`http://www.chicagotribune.com/news/nationworld/sc-dc-0323-court-business-20110322,0,4462213.story`

[7]`http://blogs.forbes.com/billsinger/2011/04/01/buffett-sokol-zicam-matrixx-supreme-court/`

case of Westboro Baptist Church hurling hompohobic invective at a soldier's funeral (Opinion PDF[8]) made mention here and there of speech which is offensive and onerous ("Because this Nation has chosen to protect even hurtful speech [. . .], Westboro must be shielded from tort liability for its picketing. . . "), but the legal logic is entirely about who had obtained what permits when and where people were standing. There, the subtext itself makes for good reading.

Because these are typically rulings about the Big Questions, like whether we can we derive cetainty out of studies rooted in probabilites, so they are much more readable than the average opinion (especially once you get into the habit of just letting the excess of citations and footnotes wash past you). So I encourage you to see how a lawyer tears apart somebody's claim that p-values provide a bright-line test for evidence's relevance. Pay especial attention to footnote six, in which Justice Sotomayor defines what a p-value is. I wish I was writing another textbook so I could cite the Supreme Court on this.

The justices instead reiterated a prior ruling that something needs to be disclosed to investors if there is "a substantial likelihood that the disclosure of the omitted fact would have been viewed by the reasonable investor as having significantly altered the 'total mix' of information made available." If you're the sort of person who thinks in terms of Frequentists vs Bayesians, that means you're a Bayesian, and that probably means that you're salivating right now, because the Supreme Court just ruled that information is relevant to the extent that it causes a reasonable person to update his or her subjective prior.

The right null for the job Following a common pattern in the medical literature, there is anecdotal evidence that Zicam caused a burning sensation followed by a loss of smell, backed up by some prior knowledge that zinc has been known to have deleterious effects on certain types of tissue. There's a small-n problem at the core of this: if one in ten thousand suffer an effect, then clinical trials of a hundred patients have no chance of passing the bright-line of $p < 0.05$, but after a million people use it, then we expect a hundred people will have suffered a permanent loss of their sense of smell.

The null hypothesis in a study is typically of the form *nothing happened, there are no differences, nothing of significance is going on*. This is a good default because your typical researcher is running a study because he or she really believes that there's something going on, and so *nothing happened* correctly sets the bar high.

For the medical literature, when it is asking whether harm is caused, this is not a helpful null hypothesis. Say that a study's null is that Drugacil does no harm, but the data finds that Drugacil kills people, with $p = 0.75$. There's a 75% likelihood that the thing about killing people was just random noise, and a skeptical researcher might retain the belief that nothing happened until given convincing evidence that something did, but I sure ain't using Drugacil. The correct null here is that harm was caused, and in an ideal world we reject it only when we are confident that there is no harm.

This isn't to say that all evidence is relevant evidence, and other inquiries in other contexts will play out differently. There's still the micronumerosity problem, potential ethical issues of such a study, et cetera. But this point is worth adding to the Supremes'

[8]`www.supremecourt.gov/opinions/10pdf/09-751.pdf`

already long list of explanations for why a bright-line p-value test doesn't work: sometimes the right null hypothesis shouldn't be that nothing happened, and sometimes, evidence that might be due to chance is still important and in need of consideration.

Why are there still all those undergrad textbooks that push for a bright-line p-value test? Because we want certainty. We don't want to live in a world where statistics only speaks in probabilities and where context always matters. But here we are.

6 Teaching this stuff

6.1 Breaking down the pipeline

30 March 2009

Let me get back to the second episode (p 35), where I pointed out the value of distinguishing between inferential and descriptive techniques.

I believe my first few tries at understanding statistics failed because I took classes that didn't make this distinction. Consider good ol' ordinary least squares (OLS), which is often all the stats an undergrad will learn. Here are the steps:

- Clean your data, producing an input matrix X and a dependent vector Y. This is via various computer-code matrix manipulations and substitutions for missing data.

- Find the line of best fit, with coefficients $\beta = (X'X)^{-1}X'Y$, using pure linear algebra.

- Test the elements of β, using methods most folks recognize as statistical, regarding comparing a statistic against a distribution.

The point here is that each of these steps is a different world from the others: computer trickery, linear algebra, and t-distributions basically have nothing in common. Like most undergrad courses, I'll pass on the first step, and assume a perfect data set. Then we're back to the distinction from prior episodes: the second step is purely descriptive, and the third step is purely inferential.

As usual, there is little or no benefit to confounding the descriptive and inferential step. They evolved separately; talking distributions provides nothing for the understanding of linear algebra; talking linear projections does nothing to forward an understanding of the Normal, t, or F distribution.

But you'd think that linear projections and t-tests are joined at the hip from the many stats classes and textbooks that present the steps above as an unbreakable pipeline.

Fun fact: errors do not need to be Normally distributed for the OLS projection to be the line of best fit (i.e., to minimize squared error). I've met a number of extremely intelligent people who thought otherwise. This even appeared in a draft textbook I was peer-reviewing last week.

This fun fact is from the Gauss-Markov theorem, which is a linear algebra and minimization exercise that has no need for math regarding distributions. But at this point, you can see where that confusion came from: when these people learned OLS, they simultaneously learned the part about projection (the Gauss-Markov theorem) and the part about hypothesis testing (based on Normally- or t-distributed errors).

I also think that there is a truly prevalent perception that that the *purpose* of OLS is the hypothesis test at the end of the pipeline, that that's all that OLS does: it tests whether one column of the X matrix affects or does not affect a column of the Y matrix. The other numbers that the software spits out—because the software also merges the descriptive and inferential steps—are just irrelevant.

The OLS pipeline goes from inputs to projection coefficients, pauses for air, then goes from projection coefficients and variances to confidence levels. Those who believe that there's no middle step, and it goes straight from inputs to confidence levels, are prone to miss out on a number of points:

- Missing the inappropriateness of OLS when Y can't be expressed as a linear combination of the elements of X. OLS is appropriate if Y is a linear function of X_1^2, but a student looking for the final confidence interval may not have an eye out for squaring or other such such transformations.

- Failing to realize that OLS is one of an infinite number of alternate models that one could use to test a hypothesis, where those tests also conclude with a t-test or F-test.

- Ignoring practical significance, such as when a coefficient is statistically significant but implies a minimal change in outputs given a reasonable change in inputs.

I could think of a few more, mostly bad habits about grubbing for p-values. Such bad habits are from a failure to balance the descriptive and the inferential.

Policy implications At this point, my recommendations should be obvious: when teaching a standard pipeline like OLS, be clear as to which steps are inferential steps, and which are descriptive. Teach them separately, as a set of pipe joints each of which has value by itself, and at the end mention that what you'd taught to that point can be welded together to form a smooth pipeline.

There are a few common ways by which parts get merged and OLS sold as an inferential technique only. I have several examples of their use on my bookshelf, and recommend that they be avoided.

The most common method of confounding is to open the section on OLS with a list of assumptions required for both description and inference. This has minor benefits over introducing each assumption as it is needed—it makes it easy to memorize the list for the test—but has the major disadvantage of not giving context as to why these assumptions to be memorized are necessary. I'm pretty sure that it's these all-assumptions-first textbooks that make it so easy for me to find people who think Normally-distributed errors somehow fit into the Gauss-Markov theorem.

Some textbooks on my bookshelf literally class regression in the inferential statistics part of the book, and heavily focus on the interpretation of those darn p-values. See above about encouraging significance-grubbing.

As you can imagine, my own writing takes pains to make the distinction, and *Modeling with Data* covers most methods twice: once purely descriptively and with no inference, once a chapter or two later covering only inference. I don't think any continuity is lost because students have to flip between p 274 and p 307.

I used OLS as an example here, but one could apply it to may parts of what's covered in a probability and statistics class. For example, all the distributions are almost always covered in one long list, even though some are for description of natural data and others are for inference using constructed statistics. As Kmenta [1986] explains, "There are no noted parent populations whose distributions could be described by the chi-squared distribution." [I like Kmenta's textbook because it does a wonderful job of telling the reader exactly whether he's talking about description or inference at any given point. Too bad it's out of print.] From a purist's point of view, all distributions are just functions, but from the student's point of view, the crucial question is what s/he will do with each distribution. If their use is different, then that needs to be made clear, meaning that methods for inference and those for description need to be clearly distinguished.

6.2 Testing the model using the model

11 February 2010

Three guys are stranded on a desert island And all they have to eat is a case of canned pears. The joke is that they're all researchers.

The physicist says: 'we can mill down these coconut husks into lenses, then focus the heat of the sun on the cans. When their temperature rises enough, the seams will burst!'

The chemists says: 'No, that'll take too long. Instead, we can refine sea water into a corrosive, that will eventually just melt the can open!'

The biologist cuts him off: 'I don't want salty pears! But I've found a yeast that is capable of digesting metals. With care and cultivation, we can get them to eat the cans open.'

The economist finally stands up and smiles: 'You are all trying to hard, because it's very simple: assume a can opener.'

[Pause for laughter.]

I've found that this joke is so commonly told among economists that you can just tell an economist 'you're assuming a can opener' and they'll know what you mean. It's also a good joke for parties because people always come up with new ways to open the can. What would the lit major do?

Cracking open a model with no tools Now back to the real world. You are running the numbers on a model regarding data you have collected. To keep this simple, let's say that you're running an Ordinary Least Squares regression on a data set of canned pear sales and education levels. You have the data set, then run OLS to produce a set of coefficients, β, and p-values indicating the odds that the βs are different from zero.

Those p-values are generated using exactly the procedure listed above: assume a distribution of the βs, write down its CDF, then measure how much of the CDF you assumed lies between zero and β.

We're still assuming a can opener. We used the assumptions of the model—that errors are normally distributed with mean zero and a variance that is a function of the data—to state the confidence with which we believe the very same model.

To make this as clear as possible: we used the model assumptions to write down a probability function, then used that probability function to test the model. But making an assumption does not add information.

Pick up any empirically-oriented journal, and in every paper, this is how the confidence intervals will be reported, by assuming that the model is true with certainty and can be used to objectively state probabilities about its own veracity.

So ¿why doesn't all of academia fall apart?

First, many of the assumptions of these models are rooted in objective fact: given such-and-such a setup, errors really will be Normally distributed. [We could formalize this by writing down tests to test the assumptions of the main model, though for our purposes there's no point—they'll just fall victim to the same eating-your-own-tail problem.] Even lacking extensive testing, if the data generation process is within spitting distance of a Central Limit Theorem, we'll give it benefit of the doubt that there is an objective truth to the distribution.

Second, we can generalize that point to say that the typical competently-written journal article's assumptions are usually pretty plausible, or at least do little harm. When they report that one option is more likely than another, that is often later verified to actually be true, though the authors had used subjective tools to state subjective odds.

Third, we shouldn't believe p-values—or any one research study—anyway. A model with fabulous p-values will increase our subjective confidence that something real is going on. But if you read that a p-value is 99.98%, ¿do you really believe that in exactly two out of 10,000 states of the world, the difference is not significant? Probably not: you just get a sense of greater confidence.

So this works because we treat the process as subjective. The authors made up a model, and used that model to state the odds with which the model is true. But if we agree that the model seems likely, and if we accept that the output odds are just inputs to our own subjective beliefs, then we're doing OK. Problems only arise when we pretend that those p-values are derived from some sort of objective probability distribution rather than the author's beliefs as formalized by the model.

6.3 Why you should teach your stats students C

5 September 2011

[This is an essay for those of you who are teaching a nontrivial amount of programming to your students. If you're teaching just enough programming to run the `regress()` function and absolutely no more, then don't worry about all this.]

I'll start with the main argument as to why you shouldn't teach your stats students C. Here are some snippets cut and pasted from Joel[1], a guru well known in the straight-up programming world.

> ...there are two things traditionally taught in universities as a part of a computer science curriculum which many people just never really fully comprehend: pointers and recursion.
>
> All the kids who did great in high school writing pong games in BASIC for their Apple II would get to college, take CompSci 101, a data structures course, and when they hit the pointers business their brains would just totally explode, and the next thing you knew, they were majoring in Political Science because law school seemed like a better idea. I've seen all kinds of figures for drop-out rates in CS and they're usually between 40% and 70%.
>
> When you struggle with [more quotidian programming issues], your program still works, it's just sort of hard to maintain. Allegedly. But when you struggle with pointers, your program produces the line **Segmentation Fault** and you have no idea what's going on...

For applied statisticians, the conversation typically ends there: C is hard because it doesn't hide pointers, and there are other languages where you don't have to think about them.[2]

The other objection, which I'll put in as an aside, is that the environment for C is entirely open, while closed environments are easier to get started with. R, matlab, &c. provide you with a single window where the commands and the output go. I think this is an outdated objection, and if you poke around, you'll find that installing a full development environment is about as easy as installing a stats package, and that IDEs and stats package GUIs have basically merged in functionality. There are so many tools for tracking down those segmentation faults in the present day that they're not really such an issue anymore.

Back to that main thread about pointers, which are the location of data rather than data itself. Why is there any value to making that distinction, and working in a language where users have to think about it? From a practical perspective, there's the simple fact that pointers will speed up your work immensely, so you can bootstrap variances from your MCMC model without tears.

But from a theoretical perspective, statistics is all about multiple levels of references to data. Above, the CompSci students had trouble with understanding the difference between data and the location of data, but if you've ever taught Stats 101, you've spent a class period watching your students get mystified by how the variance of the data and the variance of the mean of the data are different. Then you get to do that over when you teach regressions and show the students that the variance of the data, the variance of the OLS parameters, and the variance of the error term are again all different things.

[1] `http://www.joelonsoftware.com/articles/ThePerilsofJavaSchools.html`

[2] I sometimes wonder if it isn't just that *pointers pointing to data* sounds so sharp, and maybe they need another name. "An array is implemented as a bunny paw cuddling onto the first element of the array."

That is, statistics is filled with distinctions between data, statistics of data, and statistics of statistics of data. Let's say we have a simple hierarchical model, where we take the mean of each subgroup, then run a Probit on the means. The variance of those Probit parameters are statistics of statistics of statistics of data. You'll sometimes find box-and-arrow diagrams of such models that look a lot like the diagrams used to teach pointers.

So when your students are learning C and getting lost about a pointer to a pointer to data, they are getting practice in exactly the same skill they need to keep track of what's going on in nontrivial statistical models.

If you clicked through to the article by Joel above, then you saw that Joel is actually pretty pro-C as a teaching language:

> But when you struggle with pointers, your program produces the line **Segmentation Fault** and you have no idea what's going on, until you stop and take a deep breath and really try to force your mind to work at two different levels of abstraction simultaneously.

Earlier in the essay, Joel explains that "You need training to think of things at multiple levels of abstraction simultaneously, and that kind of thinking is exactly what you need to design great software architecture." It's also exactly what you need to understand a hierarchical model or even the difference between the variance of a mean and the variance of a data set. If your students are smart enough to understand statistics, they're smart enough to understand pointers. And after they get good with pointers, learning statistics will be easier.

And, as a bonus, the code your students write will be of higher quality, because they'll be writing with a full tool set, not the subset that a stats package offers under the presumption that users aren't smart enough to work with both data and references to data.

The other reason I like C as a teaching language is that it is a very simple language, with few grammatical exceptions for anything. I do all my C work with 18 keywords (which is a count way at the bottom of the rankings, which typically range between about forty and 'we stopped counting'). I'll get to the scoping constructs next time, but they're also darn simple. You won't spend another four or five sessions of class time going over objects, encapsulation, and exceptions to the object and encapsulation hierarchy, because these things are done without additional syntax. The complexity of real-world problems—string handling, matrices, vectors, regressions—happens via libraries that load more functions and structures, but leave the basic syntax intact.

In fact, the great majority of other languages and packages could be described with a sentence of the form, *this language is kinda like C, except it has an additional syntax to handle [elements] more easily.* Which is why C is a great teaching language for students who are likely to face a half-dozen other little programming languages by the time they leave grad school: a student who learns C will have the background needed to pick up all the other languages quickly and easily. So not only will they have more training in the sort of pointer-like multiple indirection that is a modern statistical model, they'll be ready to implement it in whatever tools are needed for today's project.

7

TOO MANY MODELS

7.1 Bringing theoretical models to data

1 May 2009

I had a simple agenda behind *Modeling with data*: better modeling.

Despite the title, many people miss this, what with the exposition on modeling restricted to the first few pages and the rest of the book being filled with C code. I used to think those people missed the modeling issues because of the language thing. One del.icio.us user bookmarked this site with the note: ”Statistical programming in C? What the ass?” But she did bookmark the site, and (I can tell you because of a short email exchange) did got over the foreign sound of statistics in C to like the approach.

No, I don't think reasonable people can be truly hung up over a surface issue like programming language. I now understand that the real language barrier is in the many definitions and understandings of what is a model.

When I talk to a statistician, a model means a probability distribution over elements, and that's about it. I'd start talking to a statistician about modeling subject-specific knowledge about the interaction of elements, and giant question marks would appear over his head. Which is not to say that the person is a moron, but just that his understanding of the meaning of the word *model* is much more narrowly focused than mine.

In the R package, the `model` object specifically encompasses generalized linear models (GLMs). Again, this is not to disparage R, but to show that there's a good number of people out there for whom it's perfectly OK to equate the word *model* to GLMs, because 100% of the models in their research will fall into that category.

Grab off the shelf one of those journals with “Theory” in its title, where authors can just present a theory and its implications without bringing it to data. There are such journals in any field, from economics to sociology to physics. The models are often wild and creative. Elements interact in every way imaginable. For example, I've done a lot of work with network models, where individual agents form links via iterative, nonlinear processes, regulated by whatever the author dreamed up. The models aren't necessarily complex, but they have no need to stick with simple linear components.

The empirical definition of a model and the theoretical definition don't necessarily overlap. For example, agent-based modelers naturally have agent-based simulation in mind when they hear the M-word, and many enthusiastically reject the GLMs. That attitude means that many ABMers are unfamiliar with the statistical models that were the entire world of the statisticians above. Conversely, your average statistician has zero experience with agent-based models.

Empirical implications of theoretical models [1]

The multiple meanings of *model* are a problem when the theory gains traction and is eventually brought to data. A model to the theorists includes anything under the sun, while a model in your typical stats package is a GLM. So instead of directly fitting the model, one tests its empirical implications, such as how variable A going up should cause variable B to go down. We can fit that sort of implication into a linear regression without serious violence.

But wouldn't it be great if we could fit and test the model itself?

The work I was doing that really motivated the book was on using agent-based models as probability models, which would allow for more direct testing of the model. But as above, the agent-based concept of a model and the probability concept of a model are academically disjoint: few people accept and use both concepts simultaneously.

Why not? There are many very valid reasons. There is value to specialization, and I won't claim that everybody needs to be a polyglot all the time.

But there are also many lousy reasons, based on how what we can easily theorize is so much broader than what we can easily calculate.

If your definition of a model were just OLS, you'd have no need to code anything. Mathematically, I cover that ground in two pages (pp 271–272), and implement it in code in two separate code samples, because it's so trivial that it wasn't worth revising out the redundancy. If you have the statistician's definition of *model* in mind, you're probably going to be puzzled by the book's lengthy exposition on computing the hard-to-compute.

But by the broader definition of a model, which the theoreticians in all fields are using, we are worlds away from having things so neatly boxed. That network model, however the details play out, can't be pulled off the shelf.

So that's what the book is about: my best stab at the tools you'll need to bring a new model to data (or to generate artificial data from your model), where the word *model* takes on as broad a meaning as possible.

Next time I'll talk more about the mechanics of writing code for modeling in the broader sense.

7.2 A general model object

4 May 2009

Last time, I talked about the balkanization of modeling: basic statistical models, agent-based models, many types of physical models, are all academically disjoint, by which I mean that few people simultaneously use more than one of these paradigms.

[1] EITM is the name of an ongoing series of summer institutes in political theory, in which I have participated.

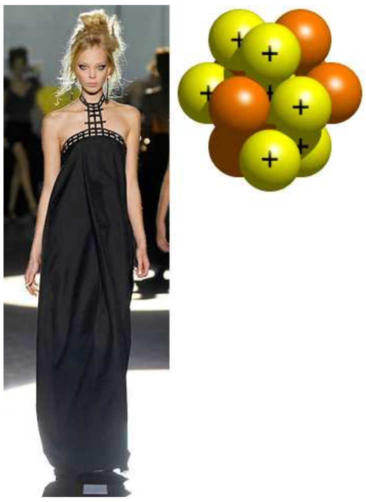

Figure 7.1: Different fields have diverse and hard-to-reconcile conceptions of a model.

This time, I want to bring it to a more practical level: if your model is anything beyond a never-to-be-estimated system of equations, it will be expressed in computer code. But our software has as much diversity in conceptions of a model as our academic departments do. If I want to do a regression, I can use R and its pleasant generalized linear model (GLM) interface; if I want to do agent-based modeling, I can use any of a number of object-friendly frameworks—but not R, which is terrible for such modeling. [We are not having the discussion about computationally-intensive modeling in R now, but trust me when I say it's not the right tool for the job.] If you're doing an EE-type simulation with polar coordinates or complex variables, then you're more likely to be doing it in Matlab, and have probably never heard of R.

The balkanization creates problems for those who want to do something outside of the group norms. If I want to use an agent-based model as a prior and then do Bayesian updating using observed data, none of the above will accommodate me. If I want to try hierarchical modeling tools to simplify and approximate a complex circuit diagram, I'll be doing a whole lot of inter-package negotiation. Generally, if I want to talk across different disciplines (or even sub-disciplines), I have to have multiple conceptions of a model in my head at once. Many humans have no problem with this, but few software packages are capable of such a feat.

A standard model form Is there a general definition of a model that we can operationalize into code, which would allow us to interchange and combine models in one place?

I've lost a lot of sleep over that question. A definition that's too general is basically vacuous—we can just say that a model somehow maps from one set of functions to another, and that'll cover anything you can think of, but won't say anything. If we restrict models to GLMs, we can get an immense amount of work done, but at the cost of balkanizing away all the fields where modeling doesn't mean regressions. The engineering challenge is in finding a decent balance between generality and practicality.

Here's what I came up with in the end. I present it to you not only because I think it's an important question, but to challenge you to think of what you would or would not include in an implementation of a general model. My own definition, which I've operationalized, reduces down to this:

> A model intermediates between data, parameters, and likelihoods.

There are four significant words in the definition, which require four more definitions:

Parameters: These you are familiar with. The mean and variance of a Gaussian distribution, the coefficients of a regression, or a simulation's tweaks about frequencies, population size, &c.

Data: Data is generally the other input: the draw from the Normal distribution whose odds you're finding or the observations for your regression. When doing estimation, we think in terms of the data being an input and the parameters being an output, but you'll see below that there's more symmetry than that; we can turn the tables and fix the parameters to produce artificial data.

Likelihoods: The odds of the given parameters and data co-occurring. For a Normal distribution, this is what you get from looking up the data point on the appropriate table. For a regression, the likelihood is calculated under the assumption that errors have a Normal distribution.

By having a likelihood, does the definition force us to stick to probabilistic models? No, because there is always some means of evaluating the quality of the model, and that can be read as a likelihood. Typically, this is a distance to some sort of ideal, or some objective to be maximized. If you don't believe me that a distance function can generate a subjective likelihood, then just take this as metaphorical (until I have a chance to post an entry on why this is the case, and why we shouldn't distinguish between likelihood and probability).

Intermediation: We could go in three meaningful directions among the above.

Data $\Rightarrow$ Parameters. This is the standard estimation problem. You have some data, and find the most likely mean and variance of the Normal distribution, or the regression coefficients that produce the line of best fit. We invariably use the likelihood function to do this.

Data + Parameters $\Rightarrow$ Likelihoods. That is, the odds of having the given inputs. These are the tables in the back of statistics textbooks for different distributions, with parameters along the columns and data values along the rows.

Parameters $\Rightarrow$ Data. Given a set of fixed parameters, we can find the most likely data, or the expected data, or make random draws to produce an artificial data set consistent with the model and parameters.

These methods are generally linked, and you can often solve one from the other. For example, if you give me a likelihood function ($D + P \Rightarrow L$), I can find you the optimal parameters via maximum likelihood estimation (MLE, $D \Rightarrow P$). This is a boon for software design, because when a model doesn't have a quick method for estimating parameters, I can fill in a default method; when it does, like the case of ordinary regression, I can use that instead of the general but computationally-intensive MLE.

You can see that there's a lot more that we want our models to do than just direct estimation of parameters from data. You may have in mind other things that one would do with a model that aren't covered; I drew the line here based on the above problem of writing a definition that is both inclusive and operationalizes into something useful.

But I should note testing, which is certainly one of the more common things to do with a model. I separate testing from the model proper, because of everything I've already said about the importance of differentiating the descriptive and inferential sides, and the prior entry (p 70) on building a pipeline with separate estimation and testing steps. From such a perspective, testing is not a part of the model, but something you can do with it.

Every test has the same form: establish a distribution; locate the data and a reference point (usually zero); establish the odds that the data and reference point differ given the distribution. I toyed with the idea of establishing a generalized testing object, but saw little benefit, being that the test is already typically a single line of code: looking up the CDF of the appropriate distribution at the given point. If you don't have a CDF on hand, you can of course generate it from the model via random draws.

[Which brings us to one more thing we often want to do with a model: generate a CDF. I implement this as a histogram, which is just another view of the same model.]

Be creative What's the benefit of this standardization of models? First, we can write methods to work with any model. Part of the estimation-testing pipeline that we often see in many stats packages is a tendency to put the code for certain methods inside the code for models with which the methods are closely associated, which means that the method isn't available when some other model wants to use it. With methods that take black-box models as inputs, we have a better chance of applying methods from discipline A to models from discipline B.

We can test models against each other. This could mean a Poisson versus an Exponential, or as above, it could mean a theoretical distribution versus a histogram fitted to the model. Or how about an agent-based model with or without some extra moving part? Most importantly, we can compare a common baseline model, like a simple linear regression, against a relatively complex simulation.

