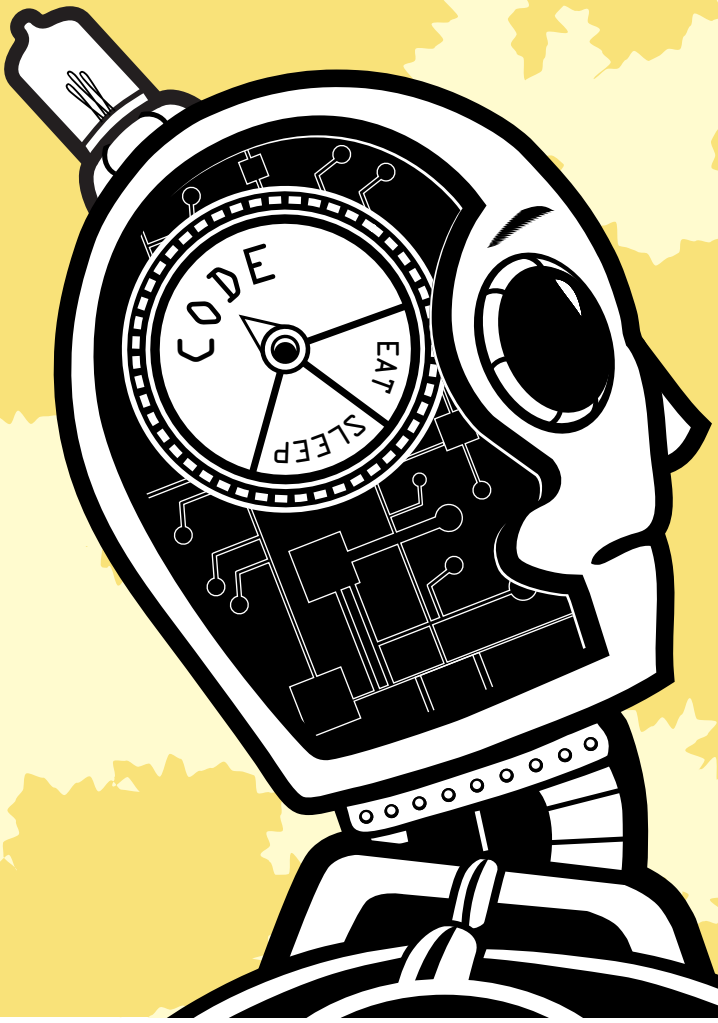
THINK LIKE A PROGRAMMER

AN INTRODUCTION TO
CREATIVE PROBLEM SOLVING

V. ANTON SPRAUL



no starch press

CONTENTS IN DETAIL

INTRODUCTION

Do you struggle to write programs, even though you think you understand programming languages? Are you able to read through a chapter in a programming book, nodding your head the whole way, but unable to apply what you've read to your own programs? Are you able to comprehend a program example you've read online, even to the point where you could explain to someone else what each line of the code is doing, yet you feel your brain seize up when faced with a programming task and a blank screen in your text editor?

You're not alone. I have taught programming for over 15 years, and most of my students would have fit this description at some point in their instruction. We will call the missing skill *problem solving*, the ability to take a given problem description and write an original program to solve it. Not all programming requires extensive problem solving. If you're just making minor modifications to an existing program, debugging, or adding testing code, the

programming may be so mechanical in nature that your creativity is never tested. But all programs require problem solving at some point, and all good programmers can solve problems.

Problem solving is hard. It's true that a few people make it look easy—the "naturals," the programming world's equivalent of a gifted athlete, like Michael Jordan. For these select few, high-level ideas are effortlessly translated into source code. To make a Java metaphor, it's as if their brains execute Java natively, while the rest of us have to run a virtual machine, interpreting as we go.

Not being a natural isn't fatal to becoming a programmer—if it were, the world would have few programmers. Yet I've seen too many worthy learners struggle too long in frustration. In the worst cases, they give up programming entirely, convinced that they can never be programmers, that the only good programmers are those born with an innate gift.

Why is learning to solve programming problems so hard?

In part, it's because problem solving is a different activity from learning programming syntax and therefore uses a different set of mental "muscles." Learning programming syntax, reading programs, memorizing elements of an application programming interface—these are mostly analytical "left brain" activities. Writing an original program using previously learned tools and skills is a creative "right brain" activity.

Suppose you need to remove a branch that has fallen into one of the rain gutters on your house, but your ladder isn't quite long enough for you to reach the branch. You head into your garage and look for something, or a combination of things, that will enable you to remove the branch from the gutter. Is there some way to extend the ladder? Is there something you can hold at the top of the ladder to grab or dislodge the branch? Maybe you could just get on the roof from another place and get the branch from above. That's problem solving, and it's a creative activity. Believe it or not, when you design an original program, your mental process is quite similar to that of the person figuring out how to remove the branch from the gutter and quite different from that of a person debugging an existing `for` loop.

