The ABC's and 3P's of Learning to Program

Terry Sergeant

Disclaimer

This document pertains to "learning to program" (in the context of a college course) and not to "programming". Some advice pertains to both, but some is excellent advice for learning to program and not particularly good advice for someone who already knows how to program and is trying to accomplish a task. There exist human beings who have learned to program who did not follow every suggestion in this document. However, these suggestions will work for just about anyone.

1 The ABC's: Thinking Right

Successfully learning to program a computer begins with a correct mental orientation.

Attitude — Programming is fun *and* hard.

You need to adopt the mindset of a *prexathlecon*:

problem-solver understand how something works and then fix it

explorer take time to look around even when the destination is unknown

athlete be dedicated to training and practice

conqueror start with intention of winning and never give up

Behavior — Programming is a skill.

For any skill you can observe this phenomenon:

1. Practice the skill.
2. Rest from practice. (Do something else or sleep).
3. Try the skill again ... and find you have improved.

The perplexing thing about this process is that the only thing to happen between step 1 and step 3 was rest! I call this a struggle-rest cycle. You need to think in terms of maximizing these struggle-rest cycles. Spending 4 hours practicing in one day and then taking three days off is not as beneficial as practicing an hour a day for four days.

Cognition — Programming requires understanding.

A programming language is an abstraction of commands that a computer performs. Learning to program requires *building an accurate mental model* that explains what will result from a particular statement. Such an understanding implies that you also know what each individual programming statement accomplishes according to your mental model.

2 The 3P's: Doing Right

Once you are thinking like a *prexathlecon* and are committed to daily practice and to building an accurate mental model it is time to focus on effective learning behaviors:

2.1 Patterns

There are two patterns you should use:

practice-repeat-memorize When you are learning a programming concept you try it out. Once you have code that works don't think of yourself as being finished. (Would a basketball player stop practicing free throws once they make one?) Instead, try it again (from memory). Type the code multiple times and each time try to recall the correct syntax without looking at anything you've done before. Having syntax memorized is essential as you progress to increasingly complex topics.

code-predict-execute-adjust Every time you try out programming statements begin by predicting what will happen when the program runs. Don't ever run a program without first making a prediction! Once you execute the program compare the actual result with your prediction. If they match that's great. If they don't match that's good too because you have credible feedback that your mental model was inaccurate and needs adjustment. Continue the code-predict-execute-adjust pattern until your mental model agrees with reality.

2.2 Practices

- *Just Start* Don't feel like you have to know every detail of every step needed to solve a problem before starting. If you are trying to make your way through a foggy landscape you won't be able to see your destination, but if you never move you'll never be able to see anything more than you see right now. Start moving in some direction ... and if you approach a cliff you'll learn where the cliff is and that you need to try a different direction.

- *Know the Type* Anytime you have a variable (or a value returned by a function) you should know it's type!! You should practice saying the type out loud. Once the type has been identified you will immediately inherit all knowledge about values of the specified type ... which will determine what makes sense as you move forward.

- *Draw The Picture* Ultimately programming statements use values that have been stored in memory. Representing the computer's memory with accurate diagrams can explain clearly what may otherwise be confusing behavior. It does require a bit of effort to learn to draw pictures accurately but doing so will be an important step in your effort to build an accurate mental model. Remember: "No Picture ... No Worky".

- *Take Baby Steps* Don't type beyond your accurate/verified mental model! If you can't accurately predict what a line of code will do then you need to apply the code-predict-execute-adjust pattern to that one line of code until you know what it is doing.

- *Ask the Lobster* I've had students come to my office asking for help and as they walk me through their code they answer their own question! So, instead of asking a person you can also ask an inanimate object (we use a lobster). For this technique to work you need to explain your code (out loud) step-by-step and believe the lobster can help.

- *Version Control* Version control is your friend—not busy work! In a decent sized program you might write for a course there are typically four or five pretty clearly delineated steps that you can get working and test individually. As you complete each step you should take time to commit (and push) your changes. Don't wait until the entire program works to do a commit.

2.3 Pitfalls

Here are some behaviors to avoid:

- *Be in a hurry.* When learning to program the goal is not to "finish" ... *the goal is to learn how to program.* Besides, computers can sense hurry like an attack dog can smell fear!

- *Copy and paste.* When learning to program the goal is not to "save time" ... *the goal is to learn how to program.* By typing code from memory rather than copy-paste you are participating in the practice-repeat-memorize pattern. (The same concept applies to copying programs or parts of programs in other ways as well.)

- *Get help.* Remember, this is a behavior to avoid! When learning to program the goal is not to "finish" ... *the goal is to learn how to program.* Don't use Google. Don't ask a friend. Don't email your instructor. Instead apply the patterns and practices listed above to figure it out on your own. Normally, if you're stuck on a single issue for a while it is because you have taken too big of a step or that your mental model is lagging behind the topics you are undertaking. If you *do* need to get help and you've already asked the lobster then try to get the smallest amount of help needed to get you moving forward. BOTTOM LINE: It is okay to be stuck for a while. The act of trying to get unstuck will modify your mental model and will improve your problem-solving skills.