My goal in all of this was to use a simulation to generate a probability distribution. In the format here, that makes sense, and is easy: just specify a likelihood function based on a distance to an optimum, use it to estimate the parameters of the model, then use those optimal parameters to find likelihoods for other purposes. Because the model is a black box with a limited set of interfaces (like the likelihood function and the parameter estimation routine), we don't have to care about the methodological innards of the model, and can use it as we would a Poisson distribution.

For many purposes this black-boxing is exactly the right way to deal with a plethora of models. Excluding some models from some uses because of their methods of computation is just blunt bigotry, which has no place in our enlightened era. There are still some practical considerations about what works best in a given situation, but you probably thought about those things during the pencil-and-paper stage of the project.

It is not the place of the software to place restrictions on your creativity.

[Not one to just chat about the theory, this is a bird's-eye view of the `apop_model` object actually implemented in the Apophenia library. There are still technicalities to be surmounted, which I discuss in the coder-oriented design notes[2].]

7.3 Multiple imputation routines

27 September 2010

Imputation is the statistician's term for what everybody else calls filling in missing data. In some contexts, imputation is just a part of the background routine, whether you're willing to admit it or not. Are you giving the mean of a set of observations and just ignoring the missing values? That's equivalent to imputing each missing value as having the value of the mean. The expected value is expected to be right, but the variance for a list of 100 items where ten were filled in at the mean is smaller than the variance for a list of 90 known items (which is what results from *listwise deletion*). Which is the right variance? We can't answer that question with the information to this point, because we don't have a model of why the missing data is missing or how it differs from the rest of the data.

To fully flesh out the situation, we'll need two models: one model that is the data generation story we want to tell, like a linear regression story or draws from a Multivariate Normal distribution, and the other model is the one that explains how we filled in the missing data. That model is probably not much like the model that you intend to estimate; it is probably much simpler, like a plain Multinomial distribution. After you make a filling-in-the-blanks draw from the distribution, you can find the variance of the statistic you want for the overall distribution based on the now-complete data; and for several draws from the fill-in distribution, you can find the variance of that statistic across data sets.

I chose that *within/across* language to parallel the language of within group/across group variance calculations from the ANOVA world, because the process here is analogous to the process there.

That's the whole story: specify a model by which your data was generated, then use that model to fill in a series of data sets, calculate your preferred statistic for each, and compute total variance as the sum of within-imputation and across-imputation variance.

¿Is it Bayesian? Really, the storyline is just the convolution of two models. Let the parent model be a distribution over the outcome $f(out|d, md)$, where md is missing data, and let md have distribution $g(md)$. To observe the final outcome of the model, you'll need to find the convolution, $f \circ g$, which, keeping to blog-level notational precision and taking the observed data as having probability one, one could write as the overall joint distribution $f(out, d, md) = f(out|d, md)g(md)$ At this point, it should start looking like Bayes's rule as presented in the typical Bayesian updating setup. The

[2]`http://apophenia.sourceforge.net/apop_notes.html`

coincidence here is that both Bayes's rule and the missing data are convolutions of two models, and therefore both will take this form.

That's convenient because our Bayesian friends have been spending the last few decades putting real work into developing the computational tools one would require for convoluting two probability distributions. So if you wanted to, you could think of the missing data problem as a Bayesian problem, pull out your generalized Bayesian solver (which probably means your MCMC routine), and get results. If you don't have a generalized solver, you can set up your missing data models so that the two distributions are from the table of conjugate distributions, and calculate results for the output distribution with pencil and paper.

The direction you take and tools you use probably depends on what you want your output to be. If you want the entire distribution, then a Bayesian-style technique is your only bet. If you just want total variance for a statistic, then the full convolution is overkill, and the method I mentioned at the head of this entry, where you simply generate a half-dozen full data sets and sum up the within/across variances, will give you the most accuracy for your computational buck.

There are several choices in naming a setup: the context, the methods used, et cetera. The real crux of the missing data problem is in specifying an auxiliary model for the missing data to accompany your main model, making it just one of a wide range of methods that join two complementary models. But in the textbooks, you'll find the process named after the method of calculating the post-convolution statistic's variance, *multiple imputation*, although there are other methods to doing the model-merging calculations.

7.4 Multiple imputation's setting

24 October 2010

This is really part II of the last segment on multiple imputation (p **??**). M.I. or one of its friends is really essential for honest analysis. If you have missing data, you have some model for filling it in, and it's better to measure the variability added by the missing data model than to just ignore it and pretend the values you filled in are correct with certainty.

So, then, ¿why aren't these techniques absolutely everywhere missing data is found? People in some fields, like your survey jockeys, are entirely familiar with this problem, and would never fill in a value without properly specifying that model. Other fields and the systems that support them expect you to reinvent the tools as needed.

There's a multiple imputation function for Apophenia, which basically does the last step or two of the multiple imputation process for you: given a series of fill-ins, find the statistic for each, and apply the within/across variance formula. It was a bear to write, and not because the math is all that hard. In fact, Apophenia is built from the ground up around the use of models in the sort of plug-in format, so if there's any stats system out there where writing a function to find a statistic using a parent model crossed with a fill-in model, it'd be this one. But look at how much has to be specified: parent model, possibly one model for each column of missing data, a statistic that uses all of those models at once, and the base data in a format that the first three items know how to work with. All of these—especially the aggregation of several models for each variable into a unified missing-data story—have to be specified and tested by the user before calling the multi-impute function.

As above, Apophenia has a standardized model object that can be sent to functions and thrown around internally without much fuss; to the best of my knowledge, it is currently unique in that respect, so other systems need to come up with a fussier means of specifying how the multi-impute function is to make its draws and aggregate everything together.

There's an R package named *mi* which solves the problem by requiring the user to use specific set of models, based on a Bayesian framework preferred by some of the pioneering works in multiple imputation, and as per the last episode, the combination of models can easily make use of Bayesian model-combining techniques. To use the package, you pick from a short list of named models, and away you go.

Quick—if you start with a Dirichlet prior with parameters $[\alpha_1, \alpha_2]$ and a Multinomial likelihood function $[\beta_1, \beta_2]$, what will the posterior look like? OK, time's up: it's a Dirichlet distribution with parameters $[\alpha_1 + \beta_1, \alpha_2 + \beta_2]$. So using this specific form dodges a computational bullet, so it's the sort of thing that is the focus of the `mi` package. I'm being a little unfair with this example, because the package allows much more flexibility than just this simplest of model combinations, but it's also a far cry from accepting any type of input model crossed with any type of fill-in model. R is Turing Complete, so you can do it, but expect to start from near zero and brush up on your S4 object syntax.

By which I mean to say that crossing one model against another is basically the limit of what we can organize with the tools we have today, which is a little sad.

Presentation But let's say that you're one of those people who assumes away the problem of organizing two models (one of which is a compound model for several variables) plus a statistic-calculating function plus a data set as just trivial and to be assumed away; then you still have (1) the problem of having users understand that these are the inputs they are to provide. Remember that most users got an education in a traditional statistics course that taught a series of plug-and-play finalized techniques and procedures. If all you know is OLS, then estimating an aggregate of OLS and missing data generation process is mindblowing.

Then, (2), there's the output problem. There are a number of possible outputs: most verbose would be to (A) actually report the several imputations for every data point, or you could (B) report the variance of each imputed value, or you could (C) report the larger variance of the final statistic and not bother with the internal workings.

Option (A) is the most voluminous data, and has been advocated by a decent number of people, especially for the case where there are separate people on the data-gathering side and the data-analysis. The gathering side could give ten filled-in data sets to the analysis side, and leave the analyst side to calculate the statistic of its choosing and trust that it will apply the simple total across/within variance equation correctly.

Option (B) is a middle-ground that gets difficult. We want to know how well the imputed values are fitting in, especially when the imputations are more complex than a simple multivariate Normal. This is where good data visualization comes in. We've had literature on the problem of presenting too many data points for decades now, and we have established means of coping. But in this case, the data is both voluminous and complex: each point has a span around it, and data may or may not be missing on different dimensions, meaning that that different confidence blobs may have different dimension. This is something we're still working on.

The `mi` package focuses on this, providing a heap of plots of the imputations for use in diagnosing problems. The authors do a fine job of giving good views of what is fundamentally too much information, but it's still a lot to digest, and would be hard to throw into a journal article with only a few sentences of explanation.

Option (C) is certainly the easiest to the consumer, because everything has finally been summarized to a single variance, and the reader doesn't have to care about whether the main cause was within-imputation or across-imputation variance (though that'd be easy enough to report as well). Now your only problem is to explain to readers that your variance is larger than the next guy's because you took into account problems that the other guy didn't.

The problems for all of these options at these various levels of aggregation are real but surmountable. In each case, the problem is in education: the end-user, whether an analyst or a package user or a reader, needs to understand that we're combining a parent model with a data generation model, what a good or bad data generation model will look like, and how to fit this combination model into a world where most models are just a single surface model. That is, multiple imputation is also at the edge of what a typical statistics education can accommodate.

8 Technique

8.1 Using a program as a library

13 April 2009

I often have this scenario: I have an analysis to do using some quirky data set. The first step in every case is to write a function to read in and clean the data. Along the way to doing that, I'll write functions producing summary statistics sanity-checking the data and my progress.

At this point I can get to the actual process of producing a descriptive model, and then testing that model's claims. This will all be in `modelone.c`

Next week, I have an idea for a new descriptive model, which will naturally make heavy use of the existing functions to clean data and display basic statistics. So how can I most quickly get to those functions while doing minimal damage to the original program?

In the context of C and many, many other systems, the only difference between a function library and a program is that a program includes a `main` function that indicates where execution should start. So the problem is basically in making sure that at compilation, there is exactly one version of `main` visible to the compiler at a time. Here are a few options, all of which are appropriate in some circumstances, though I'll focus on the last here.

Option one: Simply add more functions embodying the new model to `modelone.c`, and add a command-line switch to select the model.

Pros: immediate (especially if you don't use `getopt` to parse the command line). Cons: gets messy very fast. Something about a single several-page code file discourages reading.

Option two: Move all of the more useful functions in `modelone.c` to a second file, `model_lib.c`, write a header, and then #include the header in both `modelone.c` and the new `modeltwo.c`. Pros: very organized. Cons: can take time to do it right, which may not pay off for an isolated project.

Option three: Conditionally comment out the `main` function. Here is a skeleton for `modelone.c`:

```
void read_data(){
    ...
}

#ifndef MODELONE_LIB
int main(){
    ...
}

#endif
```

If MODELONE_LIB is not defined (note the use of #ifndef rather than #ifdef), then main will appear as normal, so you can compile modelone.c as normal.

If that variable is defined, then main will be passed over—and suddenly you have a library instead of a program. So modeltwo.c will look like this:

```
#define MODELONE_LIB
#include "modelone.c"

void run_second_model(){
    ...
}

int main(){
    read_data();
    run_second_model();
}
```

We've successfully used modelone.c, which had been a program file, as a library file.

Pros: you don't have to rewrite modelone.c, save for adding the if/endif. Thus, this is fast and doesn't require any re-testing of your original work. Cons: if you left a lot of globals floating around in modelone.c, you now have all of those globals floating around in your second model. This will be a good thing for some globals, but side-effects may creep in if you aren't aware of what else you're bringing in.

Adding main to a library In the other direction, there's good reason to have a self-executing library: testing. Rather than writing the actual library and then a separate file for testing, just put all the tests at the bottom of the library file, along with a main routine to run them all. Here, the default usage is to not run main, so surround it with #ifdef RUN_LIB_TESTS ... #endif, and then define RUN_LIB_TESTS only during testing. You may be able to surround all of the testing functions in the #ifdef, so non-testing library users can't see any of the test functions at all.

You can define RUN_LIB_TESTS either via a #define line at the top of the file that you keep normally commented out, or during compilation, by specifying -DRUN_-LIB_TESTS among the C flags to GCC (or via comparable means for other compilers).

8.2 Making integers two-dimensional

21 April 2009

This is a note about the generally-maligned modulo operation. For the most part, we just use it to get every nth item from a list. Or we might want to shove something into a numerical limit; e.g., I used it the other day for a check digit. The typical check-digit scheme consists of summing a list of elements and then taking that sum mod ten to reduce it to a single digit, or mod 11 to reduce it to a single digit where 10=X.

The use I'll focus on here is in jumping dimensions. Now and then, you find yourself in a linear space, but need to implement two-, three-, or n-dimensional data. Through creative use of the modulo operator, you can easily turn 2-D into 1-D and vice versa.

You'd normally use a double loop to touch every element of a matrix—one loop for the rows and one for the columns. But you can do the same using a single loop and integer division. For `int ct` = 0, 1, 2, 3, 4, ..., the pair (`ct/3`, `ct%3`) takes on the values (0, 0), (0, 1), (0, 2), (1, 0), (1, 1), (1, 2), (2, 0), ... This looks a lot like a double-loop with two variables, and you can use it to cover the same ground. That is, the following two loops behave identically:

```
int cols = m->size1;
int rows = m->size2;

for(int i=0; i < rows; i++)
    for(int j=0; j < cols; j++)
        gsl_matrix_get(m, i, j);

for(int i=0; i < cols * rows; i++)
    gsl_matrix_get(m, i/cols, i%cols);
```

To help you verify this, here's the table that both sets of loops would traverse, with both the one-dimensional index and the coordinate pair:

	$i\%3 = 0$	$i\%3 = 1$	$i\%3 = 2$
$i/3 = 0$	0 (0, 0)	1 (0, 1)	2 (0, 2)
$i/3 = 1$	3 (1, 0)	4 (1, 1)	5 (1, 2)
$i/3 = 2$	6 (2, 0)	7 (2, 1)	8 (2, 2)
$i/3 = 3$	9 (3, 0)	10 (3, 1)	11 (3, 2)

If you (or your students) are a visual learner, then the mod-as-table form gives you a potentially more comprehensible way to think about an operator with which we have limited day-to-day experience. Once you have the integers in a table, the modulo operation becomes an axis along a space. For example, the condition `if ((x % 3) == 1)` has a simple physical interpretation: it's just the second column of the table.

Returning to code writing, I'm not presenting this as a clever way to save a line of code: the (int division, modulo) version is typically bad form relative to the simple double-loop. But situations come up reasonably often when you need to put two-dimensional data into a one-dimensional space, and integer arithmetic is the way to do it.

In fact, the pattern continues for more dimensions. Let i be the one-dimensional index the system is handing you, and let dn be the size of the nth dimension in the array you would like to express. Here is the pattern of the coordinates:

2-D:				(i/d1,	i%d1)
3-D:			(i/(d1*d2),	i/d1%d2,	i%d1)
4-D:		(i/(d1*d2*d3),	i/(d1*d2)%d3,	i/d1%d2,	i%d1)
5-D:	(i/(d1*d2*d3*d4),	i/(d1*d2*d3)%d4,	i/(d1*d2)%d3,	i/d1%d2,	i%d1)

Next time, a digressive note about integer arithmetic and programming languages.

8.3 Overloaded with operator overloading

29 April 2009

Last time I discussed some pleasant uses of integer division, but I think most of us really think of it as an annoyance. We don't expect all the decimals to be truncated. If I type in `3/2`, I expect `1.5`, darn it, not `1`.

Indeed, this is an annoying gotcha to C and other integer-arithmetic languages, and more broadly, it shows us the dangers of *operator overloading*. O.o. is when an operator, like `/`, does something different depending on the types involved. For two integer types, the slash does the divide-and-truncate operation, and for anything else it does the usual division.

Oh, you can do pointer arithmetic, wherein you add a pointer and an integer. This too can lead to confusion: given `int x=3`, if you should mistakenly ask for `&x+3`, you don't get a compilation error, and the system just steps forward three steps as requested (which may or may not segfault).

Other languages actually *encourage* o.o., and give you tools to create a different meaning for `/` for any given pair of types. Well, you just saw the tradeoff: we can do things and create meaning for something that had been incoherent (like a pointer plus an integer), but if our expectations are wrong, then we have that much less keeping us from doing the wrong thing.

Human language is very redundant, which is a good thing. Redundancy is a good thing because it allows error-checking. When Nina Simone says *ne me quitte pas*, it's OK if you space out at the beginning, because ... *me quitte pas* has the *pas* to indicate negation. It's OK if you space out at the end, because *ne me quitte* ... has the *ne* to indicate negation.

Programming languages don't do this. We express negation exactly once, typically with only one character (!), and don't worry about things like case and gender. So if you space out in the first half of writing a line of code, there's nothing to call you on errors.

I'm not talking about sex enough on this blog, so here are some words for genitalia. The Spanish for penis is *pene*, masculine; the feminine equivalent *vagina*, is gramatically feminine. But as you can imagine, there are vulgar forms for when these terms sound too medicinal: the masculine *pene* becomes *polla* (f), and the girl-parts become the masculine (and very vulgar) *coño* (m). I could think of no better demonstration of how little gender in grammar has to do with actual gender. Instead, just think of them as noun classes.

So it's not about boys versus girls, but about redundancy, and giving the listener a few more clues about what the person across the room is trying to get across.

Programming languages *do* have genders, except they're called types. Generally, your verbs and your nouns need to agree in type (as in Russian, Amharic, Arabic, Hebrew, among other languages). That means redundancy, and perhaps a different verb form for the same action when executed on different types. With this redundancy, you'd need `matrix_multiply(a, b)` when you have two matrices, and `complex_multiply(a, b)` when you have two complex numbers (however expressed).

With operator overloading, of course, you don't need any of that. Express matrix multiplication as `a * b` and complex multiplication as `a * b`. This is much more brief, but you've lost redundancy.

I've said it above, but let me say it again: redundancy is a good thing. It'd be hard to confuse a complex scalar with a real matrix, but it's darn common to confuse a pointer-to-int and an int, or take a one-dimensional matrix to be a vector. As you add types, it only gets worse, and some systems will give you a list, vector, and unordered list to confuse, and the power to multiply together any two of them with `a * b`.

From here on to more complex types, there are a lot of subtleties involved. SQL, the language for manipulating database tables, is based on an algebra, meaning that there is an operation that maps to addition, an operation that maps to multiplication, a distributive property, et cetera. What if SQL were expressed as such, so you would write joins as `t1 * t2` instead of the verbose `select ... where t1.x = t2.x` form we do use? Things would be a lot more brief, but not necessarily any easier to read, write, or understand, because the `*` operator doesn't give you any information about what a product means in this context on these types. You just have to have the documentation open or have memorized the rules. The typical form for the rule is something like, 'It's sort of like multiplying scalars, but for the following additional rules and caveats....'

So, once more, redundancy is good, because the metaphor between the product in the real scalar context and the product in the context of the new type is probably only partially correct.

So there's the tradeoff: you've saved space on the page, and didn't have to type much of anything, but have lost all redundant hints that `b` is actually a list and not the vector you thought it was.

From here, the final decision is entirely subjective. I am a klutz and often commit the sort of errors I describe above, so I benefit heavily from a redundant language. You may be working primarily with only a few types that are hard to confuse, in which case all of my warnings are not an issue, and you only benefit from the brevity. But from the frequency of kvetching about how `int / int` behaves differently from `float / float`, it seems a lot of people lean toward preferring redundancy.

This is the only real example of o.o. I can think of in C. The `*` gets reused as binary multiplication and unary pointer-dereference, but those are very different actions and there's never confusion.

8.4 Structures

24 March 2011

First, let us go over the type system of every computing language in use today:

- There are basic types. These are numbers and words, in some form.
- There are lists or arrays of items, which have arbitrary length, and are indexed by a number.
- There are lists of items which are indexed by a name of some sort: dictionaries, hashes, or `structs`.

If you carry this list with you, it'll be easier to get a start in any new language. What are its basic types, how do we manipulate a list of identical types, and how do we aggergate diverse types into a structure of named elements?

[This is a lead-in for a longer series, by the way.]

Here's how the three types of type work out in practice among some popular languages:

	basic types	indexed lists	named lists
C family	int, float, char	arrays, pointers	structs
Lisp	numbers, strings	list	list
Awk	numbers, strings	—	array
Perl	$numbers, $strings	@array	%hash
Python	numbers, strings	list, tuple	dictionary
FORTRAN 77	int, float, char	arrays	—

This is off the top of my head—no need for irate letters about the things I approximated to keep the table short. But the organizational problem is the same in all cases: we sometimes have a bunch of homogeneous items, and we sometimes have disparate items, and we need a syntax for both types of organization.

There are a few quirks worth noting. Lisp is famous for using the same structure to handle both cases, which is all very neat and clean.

Awk is amusing in that it only has compound types indexed by a text name. You can fake normal indexed arrays with the naming scheme of `"1", "2", "3", ...`. Here's some sample code, set up so you can cut and paste it onto your command line:

```
echo '
BEGIN { try[0] = "zero";
        try[1] = 1;
        try[2] = 2;
        try["2"] = 8;
        for (i=0; i<=2; i++)
            print try[i];
} ' | awk -f '-'
```

You will see that `try[2]` will print `8`, meaning that the index is a string, even when you wrote it as a number. So that's nifty: if we have the dictionary/struct/hash, then we can fake simple arrays with them.

The table above counfounded two things: how we refer to an item (by number or a name), and whether the collection is homogeneous or of diverse types. For example, the awk code put both a string and integers into the same array, so awk's hashes are in the heterogeneous-holding category. For array-by-numeric-index, having homogeneous types is pretty much the norm, because if you're referring to a list of items that are only distinguished by which is first, second, third, ..., then they're probably pretty darn similar, type-wise (not to mention efficiency issues). Some languages do allow a list of differently-typed elements to be thrown into a list, so you could have `box[0]` be the box name, `box[1]` be a sublist of height/length/width, and `box[2]` be a pointer to the next box, but this is terrible form, and should be relegated to being an occasional lazy convenience. [I in fact did this in some code earlier today, so I can only be so righteous.]

I'll finish the thought next time, when I cover the named-index types: `structs` and dictionaries, and how these types are and are not similar across languages.

8.5 Structs versus dictionaries

25 March 2011

This continues the last entry (p **??**) giving a simplified (but sufficient) view of compound types across all languages. Every language has lots of cute tricks, but if you know what sort of means your language has of representing lists of numerically indexed elements, and how it deals with lists of named elements, then you can fake yourself a pretty long way along.

In the last episode, I equated `structs` and dictionaries, which may seem odd to those of you who have used both. Their intent tends to be different, as revealed by their names: `struct`s are for structured collections of data, while dictionaries are for long lists of named elements.

[The internal workings are certainly different, but the point of this post is that this paragraph on internals is a digression which you can skip. A `struct` is a variant of an array, in that it's a sequence of items at some spot in memory; the only difference is that the position has a name instead of a number, and the distance from one item to the next, in terms of transistors on the memory chip, isn't constant. A dictionary or hash is a higher-level structure, maybe a linked list or something like what I sketch out below, which somehow associates a name (a text string) with each item. Comparing two strings is computationally expensive, so the strings are typically munged into a more easily compared number—a hash. So the `struct` is a variant of the array that allows variable-length elements and names in your source code; the dictionary is a high-level data structure that happens to use strings as labels.]

All those differences aside, they do share much in common. You'll notice, for example, that none of the languages in the list from last time have both a fixed `struct` and a dictionary built in to the language: if there's a dictionary or hash, then that serves as the vehicle by which complex types get constructed. In the other direction, though, you can't use a `struct` to generate an especially long list of named elements, like a *bona fide* dictionary of English words and their definitions.

Or to put this another way:

	array	`struct`	dictionary
many homogeneous elements	yes	no	yes
some heterogeneous elements	don't	yes	yes

I explained the *don't* entry last time: your language may allow a numerically indexed array to hold a long list of heterogeneous elements, but this is lousy form; more below. The *no* entry is because `struct` declarations can only be so long here in practical reality.

At this point, some of the fans of the newer languages declare victory—the dictionary does more than the `struct`. But this is using only what is built in to the grammar of the language.

A dictionary is an easy structure to generate given what we have in the static-struct languages. Here's some C code; for consistency with the awk example from last time, you can cut and paste it onto your command line.

```
echo '
#include <stdio.h>

typedef struct {
   char *key;
   void *value;
} keyval;

int main(){
   int zero = 0;
   float one = 1.0;
   char two[] = "two";

   keyval dictionary[] = {{.key="zeroth", .value=&zero},
                          {.key="first", .value=&one},
                          {.key="second", .value=&two}};

   printf("keyval %s: %i\n", dictionary[0].key,
                    *(int*)dictionary[0].value);
   printf("keyval %s: %g\n", dictionary[1].key,
                    *(float*)dictionary[1].value);
   printf("keyval %s: %s\n", dictionary[2].key,
                    (char*)dictionary[2].value);
}
' | gcc -xc '-'; ./a.out
```

Once you write a `find_key` function, this can work as a full-blown dictionary. [The thing about knowing the types on output can also be worked around via creative macros, but for most applications you don't need to.] Writing this function is left as an exercise to the reader, but it's just an instructional exercise, because fleshing this out and making it bulletproof has already been done by other authors; see the GLib's keyed data tables or `GHashTable`, for example. The point here is simply that having compound structs plus simple arrays equals a short hop to a dictionary. If you are coming from a dictionary-heavy idiom to the C family, then you'll have to split your dictionary uses into `struct`-like short lists and long lists, and use a structure out of GLib (or Boost, or whatever is appropriate) for the long homogeneous lists with a name index.

OK, so we've seen (last time) how Awk uses named lists to fake numbered lists. We put named lists into numbered arrays to generate key-value lists. What if we have only simple numbered arrays?