Most programming books, though, focus their attention on syntax and semantics. Learning the syntax and semantics of a programming language is essential, but it's only the first step in learning how to program in that language. In essence, most programming books for beginners teach how to read a program, not how to write one. Books that do focus on writing are often effectively "cookbooks" in that they teach specific "recipes" for use in particular situations. Such books can be quite valuable as time savers, but not as a path toward learning to write original code. Think about cookbooks in the original sense. Although great cooks own cookbooks, no one who relies upon cookbooks can be a great cook. A great cook understands ingredients, preparation methods, and cooking methods and knows how they can be combined to make great meals. All a great cook needs to produce a tasty meal is a fully stocked kitchen. In the same way, a great programmer understands language syntax, application frameworks, algorithms, and software engineering principles and knows how they can be combined to make great programs. Give a great programmer a list of specifications, turn him loose with a fully stocked programming environment, and great things will happen.

In general, current programming education doesn't offer much guidance in the area of problem solving. Instead, it's assumed that if programmers are given access to all of the tools of programming and requested to write enough programs, eventually they will learn to write such programs and write them well. There is truth in this, but "eventually" can be a long time. The journey from initiation to enlightenment can be filled with frustration, and too many who start the journey never reach the destination.

Instead of learning by trial and error, you can learn problem solving in a systematic way. That's what this book is all about. You can learn techniques to organize your thoughts, procedures to discover solutions, and strategies to apply to certain classes of problems. By studying these approaches, you can unlock your creativity. Make no mistake: Programming, and especially problem solving, is a creative activity. Creativity is mysterious, and no one can say exactly how the creative mind functions. Yet, if we can learn music composition, take advice on creative writing, or be shown how to paint, then we can learn to creatively solve programming problems, too. This book isn't going to tell you precisely what to do; it's going to help you develop your latent problem-solving abilities so that you will know what you should do. This book is about helping you become the programmer you are meant to be.

My goal is for you and every other reader of this book to learn to systematically approach every programming task and to have the confidence that you will ultimately solve a given problem. When you complete this book, I want you to *think like a programmer* and to *believe that you are a programmer.*

About This Book

Having explained the necessity of this book, I need to make a few comments about what this book is and what it is not.

Prerequisites

This book assumes you are already familiar with the basic syntax and semantics of the C++ language and that you have begun writing programs. Most of the chapters will expect you to know specific C++ fundamentals; these chapters will begin with a review of those fundamentals. If you are still absorbing language basics, don't worry. There are plenty of great books on C++ syntax, and you can learn problem solving in parallel to learning syntax. Just make sure you have studied the relevant syntax before attempting to tackle a chapter's problems.

Chosen Topics

The topics covered in this book represent areas in which I have most often seen new programmers struggle. They also present a broad cross-section of different areas in early and intermediate programming.

I should emphasize, however, that this is not a "cookbook" of algorithms or patterns for solving specific problems. Although later chapters discuss how to employ well-known algorithms or patterns, you should not use this

1

STRATEGIES FOR PROBLEM SOLVING

This book is about problem solving, but what is problem solving, exactly? When people use the term in ordinary conversation, they often mean something very different from what we mean here. If your 1997 Honda Civic has blue smoke coming from the tailpipe, is idling roughly, and has lost fuel efficiency, this is a problem that can be solved with automotive knowledge, diagnosis, replacement equipment, and common shop tools. If you tell your friends about your problem, though, one of them might say, "Hey, you should trade that old Honda in for something new. Problem solved." But your friend's suggestion wouldn't really be a *solution* to the problem—it would be a way to *avoid* the problem.

Problems include constraints, unbreakable rules about the problem or the way in which the problem must be solved. With the broken-down Civic, one of the constraints is that you want to fix the current car, not purchase a new car. The constraints might also include the overall cost of the repairs, how long the repair will take, or a requirement that no new tools can be purchased just for this repair.

When solving a problem with a program, you also have constraints. Common constraints include the programming language, platform (does it run on a PC, or an iPhone, or what?), performance (a game program may require graphics to be updated at least 30 times a second, a business application might have a maximum time response to user input), or memory footprint. Sometimes the constraint involves what other code you can reference: Maybe the program can't include certain open-source code, or maybe the opposite—maybe it can use only open source.

For programmers, then, we can define *problem solving* as writing an original program that performs a particular set of tasks and meets all stated constraints.

Beginning programmers are often so eager to accomplish the first part of that definition—writing a program to perform a certain task—that they fail on the second part of the definition, meeting the stated constraints. I call a program like that, one that appears to produce correct results but breaks one or more of the stated rules, a *Kobayashi Maru*. If that name is unfamiliar to you, it means you are insufficiently familiar with one of the touchstones of geek culture, the film *Star Trek II: The Wrath of Khan*. The film contains a subplot about an exercise for aspiring officers at Starfleet Academy. The cadets are put aboard a simulated starship bridge and made to act as captain on a mission that involves an impossible choice. Innocent people will die on a wounded ship, the *Kobayashi Maru*, but to reach them requires starting a battle with the Klingons, a battle that can only end in the destruction of the captain's ship. The exercise is intended to test a cadet's courage under fire. There's no way to win, and all choices lead to bad outcomes. Toward the end of the film, we discover that Captain Kirk modified the simulation to make it actually winnable. Kirk was clever, but he did not solve the dilemma of the *Kobayashi Maru*; he avoided it.