3 A Typical Week in a Programming Course

Here's how these right ways of thinking and right ways of doing might play out in the context of programming course.

Day 1 In class the instructor introduces how to use various methods in Java's String class. You understood the examples as the instructor presented them but you know that seeing examples and being able to use those commands from memory are not the same thing. So, you create a small program (from scratch) for the purpose of playing with those commands. You focus on one command at a time trying to remember the syntax from memory. (This utilizes the practice-repeat-memorize pattern and the code-execute-predict-adjust pattern.) You check your notes when you get stuck. You draw a picture of a string (including its indexes) and use it to explain/predict how the various string commands will function. These steps take 30-45 minutes. It's time to do something else for a while. Later that day you come back to your program and, having learned the commands, are ready to start on the homework assignment. So, you read the assignment and plan how you will structure the assignment (methods, parameters, variables, classes) and identify what new commands will be needed to solve it. Write (on paper) your overall design (this takes another 30-45 mintes). Time spent: 1.5 hours.

Day 2 In reading the assignment the day before you realized that some parts of the assignment cannot be finished until more is discussed in lecture. There are, however, quite a few parts of the assignment you can do, so you get started on those parts. Review your design and then start writing code. Use baby steps and employ relevant practices as well as the code-execute-predict-adjust pattern. At each step along the way take time to commit and push your work. Let's say you are working on a function and you can't get it to behave the way you want. Follow this checklist:

1. Mentally step through your program and as you do so draw a picture of the list/array/objects your program creates. If you can't find a statement that creates those structures then don't draw them.
2. When walking through code say (out loud) the type of every variable.
3. If, as you walk through the code, you encounter a statement you don't understand then stop and write a separate program for the purpose of testing that single command until you understand it. Then continue walking through your code.
4. Put print statements in your program (or use the debugging features of your IDE to track variables) and predict the output and reconcile differences. For example if you get Java's dreaded "null pointer exeception" when you run your program it will (mercifully) give you a line number. Go to the specified line number and identify which variable was null. Identifying the null variable explains why you got the null pointer exception, but the real question is this: "Why was that variable null?" Presumably you were expecting the variable to point to an object and instead it is null. So, trace backwards through your program to see where you set that variable to start with.
5. If you're still stuck ask the lobster. As you walk through the section of code that isn't working explain out loud to the lobster what each statement is doing and see what lights come on.
6. If the lobster doesn't help then take a 30 minute break and come back and walk through the code again. If you still can't figure out the issue then it may be time to invite some human intervention.

Let's suppose your day ends in the "worst case scenario" that you got stuck and couldn't get past the issue. So, you will need to get outside help. Remember, as you finish each section of your code take time to document it and commit/push your changes. Total time spent: 2 hours.

Day 3 After a good night's rest take a moment to revisit the code that isn't working. If you don't have new insights ask for help (from instructor or fellow student). Once you get over the hump and reach the next milestone be sure to commit/push your changes. Today was another class day and more commands/concepts were introduced. You take time to play around with the relevant commands that are needed to complete the homework. Take time to write another (from scratch) program for the purpose of playing with/learning the new commands. Then take a moment to revisit your initial design which can be enhanced now that you know more. Time spent: 1.5 hours.

Day 4 Continue working on the homework using the techniques described above. Let's say you get stuck again and re-apply the checklist (draw, print, lobster, help). Time spent: 1.5 hours.

Day 5 Revisit the code that wasn't working ... if no new insights then get help. Once you get over the hump be sure to commit/push. If new commands are introduced in class then play around with them. Today you will likely finish the homework. Time spent: 1.5 hours.

Day 6 Take time to re-read the assignment and make sure your program meets all requirements. Commit any tweaks that are needed. Time spent: 1 hour.

Day 7 Rest.

In this scenario you will have spent about 9 hours outside of class. The truth is that most people who invest 1 to 2 hours a day in an undergraduate programming course will have good results. So, you might be able to spent closer to 1 hour per day (outside of class) and have success. As discussed earlier, spending 1 hour a day is not the same as spending 6 hours one afternoon. Also, it matters *how* you spend your time. If you spend it actively learning concepts presented in class rather than looking online at other people's solutions to your homework assignment then you will develop good skills. If instead you focus on "finishing the homework" you will find yourself on the wrong side of a snowball on its way down a hill.

The bottom line is that programming is a valuable skill precisely because it leverages very powerful hardware and is a difficult skill to master. Lots of people have started down the path of learning to code because they heard about the good jobs and high pay but they gave up. If you learn the ABC's and apply the 3P's presented here you will join the ranks of those who can say with confidence: "Yes, I know how to program a computer!"