FORTRAN 77 (which is not Fortran 90 or later versions) lacks the ability to declare complex types. This is true of many of the punched card languages first developed in the '60s and 70s. If you want a structure listing dimensions one through three, population count, and workspace size, then declare an integer array of size 5 and just remember that `iv[1]` through `iv[3]` are the dimensions, `iv[4]` represents population, `iv[5]` represents workspace size, and so on. This is what I'd above referred to as bad form, and the language all but forces you to do it. And now that you have your array of integers, set up another array `fv` for the floating-point values. [Exercise: given only numerically-indexed arrays of homogeneous types, how would you set up a key/value structure? If you wind up with four or five arrays, is there any way to bind them into one parent structure, so you don't have to send all those arrays to every function that uses the structure? [answer: no, not in F77.]] If you want a linked list where item 4 points to item 2 which points to item 6, then declare an array of integers and write 2 in location 4 and 6 in location 2, and perhaps -1 in location 4 to indicate the start of the list. If that sentence confused you, try writing (or debugging) a whole program in that style.

Here's an actual code snippet, a function call cut and pasted from archives of FORTRAN routines (I ran it through `f2c`; the R project uses this in the orignal FORTRAN):

```
ehg131(xx, yy, ww, &trl, diagl, &iv[20], &iv[29], &iv[3], &iv[2],
&iv[5], &iv[17], &iv[4], &iv[6], &iv[14], &iv[19], &wv[1] ,
&iv[iv[7]], &iv[iv[8]], &iv[iv[9]], &iv[iv[10]], &iv[iv[22]],
& iv[iv[27]], &wv[iv[11]], &iv[iv[23]], &wv[iv[13]], &wv[iv[12]],
& wv[iv[15]], &wv[iv[16]], &wv[iv[18]], &i__1, &wv[3], &wv[iv[26]],
&wv[iv[24]], &wv[4], &iv[30], &iv[33], &iv[32], &iv[41], &iv[iv[25]]
&wv[iv[34]], &setlf);
```

I could explain what `iv[1]` through `iv[41]` stand for, but it wouldn't help. This is as write-only as code gets.

I've made some effort to translate some such code to C, and it's something I really regret. The problem with code like this is not that it's in a now-unpopular language, but that it's in a language that doesn't support named, heterogeneous data structures.

There's more: objects (structs/dictionaries with functions in them), variants like Python's tuples, sets, bags, and everything else you learned about in your data structures textbook. But if you're a tourist to a new language and have to get things done fast, the above will be a good start. In an episode or two I'll really expand this point.

8.6 Easy re-typing with designated initializers

1 November 2009

This column is about dealing with multiple formats for the same thing. To give the simplest example, consider a plain old list of numbers. The raw representation is an array, where the numbers are a sequence in memory. But then you don't know how long the thing is, so you need to also have a note somewhere as to its size. Maybe you want to name it, or treat it like 2-D matrix. Next thing you know, you've got a long list of extra data taped to that simple list.

Now you've got design problems. In terms of the systems I work with, you've got several levels of intent and complexity, including the simple `double *`, the `gsl_vector*`, the `gsl_matrix*`, and the `apop_data*` structures, any of which could be used to represent a few numbers.

These different structures aren't just there for fun: a scalar doesn't necessarily behave like a 1×1 matrix or a one-unit vector.

- a scalar times a $N\times1$ column vector is usually read to produce a scaled $N\times1$ vector.
- A 1×1 matrix $\cdot$ a N-unit vector is a similar scaling operation, but here we'll have to assume that the vector is a row vector, and the output will be either a 1×N matrix or a N-element vector understood to be a row, depending on custom.
- A vector dot a vector is usually taken to mean a row vector x dot a column vector y, producing $x_1y_1+x_2y_2+\cdots+x_ny_n$. But a vector of length one doesn't match dimensions with a vector of length $N > 1$, so in this case we'd just throw an error.

There are already a lot of subtleties, like whether we want to be explicit about whether a vector is a row or a column, or just assume that it'll do whatever is needed to conform, or whether the output wants to be a scalar, vector, or matrix.

Dealing with complexity These different, sort of overlapping types are necessary, but they inevitably add complexity to the system. There are some methods for dealing with these different types, all of which have their benefits and bugs.

- Just make everything the most inclusive structure. Pros: users don't have to think: everything is a named N-dimensional frame of long long double-precision floating-point numbers, labeled with arbitrary-length strings, and there's no need to worry about sub-types and such. Cons: writing down the number 14 is now a massive production. Now you couldn't distinguish a scalar from a 1×1 matrix if you wanted to.
- Overload functions, so a function can take any representation of a list of numbers as input, and handles the differences internally. This gives surface ease, because the function user usually doesn't have to think too hard about types. But if it's a

`double*` you still need to remember to send in an extra length parameter, and it's hard to encode the above scalar/vector/matrix subtleties into such a system, because you're never quite sure how a function will read your inputs. The bugs produced by subtle differences like these are, in my experience, among the most difficult to debug.

- Have the user do the type-casting between things: pulling smaller elements out of the larger structs, and building purpose-built parent structures to wrap the smaller stuff. Cons: you need to know the structures, and have to do the work of explicitly stating things. Pros: the process of subsetting takes zero computer time, and the process of wrapping is not necessarily annoying, as discussed below.

None of these methods are ideal, and which devil you choose is a matter of local considerations, practical issues, and personal taste. I gravitate toward the third, wherein the user is expected to know the darn underlying hierarchy of types, and deal accordingly. Why? Because I've found that systems that hide that hierarchy from you do a lousy job of it. In case this essay isn't long enough, have a look at this essay on the law of leaky abstractions[1], which explains that you can get away with not explicitly acknowledging the different types for a while, but eventually you're going to have to confront the differences. With good technique, it's not hard to switch types on the fly.

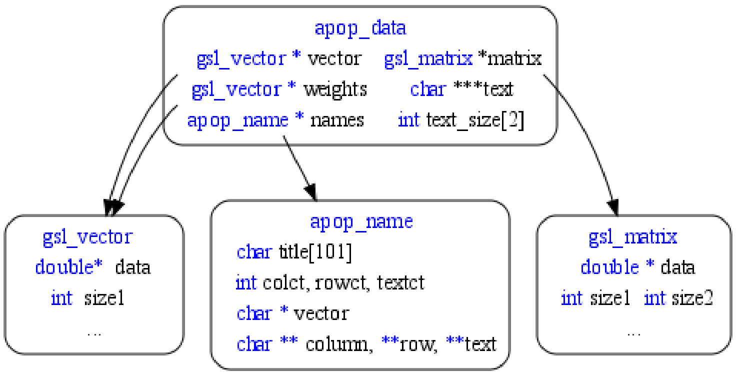

Figure 8.1: The `double*` to `gsl_matrix/_vector` to `apop_data` hierarchy.

Making it easy Getting elements out of a structure is pretty easy: just point to it. Because you can have multiple pointers pointing to the same thing, it's easy to rename something that is deeply nested inside the hierarchy. E.g., an `apop_model` (which would float above the type diagram above) holds parameters in an `apop_data` set, which holds a `gsl_vector`, which holds a list of `double`s. So:

```
double *list = my_model->parameters->vector->data;
```

[1]`http://www.joelonsoftware.com/articles/LeakyAbstractions.html`

```
do_something(list);
do_more(list[3]);
```

Since the new name is just a spare pointer to the same data, all changes to the data (without moving the pointers themselves) happen as expected, you didn't copy any data, and you don't have to free anything at the end. Clean and simple.

Going up the hierarchy is more difficult, because you need to add all that extra data yourself; one paragraph was enough to cover going down the tree, and the rest of this column will be about going up the tree. I'll start from the most verbose, and work my way toward the easier methods, so don't get discouraged by the part where I use ten lines to take a dot product—I'll have it back down to one in the end. [As we C users like to say, there are an infinite number of ways to do it.]

There are functions to wrap things. For example, the `apop_dot` function has a quite clean syntax that takes in two `apop_data` structs (plus optional parameters indicating transposition), but if your data isn't in that input format, you'll need to wrap it. Here's an example where we know that we're multiplying a 3×2 matrix against a two-element column vector, using a function to copy data to the right structure:

```
apop_data *a_dot(double *set1, double *set2){
    apop_data *d1 = apop_line_to_data(set1, 0, 3, 2);
    apop_data *d2 = apop_line_to_data(set2, 2, 0, 0);
    apop_data *out = apop_dot(d1, d2);
    apop_data_free(d1);
    apop_data_free(d2);
    return out;
}
```

If you're just doing this once, the deallocations at the end may be optional, but if you're writing a function to be called a million times, they'll become essential.

For matrices or vectors, you could produce a dummy wrapper and then point to the data. But don't forget to unlink before calling the free function, lest you lose the original data:

```
apop_data *another_dot(gsl_vector *v, gsl_matrix *m){
    apop_data *dummy1 = apop_data_alloc(0,0,0);
    apop_data *dummy2 = apop_data_alloc(0,0,0);
    dummy1->vector = v;
    dummy2->matrix = m;
    apop_data *out = apop_dot(dummy1, dummy2);
    dummy1->vector = NULL;
    dummy1->matrix = NULL;
    apop_data_free(dummy1);
    apop_data_free(dummy2);
    return out;
}
```

That is unabashedly a lot of work for one dot product.

The first way in which we can save the trouble of deallocating is to use the `static` keyword to guarantee that a shell will always be on hand to fill. [If you're not familiar with static variables, see pp 39–40 of *Modeling with Data*.]

I do this sort of thing so often that I even have a convenience macro to simplify the process.

```
#define Staticdef(type, name, def) static type name = NULL; \
                                   if (!(name)) name = def;

apop_data *easier_dot(gsl_vector *v, gsl_matrix *m){
    Staticdef(apop_data*, dummy1, apop_data_alloc(0,0,0));
    Staticdef(apop_data*, dummy2, apop_data_alloc(0,0,0));
    dummy1->vector = v;
    dummy2->matrix = m;
    return apop_dot(dummy1, dummy2);
}
```

The next step in the chain is to just produce that dummy structure on the fly, which is where designated initializers come in.

```
apop_data *easiest_dot(gsl_vector *v, gsl_matrix *m){
    apop_data dummy1 = {.vector = v};
    apop_data dummy2 = {.matrix = m};
    return apop_dot(&dummy1, &dummy2);
}
```

What just happened: we used designated initializers [p 32 of *Modeling with Data*] to allocate a structure and fill one element. The elements not explicitly mentioned are zero, so we don't have to worry about them. This works for the `apop_data` structure because it is designed to be OK with being mostly empty; below we'll see some structures that are a bit more needy.

That trick allocated an `apop_data` struct, but you'll notice that every library function takes in a pointer: `apop_data*`. This distinction is why we need to use `&dummy1` instead of just `dummy1` when making the function call. But this setup means that we don't have to deallocate anything at the end: the structure is cleaned up automatically when the function exits.

Some people are lines-of-code averse, and really hate the idea of having those extra lines of code producing extra structures. So, just do it in place:

```
apop_data *one_line_dot(gsl_vector *v, gsl_matrix *m){
    return apop_dot(&((apop_data) {.vector = v}), &((apop_data) {.matrix = m}));
}
```

I like the three-line form better, myself, partly because I need the `(apop_data)` type cast when not on the declaration line. Maybe some macros will clean up the second form:

```
#define d_from_v(v) &((apop_data) {.vector = v})
#define d_from_m(m) &((apop_data) {.matrix = m})

apop_data *one_line_dot(gsl_vector *v, gsl_matrix *m){
    return apop_dot(d_from_m(m), d_from_v(v));
}
```

Going from a raw array to the GSL's vectors and matrices require a little more care, because you'll need to add some metadata: the number of rows/columns, and the requisite jumps.

```
apop_data *a_dot_again(double *set1, double *set2){
    gsl_vector m = {.data = set1, .size1=3, .size2=2, .tda = 3};
    gsl_vector v = {.data = set2, .size=2, .stride = 1};
    return apop_dot(d_from_m(m), d_from_v(v));
}
```

The `tda` (trailing dimension of array) and `stride` elements tell the system how to convert the 1-D layout in memory into the right shape. For subvectors and submatrices, the jumps may take different forms, but for our purposes, the tda is always equal to the row size, and the stride is always one. With that in mind, we can wrap these details in macros, and daisy-chain it all together:

```
#define v_from_a(v, size) &((gsl_vector) {.data = (v),\
                      .size =(size), .stride = 1})
#define m_from_a(m, size1, size2) &((gsl_matrix) {.data = (m),\
                      .size1 =(size1), .size2=(size2), .tda = (size1)})

apop_data *a_dot_again(double *set1, double *set2){
    return apop_dot(d_from_m(m_from_a(set1, 3, 2)), d_from_v(v_from_a(set
}
```

In the end, you're still going to have to climb your way up the hierarchy a few steps for this array-to-data case to work. It's up to you if you want to take that last step and write a more macros:

```
#define dv_from_a(a, size) d_from_v(v_from_a(a, size))
#define dm_from_a(a, size1, size2) d_from_m(m_from_a(a, size1, size2))
```

All of these macros are cheap, in the sense that they allocate short structures and don't copy any of your data. Also, they're a whole lot shorter than the ten-line version.

On the con side, I think there exist people who would call them bad style, because you're not using the formal methods of allocation (e.g., `gsl_vector_alloc`), and are thus bypassing checks that things are OK. Situations that depend on those ignored structure elements having non-`NULL` values may surprise you in odd cases.

There's the problem that by skipping the setter functions, you're assuming knowledge of the internal structure of the struct that shouldn't be your problem—which is true: you the user shouldn't really have to care about `tda`s. At least you can look those details up once and hide them in a macro. [This argument usually continues that the underlying structures might change as the designers come up with new ideas, but this is not seriously an issue. Early on, structures change, but at this point, the GSL and even Apophenia have a sufficiently large base of users that arbitrarily screwing around with core structures is a social impossibility. So, frankly, the macros here are not as bad form as the textbooks say they are.]

Summary paragraph: There's real benefit to having different types: a scalar is just not a 1×1 matrix or a one-item vector, so we need to be able to specify all these structures. But, as a direct corollary, we need to be able to easily jump between structures as necessary. In this column, I gave you nine examples of how to take a dot product, depending on the inputs. Our pals designated initializers and compound literals saved the day, because they let us set up a quick structure, fill it, and use it without worrying about memory and deallocation. You can apply these tricks in a variety of situations; for those of you who might follow exactly the array-to-matrix-to-data forms above, you will find all the above macros in one cut-and-pasteable block in the web version.

8.7 Scope in C

7 September 2011

OK, here goes: all of the rules for variable scope in C.

- A variable never has scope in the code before it is declared. That would be silly.
- If a variable is in curly braces, then at the closing curly brace, the variable goes out of scope. Semi-exception: `for` loops and functions have variables in parens just before their opening curly brace; variables declared within the parens have scope as if they were declared inside the curly braces.
- If a variable isn't inside any curly braces, then it has scope from its declaration to the end of the file. Semi-exception: you can use the `extern` keyword to refer to a variable in another file.

OK, you're done.

There is no class scope, prototype scope, friend scope, namespace scope, dynamic scope, extent issues, or special scoping keywords or operators (beyond those curly braces). Does lexical scoping confuse you? Don't worry about it. If you know where the curly braces are, you can determine which variables can be used where.

In fact, most C textbooks (including *Modeling with Data*) make this more complicated than necessary by talking about functions as separate from curly-brace scope, rather than being just another example. Here is a sample function, to sum all the values up to the input number:

```
int sum (int max){
    int total=0;
    for (int i=0; i<= max; i++){
        total += i;
    }
    return total;
}
```

Then `max` and `total` have scope inside the function, by the curly-brace rule and the semi-exception about how variables in parens just before the curly brace act as if they are inside the braces. The same holds with the `for` loop, and how `i` is born and dies with the curly braces of the `for` loop.

In fact, forget about where they taught you to put curly braces, and let's just throw them in wherever we want some more scope restrictions.

You might want them around macros, for example. I'll have more examples in the near future, but here's a simple one:

```
#include <stdio.h>

#define sum(max, out) {             \
    int total=0;                    \
    for (int i=0; i<= max; i++){    \
        total += i;                 \
    }                               \
    out = total;                    \
}

int main(){
    int out;
    int total = 5;
    sum(5, out);
    printf("out= %i original total=%i\n", out, total);
}
```

I just turned the above function into a macro, but even as a macro it still needs an intermediate variable for summing elements. Putting the whole macro in curly braces allows us to have an intermediate variable named `total` independent of whatever is going on outside the macro.

Using `gcc -E curly.c`, we see that the preprocessor expands the macro as below, and following the curly braces shows us that there's no chance that the `total` in the macro's scope will interfere with the `total` in the `main` scope:

```
int main(){
    int out;
    int total = -1;
    { int total=0; for (int i=0; i<= 5; i++){ total += i; } out = total;
    printf("out= %i total=%i\n", out, total);
}
```

[But we aren't protected from all name clashes. What happens if we were to write `int out, i=5; sum(i, out);`?]

Summary paragraph: C is awesome for having such simple scoping rules, which effectively consist of finding the end of the enclosing curly braces or the end of the file. You can teach the whole scoping system to a novice student in maybe ten minutes. For the experienced author, the rule is more general than just the curly braces for functions and `for` loops, so you can use them for occasional additional scoping restrictions in exceptional situations.

8.8 Object-oriented programming in C

22 April 2010

Here are notes on object-oriented programming (OOP) in C, aimed at people who are OK with C but are primarily versed in other fancier languages.

The OO framework is in some ways just a question of philosophy and perspective: you've got these blobs that have characteristics and abilities, and once you've described those blobs in sufficient detail, you can just set them off to go running with a minimum of outside-the-blob procedural code. If you want to be strict about it, the objects only communicate by passing messages to each other. All of this is language-independent, unless you have a serious and firm belief in the Sapir-Whorf hypothesis.

Scope Much of object-oriented coding is distinguished via a method of scoping. Scope indicates what, out of the thousands of lines of code and dozens of objects you've written down, is allowed to know about a variable. The rule of thumb for sane code-writing is that you should keep the scope of a variable as small as possible to get the job done. Think of a function as a little black box: you want it to have just as many exposed parts as are necessary to interoperate with the outside world.

From the OOP perspective, this translates into dividing variables into private variables that are only internal to the object, such as the internal state of the car's motor, and things that the whole world can use, such as the location of the car. Thus, every OO language I can think of defines `public` and `private` keywords.

But wait, there's more: sometimes, you really have to break the rules, just this once, and check the internal status of the motor. You can make the status variable global, defeating the whole mechanism, or you can define a `friend` function. Below, we'll have inheritance, and will also need `protected` scope. Sometimes, the `::` operator will get you out of a jam.

That is, we can divide the OOP additions to C's syntax into two parts: syntax to give you stricter, finer control over scope, and syntax to override those stricter controls.

How does C do scope, given that it has (depending on how you count) about two keywords for scope control? The scoping rules for C are defined by the file. A variable in a function is visible only to the function; a variable outside the functions, at the top of a file, is visible only in that file.

A typical `file.c` will have an accompanying `file.h` that simply declares variables and functions. If another file includes `file.h`, then that file can see those variables and functions as well. Thus, the private variables are invisible outside the file, and the public variables declared in the header can be used by the other files where you choose to include `file.h`.

The variables in the header file need to be declared with the `extern` keyword, e.g. `extern float gas_gauge`, which indicates that the variable is actually declared in a single `.c` file, say `dashboard.c`, but another `.c` file that includes this header, maybe `motor.c` is being made aware that there is an floating-point variable named `gas_gauge` declared somewhere external to `motor.c`. As noted, the custom is to put all these `extern`s in a header file and be done with it, but if you want to have especially fine control of scope across files/objects, then you can insert exactly as many `extern` declarations as you need exactly where you need them; similarly for function declarations.

The naming thing Objects let you name functions things like `move` and `add` and never worry about interfering with fifty other functions with the same names. This is nice, but there is a simple C custom to take care of that: prepend the object name. Instead of the C++ `my_data.move()`, where you just understand that this `move` function refers to an `apop_data` object, you'd have a function with a name like `apop_data_move(my_data)`. There ya go, crisis averted: no name space clashes. Some readers somewhere may complain that the name-prepending is ugly, to which I respond: care less.

But seriously, go have a look at Joel the guru[2] for more on how wonderful naming similar to this can be.

C already has a scoping system comparable to that of C++ if you use the one file-one object rule and a few customs in naming. Adding a whole new syntax for scoping on top of this is basically extraneous, and could create confusion now that you've got two simultaneous scoping systems in action.

Inheritance and overloading Overloading functions and operators is dumb. Joel's article above has a humorous bit about this, which opens: "When you see the code `i = j * 5;` in C you know, at least, that `j` is being multiplied by five and the results stored in `i`. But if you see that same snippet of code in C++, you don't know anything. Nothing." The problem is that you don't know what `*` means until you look up the type for `j`, look through the inheritance tree for `j`'s type to determine *which version* of `*` you mean, et cetera.

Say you have a `blob` object which includes a `cleave` method that splits the blob in half, and a `blobito` object that includes a `cleave` method that binds together internal elements. You have a `blob` object named `my_b`, and, *faux pas*, think that it is a `blobito` object. You call the `my_blob.cleave()` function, expecting that `my_b.size` will double, but instead it halves.

This may sound like a silly example, but from my experience, the most common use of OO machinery such as inheritance is where two objects are very similar but subtly different. Textbook examples: `accountant` and `programmer` objects that both inherit from the `officedrone` object, or from the U.S., `state` and `district`

[2] `http://www.joelonsoftware.com/articles/Wrong.html`

objects that inherit from the generic `us_division`, and are identical except that the `district` object has no `senators` or `representatives`.

Those situations where two objects are similar and therefore easily confused are the ones where we most need a syntax that breaks when we make a mistake in guessing the type. If you were doing this in C, you would be notified of your error at compile time (because you'd be calling `blobito_cleave(my_blob)` when you should be calling `blob_cleave(my_blob)`). In many interpreted languages, you would be notified of your error at run time, or sooner depending on the language. In C++, with appropriately defined methods, you would never, ever be notified of your error. That is, operator overloading allows you to bypass a large number of safety checks.

I promised you notes on how C does it, not rants about overloading, so let us move on to Option B: inheritance via composition. For example, Apophenia has an `apop_data` type:

```
typedef struct apop_data{
    gsl_matrix  *matrix;
    apop_name   *names;
    char        ***categories;
    int         catsize[2];
} apop_data;
```

[This essay was originally written in January 2006, and Apophenia has evolved and stabilized since then. So the sample code is not apop-accurate, but is still fine for getting across the principles discussed here.]

In OOP-speak, the `apop_data` structure is a multiple-inheritance child of the `gsl_matrix` and `apop_name` structures (plus an array of strings). All of the functions that operate on these parent objects can act on elements of the child `apop_data` structure, and life is good. To go further with OOP jargon, C lets you extend a structure via a *has-a* mechanism: we have a `gsl_matrix` and want to give it names, so we create a structure that *has-a* matrix and names. Typical OOP languages allow you to extend via *is-a*, wherein your `named_matrix` *is-a* `gsl_matrix` plus the additional elements to add names. I'm no OO pro, but I think you're supposed to read those sentences like the Italian chef in a Disney movie.

On the one hand, having only *has-a* to work with means that if a function acts on a `gsl_matrix *` you can't transparently call, e.g., `apop_pca(apop_data_set)` [PCA=principal component analysis]; you have to know that there's a `gsl_matrix` inside the data set and that's what's being operated on: `apop_pca(apop_data_set->matrix)`. On the other hand, you can not accidentally call the wrong instance of the function and then spend an hour wondering why the function didn't operate the way you'd expected.

So on the minus side, the internals of the object aren't hidden from you—but on the plus side, things aren't hidden from you.

The void, templates And finally, for when you really don't want to deal with types, there's the `void` pointer. Here's a snippet from an early draft of Apophenia's `apop_model` type:

```
typedef struct apop_model{
    char    name[101];
    apop_model * (*estimate)(apop_data * data, void *parameters);
    ...
} apop_model;
```

Two things to note from this example. First, including a function inside a struct is a-OK. We'll declare a GLS estimation function as `static apop_estimate * apop_estimate_GLS(apop_data *set, gsl_matrix *sigma)`, declare the model via something like: `apop_model apop_GLS = {"GLS", apop_estimate_-GLS, ...};` and then we can call `apop_GLS.estimate(data, sigma);` just like we would in C++-land.

[If you didn't follow the syntax of declaring hooks for functions, see p 190 of *Modeling with Data*.]

Second, there's the `void` pointer at the end of the declaration of the `estimate` method in the structure. Notice that that second argument of `apop_estimate_GLS` is typed as a `gsl_matrix *`, even though we're plugging it in where the template asked for a `void *`. [Non-OOP quiz question for the statisticians: why is this a terrible way to implement GLS?] Other models require different parameters, like the MLE functions take parameters for the search algorithm, but they're also called via the same `model_in-stance.estimate(data, params)` form.

It's up to you, the user, to remember what types make sense for what models, because the `void` pointer is your way of saying ”Dear C type-checker: leave me alone.” The type-checker will still check that you're sending a pointer and not data, but from there you're free to live it up and/or segfault.

The void pointer is how you would implement template-like behavior. For example, here is a linked list library (gzipped source) that I wrote when I was avoiding harder work. It links together void pointers, meaning that your list can be a linked list of integers, strings, or objects of any type. How's that for a nice, concrete example.

The primary benefit from C++'s template system over using `void` pointers is that the template system will still check types. Personally, I've rarely had problems. If I have a list named `list_of_data`, I know to not add `gsl_matrix`es to it. Not having type-checking means that it's up to me to make sure that the wrong thing is never the intuitive thing to do.

By the way 1: notice how `apop_estimate_GLS` is declared to be `static`, so outside the file it's only accessible as the `apop_GLS.estimate()` method.

