by daniel chudnov
Information Technology Specialist
Office of Strategic Initiatives
Library of Congress

Getting Over the Hump

I'M OBSESSED WITH THE IDEA OF LIBRARIANS BECOMING BETTER CODERS.

I'm obsessed with the idea of librarians becoming better coders. It's not something every librarian needs to do, but if even a small fraction of our colleagues can do a little more "under the hood" of the systems we depend on to deliver services to our communities, we'd be in a lot better shape. Since my first week of library school through to my current job, friends and colleagues from libraries all over have shared their frustrations about how difficult it can be to build programming skills beyond superficial "script tweaking" and the like. They'd like to be able to do more, but they're always running into stumbling blocks, whether it's a lack of mentoring, insufficient organizational support, or the difficult nature of some technical challenges. Many describe how there's a big hump in the learning curve and how just when they think they've made progress toward better understanding and mastery of technical skills and concepts, they slip back down again for one reason or another.

It's frustrating in both directions. Somehow between library school and now, I got over that hump. It would be great if more of us could do the same. Our profession, and the services we provide, would be stronger for it, and the work itself could become more rewarding for those developing and building new skills to develop and build new services. Selfishly, I'd just like to increase the pool of people I can connect with at peer institutions about this stuff.

I've tried my hand at a few ways of helping colleagues get over the hump. I'm not very good at it yet, but I will keep trying. So far I've written columns here, prepared an instructional video, and given a workshop at a conference. Through these, on occasion, one method or another reaches somebody and that light bulb goes off. You can tell because you hear something like "Oh! That's how that works!" when they connect one concept with another. Then they're off and running.

But that doesn't happen very often.

Usually, there's a lot more of "This doesn't seem to be working" and "But wait, why does this do that again" and "I still don't get it" and, worst of all, "I give up." The last of these is a variation on a theme I've also heard throughout my career, that "that techie stuff is too hard for me." If you really believe that this stuff is too hard for you, you can stop reading right now. I don't think I can help you. But if you don't believe it's too hard for you, if you want to get better, if you're willing to keep trying until you "get it," whatever "it" is, I've included here a number of tips to help convince you that you can and will learn what you need to know to make progress, to complete projects, and, eventually, to feel like you really know what you're doing.

Stop Fretting and Open the Editor Already

Sometimes I think about a problem for so long I forget to actually get started working on it. I worry about how this approach won't

work because it didn't work for me on that other project, or about how if I have to change things later, I'll have to redo it all. These are excuses for not getting started. It's not a cliché to remember what Lao Tzu wrote: "A journey of a thousand miles begins with a single step." Open up your text editor. Start nibbling at a piece of the problem. Create a way to make some simple, definite progress. There, your problem just got a little smaller. Now we're getting somewhere!

Build Something You Need

There's a common phrase among hackers that summarizes the different motivation you have when you're building something for yourself versus building something for someone else. It's "scratching your own itch." If you work as a programmer, you will spend a lot of your time building things for other people. But when you're learning how to be a programmer, you're your own best audience. There are a few reasons for this. First, if you're the primary user, you will minimize the time between changes you make in code and getting feedback from your users. Tweak it, run it, see if it works the way you want, tweak it again. It's a simple cycle. Second, you need to develop experience in going from thinking "There should be a tool for that" to thinking "I can build a tool for that" to thinking "I can build a really good tool for that." The more you use it, the more ways you will find to make it better. Finally, this cycle of improvements will take you from being able to make something work to figuring out how to make it work better. You'll reach a point where you see that some of the assumptions you made about your original need and what you ended up using it for were wrong. When you face that, you'll make new decisions to change your code to make it fit better. This might be the most important moment: gaining insight into understanding user needs and finding a design that fits. The next time you go to build a tool for yourself or someone else, all that experience should pay off.

Don't Imagine There's a Perfect Solution

I used to think that because I was reading about some newly hyped standard or framework that if I didn't use it, I was missing something. I also used to think that I should try to use the languages and supporting tools that everybody else seemed to be using. Neither of these turned out to be true. When I started as a programmer, Perl and Java were the most popular languages and Emacs and some proprietary IDE I can't recall were the most popular development tools. I never became very good at any of those. Now my main weapons of choice are Python and Vim, and though they don't do everything, they do most of what I need. Similarly, just because somebody else built a system using an Oracle or MySQL back end doesn't mean *you* have to use an Oracle or MySQL back end. Sometimes simple files on disk can solve a problem better than a database. XML fits a lot of problems, but anywhere XML can be used, there are other options too. We all use the web all the time, but not everything needs a web interface. And like all of these choices about how and what, any 10 programmers can usually come up with 16 solutions to any problem, all of which could plausibly work. Coding is equal parts engineering, craftsmanship, and artistry, and if your strength is one of those but not the others, you'll write a program that is different from a program written by those with the other strengths. Does it work as intended? Do you understand how it works? If you can answer yes, the rest is details.

Ask for Help

In March I wrote about the microlevel issues involved in getting stuck in the middle of a code problem and offered the same suggestion: Ask for help. The same advice applies to the more macrolevel question "How to I improve overall?" Find somebody with more experience than you whom you trust and ask them to review your code with you. If you think something you've written could be made better, find an appropriate forum and share a snippet with a focused question or two summarizing your concerns. If there's a deeper concept or fundamental theory that you don't understand, find a how-to book, a textbook, or a local class you can attend. Most concepts in computer science and programming have theoretical underpinnings that have proven to be solid over at least a few decades now. There probably are resources available and experienced practitioners somewhere near you. Don't hesitate to look for them.

Expect to Make Changes

Software changes. When you think it's working, you'll find a case where it doesn't. When you think it's broken, you'll find that you're just off slightly from the right solution. When you think it's too slow, you'll realize that it doesn't need to be any faster than it already is. Don't worry about getting it all right up front. It's easy to change. After you open up your editor and start building small pieces of solutions, string these pieces together so you can see how something flows from one end to the other, even if it doesn't solve the whole picture yet. If it looks like you're on the right track, keep moving piece by piece, but don't think you'll ever get everything to be perfect, right, or reliable. You'll always find something you can do to improve things later. In the meantime, you can prepare for eventual change by using version control, by using consistent naming conventions and code styles, and by commenting on your code. Think about your code not as something the computer has to read

immediately but as something you're going to have to read a year from now. If you don't look at it for a year, will you remember how it works, what that variable means, or why you structured things that odd way? It's best to leave a comment, rename the variable to something meaningful, or clean up the structure to explain it to your future self now.

You'll Never Be 'Done'

Expecting to make changes is a habit you should encourage. Everything in the last paragraph doesn't just apply during the days, weeks, or months when you're most intensively working on a software project. It really does happen that 5 or 8 years down the road when you might have to reach back into something you haven't used in all that time. Or your users might have used something you wrote so much more than you expected that your assumptions about something as simple as generating unique identifiers turned out to be wrong, only after 8 years of it appearing to be right. It's the blessing and the curse of software in a nutshell. There's always something more to do, some other way to make it better. I think that's the trick to getting over the hump as a new programmer too. There's always another hump after this one. Get over the one in front of you first, and the confidence and experience you gain will help you immensely the next time out. ■

Daniel Chudnov is a librarian working as an information technology specialist in the Office of Strategic Initiatives at the Library of Congress and is a frequent speaker, writer, and consultant in the area of software and service innovation in libraries. Previously, he worked on the DSpace project at MIT Libraries and the jake metadata service at the Yale Medical Library. His email address is daniel.chudnov@gmail.com, and his blog is at http://onebiglibrary.net.