By the way 2: I can't recall ever using this, but if you wanted to, you could even type-cast inside the function:

```
void move(void *in, char type){
    if (type == 'a')
        a_move((a_type*) in);
    if (type == 'b')
        b_move((b_type*) in);
}
```

This self I've only wanted something like the `this` or `self` keyword maybe twice, but I have no idea how to gracefully implement it in C, if at all. [Maybe with the preprocessor?] So I'm open to suggestions on this one.

OK, there you have it: most of the basics of object-oriented programming implemented via relatively simple techniques in C. The moral: object-oriented coding is a method and a mindset, not a set of keywords.

Refs More essays along the same lines:
A full book[3] that goes into great detail about the above simple tricks, and also goes much further in implementing something that looks like C++.

An article that focuses on encapsulation, with some suggestions on hiding data.

Another article that blew way past my attention span, and basically shows you how to write a C++ compiler in C. Given my disdain for overloading and strict inheritance (as opposed to inheritance via composition), I wasn't really into it.

8.9 Yet another Git tutorial

8 November 2009

Git is a revision control system, meaning that it is designed to keep track of the many different versions of a project as it develops, such as the stages in the development of a book, a tortured love letter, or a program.

Here's a typical story: you begin chapter one, and commit it to the repository. Your coauthor is working on chapter two, and does the same. Tomorrow, you pull out the current version of the repository, and both chapters are now on hand for you to revise. Later, your coauthor calls and tells you that she was robbed and used her laptop to block a bullet, and you reassure her that it's no problem, because the draft is safely stored in the repository. She gets a new laptop, checks out the current state of the project, and is back to revising your and her work so far.

I typically put even small solo projects under revision control, because it makes me a better writer/coder: I'm more confident deleting things when I know that they're safely stored should I want them back. Git makes this easy, as you'll see below.

Git's history is relevant to its use. It was written by the guy who originally wrote Linux, Linus Torvalds, with the intent of supporting Linux development. Linus is a communist in the best possible way, and thus pushes Git at a means of easy collaboration among equals. So if you like the idea of collaborative development, and especially if you're a computer geek, then Git will enthuse you.

To me, the most interesting thing about Git is just how many tutorials there are about it. It's a complex system, and people have interesting reactions to it. Some tutorials break through the complexity by just giving *to do this, type this* instructions; others get enthusiastic and effuse about the clever structure of the system more than showing you how to use it. For my purposes, I need something to explain the underlying concepts, and not condescend to the reader, but not confuse the story with all the

[3]`http://www.planetpdf.com/developer/article.asp?contentid=6635`

details that are basically irrelevant to those who don't need to create clean patches (or know what a patch is) and don't see a need to cryptograpically sign our book chapters.

I will assume that you are familiar with the basics of POSIX directories, files, text editing, and the usual basic commands like `cp`, `mv`, `rm`, and so on.

The structure A revision control system does two difficult tasks: it has to organize a pile of slightly different versions of a project, and it has to merge together revisions, say when you and a pal are separately working on the same project and finally come together.

That first part is already enough to get lost: you have the version of the file before you, the fifteen earlier versions you wrote, and the twenty earlier versions that your colleague wrote. As noted above, the modern trend is toward distributed version control, meaning that you may have a repository in your home directory, and there may be a remote repository, and all those repositories have some number of versions. All this multiplicity is great because every little change is tracked, history can be interrogated and compared, and completely screwing up your local repository just means you have to start over with a new copy.

But all this multiplicity means that you need to know the address of where you are and where you want to go—a typical simple setup, like the one in Figure 8.2, is already overwhelming if you don't know how to get around. So, we will begin with an overview of the repository-branch-version-file hierarchy that you will be navigating.

- There may be several *repositories.*
- Each repository holds several *branches.*
- Each branch holds several *versions* (aka *commit objects*).
- At any one time, you are looking at exactly one version of the project, and the several files that comprise that version.

The repository The repository is a pile of versions, in a binary format that Git can read and you can't. Here is how you would create a new one, in a given directory that will be the repository from then on:

```
mkdir new_project
cd new_project
git init
```

You now have a blank repository. [If you are putting an existing project under revision control, you will need to add existing files to the index with `git add .`; see below.] Git stores all its files in a directory named `.git`, where the dot means that all the usual utilities like `ls` will take it to be hidden; after the init step, you can look for it via, e.g., `ls -a`.

This is the first embodiment of Linus's egalitarian communism: he wanted to make it easy to create lots of repositories on lots of machines. The key to this is that the `.git` subdirectory holds all the information, so copying your project directory (including the `.git` subdirectory) generates a new, entire respository. If you want to back up your project and repository, just recursively copy your project directory:

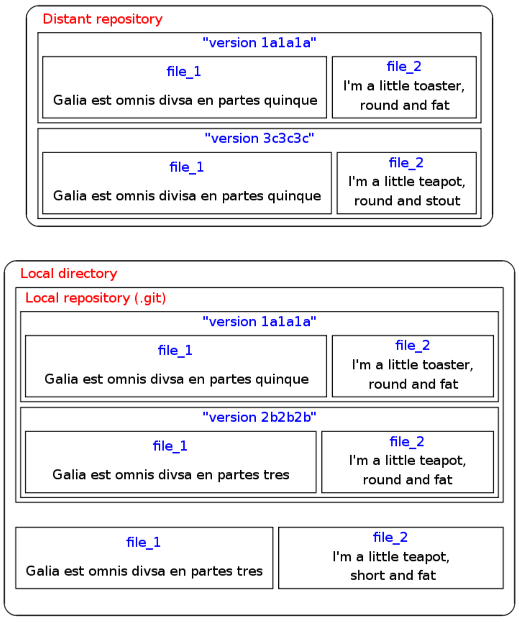
Distant repository
"version 1a1a1a"
file_1
Galia est omnis divsa en partes quinque
file_2
I'm a little toaster, round and fat
"version 3c3c3c"
file_1
Galia est omnis divisa en partes quinque
file_2
I'm a little teapot, round and stout
Local directory
Local repository (.git)
"version 1a1a1a"
file_1
Galia est omnis divsa en partes quinque
file_2
I'm a little toaster, round and fat
"version 2b2b2b"
file_1
Galia est omnis divsa en partes tres
file_2
I'm a little teapot, round and fat
file_1
Galia est omnis divsa en partes tres
file_2
I'm a little teapot, short and fat

Figure 8.2: ¡Multiplicity! Here, there are three places where a version of the project could be: a distant repository, in your local directory in a repository named `.git`, or checked out in your local working directory. In the story so far, there are four versions: `1a1a1a`, which is present in both the local and remote repositories; `2b2b2b`, where you reallocated Gaul and became a teapot; `3c3c3c`, where someone else fixed a typo in *divsa* and checked it in to the distant repository; and the version you are looking at in your current directory.

```
cp -r new_project project_bkup
```

Versions Now create a new file for yourself, using your favorite text editor or what-have-you.[4] Git doesn't know about this file yet; you have to tell the machine about it using the index (discussed below). For now, try

```
git add .
```

to add to Git's index everything in the current working directory (in UNIX-speak, `.`).

Now you can save your work to the repository in your working directory, or in revision control jargon, commit your changes:

```
git commit -a -m "What I've been doing up until this check-in."
```

[4]Git, like every revision control system I know, works best around the POSIX paradigm of relatively short lines of text. Computer code naturally looks like this, as does typical human-written plain text (with linewrap enabled so you don't have one line per paragraph). If you're using a binary format like a word processor document, then Git will have trouble making meaningful merges, and you're basically stuck using whatever revision control the word processor vendor gave you (if any).

You can make more changes to your text file, and then re-run `git commit -a -m "..."` as often as you wish, thus creating a history of commits that you'll be able to refer to below. Notice that once you've put a file into the index, you don't need to re-run `git add ...`.[5]

I've been avoiding the term *commit objects* as being a little too jargon, but the jargon does get across the idea of a single committed blob, which is treated as a unit for our purposes. You can use `git log` to get the list of commits in your history. The log shows two relevant pieces of information: the 40-digit SHA1 cryptographic hash, and the human-language message you wrote when you did the commit. The SHA1 hash is a computer-scientist clever means of solving several problems, and is the name you will use for the commit. Fortunately, you need only as much of it as will uniquely identify your commit. So if you want to check out revision #fe9c49cddac5150dc974de1f7248a1c5e3b33e89, you can do so with

```
git checkout fe9c
```

With that command, you've gone back in time, and have in your current directory whatever you had back then. Take notes, copy off that paragraph you wish you hadn't deleted, then

```
git checkout master
```

to return to the head of the master branch (which is where you started off, being that we haven't discussed creating new branches yet).

[I suggested that you take notes and make observations, but not that you change anything. ¿What would happen if you were to build a time machine, go back to before you were born, and kill your parents? If we learned anything from science fiction, it's that if we change history, the present doesn't change, but a new alternate history splinters off. So if you check out an old version, make changes, and check in the changed version, then you've created your first branch off of the master branch.[6]]

Git is designed to make it as easy as possible to bounce back to an alternate version and bounce back to where you were, as often as you need. But this may not be ideal, because you may want to have both versions living side-by-side. The easiest way to do this is to just make yourself a second repository.

```
cp -R /path/to/maindir new_tempdir
cd new_tempdir
git checkout fe9c
```

Now you can do side-by-side comparisons between the main version in the main repository and your disposable copy. [There's also a `git clone` command, see below, that does about the same as this in a slightly slicker manner.]

[5]Much of git's advanced technique is about rewriting the history to produce a smoother course of events. This document makes a point of not worrying about the history, but I will mention one nice feature to keep your log from filling up with small commits. After you commit, you will almost certainly slap your forehead and realize somthing you forgot. Instead of just doing another commit, you can do `git commit --amend -a` to add to your last commit.

[6]You haven't named your branch, which can create problems. Getting ahead of the story, if you ever call `git branch` and find that you are on `(no branch)` then you can run `git branch -m new_branch_name` to name the branch you've just splintered off to.

I've found the ability to quickly jump around in time to be immense fun, and has made me a more confident editor. When in doubt, I make the cut, and know that the worst punishment for an error is the small bother of checking out an old version.[7]

The index versus your files We've looked at prior check-ins; now ¿what will the *next* check-in look like? Git maintains what is called the index, which is the nascent list of files that will become a commit object when you next call `git commit`. That index is not identical to the files you see when you do a directory listing, for a few reasons. First, many systems produce annoying files like log files, object code, and other mid-processing cruft, and you don't want those taking up space in the repository. There are also advanced commit strategies wherein you may change several files, but want to save only the changes you'd made in one or two.

Regardless of the rationale, bear in mind that the working directory you are looking at is probably a mix of files Git is tracking and files Git doesn't care about. If you want to add a new file to the repository, remove an old one, or fix the name of a file, you'll need to do one of:

```
git add newfile
git rm oldfile
git mv flie file
```

so that the change is evident both in the working directory and in the index. As for files that you are modifying but are not shuffling around in the filesystem, you technically have to add those one by one as well by running `git add modified_-file` with every single commit, but the `-a` in the command `git commit -a` tells git to save all modifications on known files (including removal), thus saving you all that tedium. In my own work, I have never encountered a reason to commit without the `-a` flag, but perhaps you may one day run across something.

But do remember that `git add, rm, mv` only change the index of what the next commit will look like. The commit won't actually happen until you say so with `git commit -a`.

Branches To this point, you have been alternating between doing work and saving it via `git commit -a`, thus producing a sequence of committed versions of your program. That's a branch.

Perhaps *thread* would be a better term, being that this represents a single thread of your work conversation. By default, you are on a branch named *master.* Other threads come up in two manners: your own work may digress, or you may have colleagues who are following their own threads. The typical story in your own work would be that you are trying something speculative, which may or may not work. By creating a new branch, you are ensuring that you have something stable in the master branch at all times; you can merge the experimental branch back into the master thread later if all works out (where merging will be covered below).

¿What branch are you on right now? Find out with

[7]See also, e.g., `git show fe9c:oldfile` to just display a single file from an old version without doing a full checkout.

```
git branch
```

which will list all branches and put a * by the one that is currently active.

Now create a new branch. There are two ways to do it:

```
git branch new_leaf
git checkout new_leaf

#or equivalently:

git checkout -b new_leaf
```

There are really two steps here: establishing a new branch in the repository, then changing your working version to make use of that branch. The two-command version makes that explicit; the single `checkout -b` form is provided because it's so common to want to immediately use the branch that you are creating.

Having built a new branch, you can switch between branches easily. E.g., to switch back to the master branch:

```
git checkout master
```

You can see that the branch checkouts, like `git checkout newleaf` or `git checkout master`, have the same syntax as that infernal SHA1 syntax like `git checkout fe9c`. The reason is that the name of the branch is really just a synonym for the SHA1 hash that is the last item on that branch (aka the *branch head*). Use them interchangeably, though I'm guessing you'll lean toward using the branch name.

Merging To this point, everything has been about creating new versions, and jumping around between versions. Now for the hard part: you have a version, your colleague has a version, and they differ.

The command is simple enough. You have on your screen a current version, and you want to fold in the revisions from version fe9f. Then

```
git merge fe9f
```

will do the work. Of course, you can use a branch name as well, like `git merge new_leaf`.

For some things, the system will have no problem merging together the two threads: if your coauthor was working only on the intro to chapter three and you were working only on chapter three's conclusion, that's easy to merge.

But if you were both wrestling with the same paragraph, then the computer will be confused. It will tell you that there are conflicts, and write both into the file in your current directory. You will then have to open the file(s) in your text editor, and find the place where git wrote both versions for you to compare and choose from.

If you were to merge revision `3c3c3c` from the remote repository into the current working revision (i.e. the head), then `file2` would probably wind up looking something like this:

```
I'm a little teapot,
<<<<<<< HEAD
short and stout
=======
round and fat
>>>>>>> 3c3c3c
```

Here, Git finds a single line with two different versions, and it can't rely on timing or other heuristics to pick one. So, it shows you the two versions, and it is up to you as a human to decide what to do. The solution will often require human contact with another author (IM is a perfect medium for this). Git can't call your coauthor, but it can at least point you to the exact line where differences exist.

The other type of conflict, which is just annoying, is when your colleague has renamed a file or moved it from one directory to another. Git typically won't just move the darn file for you, but will instead list it as a conflict for you to deal with. Moving files can create other awkward issues; for example, if you are doing `git pull` from a subdirectory that your coauthor has deleted, you'll get entirely confused errors.

Here is the procedure for committing merges:

1. `git merge a_branch`.
2. Get told that there are conflicts you have to resolve.
3. Check the list of unmerged files using `git status`.
4. Pick a file to manually check on. Open it in a text editor and find the merge-me marks if it is a content conflict; move it into place if it's a file name or file position conflict.
5. `git add your_now_fixed_file`.
6. Repeat steps 3–5 until all unmerged files are checked in.
7. `git commit` to finalize the merge.

I have always found merging to be unnerving. There is a computer modifying your files, without even telling you what it is modifying. Unlike the long list of versions and your endless power to shunt branches, the merge algorithm is more-or-less a black box, and you just have to trust it. In that context, it's a somewhat good thing that the machine sometimes refuses to auto-merge and demands human attention. If the computer does go too far and makes a total mess of things, you can take recourse knowing that you have the previous version safely stowed.

The stash It doesn't take long working with Git to discover that it doesn't like doing anything when there are uncommited changes in the current working directory. It typically asks you to commit your work, and then do the checkout or such that you had intended.

One thing you can do in this case is a variant of the merge routine above: ask `git status` which files are tracked but modified; `git add` those files; then `git` `commit` your changes. Once your working tree, the index, and the latest commit are all in harmony, you can go back to your original plan.

Another sometimes-appropriate alternative is `git reset --hard`, which takes the working directory back to where you had last checked out. If the command sounds severe, it is because you are about to throw away all work you had done since the last checkout.[8]

The other option is the *stash*, a quick-to-use branch, with a few special features, like retaining all the junk in your working directory. Here is the typical procedure:

```
git stash
git checkout newleaf #or another commit, or what-have-you
#do work here
git checkout master #or the branch you had stashed from
git stash pop
```

So this is the above procedure of checking out, doing work, and then returning to the current version, but you stash your in-progress working directory beforehand and pop it back into place afterward.

Popping the stash works by merging the stash's semi-branch back to whatever is currently checked out, which is why you have to check out the commit you had been on before going exploring in the history: doing the merge is trivial if you have checked out the commit that you had been on when you started, and could be a mess if you are elsewhere. [The ability to apply the stash onto a separate commit allows for creative merging strategies which you may find use for if you are feeling clever.]

Remote repositories So far, I've talked only of checking out versions and branches that are in the respository of the directory you are in right now. But you can copy branches across repositories, which is how sharing happens. As alluded above, there can be amusing reasons for cloning a directory to another directory, and then merging changes between them.

To do all this, you need to be able to copy a branch from another repository. And to do that, you will need to name the other repository. Do this via

```
  #for an on-disk remote repository:
git remote add my_copy /path/to/copy

  #and for a distant remote repository:
git remote add distant_version http://...
```

That is, you will give a nickname for the repository, then a locator. There are many options for locators; the odds are good that the maintainer of any given repository handed you a locator to use, so I won't bore you with a list of options here.

[8]By the way, you can reset individual files by just checking them out. I recommend this syntax: `git checkout -- one_file`, after which `one_file` has reverted to its state as of the last checkin, and all changes lost.

If you are joining a project that already has a repository, running `git clone /path/to/copy` will set up a local copy and add a `remote` label of `origin`, set to the location of that parent repository.

A plain `git remote` will give you a list of remote repositories your repository knows about. You will probably just assign one remote and be done with it, but you have the power to live Linus's dream of concurrently passing files among several of your peers' several repositories.

Having established a remote, you also have more branches to choose from: try `git branch -r` to list remote branches (or `git branch -a` to list all branches, local and remote).

There are a few ways to get a remote branch:

```
git checkout -b new_local distant_version/master
 #or
git pull distant_version master
```

The first version uses the plain checkout mechanism using the remote tag you got via `git branch -r`, and uses the `-b` flag to create a name for the new branch you are about to create (otherwise you'd be stuck on `(no branch)`). The `pull` version merges into whatever you are working on now. You are probably getting things from the repository to bring your own work back up-to-date and in sync, so you probably want to use `pull` instead of `checkout`.

The converse is `push`, which you'll use to update the repository with your last commit (not the state of your index or working directory).

```
git push distant_version
```

When somebody had made another commit to the repository while you were working, then you will first need to do another `git pull`, slog through the merging procedure, and then push back the cleaned-up final version. This is common in team projects, and the error you get (about fast-forward merging) is entirely unhelpful.

git help That's all I'm giving you, and it should be enough for you to keep versions of your work, confidently delete things, merge in your colleagues' work, and be able to keep your bearings in Git's repository/branch/version/index system. From there, `git help` and your favorite search engine will teach you a whole lot of ways to doing these things more smoothly, and many of the tricks that I didn't cover here.

8.10 Git status interactive

12 December 2009

One of the first things that struck me as nice about Git was the status command, which produces something just shy of a script for revising the status of all the files. It even gives you tips about how to do common tasks.

I got even more excited when I saw `git rebase --interactive`, which generates a semi-script, opens it for you to edit, and then runs the thing automatically. That was smooth.

So I expected there'd be a similar procedure like `git status --interactive`, which, if it existed, would work like this:

- You type `git istatus`.
- Your favorite editor opens. There, you see the output from `git status`, plus instructions for some basic commands: put an `a` at the head of a line to add a file, an `i` to ignore it from now on, an `ea` to edit then add (which you'll do if you're merging), an `r` to remove the file from the repository, and so on.
- You exit, and your instructions are run.

Git doesn't do that. So I wrote a demo script to make that happen, git-status-interactive[9].

Click that link to save the script to your hard drive, and make it executable via the usual `chmod 755 git-status-interactive`. You probably want to alias the script using Git's aliasing system. For example, to allow the `git istatus` command I'd shown above, try this command from your bash prompt, in a single git repository:

```
git config --add alias.istatus \!/your/path/to/git-status-interactive
```

Or if you have the permissions to make global changes to the git config:

```
git config --global --add alias.istatus \!/your/path/to/git-status-intera
```

Some further notes The script is a demo—dead simple, with no serious error checking. To some extent it's a feature request: Dear Git team, please implement something like this in Git, but competently. Also, dear readers, please drop me an email if you've improved this thing for the better.

[By the way, Git does have `git add -i`, which behaves very differently from the edit-a-generated-file mechanism from `git rebase --interactive`. `git add -i` doesn't let me tick off files to ignore, and doesn't help immensely during merging; though it will give you more control when adding, like committing changes to sections of a file.]

Apart from `git status` and the shell, I use exactly one program to make this happen: Sed. The prep step runs Sed to take in the output of `git status` and then remove non-comment lines and insert instructions; the post-editor step run Sed to replace the one-character markers with the full commands. That's all.

Because the modified file just runs as a shell script, you can add other commands as you prefer. For example, replacing the # at the head of the line with an `rm` turns it into a standard remove command, or you can `mv` a file that git complains is in the wrong place (probably due to merging issues), et cetera.

In case you missed the link in the text above, download git status interactive[10] here.

[9] `http://modelingwithdata.org/asst/git-status-interactive`
[10] `http://modelingwithdata.org/asst/git-status-interactive`

8.11 Better variadic functions in C

3 June 2009

I really dislike how C's variadic functions are implemented. I think they create lots of problems and don't fulfil their potential. So this is my effort to improve on things.

A variadic function is one that takes a variable number inputs. The most famous example is `printf`, where both `printf("Hi.")` and `printf ("%f %f %i\n", first, second, third)` are valid, even though the first example has one input and the second has four.

Simply put, C's variadic functions provide exactly enough power to implement `printf`, and nothing more. You must have an initial fixed argument, and it's more-or-less expected that that first argument provides a catalog to the types of the subsequent elements, or at least a count. In the example above, the first two items are expected to be floating-point variables, and the third an integer.

There is no type safety: if you pass an `int` like 1 when you thought you were passing a `float` like 1.0, results are undefined. If you think there are three elements passed in but only two were passed in, you're likely to get a segfault. Because of issues like this, CERT, the software security group, considers variadic functions to be a security risk[11] (Severity: high. Likelihood: probable).

I understand that the designers of the system are reluctant to impose too much magic to make variadic functions work, like magically dropping into place an `nargs` variable giving an argument count. So today's post is an exercise in how far we can get in implementing decent variadic functions using only ISO C. To give away the ending, I manage some of the things we use variadic functions for in a safe and more convenient manner—optional arguments work very well—but it takes many little tricks, and I'm still short of true `printf` functionality.

Designated intializers First, a digression into a pair of nifty tricks that C99 gave us: compound literals and designated initializers. I find that many people aren't aware of these things, because they're learning C from textbooks written before 1999, and using compilers that may not use the 1999 standard by default.

Darn it people, it's been a decade. This is not new.

The idea is simple: if you have a `struct` type, you can use forms like these to use an anonymous struct wherever it's appropriate:

```
typedef struct {
    int first, second;
    double third;
    gsl_vector *v;
} stype;

stype newvar = {3, 5, 2.3, a_vector};
```

[11] `https://www.securecoding.cert.org/confluence/display/seccode/DCL11-C.+Understand+the+type+issues+associated+with+variadic+functions`

```
stype nextvar = {3, 5};
newvar = (stype)  {.third = 3.12, .second=5};
function_call( (stype) {.third = 8.3});
```

In each case, a full struct is set up, and the compiler is smart enough to know what goes where among those elements you specified, and sets the other elements to zero or NULL.

These sorts of features that we have for initializing a `struct` are exactly the sort of thing many more recent languages put into their function calls: default values are filled in, and named elements are allowed via designated initializers.

At the end of the example, I put a compound literal inside a function call, so we are technically calling a function using these pleasant variable-input features, but it's not yet looking much like `printf`.

Cleaner function calls We can clean up the struct-to-function trick to get a lot closer to variadic functions. Here's the agenda for making this work:

- For each function, set up a struct where the elements of the struct are the inputs to the function.
- Produce a shadow function whose sole input is that struct, which sets the default vaules and then calls the original function.
- Write a wrapper macro so that the instead of the user having to type the full compound literals form `f( (ftype) {arg1, arg2})`, they can just type the usual `f(arg1, arg2)`.

So, here it is. The first third is a set of general macros, the second third sets up a single function, and the last third actually makes use. This program should compile with any C99-compliant compiler. After the code, I'll have some detailed notes to walk you through it.

```
#define varad_head(type, name) \
         type variadic_##name(variadic_type_##name x)

#define varad_declare(type, name, ...) \
        typedef struct {            \
                    __VA_ARGS__       ;  \
                } variadic_type_##name;     \
    varad_head(type, name);

#define varad_var(name, value) name = x.name ? x.name : (value);
#define varad_link(name,...) \
        variadic_##name((variadic_type_##name) {__VA_ARGS__})


///////////////////// header + code file
```

```
varad_declare(double, sum, int first; double second; int third;)
#define sum(...) varad_link(sum,__VA_ARGS__)


varad_head(double, sum) {
    int varad_var(first, 0)
    double varad_var(second, 2.2)
    int varad_var(third, 8);

    return first + second + third;
}


////////////////////// actual calls
#include <stdio.h>

int main(){
    printf("%g\n", sum());
    printf("%g\n", sum(4, 2));
    printf("%g\n", sum(.third=2));
    printf("%g\n", sum(2, 3.4, 8));
}
```

• There are three macros in the first section, roughly corresponding to the three steps of the agenda. `varad_declare` declares a special type and a function to use that type. Notice that the third and later arguments to the macro go into the struct, not a function header, so variables are separated by semicolons. `varad_var` sets default values for each variable. `varad_link` is used to clean up the function call.

• The second third sets up a single function. The bulk declares that intermediate function that takes in a struct, sets default values, and calls the real function.

• There is one more macro in this section, which needs to be rewritten for every new function. It'd be great if there were a macro to just churn out this trivial macro for each new function, but you can't write macros that generate macros. Why not? I dunno. Seems like it wouldn't be a big deal for the preprocessor, but them's the rules. The too-simple preprocessor is my second big complaint about C.

• The main part of the intermediate function has a line for each element of the struct, declaring an intermediate variable and setting a default value. The compiler gave missing elements a default value, but we often want the default to be something other than zero. We can also have more intelligent defaults based on variable information, like maybe `int varad_in(third, first * 3)`.

• The third part is a call to the function we've set up, and you can see that it works great: we can give it no arguments, all arguments, named arguments, or whatever else seems convenient, with no regard to the internal guts from prior sections.

Infinite input OK, so far, the result looks much more modern relative to C's standard fixed inputs. It allows optional arguments, and named arguments. It checks types, and complains during compilation if you've got mismatched types, meaning that a lot of the security holes of the standard variadic form are gone.

But we want more from our variadics than just optional arguments: we'd like to specify arbitrary-length lists. Can we declare a structure that could take an arbitrary number of inputs, such as a function to sum n inputs?

The short answer is no. [The long answer: the last element of a struct can be an array of indeterminate size, to be allocated at compile-time. When the anonymous struct is being generated for the function call, a compiler could count the elements at the end of the list and allocate the variable-size array appropriately.

However, this depends on whether the anonymous struct is dynamic or static. By static, I mean something produced at the initialization, like the constants or global variables; by dynamic, I mean variables that are initialized along the way during the run. For static variables, the variable-length last argument will be stuck in the form set at the first allocation; for dynamic variables, there are more options. So what are the anonymous structs used for the function calls here? It is my reading that the ISO C standard doesn't demand things one way or the other, so we can't rely on dynamic allocation of the type we'd get elsewhere via a line like `x = (structtype) {1, 2, {3, 5, 9, 10}}`, where the variable-lenght allocation would be valid.

Also, for an array of fixed length, you can usually get the size by `sizeof(list)/sizeof(list[1])`. So if the system allocated the right-sized list, you wouldn't even need a separate `nargs` element taking up space.]

We're instead stuck just making up a size for an element of the struct, like 1,000, and letting the compiler pad it with zeros. Given that we generally use variadic arrays for lists of items hand-typed into the code, and array inputs for lists of truly arbitrary length, we can probably get away with 1,000 inputs max, but it's certainly not ideal.

```
//Put the macros from the first third above in "variadic.h".
#include "variadic.h"

varad_declare(double, sum, int first;
                double second; int third[1000];)

#define sum(...) varad_link(sum,__VA_ARGS__)
varad_head(double, sum) {
    int varad_var(first, 0)
    double varad_var(second, 2.2)
    int * varad_var(third, NULL);

    double sum = first + second;
    for (int i=0; i< 1000; i++)
        sum += third[i];
    return sum;
}

int main(){
    printf("%g\n", sum());
    printf("%g\n", sum(4, 2));
    printf("%g\n", sum(.third={2}));
    printf("%g\n", sum(2, 3.4, {8, 8}));
}

#endif
```

So the `third` element can have variable length, as desired. If you're not sure of the type coming in, the last element can be an array of `void *`, where `void *` is your signal to the compiler that you're willing to take your chances on types and have a catalog or system on hand to do your own casts.

How're we doing? So there's the story: we can do a lot better with our variable-length function calls than we do, and it's not even something involving crazy re-writing of everything. Standard C already gives us the tools to go 90% of the way.

However, it's a frickin' pain to set up. C's preprocessor is limited, and we had to write several macros to make this happen. For every function, the namespace has to have another auxiliary function and a type floating around. You won't notice this normally, but it can create quirks in the debugger and other places that expect a little more normalcy out of the code base.

Apophenia uses this setup for a few dozen functions, but with a few more tricks. Notably, everything is wrapped in `#ifdef`s to let everything degrade to the standard function call if needed. Many things beyond the compiler eat C code, like documentation generators, interface generators, &c. Even though all the above is 100% standard C compliant, some systems like the setup more than others. [I also wrote a sed script to generate all this boilerplate from appropriate markers. The script also gets around the problem that we can't use the preprocessor to generate macros.]

OK, summary paragraph: we need to fix C's variadic function calling scheme, which is built around `printf` to the detriment of many other possibilities, and even to the detriment of security. Being that we can already do most of what we want via ISO C99, we can fix them without introducing incompatibilities or changing the character of C. But given the amount of extras and tricks involved, and given that we still don't quite achieve proper variadic functions, there'd need to be some fixes in the language itself to update variadic functions to a modern form.

8.12 Tip 1: Use a makefile

1 October 2011

level: Start here
purpose: Worry about compilation details once and only once

On the scale from *works out of the box* on one end to *infinitely configurable and tweakable* on the other, the C compiler is far on the tweakability side. But correctly tweaking the compiler is a problem you need to solve once, and then get on with your life. The solution is the *makefile*, which is basically an organized set of variables and shell scripts.

If your program has only one `.c` file, here's the `makefile` for you.

```
OBJECTS =$(P).o
CFLAGS = -g -Wall -std=gnu99 -pthread -O3
LIBS=

c: $(OBJECTS)
    gcc $(CFLAGS) $(OBJECTS) $(LIBS) -o $(P)

$(OBJECTS): %.o: %.c
    gcc $(CFLAGS) -c $< -o $@
```

Usage:

- Once ever: save this (with the name `makefile`) in the same directory as your `.c` files. Make certain the `gcc` lines are indented with a tab, not spaces. [Otherwise you'll get a `missing separator` error. Yes, this is stupid.]
- Once per session: Set the variable `P` as the name of your program from the command line: `export P=your_program` (not `your_program.c`).
- Every time you need to recompile: type `make`

Tweaks:

- If you have a second (or more) C file, add `second.o third.o`, et cetera on the `OBJECTS` line (no commas, just spaces between names).
- If, when you run the debugger, you find that too many variables have been optimized out for you to follow what's going on, then remove or comment out the `-O3` bit (which sets Optimization level three).
- If you are using a not-entirely-standard library of functions, then you will need to add the library on the `LIBS` line and the include path on the `CFLAGS` line. Try typing `pkg-config` on your command line; if you get an error about specifying package names, then great, you have pkg-config and can use it like:

  ```
  LIBS=‘pkg-config --libs apophenia glib-2.0‘
  CFLAGS=[everything above plus:] ‘pkg-config --cflags apophenia glib-2.
  ```

  If you get an error about `pkg-config` not being found, you'll have to specify each library and/or its locations:

  ```
  LIBS=-L/home/b/root/lib -lweirdlib
  CFLAGS=[everything above plus:] -I/home/b/root/include
  ```

  Tune in next time for an extended example.
- After you add a library to the `LIBS` and `CFLAGS` lines and you know it works on your system, there is little reason to ever remove it. Do you really care that the final executable might be 10 kilobytes larger than if you customized a new makefile for every program?

OK, you're done! After you `export P=your_program`, you can run `make` and watch the compiler run and/or spit out errors, and never worry about what all that junk in the makefile actually means.

There will be limited plugs for *Modeling with Data* in this tip-a-day series, but I think it's worth mentioning that Appendix A goes into great detail about tweaking the makefile to do great things.

To do:
Here's the world-famous `hello.c` program, in two lines:

```
#include <stdio.h>
int main(){ printf("Hello, world.\n"); }
```

Save that and the `makefile` to a directory, and try the above steps to get the program compiled and running.

8.13 Tip 2: Use libraries

3 October 2011

level: You have the syntax down and want to get real work done
purpose: Use 40 years of prior scholarship to your advantage

Twenty years ago, it was evidently pretty difficult to pull down a good library of functions and make use of them in your current project. I say this because I couldn't find any C tutorials from the period that show you how to use a non-standard library to do real work. Which is why you can find C detractors who will say self-dissonant things like *C is forty years old, so you have to write every procedure from scratch in it.*

Now, it's easy. We have the GNU to thank for much of this, because free libraries now outnumber for-pay libraries, and so there are package managers and other such systems to let you pull down a library with a few mouse clicks. If you have to create windows for your program, deal with XML, encode audio streams to MP3, or manipulate DNA sequences, ask your package manager for a library before you start from zero.

Besides the politics of free/open source/libre/whatever, the GNU also has a set of tools that will prepare a library for use on any machine, by testing for every known quirk and implementing the appropriate workaround, collectively known as the `autotools`.

Writing a package to work under autotools is, um, hellish, but the user's life is much easier as a result. Also, it's a logical extension to Tip #1, because now that you know that having a makefile will simplify compilation, it's only logical that you'd use a tool to generate a makefile for you.

To do:
Let's try a sample package, shall we? The GNU Scientific Library includes a host of numeric computation routines. If you ever read somebody asking a question that starts *I'm trying to implement something from* Numeric Recipes in C..., the correct response is *download the GSL, because they already did it for you.*

One of the things I ♡ about using POSIX[12] is that I can give people unambiguous and quick tech support over IM. No *click here, then look for this button* stuff, just paste this onto the command line (assuming you have root privileges on your computer):

```
wget ftp://ftp.gnu.org/gnu/gsl/gsl-1.15.tar.gz #download
tar xvzf gsl-*gz    #unzip
cd gsl-1.15
./configure         #determine the quirks of your machine
make                #compile
make install        #install to the right location---if you have permissi
```

If you get an error about a missing program, then use your package manager to obtain it and start over. [Package managers are one of those places where I can't just tell you what to type, which is one solid reason why I'm not using one here.]

If you are reading this on your laptop, then you probably have root privileges, and this will work fine. If you are at work and using a shared server, the odds are low that you have superuser rights. If you don't have superuser rights, then hold your breath for two days until Tip #3.

Now you'll need to indicate in your makefile that you will be linking programs you write to the library you just installed. The makefile from Tip #1 had a blank LIBS line; this is where you start filling it in.

If you have pkg-config on hand, then use it like so:

```
LIBS=‘pkg-config --libs gsl‘
```

When you add new libraries, add them to the list like so:

```
LIBS=‘pkg-config --libs gsl sqlite3 apophenia‘
```

If you're on a system without pkg-config, you'll need to explicitly specify which libraries you need:

```
LIBS=-lgsl -lgslcblas -lm
```

Every time you install a new library, you will always need to add at least one item to the `LIBS`, like the `-lgsl` part. The GSL has a quirk that it requires a BLAS (basic linear algebra system), and `-lm` is the standard math library.

Did it install? The numeric integration documentation[13] has a sample program that integrates the function specified at the top of the file. It's a good example because I get the impression that numeric integration is the sort of thing that I feel people often re-implement in C (and I already have you reading the manual—there's a lot there). Paste it into a file and use your library-improved makefile to test and install it.

[12]UNIX is a trademark of AT&T; POSIX (Portable Operating System Interface (the X goes uneXplained)) is a more general descriptor for things that are UNIX-like, including Linux, BSD, Mac OS X, &c.

[13]`http://www.gnu.org/s/gsl/manual/html_node/Numerical-integration-examples.html`

8.14 Tip 3: Use libraries (even if your sysadmin doesn't want you to)

3 October 2011

level: library user
purpose: use the Machine to rage against the Man

You may have noticed the caveats in the last entry about how you have to have root privileges to install to the usual locations on a POSIX system. But you may not have root access if you are using a shared computer at work, or you have an especially controlling significant other.

Then you have to go underground, and make your own private root directory.

The first step is to simply create the directory:

```
mkdir ~/root
```

I already have a `~/tech` directory where I keep all my technical logistics, manuals, and code snippets, so I made a `~/tech/root` directory. The name doesn't matter, but I'll use `~/root` below.

[Your shell replaces the tilde with the full path to your home directory, saving you a lot of typing. But other programs, like `make`, may or may not recognize the tilde as your home directory.]

The second step is to add the right part of your new root system to all the relevant paths. For programs, that's the `PATH` in your `.bashrc`:

```
PATH=~/root/bin:$PATH
```

By putting the `bin` subdirectory of your new directory before the original `PATH`, it will be searched first and your copy of any programs will be found first. Thus, you can substitute in your preferred version of any programs that are already in the standard shared directories of the system.

For libraries you will fold into your C programs, note the new paths to search in the makefile you wrote in Tip #1 and added to in Tip #2:

```
LIBS=-L/home/your_home/root/lib <plus the other flags, like -lgsl -lm ...>
CFLAGS=-I/home/your_home/root/include <plus -g -Wall --std=gnu99...>
```

[Again, Appendix A of `Modeling with Data` goes into detail on dealing with paths and environment variables.]

The last step is to install programs in your new root. If you have the source code and it uses autotools, all you have to do is add `--prefix=~/root` in the right place:

```
./configure --prefix=~/root
make
make install
```

You didn't need `sudo` to do the install step because everything is now in territory you control.

Now that you have a local root, you can use it for other systems as well, such as adding a subdirectory for R packages (e.g., `mkdir ~/root/rlib`) and notifying R about them by adding `R_LIBS=~/root/rlib` to the `~/.Renviron` file.

Because the programs and libraries are in your home directory and have no more permissions than you do, your sysadmin can't complain that they are an imposition on others. If your sysadmin complains anyway, then, as sad as it may be, it may be time to break up.

8.15 Tip 4: Don't bother explicitly returning anything from `main()`

7 October 2011

level: Hello, world.
purpose: Eliminate a line of unnecessary code from every program

Much of the tip-a-day series will be about making your life with C more like your life with quick-and-dirty scripting languages. Toward the goal of having fewer lines of code, let's shave a line off of every program you write.

Your program must have a `main` function, and it has to be of return type `int`, so you must absolutely have

```
int main(){ ... }
```

in your program.

You would think that you therefore have to have a `return` statement that indicates what integer gets returned. However, the C standard knows how infrequently this is used, and lets you not bother: "...reaching the } that terminates the main function returns a value of 0." (C standard §5.1.2.2.3) That is, if you don't write `return 0;` as the last line of your program, then it will be assumed.

Tip #1 showed you this version of `hello.c`, and you can now see how I got away with a `main` containing only one line of code:

```
#include <stdio.h>
int main(){ printf("Hello, world.\n"); }
```

A few tips from now, we'll have cut this down to only one line.

To do:
Go through your programs and delete this line; see if it makes any difference.

8.16 Tip 5: Initialize wherever the first use may be

9 October 2011

level: still pretty basic
purpose: not think about declarations so much

I see code like this pretty often:

```
int main(){
    char *head;
    int i;
    double ratio, denom;

    denom=7;
    head = "There is a cycle to things divided by seven.";
    printf("%s\n", head);
    for (i=0; i< 10; i++){
        ratio = i/denom;
        printf("%g\n", ratio);
    }
}
```

We have three or four lines of introductory material (I'll let you decide how to count the white space), followed by the routine.

This is somewhat a matter of style, but I think this looks archaic, and I've heard from a few folks who learned to code via untyped scripting languages for whom the introductory declarations are a direct and immediate turn-off. Variables still have to have a declared type, but here's how I'd write the code to minimize the burden:

```
int main(){
    double denom=7;
    char *head = "There is a cycle to things divided by seven.";
    printf("%s\n", head);
    for (int i=1; i<= 6; i++){
        double ratio = i/denom;
        printf("%g\n", ratio);
    }
}
```

Here, the declarations happen as needed, so the onus of declaration reduces to sticking a type name before the first use. If you have color syntax highlighting, then the declarations are still easy to spot (and if you don't have color, golly, get a text editor that supports it—there are dozens to hundreds to choose from!).

Also, by the rule that you should keep the scope of a variable as small as possible, we're pushing the active variable count on earlier lines that much lower. When you have a too-long function running a page or two, this can start to matter. As for the index, it's a disposable convenience for the loop, so it's natural to reduce its scope to exactly the scope of the loop.

Did you notice how I've been specifying `-std=gnu99` in the sample makefiles and other calls to `gcc`? Declaring the iterator variable for the `for` loop wasn't valid in the mid-1990s, and GCC is still stuck in using that as the norm. It's annoying, I know, but that's a minor kvetch about a program as awesome as `gcc`. If you're not already compiling via an alias or a makefile, you might want to add `alias gcc="gcc -std=gnu99"` to your `.bashrc` to get GCC to always use the 1999 standard.

8.17 Tip 6: Aggregate your includes

11 October 2011

level: casual user
purpose: stop thinking about standard header files

There was once a time when compilers took several seconds or minutes to compile even relatively simple programs, so there was human-noticeable benefit to reducing the work the compiler has to do. My current copies of `stdio.h` and `stdlib.h` are about 1,000 lines long [try `wc -l /usr/include/stdlib.h`] and `time.h` another 400, meaning that

```
#include <time.h>
#include <stdio.h>
#include <stdlib.h>
int main(){
    srandom(time(NULL));
    printf("%li\n", random());
}
```

is actually a ∼2,400-line program.

You see where this is going: your compiler doesn't think 2,400 lines is a big deal anymore, and this compiles in under a second. So why are we spending time picking out just the right headers for a given program?

To do:
Write yourself a single header, let us call it `allheads.h`, and throw in every header you've ever used, so it'll look something like:

```
#include <math.h>
#include <time.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <gsl/gsl_rng.h>
```

I can't tell you exactly what it'll look like, because I don't know exactly what you use day to day.

Now that you have this aggregate header, you can just throw one

```
#include <allheads.h>
```

on top of every file you write, and you're done with thinking about headers. Sure, it will expand to perhaps ten thousand lines of extra code, much of it not relevant to the program at hand. But you won't notice.

Having junk in the name space may have conesequences from time to time. Say that you've defined a function called `rndm`, and at some point you forget the compressed name and call `random`. As above, `random` is a real function declared in `stlib.h`, so you won't get the usual warning about using an undeclared function. I don't count this as all that much of a problem.

When I wrote the Apophenia library, by the way, I had in mind this kind of thing, so I have

```
#include <apop.h>
```

at the head of every program I write, whether I'll be using anything from the Apophenia library or not, because that includes `stdio`, `stdlib`, the stats-relevant parts of the GSL, et cetera. And then I more-or-less never think about headers.

The next tip will show how to eliminate even that `#include <allheads.h>` line at the top of all your programs.

8.18 Tip 7: Include header files from the command line

15 October 2011

level: library user
purpose: really never think about standard headers again

The GCC has a convenient flag for including headers:

```
gcc -include stdio.h
```

is equivalent to putting

```
#include <stdio.h>
```

at the head of your C file.

By adding that to our invocation of GCC, we can finally write `hello.c` as the one line of code it should be:

```
int main(){ printf("Hello, world.\n"); }
```

which compiles fine via:

```
gcc -include stdio.h hello.c -o hi
```

To do:
In tip #5, you wrote a single header file to join them all, so that you could just write

```
#include <allheads.h>
```

at the top of your program. Now you don't even need to do that: modify the makefile from tip #1 by adding `-include allheads.h` on the `CFLAGS` line.

Discussion:
My impression is that C programmers hold themselves to a much higher standard of

portability than people who write code in the typical scripting language. With no standardized environment by a central authority, it takes more discipline to ensure that everything will compile everywhere.

This tip bucks that tendency a bit, because we're saying the program will only compile correctly if you give just the right flag to the compiler (and this is even GCC specific). If this bothers you, well, skip this tip.

I'm not bothered for two reasons. First, I always attach a makefile with every program I send to anybody, because that significantly raises the odds that the other side will be able to get the program running. Second, most of what I write will never see the light of day, but is written for my own research or idiosyncratic needs. For the one-in-a-hundred stupid little C scripts for my personal use that grow into something used by others, I might take the time to explicitly `#include <apop.h>` rather than using the `-include apop.h` flag from this tip.

8.19 Tip 8: Use here scripts

17 October 2011

level: intermediate POSIX
purpose: fewer temp files floating around

I'll relate this to C in a few episodes, but this is a general feature of POSIX-compliant shells that you can use for Python, Perl, or whatever else. In fact, if you want to have a multilingual script, this is an easy way to do it. Do some parsing in Perl, do the math in C, then have R produce the pretty pictures, and have it all in one text file.

Here's a Python example. Normally, you'd tell Python to run a script via

```
python your_script.py
```

You can give the file name `'-'` to use standard in as the input file:

```
echo "print 'hi.'" | python '-'
```

[Subtip: We need `'-'` and not just - to indicate that this is plain text and not introducing a switch like the *c* in `python -c "print 'Hi'"`. Many programs follow a custom that two dashes indicate that they should stop reading switches and read subsequent inputs plain. Thus
`echo "print 'hi.'" | python -- -`
also works, but is the sort of thing that scares people.]

You could, in theory, put some lengthy scripts on the command line via `echo`, but you'll quickly see that there are a lot of little undesired parsings going on—you might need `\"hi\"` instead of just `"hi"`, for example.

Thus, the *here script*, which does no parsing at all. Try this:

```
python '-' <<"XXXX"
lines=2
print "\nThis script is %i lines long.\n" %(lines,)
XXXX
```

The `XXXX` is any string you'd like; `EOF` is also popular, and `-----` looks good as long as you get the dash count to match at top and bottom. When the shell sees your chosen string alone on a line, then it will stop sending the script to the program's stdin. That's all the parsing that happens.

discussion:
This tip is a standard shell feature, and so should work on any POSIX system. I know because the POSIX standard[14] is online and is not all that painful to read, as standards go. Unfortunately, I'm not sure how to link to a certain line, so you'll have to go searching to verify my promise that Here Documents are standard shell features.

There's also a variant that begins with `<<-`. Search the standard or ask `man bash` for details.

As another variant, there's a difference between `<<"XXXX"` and `<<XXXX`. In the second version the shell parses certain elements, which means you can have the shell insert the value of `$shell_variables` for you.

8.20 Tip 9: Compile C programs via here script

19 October 2011

level: amused beginner
purpose: Compile via cut/paste

Today's tip compiles C code pasted onto the command line. If you are a kinetic learner who picked up scripting languages by cutting and pasting snippets of code into the interpreter, you'll be able to do the same with C and your shell.

Further, a shell is glue that does traffic control among the many programs that do the real work. This tip will let you put C into that flow, so C code can be sandwiched into a single file that also has plain shell instructions or steps in Perl or R.

We're not going to use the makefile, so we need a single compilation command. To make life less painful, let us alias it. Paste this onto your command line, or add it to your `.bashrc`, `.cshrc`, &c.:

```
go_libs="-lm"
go_flags="-g -Wall -include allheads.h --std=gnu99 -O3"
alias go_c="gcc -xc '-' $go_libs $go_flags"

#C shell users do this a little differently:
set go_libs="-lm"
set go_flags="-g -Wall -include allheads.h --std=gnu99 -O3"
alias go_c "gcc -xc '-' $go_libs $go_flags"
```

where `allheads.h` is the aggregate header you'd put together in tip #6. Using this `-include` switch means one less thing to think about when writing the C code, and I've found that bash's history gets wonky when there are #s in the C code. [The `zsh` is a big winner when it comes to interactively editing histories with long here scripts, and has no problem with the #s.]

[14] `http://pubs.opengroup.org/onlinepubs/009695399/`

On the gcc line, you'll recognize the `'-'` to mean that instead of reading from a named file, use stdin. The `-xc` flags this as C code, because *GCC* stands for GNU Compiler Collection, not GNU C Compiler, and with no input filename ending in `.c` to tip it off, we have to be clear that this is not Java, Fortran, Objective C, Ada, or C++.

Whatever you did to customize the `LIBS` and `CFLAGS` in your makefile, do here. For example, here are my versions of `go_libs` and `go_flags`, because I have pkg-config installed and use GLib and Apophenia (and by implication, the GSL and SQLite) for everything:

```
go_libs="`pkg-config --libs glib-2.0 apophenia`"
go_flags="-g -Wall -include apop.h --std=gnu99 -O3"
```

Now we can use a here document to paste short scripts onto the command line. Not only do you not need a makefile, you don't even need an input file or command interpreter.

To do:
After defining the right aliases for your setup (including a reference to your aggregate header), paste this onto your command line:

```
go_c <<"EOF"
int main(){
    long int testme = 2, ct =0;
    long int *primes   = NULL;
    while(1){
        int isprime = 1;
        for (long int i=0; isprime && i< sqrt(testme) && i<ct; i++)
            isprime = testme % primes[i];
        if (isprime){
            printf("%li  \r", testme); fflush(NULL);
            primes       = realloc(primes, sizeof(long int)*(ct+1));
            primes[ct++] = testme;
        }
        testme  ++;
    }
}
EOF

./a.out #This line will wait for you to hit <ctrl>-C
```

Now you have a program that generates prime numbers faster than you can read them [stop via <ctrl>-C]. You can change `\r` to `\t` or `\n` if you want to keep a record. Until I got bored and stopped, it was testing about 100,000 numbers/second on my netbook, but you can see how it does on your machine.

Managing lines of code on the command line is not fun, so don't expect this sort of thing to be your primary mode of working. But cutting and pasting code snippets onto the command line *is* fun, and being able to have a single step in C within a longer shell script is pretty fabulous.

8.21 Tip 10: Use `asprintf` to make string handling less painful

21 October 2011

level: basic string user
purpose: don't call `malloc`

The function allocates the amount of string space you will need, and then fills the string. That means you never really have to worry about string-allocing again.

`asprintf` is not part of the C standard, but it's available on systems with the GNU or BSD standard library, which covers pretty much everybody (including Apple computers).

The old way made people homicidal (or suicidal, depending on temprament) because you first have to get the length of the string you are about to fill, allocate space, and then actually write to the space. ¡Don't forget the extra slot in the string for the null terminator!

This sample program demonstrates the painful way of setting up a string, for the purpose of using C's `system` command to run an external program. The thematically appropriate program, `strings`, searches a binary for printable plain text. I'm assuming that you're compiling to `a.out`, in which case the program searches itself for strings. This is perhaps amusing, which is all we can ask of demo code.

```
#include <stdio.h>
void get_strings(char *in){
    char *cmd;
    int len = strlen("strings ") + strlen(in) + 1;
    cmd = malloc(len); //The C standard says sizeof(char)==1, b.t.w.
    snprintf(cmd, len, "strings %s", in);
    system(cmd);
}

int main(){
    get_strings("a.out");
}
```

[Apophenia users, use `apop_system("strings %s", in)` to bypass the need to first print to a string and then call `system`. Everybody else, feel free to write your own `system_with_printf` function; it's not that hard.]

With `asprintf`, `malloc` gets called for you, which means that you also don't need the step where you measure the length of the string:

```
#include <stdio.h>
void get_strings(char *in){
    char *cmd;
    asprintf(&cmd, "strings %s", in);
    system(cmd);
}

int main(){
    get_strings("a.out");
}
```

The actual call to `asprintf` looks a lot like the call to `sprintf`, except you need to send the location of the string, not the string itself, because something new will be `malloc`ed into that location. Also, we don't need to send the length parameter like with `snprintf`, because `asprintf` is smart enough to count letters for you.

8.22 Tip 11: String literals

23 October 2011

level: intermediate string user
purpose: understand an annoying subtlety of C string handling

Here is a program that sets up two strings and prints them to the screen:

```
#include <stdio.h>
int main(){
    char *s1 = "Thread";

    char *s2;
    asprintf(&s2, "Floss");

    printf("%s\n", s1);
    printf("%s\n", s2);
}
```

Both forms will leave a single word in the given string. However, the C compiler treats them in a very different manner, which can trip up the unaware.

Did you try the sample code in tip #10 that showed what strings are embedded into the program binary? In the example here, `Thread` would be such an embedded string, and `s1` could thus point to a location in the executable program itself. How efficient—you don't need to spend run time having the system count characters or waste memory repeating information already in the binary. I suppose in the 1970s this mattered.

Both the baked-in `s1` and the allocated-on-demand `s2` behave identically for reading purposes, but you can't modify or free `s1`. Here are some lines you could add to the above example, and their effects:

```
s2[0]='f'; //Switch Floss to lowercase.
s1[0]='t'; //Segfault.

free(s2); //Clean up.
free(s1); //Segfault.
```

If you think of a bare string declared like `"Floss"` as pointing to a location in the program itself, then it makes sense that `s1`'s contents will be absolutely read-only.

[I honestly don't know how your compiler really handles a constant string, but it is a fine mental model to presume it is pointing to a point in the program, so writing upon is strictly forbidden.]

Did you think this would be a series about why C is better than every other language in every way? If so, sorry to disappoint you. The difference between constant and variable strings is subtle and error-prone, and makes hard-coded strings useful only in limited contexts. I can't think of a scripting language where you would need to care about this distinction.

But here is one simple solution: `strdup`, which is POSIX-standard, and is short for *string duplicate*. Usage:

```
char *s3 = strdup("Thread");
```

The string `Thread` is still hard-coded into the program, but `s3` is a copy of that constant blob, and so can be freely modified as you wish.

8.23 Tip 12: Use asprintf to extend strings

25 October 2011

level: basic string user
purpose: `malloc` will be lonely, because you never call it

Here is an example of the basic form for appending another bit of text to a string using `asprintf`, which, as per tip #10, can be your workhorse for string handling:

```
asprintf(&q, "%s and another_clause %s", q, addme);
```

I (heart) this for generating queries. I would put together a chain something like this contrived example:

```
int row_number=3;
char *q =strdup("select ");
asprintf(&q, "%s row%i \n", q, row_number);
asprintf(&q, "%s from tab \n", q);
asprintf(&q, "%s  where row%i is not null", q, i);
```

And in the end I have

```
select row3
from tab
where row3 is not null
```

A rather nice way of putting together a long and painful string. [I had trouble coming up with a simple example for this one that didn't look contrived. But when each clause of the query requires a subfunction to write by itself, this sort of extend-the-query form starts to make a lot of sense. Apophenia users, see also `apop_text_paste`.]

But it's a memory leak, because the blob at the original address of `q` isn't released when `q` is given a new location by `asprintf`. For one-off string generation, it's not even worth caring about—you can drop a few million query-length strings on the floor before anything noticeable happens.

If you are in a situation where you might produce an unknown number of strings of unknown length, then you will need a form like this:

```
//Safe asprintf macro
#define Sasprintf(write_to,  ...) {\
    char *tmp_string_for_extend = write_to;    \
    asprintf(&(write_to), __VA_ARGS__);   \
    free(tmp_string_for_extend);  \
}

//sample usage:
int main(){
    int i=3;
    char *q = NULL;
    Sasprintf(q, "select * from tab");
    Sasprintf(q, "%s where row%i is not null", q, i);
    printf("%s\n", q);
}
```

Discussion and caveats:
The `Sasprintf` macro, plus occasional use of `strdup`, is enough for roughly 100% of your string-handling needs. Except for one glitch and the occasional `free`, you don't have to think about memory issues at all.

The glitch is that if you forget to initialize `q` to `NULL` or via `strdup` then the first use of the `Sasprintf` macro will be freeing whatever junk happened to be in the uninitialized location `q`—a segfault.

As you learned in the last tip, the following also fails—wrap that declaration in `strdup` to make it work:

```
char *q = "select * from";
Sasprintf(q, "%s %s where row%i is not null", q, tablename, i);
```

8.24 Tip 13: Use a debugger

27 October 2011

level: absolutely basic
purpose: interact with your allegedly non-interactive program

Next time, I'll run a tip about debugging technique using GDB. But today's tip is short and brief:

Use a debugger, always.

I think some of you will find this to be not much of a tip, because who possibly wouldn't use a debugger? As a person who promotes C to people who typically are using Java or Python, I can tell you the number of people who try to write C without a debugger are myriad.

If there's a fault in Java or Python code, the machine throws up a backtrace immediately. Our coder tries C, commits a similar error, gets the entirely unhelpful `segmentation fault. Core dumped.` error, and gives up. There are more than enough C textbooks that relegate the debugger to the *other topics* segment, somewhere around Chapter 15, so it's understandable that so many people don't have the reflex of pulling up the debugger at the first sign of trouble.

[Why *Segmentation fault*, exactly? Because the computer allocates a segment of memory for your program, and you are touching memory outside that segment.]

About that *always* clause: there is virtually no cost to running a program under the debugger. I have never been able to perceive a difference in speed between running with the debugger and running without a net (and this is the sort of thing I write tests for when I can't focus on real work).

The debugger isn't just something to pull out when you really need the backtrace and variable states. It's great being able to pause anywhere, increase the verbosity level with a quick `print verbose++`, force out of a `for (int i=0; i< 10; i++)` loop via `print i = 100` and `continue`, or test a function by throwing a series of test inputs at it. The fans of interactive languages are right that interacting with your code improves the development process all the way along; they just never got to the debugging chapter in the C textbook, and so never realized that all of those interactive habits apply to C as well.

To do:
Get to know a debugger. I am a luddite, so I use GDB, but your IDE might have one built in. There are graphical front-ends to GDB with animal mascots; I never liked them well enough to switch, but they may fit your tastes perfectly.

Here are some simple things to try on any program you may have with at least one function beyond `main`. They are the absolute basics in debugging technique, so if you find that your debugging system somehow makes one of these steps difficult, then dump it and find another.

- pause your program at a certain point,
- get the current value of any variables that exist in that function;
- jump to a parent function and check variable values there;
- step past the point where you paused, one line of code at a time.

8.25 Tip 14: easier interrogations with GDB variables

29 October 2011

level: intermediate debugger
purpose: name suspects

OK, after Tip 13 I hope you're sold on the value of a debugger, and you are using it to see your data as it gets transformed by your program.

There are graphical front-ends for the debugger, which have built-in means of showing you certain common structures. But you also no doubt have your own favorite structures, and need tailored routines for viewing those structures the way you are used to seeing them.

This tip will cover some useful elements of GDB that will help you look at your data with as little cognitive effort as possible. All of the commands to follow go on the GDB command line. I assume you can already use GDB to do the exercise at the end of Tip 13.

Here's tip zero: the @ shows you a sequence of elements in an array. For example, here are a dozen of a `gsl_vector*`'s elements:

```
print *vector->data@12
```

Note the star at the head of the expression; without it we'd get a sequence of a dozen pointers. Also, herein I'll abbreviate `print` to `p`.

Next tip, which will only be new to those of you who didn't read the GDB manual, which is probably all of you. You can generate convenience variables, to save yourself some typing. For example, if you want to inspect an element deep within a hierarchy of structures, you can do something like

```
set $vd = my_model->dataset->vector->data
p *$vd@10
```

That first line generated the convenience variable to substitute for the obnoxious path. Following the lead of some shells, a dollar sign indicates a variable, and use `set` on first use. Unlike the shell, you need a dollar sign on the `set` line. The second line demonstrates a simple use. We don't save much typing here, but over the course of a long interrogation of a suspect variable, this can certainly pay off.

This isn't just a label; it's a real variable that you can modify:

```
p *$vd/14  #print the pointed to item divided by 14.
p *($vd++)  #print the pointee, and step forward one
```

That second line uses the only piece of pointer arithmetic worth knowing: that adding one to a pointer steps forward to tne next item in the list. [More on this in the next few tips.]

This is especially useful because hitting the enter key without any input repeats the last command. Since the pointer stepped forward, you'll get a new next value every time you hit enter, until you get the gist of the array. This is also useful should you find yourself dealing with a linked list. Pretend we have a function that displays an element of the linked list; then:

```
show_structure $list
show_structure $list->next
```

and leaning on the <enter> key will step through the list. [I'm not being creative here. This is still all from the manual.]

Tip 15 will be about making that imaginary function to display a data structure a reality.

But for now, here's one last trick about these $ variables. Let me cut an paste a few lines of interaction with a debugger in the other screen:

```
(gdb) p *out->parameters
$54 = {
  vector = 0x8056380,
  matrix = 0x0,
  names = 0x80561c0,
  text = 0x0,
  textsize =     {0,
    0},
  weights = 0x0,
  more = 0x0
}
```

You probably don't even look at it anymore, but notice how the output to the print statement starts with `$54`. Indeed, every output is assigned a variable name, which we can use like any other:

```
(gdb) p *$54->vector->data
$55 = 25.0001
(gdb) p $55*4
$56 = 100.0004
```

8.26 Tip 15: get `gdb` to print your structures

31 October 2011

level: intermediate debugger
purpose: look at the state of your data in different ways

GDB lets you define simple macros, which are especially useful for displaying nontrivial data structures—which is most of the work one does in a debugger. Gosh, even a simple 2-D array hurts your eyes when it's displayed as a long line of numbers.

The facility is pretty primitive. But you probably already wrote a C-side function that prints any complex structures you might have to deal with, so the macro can simply call that function with a keystroke or two. For example, I use `pd` to print `apop_data` structures via this macro:

```
define pd
    p apop_data_show($arg0)
end
document pd
```

```
Call apop_data_show to display an apop_data set.
E.g., for a data set declared with apop_data *d, use pd d.
end
```

Put these macros in your `.gdbinit`.

Notice how the documentation follows right after the function itself; view it via `help user-defined` or `help pd`. The macro itself just saves a few keystrokes, but because the primary activity in the debugger is looking at data, those little things add up.

To give a more involved example from the data structures I deal with in my work, the `apop_model` object has a list of settings groups—doesn't this already sound like a pain to inspect? So I wrote the following macro, with its accompanying documentation. This was enough to turn debugging these settings groups from pulling teeth to, um, cuddling puppies.

I don't expect you to follow the details of what it does, but as a somewhat exceptional case, you can see how much you can do: if argument one is `apop_mle`, then `$arg1_settings` → `apop_mle_settings`, and turning the same text into a string isn't an awkward exception like with C's macro processor.

```
define get_group
    set $group = ($arg1_settings *) apop_settings_get_grp( $arg0, "$arg1"
    p *$group
end
document get_group
Gets a settings group from a model.
Give the model name and the name of the group, like
get_group my_model apop_mle
and I will set a gdb variable named $group that points to that model, whi
like any other pointer. For example, print the contents with
p *$group
The contents of $group are printed to the screen as visible output to thi
end
```

I partly needed this because you can't use preprocessor macros at the GDB prompt—they were subbed out long before the debugger saw any of your code, so if you have a valuable macro in your code, you may have to reimplement it in GDB. At least writing these things is quick.

For more examples, I put a list of my favorite gdb macros[15] for the GSL, SQLite, and Apophenia elsewhere on this site.

One last unrelated GDB tip, and then we can go back to C technique next time. Add this line to your `.gdbinit` to turn off those annoying notices about new threads:

```
set print thread-events off
```

[15] `http://modelingwithdata.org/appendix_o.html\#gdbinit`

8.27 Tip 16: All the pointer arithmetic you need to know

2 November 2011

level: basic
purpose: save you reading and cognitive effort

Here's my theory of why I like C despite the common wisdom that it is terrible: I didn't learn C in a classroom. When you learn this stuff on the streets, you skip the parts of the textbook that that aren't necessary for survival, at which point you have a pretty lean and fun language.

Kernighan & Ritchie (and by extension, lots of standard C textbooks) use a lot of paper expressing love for how pointer arithmetic works. If you're reading this blog, then you need none of it.

It's really amusing stuff. An array element consists of a base position plus an offset. This was all designed in the 1970s, so implementing an array as a block of memory and its elements as offsets made sense to the sort of people who spend their mornings writing assembly code. But it was also sort of mathematically clean and appealing. You could declare a pointer `double *p`; then that's our base, and you can use the offsets from that base as an array: the contents of the first element is `p[0]`, the contents of the second `p[1]`, et cetera. So we've implemented the distinction between data and the location of data, and got arrays for free in the process.

Or you could just write the base plus offset directly and literally, via a form like `(p+1)`. As your textbooks will tell you, this is valid C, and in fact `p[1]` is exactly equivalent to `*(p+1)`, which explains why the first element in an array is `p[0] == *(p+0)`. K & R spend about six pages on this stuff [sections 5.4 and 5.5].

This is a bit like how Latin is taught, versus every other language. In your Spanish class, you start off with usage. For arrays, something like:

- Declare pointers either via the dynamic form, `double *p` or the static form like `double p[100]`. We'll worry about the distinction later.
- In either case, the nth array item is `p[n]`. Don't forget that the first item is zero, not one; it can be referred to with the special form `p[0] == *p`.
- If you need the address of the nth element (not its actual value), use the ampersand: `&p[n]`. Of course, the zeroth pointer is just `&p[0] == p`.

Weeks later, when you can confidently ask donde estan los aseos, your Spanish class teaches you about the difference between different types of future tense. Meanwhile, in Latin class, you *start* with learning about the ablative case, and then learn Latin as an application of all that grammar you saw. Jumping the metaphor again, your average C textbook opens the section on pointers with a diagram showing a series of memory registers, while your typical Python textbook never gets into the details of implementation at all.

Since this is a tip-a-day blog, and you're probably reading *don't read the section on pointer arithmetic* as more a rant than a tip, I'll throw in one nice trick: you don't need an index for `for` loops that step through an array. Here, we use a spare pointer that starts at the head of a list, and then step through the array with `p++` until we hit the `NULL` marker at the end.

```
#include <stdio.h>

int main(){
    char *list[] = {"first", "second", "third", NULL};
    for (char **p=list; *p != NULL; p++){
        printf("%s\n", p[0]);
    }
}
```

It's nice that we don't have to bother with a counter, as we would in any other language, but then, it's not much of a payoff to six pages of pointer arithmetic lessons. Exercise: how would you implement this if you didn't know about `p++`?

Oh, and as for bit-shifting operators, like bitwise XOR and shift-register-left, I have written tens of thousands of lines of C code and used one maybe once. Just skip those sections entirely.

8.28 Tip 17: Define a string type

4 November 2011

level: still basic
purpose: slightly less confusing declarations

Here's a line to paste into the unified header you wrote to include at the head of all your programs back in Tip 6:

```
typedef char* string;
```

Pretty simple, but it will make your code more readable.

The sample code from last time is pretty short, so I'll reprint the whole program. You may find it to be hard to read (I was using it as an example of a point I was calling obscure), but we'll try to fix that:

```
#include <stdio.h>

int main(){
    char *list[] = {"first", "second", "third", NULL};
    for (char **p=list; *p != NULL; p++){
        printf("%s\n", *p);
    }
}
```

Do the declarations communicate to you that `char *list[]` is a list of strings, and that `*p` is a string?

Now use typedef to replace `char *` with `string`. There are fewer stars floating around; `p` is more clearly a pointer and `*p` the data at the pointer:

```
#include <stdio.h>
typedef char* string;

int main(){
    string list[] = {"first", "second", "third", NULL};
    for (string *p=list; *p != NULL; p++){
        printf("%s\n", *p);
    }
}
```

The declaration line for `list` is now as easy as C gets, and clearly indicates that it is a list of strings, and the snippet `string *p` should indicate to you that `p` is a pointer-to-string, so `*p` is a string.

In the end, you'll still have to remember that a string is a pointer-to-`char`; for example, `NULL` is a valid value.

I find that most folks (myself included) have no serious problem with pointers until they get to pointers-to-pointers and further depth beyond that. With strings as `char*`s, you hit that multiple-star wall much earlier than there's any reason to. And since t (p **??**)hrough 12 already went over how to deal with strings without ever calling `malloc`, we might as well start using a non-pointer type name for them.

To do:
Try declaring a 2-D array of strings, using the typedef above plus

```
typedef stringlist string*
```

Is it easier to work out how to allocate its parts?

8.29 Tip 18: Declare arrays when you know their size

6 November 2011

level: still basic
purpose: save the memory register stuff for when you really need it

You can allocate arrays to have a length determined at run time.

I point this out because it's easy to find texts that indicate that you either know the size of the array at compile time or you've gotta use `malloc`. But it's perfectly fine to delay initialization of the array until you find out its size. [Again, this is the difference between C in the 1970s when this either-or choice was real, and the C of this millennium.]

For example, here's a program (found via One Thing Well[16]) that will allow you to run several programs from the command line in parallel[17]. The intent of the following snippet (heavily edited by me) is to get the size of the array from the user using `atoi(argv[1])` (i.e., convert the first command-line argument to an integer), and then having established that number at run-time, allocate an array of the right length.

[16] `http://onethingwell.org/post/9960491695/parallelize`
[17] `http://www.marco.org/2008/05/31/parallelize-shell-utility-to-execute-command-batches`

```
pthread_t *threads;
int thread_count;
thread_count = atoi(argv[1]);
threads = malloc(thread_count * sizeof(pthread_t));
```

This is fine, and I don't mean to disparage the author when I rewrite this, but we can write the edited lines of code above with less fuss:

```
int thread_count = atoi(argv[1]);
pthread_t threads[thread_count];
```

There are fewer places for anything to go wrong, and it reads like declaring an array, not initializing memory registers.

The original program didn't bother freeing the array, because the program just exits. But if we were in a situation where the first would need a `free` at the end, the variable-length initialization still doesn't need it; just drop it on the floor and it'll get cleaned up when the program leaves the given scope.

By the way, you can find people online who will point out that manually allocated memory is faster than automatic. I recommend not caring. The speed difference is a few percent, not an order of magnitude (and is architecture- and compiler-dependent whether there's any difference at all). It's a subjective thing, but in this case I'll gladly trade more readable code for the small speed gain, if any.

8.30 Tip 19: define persistent state variables

8 November 2011

level: intermediate
purpose: build more self-sufficient functions

Static variables can have local scope. That is, you can have variables that exist only in one function, but when the function exits the variable retains its value. This is great for having an internal counter or a reusable scratch space.

Let's go with a traditional textbook example for this one: the Fibonacci sequence. Each element is the sum of the two prior elements, and we declare the first two elements to both be one.

```
#include <stdio.h>

long long int fibonacci(){
    static long long int first = 1;
    static long long int second = 1;
    long long int out = first+second;
    first=second;
    second=out;
    return out;
}
```

```
int main(){
    for (int i=0; i< 50; i++)
        printf("%Li\n", fibonacci());
}
```

Check out how insignficant `main` is. The `fibonacci` function is a little machine that runs itself; `main` just has to bump the function and it spits out another value. My language here isn't for cuteness: the function is a simple *state machine*, and static variables are the key trick for implementing state machines via C.

On to the tip: static variables are initialized when the program starts, before `main`, so you need to set their value to a constant.

```
//this fails: can't call gsl_vector_alloc() before main() starts
static gsl_vector *scratch = gsl_vector_alloc(20);
```

This is an annoyance (more next time), but easily solved with a macro to start at zero and allocate on first use:

```
#define Staticdef(type, var, initialization) \
    static type var = 0; \
    if (!(var)) var = (initialization);

//usage:
Staticdef(gsl_vector*, scratch, gsl_vector_alloc(20));
```

This works as long as we don't ever expect `initialization` to be zero (or in pointer-speak, `NULL`). If it is, it'll get re-initialized on the next go-round. Maybe that's OK anyway.

8.31 Tip 20: get to know static, automatic, manual memory

10 November 2011

level: intermediate
purpose: articulate why you hate C

C provides three models of memory management, which is two more than most languages and two more than you really want to care about.

static data is initialized before `main` starts. Array size is fixed at startup, but values can change (so it's not really static).

automatic is where you declare a varaible on first use, and it is removed when it goes out of scope. Most languages have only automatic-type data.

manual involves `malloc` and `free`, and is where most of your segfaults happen. This memory model is why Jesus weeps when he has to code in C.

Here's a little table of the differences in the three places you could put data:

	static	auto	manual
initialized on startup	●		
can be scope-limited	●	●	
set values on init	●	●	
`sizeof` measures array size	●	●	
persists across fn calls	●		●
can be global	●		●
set size at runtime		●	●
can be resized			●
Jesus weeps			●

Some of these things are features that you're looking for in a variable, like resizing or convenient initialization. Some of these things, like whether you get to set values on initalization, are technical consequences of the memory system. So if you want a different feature, like being able to resize in real time, suddenly you have to care about `malloc` and the pointer heap.

If we could bomb it all out and start over, we wouldn't tie together three sets of features with three sets of technical annoyances. But here we are.

All of this is about where you put your data. Variables are another level of fun:

1. If you declared your `char`, `int`, or `double` variable either outside of a function or inside a function with the `static` keyword, then it's static; otherwise it's automatic.

2. If you declared a pointer, the pointer itself has a memory type as per rule number one. But the pointer could be pointing to any of the three types of data. Static pointer to `malloc`ed data, automatic pointer to static data—all the combinations are possible.

Rule number two means that you can't identify the memory model by the notation. On the one hand, it's nice that we don't have to deal with one notation for auto arrays and one notation for manual arrays; on the other hand, you still often have to be aware of which you have on hand, so you don't get tripped up resizing an automatic array or not freeing a manual array. This is why the statement *C pointers and arrays are identical* is about as reliable as the rule about *i before e except after c*.

To do:
Check back on some code you have and go through the typology: what data is static memory, auto, manual; what variables are auto pointers to manual memory, auto pointers to static values, et cetera. If you don't have anything immediately on hand, try it with . (p **??**)

8.32 Tip 21: become a better typist

12 November 2011

level: basic computer user

purpose: gain some confidence at the keyboard

This may not be the C/POSIX tip you were expecting, but let me tell you how I taught myself to type. This is probably obvious, and I can verify from my experience teaching people how to use these systems, but comfort with the POSIX toolchain and comfort with the keyboard are closely (but imperfectly) correlated. It's hard to be comfortable on the command line if you're not comfortable typing.

If you're still a hunt-and-peck typist, then there are abundant tutorials out there to show you where the home keys are, and your search engine's recommendations are as good or better than mine. But for me, there was a point where I technically knew how to type but had hit my plateau. That's where this tip came in and made me the person I am today.

To do:
Next time you have some keyboard-oriented work to do, get a light t-shirt and drape it over the keyboard. Stick your hands under the shirt, and start typing.

The intent is to prevent that sneaking glance that we all do to check where the keys are. It turns out that the keys aren't very mobile and are always exactly where you left them. But those micropauses to check on things are how we keep our confidence and facility with the keyboard at a certain safe speed. If you're old enough to be reading this blog, then you've been looking at a qwerty keyboard for years now, and don't need those reassuring peeks.

Not being able to see will probably be frustrating for you at first, but persist through the initial awkwardness, and get to know those occasional keys that you never quite learned. When you are more confident with the keyboard, you'll have more brain power to dedicate to writing code.

8.33 Tip 22: all the casting you'll need

14 November 2011

level: intermediate
purpose: still less obsolete cruft in your life

There are two (2) reasons to cast a variable from one type to another.

First: when dividing two numbers, an integer divided by an integer will always return an integer, so the following statements will be true:

```
4/2 == 2
3/2 == 1
```

That second one is the source of lots of errors. It's easy to fix: if `i` is an integer, then i + 0.0 is a floating-point number that matches the integer. Don't forget the parentheses, but that solves your problem:

```
4/(2+0.0) == 2.0
3/(2+0.0) == 1.5
```

You can also use the casting form:

```
4/(float)2 == 2.0
3/(float)2 == 1.5
```

I'm partial to the add-zero form, for æsthetic reasons; you're welcome to prefer the cast-to-float form. But make a habit of one or the other every time you reach for that `/` key, because this is the source of many, many errors. [And not just in C; lots of other languages also like to insist that `int / int` → `int`. Not that that makes it OK.]

Second: array indices have to be integers. It's the law (C standard §6.5.2.1), and GCC will complain if you send a floating-point index. So, you may have to cast to an integer, even if you know that in your situation you will always have an integer-valued expression.

```
4/(float)2 == 2.0 //this is float, not an int.
mylist[4/(float)2]; //So this is an error: floating-point index

mylist[(int)(4/(float)2)]; //This works; take care with the parens

int index=4/(float)2;//This form also works,
mylist[index];       //and is more legible.
```

Now that I've covered both of the reasons to cast in C, I can point out the reasons to not bother. Notice that the `index` variable above was an integer, but the right-hand value was a floating-point number. C auto-casts in this case, truncating down to the right value. If it's valid to assign an item of one type to an item of another type, then C will do it for you without your having to tell it to with an explicit cast; if it's not valid, then you'll have to write a function to do the conversion anyway.

C++ isn't like this: you have to explicitly cast in all cases. Fortunately, you're writing in C, so you can ignore C++ tutorials that tell you to explicitly cast. [And as a broad rule that universally works for me: don't bother with anything that uses *C/C++* in the title.]

In the 1970s and 80s, `malloc` returned a `char*` pointer, and had to be cast (unless you were allocating a string), with a form like:

```
//don't bother with this sort of redundancy:
float* list = (float) malloc(list_length * sizeof(float));
```

You don't have to do this anymore, because `malloc` now gives you a `void*` pointer, which the compiler will comfortably auto-cast to anything.

If you check the examples above, you'll see that I even gave you options to avoid the casting syntax for the two legitimate reasons to cast: adding 0.0 and declaring an integer variable for your array indices. Bear in mind the existence of the casting form `var_type2 = (type2) var_type1`, because it might come in handy some day, and in a few tips we'll get to declarations that mimic this form. But for the most part, explicit type casting is just redundancy that clutters the page.

8.34 Tip 23: the limits of `sizeof`

16 November 2011

level: obscure
purpose: recognize bad advice about `sizeof`

This tip is about the `sizeof` operator, and one difference where the statement *arrays and pointers are identical in C* is false.

I marked this entry as *obscure* because you can live a long life without using the `sizeof` operator except inside allocations like

```
element_type * newlist = malloc(listlen * sizeof(element_type));
```

I even wouldn't fault you if you wrapped it in a macro like

```
#define Allocate(element_type, listname, listlength) \
    element_type * listname = malloc(listlength * sizeof(element_type));
```

and never typed the word `sizeof` again. [You'd need a realloc macro too. Exercise for the reader.]

To summarize today's tip: the above form is a safe use of `sizeof`, and not much else is.

We start the story under the hood. The C compiler is famously ignorant of metadata, knowing only a few facts about your data:

1. It knows the base location of your data (so you can always point to it).

2. It knows the size of one unit of your data. Recall that C relies heavily on a f (p **??**)or pulling elements of arrays and structs, so it needs to know how many bytes to step when you write `base + 3` or `mystruct.third_elmt`.

3. For static and automatic memory, it needs to know the total size that has to be automatically freed at the end of the function or at the end of the program.

That's about it.

[That is as much as the system needs for its own operation, but ¿why doesn't C provide more, like a consistent method to query the size of a block of manually allocated memory? Letting implementers of `malloc` pick their favorite means of recording the block size provided the sort of freedom that delights the system programmers, and thanks to this non-policy there are a lot of different implementations of `malloc`[18]. It's annoying, but your computer is faster for it.]

Which brings us to the `sizeof` operator. You might think `sizeof` is just another function, but it's a keyword built into the compiler, because it has to have exceptional knowledge about structure internals and is the only non-macro chance you have to operate on a type. It is a window into item #3 in the list.

Here's a trick that's often thrown around[19]: you can get the size of an automatic or static array by dividing its total size by the size of one element. This is usually via a form like

[18] `http://en.wikipedia.org/wiki/Malloc\#Implementations`
[19] `http://c-faq.com/aryptr/arraynels.html`

```
//This is not reliable:
#define arraysize(list) sizeof(list)/sizeof(list[0])
```

The denominator of the expression depends on #2 above: the system has to know the size of one element.

The numerator really depends on #3, and is where the distinction between automatic versus manually-allocated data will trip you up. What is the size of the data that C will have to free when the variable goes out of scope? For an automatic array like `double list[100]`, the compiler had to allocate a hundred `double`s, and will have to free that much space at the end of scope. For manually-allocated memory, all the system has to do at the end of the scope is destroy the pointer—freeing the data itself is your problem. So: `sizeof` will probably return 200 in the case of the auto array, and will probably return one in the case of the manual array.

Some cats, when you point to a toy, will go and inspect the toy; some cats will sniff your finger.

Here's some sample code, so you can see what your own system returns when dealing with automatic and manual memory.

```
#include <stdio.h>

#define peval(cmd) printf(#cmd ": %g\n", cmd);

int main(){
    double *liszt = (double[]){1, 2, 3};
    double list[] = {1, 2, 3};
    peval(sizeof(liszt)/(sizeof(double)+0.0));
    peval(sizeof(list)/(sizeof(double)+0.0));
}
```

You'll recognize the add-zero trick from . (p **??**) The initialization of `liszt` may be a form unfamiliar to you. For now, rest assured that it works in appropriate conditions; I'll get to it in a few tips, so think of it as foreshadowing.

When you run the program, you get two different values. The first variable is a pointer, and the second an array. The size of the first is the size of one pointer (which is appropriately half of a `double`); the size of the second is three `double`s long.

The only reason the system cares about the total size of your data is for the purposes of freeing it when leaving scope, and that's the size you're going to get when you use `sizeof`. But that is pretty much always a different purpose than what you had in mind.

[Formally, the `sizeof` operator isn't really tied to the end-of-scope freeing, but it coincides so darn well that I'm comfortable recommending it here as a functional mental model.]

This break in purposes makes `sizeof` largely useless outside of calls to `malloc`. We'd like to write a function `manipulate\_array(double in_array[])` and use `sizeof` to get the size of the input array, rather than wasting the user's time asking for the length. But that won't work because the user may send either a pointer or an array, and we won't know which. [It can also fail for other reasons that aren't worth getting into.]

8.35 Tip 24: Compound literals

18 November 2011

level: medium
purpose: create fewer one-off temp variables; lots of future uses

I know, you have no idea what the title means, but thanks for clicking through anyway.

You can write a single element into your text easily enough—C has no problem understanding `f(34)`.

But if you want to send a list of elements as an argument to a function—a compound literal value like `{20.38, a_value, 9.8}`—then there's a syntactic caveat: you have to put a sort of type-cast before the compound literal, or else the parser will get confused. The list now looks like this: `(double[]) {20.38, a_value, 9.8}`, and the call looks like

```
f((double[]) {20.38, a_value, 9.8});
```

To give a full example, say that we have a function `sum` that takes in an array of `double`s. Then here are two ways for `main` to call it:

```
#include <math.h> //NAN
#include <stdio.h>

double sum(double  in[]){
    double out=0;
    for (int i=0; !isnan(in[i]); i++) out += in[i];
    return out;
}

int main(){
    double *list = (double[]){1.1, 2.2, 3.3, NAN};
    printf("sum: %g\n", sum(list));

    printf("quick sum: %g\n", sum((double[]){1.1, 2.2, 3.3, NAN}));
}
```

The first two lines of `main` generate a single-use array named `list` and then send it to `sum`; the last line does away with the incidental variable and goes straight to using the list.

[This paragraph is arcana; you are welcome to skip it.] The first line in `main` is the typical example of an initialization via a compound literal. ¿How does it differ from

```
double alist[] = {0.1, 0.3, 0.5, NAN};
```

? This is back to the obscureness of the . (p **??**) For the typical array intialization, we have an auto-allocated array and it is named `alist`. The compound initializer generates an anonymous auto-allocated array, and then we immediately point a pointer to it. So `alist` *is* the array, while `list` is a pointer to an anonymous array. May you never be in a position where you have to care about this distinction.

There's your intro to C.I.s; I'll demonstrate many uses over the coming weeks. Meanwhile, ¿does the code on your hard drive use any quick throwaway lists whose whose use could be streamlined by a compound initializer?

8.36 Tip 25: Variadic macros

20 November 2011

level: not hard
purpose: delightful tricks to follow in the coming week

I broadly consider variable-length functions in C to be broken—I have a personal rules about the three forms I'm willing to use, which I'll maybe expound upon later.

But variable-length macro arguments are easy. The keyword is `__VA_ARGS__`, and it expands to whatever set of elements were given.

For example, here's a fast way to implement a customized variant of printf for reporting errors:

```
#define print_a_warning(...) {\
    printf("Glitch detected in %s at line %s:%i: ", __FUNCTION__, __FILE_
    printf(__VA_ARGS__); }

//usage:
print_a_warning("x has value %g, but it should be between zero and one.\r
```

The `__FUNCTION__`, `__FILE__`, and `__LINE__` macros get filled in with what you'd expect.

You can probably guess how the ellipsis (`...`) and `__VA_ARGS__` work: whatever is between the parens gets plugged in at the `__VA_ARGS__` mark.

You can have arguments before the ellipsis if you want. A full example:

```
#define print_a_warning(dostop, ...) { \
    printf("Glitch detected in %s at line %s:%i: ", __FUNCTION__, __FILE_
    printf(__VA_ARGS__);    \
    if (dostop=='s') abort(); \
}

int main(int argc, char ** argv){
    if (argc <= 1) print_a_warning('s', "argc has value %i, but it shoulc
    if (argc > 2) print_a_warning('c', "argc has value %i, but it should

    printf("arg one = %s\n", argv[1]);
}
```

8.37 Tip 26: Safely terminated lists

22 November 2011

level: follows from a (p **??**)nd
(p **??**) **purpose**: operate on lists with less risk of segfaults

Compound literals and variadic macros are the cutest couple, because we can now use macros to build lists and structures. I'll get to the structure building a few tips from now; let's start with lists.

Here's the sum function from a few episodes ago.

```
double sum(double  in[]){
    double out=0;
    for (int i=0; !isnan(in[i]); i++) out += in[i];
    return out;
}
```

When using this function, you don't need to know the length of the input array, but you do need to make sure that there's a NaN marker at the end. [You'll also have to check beforehand that none of your inputs are NaNs.] Now the fun part, where we call this function using a variadic macro:

```
#include <math.h> //NAN
#include <stdio.h>

#define sum(...) sum_base((double[]){__VA_ARGS__, NAN})

double sum_base(double  in[]){
    double out=0;
    for (int i=0; !isnan(in[i]); i++) out += in[i];
    return out;
}

int main(){
    double two_and_two = sum(2, 2);
    printf("2+2 = %g\n", two_and_two);
    printf("(2+2)*3 = %g\n", sum(two_and_two, two_and_two, two_and_two));
    printf("sum(asst) = %g\n", sum(3.1415, two_and_two, 3, 8, 98.4));
}
```

Now that's a stylish function. It takes in as many inputs as you have on hand, and operates on each of them. You don't have to pack the elements into an array beforehand, because the macro uses a compound initializer to do it for you.

In fact, the macro version only works with loose numbers, not with anything you've already set up as an array. If you already have an array—and if you can guarantee the `NAN` at the end—then you can call `sum_base` directly.

8.38 Tip 27: Foreach in C

24 November 2011

level: medium
purpose: borrow a useful scripting language construct

Last time, you saw that you can use a *compound literal* anywhere you would put an array or structure.

For example, here is an array of strings declared via a compound literal:

```
char **strings = (char*[]){"Yarn", "twine"};
```

The compound literal is automatically allocated, meaning that you need neither `malloc` nor `free` to bother with it. At the end of your function it just disappears.

Now let's put that in a `for` loop. The first element of the loop declares the array of strings, so we can use the line above. Then, we step through until we get to the `NULL` marker at the end. For additional comprehensibility, I'll `typedef` a string type, as per . (p **??**)

```
#include <stdio.h>

typedef char* string;

int main(){
    string str = "thread";
    for (string *list = (string[]){"yarn", str, "rope", NULL}; *list; lis
        printf("%s\n", *list);
}
```

It's still noisy, so let's hide all the syntactic noise in a macro. Then `main` is as clean as can be:

```
#include <stdio.h>
//I'll do it without the typedef this time.

#define foreach_string(iterator, ...)\
    for (char **iterator = (char*[]){__VA_ARGS__, NULL}; *iterator; itera

int main(){
    char *str = "thread";
    foreach_string(i, "yarn", str, "rope"){
        printf("%s\n", *i);
    }
}
```

To do:
Rewrite the macro to use `void` pointers rather than strings. If you're the sort of person who never tries even the easy exercises when reading tutorials then (1) I hate you, and (2) just wait until next time, when I'll use the solution to this as part of the next tip.

8.39 Tip 28: Vectorize a function

26 November 2011

level: easy if you've read all the medium tips so far
purpose: make your code and math look more similar

The `free` function takes exactly one argument, so we often have a part at the end of a function of the form

```
free(ptr1);
free(ptr2);
free(ptr3);
free(ptr4);
```

'!How annoying! No self-respecing LISPer would ever allow such redundancy to stand, but would write a macro to allow a vectorized `free` function that would allow:

```
free_all(ptr1, ptr2, ptr3, ptr4);
```

If you've been following along since , (p **??**) then the following sentence will make complete sense to you: we can write a variadic macro that generates an array (ended by a stopper) via compound literal, then runs a `for` loop over the array. Here's the code:

```
#define fn_apply(fn, ...) { \
    void *stopper_for_apply = (int[]){0};    \
    void **list_for_apply = (void*[]){__VA_ARGS__, stopper_for_apply}; \
    for (int i=0; list_for_apply[i] != stopper_for_apply; i++) fn(list_for_apply[i
}
```

We need a stopper that we can guarantee won't match any in-use pointers, including any `NULL` pointers, so we use the compound literal form to allocate an array holding a single integer, and point to that. Notice how the stopping condition of the `for` loop looks at the pointers themselves, not what they are pointing to.

Usage so far:

```
fn_apply(free, ptr1, ptr2, ptr3, ptr4);
```

If we want this to really look like a function, then we can do that via one more macro:

```
#define free_all(...) fn_apply(free, __VA_ARGS__);

//We can wrap this foreach around anything that takes in a pointer.
//For GSL and Apophenia users, let's define:
#define gsl_vector_free_all(...) fn_apply(gsl_vector_free, __VA_ARGS__);
#define gsl_matrix_free_all...) fn_apply(gsl_matrix_free, __VA_ARGS__);
#define apop_data_free_all...) fn_apply(apop_data_free, __VA_ARGS__);
```

Adding it all up:

```
#include <stdio.h>
#define fn_apply(fn, ...) { \
    void *stopper_for_apply = (int[]){0};    \
    void **list_for_apply = (void*[]){__VA_ARGS__, stopper_for_apply}; \
    for (int i=0; list_for_apply[i] != stopper_for_apply; i++) fn(list_fo
}

#define free_all(...) fn_apply(free, __VA_ARGS__);

int main(){
    double *x= malloc(10);
    double *y= malloc(100);
    double *z= malloc(1000);

    free_all(x, y, z);
}
```

If the input isn't a pointer but is some other type (`int`, `float`, &c), then you'll need a new macro. Implementing this (either for one new type or taking a type as an argument to the macro) is left as an exercise for the reader. I showed you a version with strings in the . (p **??**) Also, if you want compile-time warnings about type errors, then you can rewrite the function here to use `gsl_vector *` pointers, `gsl_matrix *` pointers, . . . , instead of just `void*` pointers.

You could rewrite this to return a value for each input item, but at this point we might be stretching what we want a macro to do. If you're a fan of the Apophenia library, it has the `apop_map` function (and friends) to do this sort of thing with vectors and matrices.

8.40 Tip 29: Preprocessor tricks!

28 November 2011

level: getting advanced
purpose: turn code into more code

The token reserved for the preprocessor is the octothorpe, #, and the preprocessor makes three (3) entirely different uses of it.

You know that a preprocessor directive like `#define` begins with a # at the head of the line. Whitespace is ignored, so here's your first tip: you can put throwaway macros in the middle of a function, just before they get used, and indent them to flow with the function. According to the Old School, putting the macro right where it gets used is against the Correct organization of a program (which puts all macros at the head of the file), but having it right there makes it easy to refer to and makes the throwaway nature of the macro evident.

Next use of the #: in a macro, it turns input code into a string. Here's the code from , (p **??**) slightly rewritten:

```
#include <stdio.h>

int main(){
    #define peval(cmd) printf(#cmd ": %g\n", cmd);
    double *liszt = (double[]){1, 2, 3};
    double list[] = {1, 2, 3};
    peval(sizeof(liszt)/(sizeof(double)+0.0));
    peval(sizeof(list)/(sizeof(double)+0.0));
}
```

When you try it, you'll see that the input to the macro is printed as plain text, because #`cmd` is equivalent to `cmd` as a string.

So `peval(list[0])` would expand to

```
printf("list[0]" ": %g\n", list[0]);
```

Does that look malformed to you, with the two strings `"list[0]" ": %g\n"` next to each other? Next preprocessor trick: if two literal strings are adjacent, the preprocessor merges them into one: `"list[0]: %g\n"`. And this isn't just in macros:

```
printf("You can use the preprocessor's string "
       "concatenation to break long lines of text "
       "in your program. I think this is easier than "
       "using backslashes, but be careful with spacing.");
```

Conversely, you may want to join together two things that are not strings. Here, use two octothorpes, which I will herein dub the hexadecathorpe : ##. If the input is `LL`, then when you see `name ## _list`, read it as `LL_list`, which is a valid and useful variable name.

Gee, you comment, *I sure wish every array had an auxiliary variable that gave its length.* OK, let's write a macro that declares a local variable ending in `_len` for each list you tell it to care about. We'll even make sure every list has a terminating marker, so you don't even need the length.

That is, this macro is total overkill, but does demonstrate how you can generate lots of little temp variables that follow a naming pattern that you choose.

```
#include <stdio.h>
#include <math.h> //NAN

#define Setup_list(name, ...) \
    double *name ## _list = (double [])(__VA_ARGS__, NAN};   \
    int name ## _len = 0; \
    for (name ## _len =0; !isnan(name ## _list[name ## _len]); \
                         name ## _len ++) /*do nothing.*/;


int main(){
    Setup_list(items, 1, 2, 4, 8);
    // Now we can use items_len and items_list:
    double sum=0;
    for (double *ptr= items_list; !isnan(*ptr); ptr++)
        sum += *ptr;
    printf("total for items list: %g\n", sum);


    // Some systems let you query an array for its
    // own length using a form like this:
    #define Length(in) in ## _len

    sum=0;
    Setup_list(next_set, -1, 2.2, 4.8, 0.1);
    for (int i=0; i < Length(next_set); i++)
        sum += next_set_list[i];
    printf("total for next set list: %g\n", sum);
}
```

Discussion:
The macro above really is pretty bad form, and tries to hard for the sake of demonstrating multiple new variables. But there are some in the Old School who eye all macros warily. C is built upon *expressions*, which evaluate to produce other expressions, and can be plugged in wherever. That's why you can write things like `if (x = (y==f(z))...` and it all makes sense (or at least, compiles correctly). The macros above don't produce expressions, but are blocks of code that generate new variables and do all sorts of math along the way.

Macros are a pain to debug and can do tricky things, so here's my subjective style tip: lean toward macros that aren't expressions, and which can't be used in the middle of a stream of math even if you wanted to. When something breaks, you'll need to hunt through the macro to find the error; it's a pain, but at least you have the block of code the macro produces isolated.

8.41 Tip 30: Use Apophenia to read in data and configuration info

30 November 2011

level: Basic for anybody dealing with data
purpose: Use libraries!

This is a special case of a (p **??**)bout using pre-existing libraries wherever possible. After all, C's big edge is that it's been around for forty years; that's a lot of time for useful libraries to get written.

Reading in text is an especially difficult problem that everybdoy has to deal with so it is especially library-appropriate. Despite my self-conscious desire to not self-promote, I'm gonna tell you that Apophenia does a decent job with this.

First, let's generate a data set. I'll wrap it up as a here document, as per , (p **??**) so you can just paste this onto the command line:

```
cat > text_data << "."
left|middle|right
2|5| 12
3|8|9
3|8|Galia est omnis divisa en partes tres
.
```

The sample data shows the first tip for the day: use pipes as field delimiters. Pipes really look like the bounds between fields, and they rarely appear in the data you're putting into a text file. The default for so many systems is commas or tabs, both of which are just asking for glitches.

Reading a data set to a matrix is pretty trivial via Apophenia. In this example, I'll stretch it out by first reading into the database (instead of directly using `apop_text_to_data`, which would save two lines of code but lose the non-numeric input). [And remember a (p **??**)bout compiling C code via here document? It's how I test all the sample code I put here, and is still an easy way for you to try it all out.]

```
#include <apop.h>

int main(){
    sprintf(apop_opts.input_delimiters,"|");
    apop_text_to_db("text_data", "datatab");
    apop_data *indata = apop_query_to_data("select "
                            "left, middle from datatab");
    Apop_col(indata, 0, firstrow);
    Apop_col(indata, 1, secondrow);
    printf("first column sum: %Lg\n", apop_sum(firstrow));
    printf("second column sum: %Lg\n", apop_sum(secondrow));
}
```

If you installed the Apophenia library, then you also have the command-line `apop_text_to_db`, which just runs the C function in the second line of `main`.

8.42 Tip 31: Use the database for configuration info

2 December 2011

level: writing programs with lots of options
purpose: avoid writing yet another config system

An easy text-to-database conduit isn't just for data sets. Your project may also need an extensive set of configuration details. Simulations are especially prone to this:

how many periods should run, how many agents should we start with, what percent of agents will be type 1, et cetera. These can all be expressed as plain text.

You could thus write a text file where each line is a key, followed by a colon, followed by the key's value. Read in the text file using `apop_text_to_db` with the delimiter set to `:`, and you now have a key/value database with all of your configuration info.

Let's re-do the Fibnoacci example from . (p **??**) First, a config file, here-document-ified for your cutting and pasting convenience:

```
cat > fib_config << "."

title: The Fibonacci sequence
how many: 20

#In my example, I started the sequence with 1, 1, but the
#Fib. sequence formally starts with 0, 1.
first: 0
second: 1

#Or uncomment these to try Lucas numbers, which start with:
#first: 2
#second: 1
#title: The Lucas sequence

#The setup below will use only the first instance of any given key.
.
```

Our keys can have spaces (and basically any other odd characters that make the names human-friendly), because they're text to be read into the database, not variable names. Apophenia follows the shell convention that #s indicate comment lines.

Now for the program that uses the config file. The big change from the version in Tip #19 (where this code was used to introduce static variables for state machines) is the `Get_float_key` macro used throughout, and that `main` starts by reading in the database and setting defaults. If you recall the , (p **??**) the macros should be easy to read.

```
#include <apop.h>

#define Get_float_key(k) \
    apop_query_to_float("select value from config where key='" #k "'")

//Get_text_key leaks. New School Tip: the leak is too small to be worth c
#define Get_text_key(k) \
    apop_query_to_text("select value from config where key='" #k "'")->te

#define Check_key(k, default)   \
    if (!apop_query_to_float("select count(*) from config where key='" #k
        apop_query("insert into config values ('" #k "', '" #default "')");

#define Staticdef(type, var, initialization) \
    static type var = 0; \
    if (!(var)) var = initialization

long long int fibonacci(){
    Staticdef(long long int, first, Get_float_key(first));
    Staticdef(long long int, second, Get_float_key(second));
    long long int out = first+second;
    first=second;
    second=out;
    return out;
}

int main(){
    sprintf(apop_opts.input_delimiters,":");
    apop_text_to_db("fib_config", "config", .field_names = (char*[]){"key", "value
    Check_key(title, Fibbo sequence);
    Check_key(first, 0);
    Check_key(second, 1);
    Check_key(how many, 10);

    printf("%s\n", Get_text_key(title));
    int max=Get_float_key(how many);
    for (int i=0; i< max; i++)
        printf("%Li\n", fibonacci());
}
```

Great, now you can endlessly tweak your program's parameters without recompiling. If you have to keep a dozen different variant runs organized, you can write a single script that precedes each run of your program with a new config file generated via here document.

There are lots of other ways to read in config data; have a look at *Modeling with Data* pp 203–210 for several. But here's the short version: if you're using `fscanf` or `getchar`, there are higher-level tools that will do the work for you.

8.43 Tip 32: Get to know your shell

4 December 2011

level: basic command-line habitant
purpose: use the conveniences available to you

My favorite lines from this video about UNIX[20]:

[20] http://www.youtube.com/watch?v=JoVQTPbD6UY

"We are trying to make computing as simple as possible."
". . . we wanted. . . not just a good programming environment. . . but a system around which a community could form—a fellowship."
"The UNIX operating system is basically made up of three parts: the kernel . . . the shell. . . the various utility programs, which perform specific tasks like editing a file or sorting a bunch of numbers or making a plot."

The first two are just something to make you go *hmm*, but the third is a real statement of policy: the shell that you just use to `cd` and `ls` around before pulling up your editor was intended to really be a core part of how you do work on a UNIX box. As such, the shell does a lot more than just walk around the filesystem and start programs.

A POSIX-standard shell will have

1. an interactive front-end—the command prompt—which may include lots of user-friendly tricks,
2. a system for recording and re-using everything you typed—history,
3. abundant macro options, in which your text is replaced with new text—i.e., an *expansion* syntax, and
4. a Turing-complete programming language.

There is *a lot* of shell scripting syntax, but the next four tips will cover a few pieces of low-hanging syntactic fruit for the above four categories. There are many shells to be had (and my last tip will be that you try a different one from the one you're using now), but most of these tips are are POSIX-standard, and so work in any shell.

bang star I'll start with item #2 from the list: history. If you don't want to reach all the way over to the up arrow, `!!` will repeat the prior command. [This `!` stuff isn't POSIX-standard, but seems pretty standard across shells.] I find this useful when I'm editing the source code for `goprogram.c` in one window and running it in another. Here's what I usually wind up typing in that run window's command prompt to compile and run the program over and over:

```
make; ./goprogram
!!
!!
!!
!!
```

Now divide the command line into the first item (the command), and everything else (the command arguments). You can paste the command arguments into the current line with `!*`. To make a directory and then step into it without retyping:

```
mkdir /home/b/tech/code_snippets/try_this
cd !*
```

If you don't have an edit window/run window setup, then you can alternate between editing a Python script and running it with:

```
vi a_script_that_I_am_writing.py
python !*
vi !*
python !*
```

where you can replace `vi` with the editor of your choice. If you are using `vi` itself, then you can of course run an executable script without leaving the editor via the `:!. %` command. `vi` is wonderful like that.

[In many shells, `!p` re-runs the last command that started with a *p* (and `!pyth` pulls up the last command that started with *pyth*, but why do extra typing). In which case you could turn the above sequence into `vi script`, then `python !*`, `!v`, `!p`, `!v`, `!p`. But use this form sparingly and don't just fish through your history, because `!r` might pull up an `rm *` you forgot about.]

fc This is a command for turning your noodling on the shell into a repeatable script. Try

```
fc -l   #the l is for 'list' and is important
```

You now have on the screen a numbered list of your last few commands. Your shell may let you type `history` to get the same effect.

You can write history items 100 through 200 to a file via `fc -l 100 200 > a_script`. Cut out the line numbers [or use `fc -n -l` to not print them to begin with], remove all your experiments that didn't work, and you've converted your futzing on the command line into a clean shell script.

In most shells [not POSIX-standard, but if it works, use it], you can run the shell script via `source a_script`, or the convenient shorthand `. a_script`, which trades comprehensibility for brevity.

If you omit the `-l` flag, then `fc` becomes a much more immediate and volatile tool. It pulls up an editor immediately (which means if you redirect with `>` you're basically hung), doesn't display line numbers, and when you quit your editor, whatever is in that file gets executed immediately. This is great for a quick repetition of the last few lines, but can be disastrous if you're not careful. If you realize that you forgot the `-l` or are otherwise surprised to see yourself in the editor, delete everything on the screen to prevent unintended lines from getting executed.

But to end on a positive note, `fc` stands for *fix command*, and that is its simplest usage. With no options it edits the prior line only, so it's nice for when you need to make more elaborate corrections to a command than just a typo.

8.44 Tip 33: Replace shell commands with their outputs

6 December 2011

level: you want something more than pipes
purpose: use outputs as inputs to the next step

, (p **??**) I gave you a four-item list of things your shell can do. Number three was expansions: replacing certain blobs of text with other text.

Variables are a simple expansion. If you set a variable like

```
onething="another thing"
```

on the command line [C shell users: `set onething="another thing"`], then when you later type

```
echo $onething
```

then `another thing` will print to screeen.

Shell variables are a convenience for you to use while working at the command prompt or throwing together a quick script. They are stupendously easy to confuse with *environment variables*, which are sent to new processes and read via a simple set of C functions. Have a look at Appendix A of *Modeling with Data* for details on turning shell variables into environment variables.

Also, your shell will require that there be no spaces on either side of the `=`, which will annoy you at some point. [This rule is for the purposes of supporting a feature that is mostly useful for makefiles.] But there you have it: our easiest and most basic substitution of one thing for another.

[Isn't it conveniently nifty that the $ is so heavily used in the shell, and yet is entirely absent from C code, so that it's easy to write shell scripts that act on C code (like in) (p **??**), and C code to produce shell scripts? It's as if the UNIX shell and C were written by the same people to work together.]

For our next expansion, how about the backtick, which on a typical keyboard shares a key with the ˜ and is not the more vertical-looking single tick `'`. [The vertical tick indicates that you don't want expansions done: `echo '$onething'` will actually print `$onething`.] The backtick replaces the command you give with the output from the command, doing so macro-style, where the command text is replaced in place with the output text. Here's an example in which we count lines of C code by how many lines have a `;`, `)`, or `}` on them; given that lines of source code is a lousy metric for most purposes anyway, this is as good a means as any, and has the bonus of being one line of shell code:

```
 #count lines with a ), }, or ;, and let that count be named Lines.
Lines=`grep '[)};]' *.c | wc -l`

 #count how many lines there are in a directory listing; name it Files.
Files=`ls *.c |wc -l`

echo files=$Files and lines=$Lines

 #Arithmetic expansion is a double-paren.
 #In bash, the remainder is truncated; more on this later.
echo lines/file = $(($Lines/$Files))

 #Or, use those variables in a here script.
 #By setting scale=3, answers are printed to 3 decimal places.
```

```
bc << ---
scale=3
$Lines/$Files
---
```

OK, so now you've met variable substitution, command substitution, and in the sample code I touched on arithmetic substitution for quick desk calculator math. That's what I deem to be the low-hanging fruit; I leave you to read the manual on alias expansion, history expansion, brace expansion, tilde expansion, parameter expansion, word splitting, pathname expansion, glob expansion, and the difference between `" "` and `′ ′`.

8.45 Tip 34: Use the shell's for loops to operate on a set of files

8 December 2011

level: you use the shell enough to be frustrated by it
purpose: do the same thing to a bunch of files

Continuing on with the discussion of getting more from the shell (begun in) (p **??**), let's get to some proper programming, with if statements and for loops.

But here's your first tip about programming using your shell's language: don't do it. The shell is Turing complete, and has variables and functions that look like those in any other language, but it's especially easy to write unmaintainable code in the shell. Scope is awkward—pretty much everything is global. It's a macro language, so all those things that they warned you about when you write two lines of C preprocessor code are relevant for every line of your shell script. There are little tricks that will easily catch you, like how you can't have spaces around the `=` in `onething=another`, but you must have spaces around the `[` and `]` in `if [ -e ff ]` (because they aren't characters—they're kewords that just happen to not have any human-language characters in them). Write shell scripts to automate what you would type at the command line, and if you need to go further take the time to switch to Perl, Python, &c.

for loops My vote for greatest bang for the buck from having a programming language that you can type directly onto the command line goes to running the same command on several files. Here, let's back up every `.c` file the old fashioned way, by copying it to a new file with a name ending in `.bkup`:

```
for file in *.c;
do
 cp $file ${file}.bkup;
done
```

You see where the semicolon is: at the end of the list of files the loop will use, on the same line as the `for` statement. I'm pointing this out because I find it to be hopelessly counterintuitive, especially when we cram this onto one line:

```
for file in *.c; do cp $file ${file}.bkup; done
```

It somehow bothers me that the `do` is right there with the command, but there you have it.

For your scientific computing needs, the `for` loop is useful for dealing with a sequence of N runs. By way of a simple example, let's search our C code for digits, and write each line that has a given number to a file:

```
for i in 0 1 2 3 4 5 6 7 8 9; do grep $i *.c > lines_with_${i}; done
wc -l lines_with*  #a v. rough histogram of your digit usage.
```

Testing against Benford's law is left as an exercise for the reader.

The curly braces in `${i}` are there to distinguish what is the variable name and what is subsequent text; you don't need it here, but you would to make a file name like `${i}lines`.

You may have the `seq` command installed on your machine—it's BSD standard but not POSIX standard. Then we can use backticks to generate a sequence:

```
for i in ‘seq 0 9‘; do grep $i *.c > lines_with_${i}; done
```

Running your simulation a thousand times is now trivial:

```
for i in ‘seq 1 1000‘; do ./run_sim > ${i}.out; done

#or append all output to a single file:
for i in ‘seq 1 1000‘; do echo run $i >> sim_out; ./run_sim >> sim_out; d
```

8.46 Tip 35: Use the shell to test for files

10 December 2011

level: you want some automation out of your shell
purpose: look before you leap

, (p **??**) I discussed how the shell can be used as a Turing-complete programming language (and advised you to not use it as such). For example, you could automate the sort of thing you'd type at a command line, like setting up a sequence of runs of a program.

Now let's say that your program relies on a data set that has to be read in from a text file to a database. You only want to do the read-in once; in pseudocode: if database exists do nothing, else generate database from text.

On the command line, you would use `test`, a versatile command typically built into the shell. Run a quick `ls`, get a file name you know is there, and use `test` like this:

```
test -e a_file_i_know
echo $?
```

By itself `test` outputs nothing, but since you're a C programmer, you know that every program has a `main` function that returns an integer, and we will use only that return value here. Custom is to read the return value as a problem number, so 0=no problem, and in this case 1=file does not exist. [Which is why, as per , (p **??**) the default is that `main` returns zero.] The shell doesn't print the return value to the screen, but stores it in a variable, `$?`, which you can print via `echo`.

OK, now let us use it in an `if` statement to act only if a file does not exist. As in C, `!` means *not*.

```
if test ! -e a_test_file; then
    echo test file had not existed
    touch a_test_file
else
    echo test file existed
    rm a_test_file
fi
```

Notice that, as with the `for` loops from last time, the semicolon is in what I consider an awkward position, and we have the super-cute rule that we end `if` blocks with `fi`. To make it easier for you to run this repeatedly using `!!`, let's cram it onto one margin-busting line. The keywords `[` and `]` are equivalent to `test`, so when you see this form in other people's scripts and want to know what's going on, the answer is in `man test`.

```
if [ ! -e a_test_file ]; then echo test file had not existed; touch a_test_file; e
```

The multi-line version would make a fine header for the script from last time: start with some `if` statements to check that everything is in place, then run a `for` loop to run your program a few thousand times.

The condition is considered to be true when the evaluated expression is zero (=no problem), and false when it is nonzero (=problem). So outside of the `test` command you can think of the typical if statement as *if the program ran OK, then…*, which makes it perfect for error checking:

```
#generate some test files
mkdir a_test_dir
echo testing ... testing > a_test_dir/tt

#Remove the test files iff they were archived right
if tar cz a_test_dir > archived.tgz; then
    echo Compression went OK. Removing directory.
    rm -r a_test_dir
else
    echo Compression failed. Doing nothing.
fi
```

[If you want to see this fail after running once, try `chmod 000 archived.tgz` to make the destination archive unwriteable, then re-run.]

8.47 Tip 36: Try a new shell

12 December 2011

level: command-line habitant
purpose: get comfortable in your shell

I t (p **??**)his set of shell tips with a list I made up: the shell provides (1) facilities for interactive comfort, (2) history, (3) a ton of macro-like expansions, and (4) programming standards like `for` loops and `if` statements. I then gave you tips for parts (2), (3), and (4).

Here's my tip for item (1): try a new shell. There is no particular reason for the interactive features to stick to any one standard, because it is by definition not the programmable and scriptable part of things, so some shells provide much more user comfort than others. If some shell wants to have an animated paperclip in the corner kibbitzing your work, who're they hurting?

The shell you're using now is probably bash, the Bourne-again shell, so named because it's a variant of Stephen Bourne's shell for the original UNIX. It is part of the GNU project, and so has very wide distribution. But wide distribution means that it can't be too experimental with new features and comforts, because it has to run everywhere, and for every stupid quirk somebody has written a *really important* shell script that depends upon that quirk. I have my own complaints: GNU tools tend to provide half-hearted support for the `vi` keymap (which I have very much internalized), and the manual page is amazing for having absolutely no examples.

If you are on a Mac with software a couple of years old, then you may be on a version of the C shell. The C shell is really not an acceptable shell anymore, and you will especially reap benefits from dropping it.

Notably, there's tab completion. In bash, if you type part of a file name and hit `<tab>`, the name will be autocompleted if there's only one option, and if not, hit `<tab>` again to see a list options. If you want to know how many commands you can type on the command line, hit `<tab><tab>` on a blank line and bash will give you the whole list.

There are two types of shell users: those who didn't know about this tab-completion thing, and people who use it *all the time on every single line*. Now and then I wind up on a machine (like an old Mac) that somehow doesn't have tab completion, and I can only go about fifteen minutes trying to work with it before winding up in the fœtal position.

But shells beyond bash go further. When you type `ls -<tab>` in the Z shell, it will check the help pages and tell you what command line switches are available, and when you type `make <tab>` it will read your makefile and tell you the possible targets. The Friendly Interactive shell (fish) will check the manual pages for the summary lines, so when you type `man l<tab>` it will give you a one-line summary of every command beginning with L, which may save you the trouble of actually pulling up any man page at all.

Here are a few zsh tips, that give you a hint as to what switching from bash can get you. There are lots of other shells out there; I'm writing about zsh because it's the one

I know best [and it gets the vi keymap right].

You can find many pages of Z shell tips[21], or maybe check out this 14-page Zsh reference card[22] [PDF]. So there goes parsimony—but why bother being Spartan with interactive conveniences. [If you have Spartan æsthetics, then you still want to switch out of bash; try `ash`.] Much of the documentation is taken up by options you can add to your `.zshrc` (or just type onto the command line); here are the two you'll need for the examples below:

```
setopt INTERACTIVE_COMMENTS
#now commends like this won't give an error
setopt EXTENDED_GLOB
#for the paren-based globbing below.
```

Expansion of globs, like replacing `file.*` with `file.c file.o file.h` is the responsibility of the shell. The most useful way in which Zsh expands this is that `**/` tells the shell to recurse the directory tree when doing the expansion. A POSIX-standard shell reads ˜ to be your home directory, so if you want every `.c` file anywhere in your purview, try

```
ls ˜/**/*.c
```

Remember last time how we backed up our `.c` files? Let's do that with every last one of 'em:

```
for ff in ˜/**/*.c; do cp $ff ${ff}.bkup; done

#What backups do we have now?
#you may need "ls --color=no"
ls ˜/**/*.c.bkup > list_of_backups
```

The Z shell also allows post-glob modifiers, so `ls *(.)` lists only plain files and `ls *(/)` lists only directories. This gets Byzantine, so try `ls *(<tab>` to get the full list of options before typing in that last `)`.

Oh, and remember from h (p **??**)ow bash only gives you arithmetic expansion on integers, so `$((3/2))` is always one? Zsh and Ksh (and I dunno which others) are C-like in giving you a real (more than integer) answer if you cast the numerator or denominator to float:

```
#works for zsh, syntax error for bash:
echo $((3/2))
echo $((3/2.))

 #repeating the line-count example from Tip 33:
Files=`ls *.c |wc -l`
Lines=`grep '[)};]' *.c | wc -l`
 #Cast to floating point by adding 0.0
echo lines/file = $(($Lines/($Files+0.0)))
```

[21] `http://grml.org/zsh/zsh-lovers.html`
[22] `http://www.bash2zsh.com/zsh_refcard/refcard.pdf`

Now the shell-as-desk calculator is usable again.

Spaces in file names can break things in bash, because spaces separate list elements:

```
echo t1 > "test_file_1"
echo t2 > "test file 2"

#This fails in bash, is OK in Zsh
for f in test* ; do cat $f; done
```

The Z shell has array variables (which you can define using parens) that don't rely on spaces as delimiters, so the above isn't a problem:

```
#Having made the files above, run this in zsh:
for f in test* ; do cat $f; done

#equivalent to:
files=(test*)
for f in $files ; do cat $f; done
```

OK, enough about the Z shell, which is the one that is currently working well for me. There are many more to be had, and the odds are good that there's a shell that will work better for you than the one that shipped with your box. Wikipedia[23] has a shell comparison chart. Perl fans, maybe try the Perl shell[24].

If you decide to switch, there are two ways to do it: you can use `chsh` to make the change official in the login system [`/etc/passwd` gets modified], or if that's somehow a problem, you can add `exec -l /usr/bin/zsh` (or whatever shell you like) as the last line of your `.bashrc`, so bash will replace itself with your preferred shell every time it starts.

8.48 Tip 37: Rename things with pointers

14 December 2011

level: just far enough to be confused by pointers
purpose: distinguish between aliasing and memory management issues

OK, where were we? At the outset, I went over some of the o (p **??**)f y (p **??**)our C programs, in case you're used to having an interpreter doing all your dirty work. I went over the subset you need to make strings tolerable (Tips , (p **??**) , (p **??**) , (p **??**) and) (p **??**). I showed you some tricks with , (p **??**) with , (p **??**) and how far you can get . (p **??**) Underlying that was that you could do so while avoiding the dreaded `malloc` and all the associated memory management. I covered ((p **??**)there's a fourth type to come). I haven't even mentioned structs yet, though my number one favorite tip is about them. That'll come next.

[23]`http://en.wikipedia.org/wiki/Comparison_of_computer_shells`
[24]`http://www.focusresearch.com/gregor/sw/psh/`

But for now, let me pick up on that thread of segregating `malloc` and memory management issues to their proper place.

When I tell my computer *set* `A` *to* `B`, I could mean one of two things:

1. Copy the value of `B` into `A`. When I do `A++`, then `B` doesn't change.

2. Let `A` be an alias for `B`. When I do `A++`, then `B` also gets incremented.

The first conceptual tip is in no way C specific: every time your code says *set* `A` *to* `B`, you need to know whether you are making a copy or an alias.

For C, you are always making a copy, but if you are copying the address of the data you care about, a copy of the pointer is a new alias for the data. That's a fine implementation of aliasing. It doesn't get awkward until you start aliasing the the location of the data, which is the start down the chain of aliasing aliases.

Other languages have different customs: LISP family languages lean heavily on aliasing and have `set` commands to copy; Python generally copies scalars but aliases lists (unless you use `copy` or `deepcopy`). Again, knowing which to expect will clear up a whole lot of bugs all at once.

[By the way, you'll now and then meet a language that does not provide any mechanism at all for one of aliasing or copying. I hate to be negative, but such languages are braindead. Do not use them for serious work. I don't care if the language is well-funded and there are conferences about it; this is an absolutely basic requirement.]

We often wind up with structures within structures within structures. Let me use Apophenia as an example, and allocate an `apop_data` set via

```
apop_data  *adata = apop_data_alloc(1,1);
```

This is an `apop_data` set that has one element, which you may sometimes need; e.g., model parameters have to be of type `apop_data`, so if your model has one parameter, this is what you've gotta have. But this declaration wraps a `gsl_matrix`. If you're mostly working with the matrix, then name it:

```
apop_data  *adata = apop_data_alloc(1,1);
gsl_matrix  *matt = adata->matrix;

//Now matrix operations are clearer:
gsl_matrix *inv = apop_matrix inverse(matt);
```

[Inverting a 1-D matrix may seem like overkill, but in a larger routine, it sure beats writing a bunch of special cases that depend on the size of the input matrix.]

In fact, the matrix is really just an array of `double`s, so the one element in that 1×1 matrix may also merit an alias:

```
apop_data  *adata = apop_data_alloc(1,1);
double *thedata = adata->matrix->data;

//Use this like any other scalar:
*thedata = 1.2;
```

Getting slices of matrices and vectors also work by generating aliases for the data.

So you've got a lot of ways of looking at your data, depending on whatever is conveninent and has meaning for you. The alias makes clear to human readers what your focus is and how you are thinking about the data.

And writing `thedata` sure is more readable than putting `adata->matrix->data` everywhere.

Oh, look: we've gotten through another tip on pointers without mentioning `malloc`—and that's really the point of this tip. The concept of manually-allocated memory and the concept of an alias are distinct, and you can readily use aliases without manual allocation.

If you manually allocate a block, you're going to feel pretty stupid if you don't have a means of referring to it, so you can't have manual allocation without a pointer aliasing the location. That's why the old school textbooks introduce pointers and manual memory management at the same time. But so what—point to whatever you want whenever you want to make something more readable.

To do:
I gave you that advice above that every time you have a line that says *set* `A` *to* `B`, you need to know whether you are asking for an alias or a copy. Grab some code you have on hand (in whatever language) and go through line by line and ask yourself which is which. Were there cases where you could sensibly replace a copying with an alias?

8.49 Tip 38: Use Valgrind to check for errors

16 December 2011

level: once you're used to the debugger
purpose: find allegedly impossible-to-find bugs

Fail fast. If something goes wrong, then you want the program to start banging pots and pans the moment it happens.

Here's how I usually start my little lecture to people about debugging technique: *you need to find the first point in the program where something looks wrong.* Good code and a good system will find that point for you.

C gets mixed scores on this. The compiler runs loads of consistency checks, and thus finds many errors before they even occur. If you are dyslexic or otherwise commit frequent spelling errors, a language that requires declared variables is essential; an implicit-declaration language will take a typo like `conut=15` and generate a new variable that has nothing to do with the `count` you meant to set.

On the other hand, C will let you allocate to the tenth element of a nine-element array, and then trundle along for a long time before you find out that there's garbage in element ten. [Some scripting languages will just auto-reallocate the array to accommodate your error, which is an enthusiastic failure to fail fast.]

Those memory mismanagement issues are a hassle, and so there are tools to confront them. Within these, Valgrind is a big winner. Get a copy via your package manager.

Valgrind runs a sort of virtual machine that keeps better tabs of what is allocated where than the real machine does, so it knows when you hit the tenth element in an array of nine items.

Once you have a program compiled (with debugging symbols included via GCC's `-g` flag, of course), run

```
valgrind your_program
```

If you have an error, Valgrind will give you two backtraces that look a lot like the backtraces your debugger gives you. The first is where the misuse was first detected, and the second is Valgrind's best guess as to where the memory you meant to write to would have been allocated. The errors are often subtle, but having the exact line to focus on goes a long way toward finding the bug. Valgrind is under active development—programmers like nothing better than writing programming tools—so I'm amused at how much more informative the reports have gotten over time, and only expect better in the future.

You can also start the debugger at the first error, by running via

```
valgrind --db-attach=yes your_program
```

With this sort of startup, you'll get a line asking if you want to run the debugger on every detected error, and then you can check the value of the implicated variables as usual. At which point we're back to having a program that fails on the first line where a problem is detected.

Valgrind also does memory leaks, via

```
valgrind --leak-check=full your_program
```

This is slower, so you might not want to run it every time, but maybe it won't make any noticeable difference for your situation. At the end, you'll have a backtrace for every leak.

I take any memory leak under maybe 100KB as ignorable noise (unless I expect that the code could someday be re-run in the center of a loop). We're not working on computers with 64kb of memory, so don't stress about every line.

8.50 Tip 39: Know the constraints of C structs

18 December 2011

level: beginner in C, versed in other languages
purpose: add elements to your structures

Adding elements to your structures is, it turns out, a really hard problem, with an abundance of different solutions, none of which are all that pleasing. This entry isn't too much of a direct tip, to tell you the truth, but is about the means of thinking about all this.

We'll start with the problem statement, which is a restatment of the t (p **??**)hat C really likes to think in base-plus-offset terms. If you have an array of ints declared via `int *aa`, then that's a specific location in memory, and you refer to `int[3]`, that is a specific location in memory `3 * sizeof(int)` past the point where `aa` is. Structured data works the same: if I declare a type and an instance of that type like this:

```
typedef struct {
    char *name;
    void *value;
} list_element_t;

list_element_t *LL;
```

then I know that `LL` is a point in memory, and `LL->value` is a point in memory just far enough past that to get past the name, i.e. `sizeof(char*)` further down from `LL`.

In terms of processing speed, this base-plus-offset setup is as fast as it gets, but it more-or-less requires knowing the offsets at compile-time. If you come in later in the game and somehow redefine `list_element_t` as not having that first `name` element, then the whole system breaks down. You could perhaps imagine extending the structure to have more elements after the fact, but now all your arrays of stucts are broken. Your structure is fixed at compile time, at which point the various offsets are measured and your source code converted to express offsets (`LL + sizeof(int)`) instead of the names we recognize (`LL->value`).

Pro: we get lots of compile-time checks, and what is probably the fastest way possible to get to your data. Con: the structrue is fixed at compile time.

There are many ways to get around the fixed structures.

C++, Java, &c Develop a syntax for producing a new type that is an instance of the type you want to extend, but which inherits the old type's elements. Pros: you can still get base-plus-offset speed, and compile-time checking; once the child structure is set up, you can use it as you would the parent. Cons: so much paperwork, so many added keywords to support the structure (where C has `struct` and its , (p **??**) Java has `implements`, `extends`, `final`, `instanceof`, `class`, `this`, `interface`, `private`, `public`, `protected`).

Perl, Python, &c A structure in any of these languages is really a list of named elements—the `struct` above would be a fine prototype for a rudimentary implementation. When you need an item, the system would traverse the list, searching for the item name the programmer gave. Pros: fully extensible by just adding a new named item. Cons: ¿when you refer to a name not in the list, is it a typo or are you adding a new element?; you can improve the name search via various tricks, but you're a long ways from the speed of a single base-plus-offset step; you'll need more C++-like syntax go guarantee that a list-as-structure has certain elements.

C All the machinery you have in C for extending a structure is to wrap it in another structure. Say that the above type is already packaged and can not be changed, but we'd like to add a type marker. Then we'll need a new structure:

```
typedef struct {
    list_element_t elmt;
    int typemarker;
} list_element_w_type_t;
```

Pros: this is so stupid easy, and you still get the speed bonus. Cons: Now, every time you want to refer to the name of the element, you need `your_typed_list->elmt->name` instead of what you'd get via a C++/Java-like extension: `your_typed_list->name`. Add a few layers to this and it starts to get annoying. You still don't get to add to the list during run-time; the only way to do this is via a list like the one the struct here describes.

If you come from one of the other traditions, don't expect the C struct to do what a list or hash does in Ruby or Perl—but you can get such a struct via the Glib library. Instead, revel in the simplicity of building structures using other structures as elements. Several of the tips to follow will help you with using nested structures without annoyance. You already saw how c (p **??**)an help here.

Bibliography

Nicholas A Christakis and James H Fowler. The spread of obesity in a large social network over 32 years. *N Engl J Med*, 357(4):370–379, 2007.

Cohen-Cole E and J M Fletcher. Is obesity contagious? social networks vs. environmental factors in the obesity epidemic. *Journal of Health Economics*, 27(5):1382–7, September 2008.

Rachel Karniol. A conceptual analysis of immanent justice responses in children. *Child Development*, 51(1):118–130, 1980.

Jan Kmenta. *Elements of Econometrics*. Macmillan Publishing Company, 2nd edition, 1986.

Caroline Mansfield, Suellen Hopfer, and Theresa M Marteau. Termination rates after prenatal diagnosis of Down syndrome, spina bifida, anencephaly, and Turner and Klinefelter syndromes: A systematic literature review. *Prenatal diagnosis*, 19(9): 808–812.

Donald A Norman. *The Design of Everyday Things*. Basic Books, 1988.

Judea Perl. *Causality*. Cambridge University Press, March 2000.