Fortunately, the problems you will face as a programmer are solvable, but many programmers still resort to Kirk's approach. In some cases, they do so accidentally. ("Oh, shoot! My solution only works if there are a hundred data items or fewer. It's supposed to work for an unlimited data set. I'll have to rethink this.") In other cases, the removal of constraints is deliberate, a ploy to meet a deadline imposed by a boss or an instructor. In still other cases, the programmer just doesn't know how to meet all of the constraints. In the worst cases I have seen, the programming student has paid someone else to write the program. Regardless of the motivations, we must always be diligent to avoid the Kobayashi Maru.

Classic Puzzles

As you progress through this book, you will notice that although the particulars of the source code change from one problem area to the next, certain patterns will emerge in the approaches we take. This is great news because this is what eventually allows us to confidently approach any problem, whether we have extensive experience in that problem area or not. Expert problem

Have you thought about it? Were you able to come up with a solution? There are two possible paths to an answer here. The first path is trial and error: attempting various Kratzz moves in a methodical way and backing up to previous steps when you reach an alarm state until you find a series of moves that succeeds.

The second path is realizing that the puzzle is a trick. If you haven't seen the trick yet, here it is: This is just the fox, goose, and corn problem in an elaborate disguise. Although the rules for the alarm are written generally, there are only so many combinations for this specific lock. With only three Kratzz, we just have to know which combinations of Kratzz in a receptor are acceptable. If we label the three Kratzz *top*, *middle*, and *bottom*, then the combinations that create alarms are "top and middle" and "middle and bottom." If we rename *top* as *fox*, *middle* as *goose*, and *bottom* as *corn*, then the troublesome combinations are the same as in the other problem, "fox and goose" and "goose and corn."

This problem is therefore solved in the same way as the fox, goose, and corn problem. We slide the middle Kratzz (goose) over to the left receptacle. Then, we slide the top (fox) to the left, holding the left alarm's suppressor as we put the top (fox) into place. Next, we start sliding the middle (goose) back to the right receptacle. Then, we slide the bottom (corn) to the left, and finally, we slide the middle (goose) to the left once again, opening the lock.

Lessons Learned

The chief lesson here is the importance of recognizing analogies. Here, we can see that the Quarrasi lock problem is analogous to the fox, goose, and corn problem. If we discover that analogy early enough, we can avoid most of the work of the problem by translating our solution from the first problem rather than creating a new solution. Most analogies in problem solving won't be so direct, but they will happen with increasing frequency.

If you had trouble seeing the connection between this problem and the fox, goose, and corn problem, that's because I deliberately included as much extraneous detail as possible. The story that sets up the Quarrasi problem is irrelevant, as are the names for all of the alien technology, which serve to heighten the sense of unfamiliarity. Furthermore, the odd/even mechanism of the alarm makes the problem seem more complicated than it is. If you look at the actual positioning of the Quinicrys, you can see that the top Kratzz and the bottom Kratzz are opposites, so they don't interact in the alarm system. The middle Kratzz, however, interacts with the other two.

Again, if you didn't see the analogy, don't worry. You'll start to recognize them more after you put yourself on alert for them.

General Problem-Solving Techniques

The examples we have discussed demonstrate many of the key techniques that are employed in problem solving. In the rest of this book, we'll look at specific programming problems and figure out ways to solve them, but first we need a general set of techniques and principles. Some problem areas

have specific techniques, as we'll see, but the rules below apply to almost any situation. If you make these a regular part of your problem-solving approach, you'll always have a method to attack a problem.

Always Have a Plan

This is perhaps the most important rule. You must always have a plan, rather than engaging in directionless activity.

By this point, you should understand that having a plan is always possible. It's true that if you haven't already solved the problem in your head, then you can't have a plan for implementing a solution in code. That will come later. Even at the beginning, though, you should have a plan for how you are going to find the solution.

To be fair, the plan may require alteration somewhere along the journey, or you may have to abandon your original plan and concoct another. Why, then, is this rule so important? General Dwight D. Eisenhower was famous for saying, "I have always found that plans are useless, but planning is indispensable." He meant that battles are so chaotic that it is impossible to predict everything that could happen and have a predetermined response for every outcome. In that sense, then, plans are useless on the battlefield (another military leader, the Prussian Helmuth von Moltke, famously said that "no plan survives first contact with the enemy"). But no army can succeed without planning and organization. Through planning, a general learns what his army's capabilities are, how the different parts of the army work together, and so on.

In the same way, you must always have a plan for solving a problem. It may not survive first contact with the enemy—it may be discarded as soon as you start to type code into your source editor—but you must have a plan.

Without a plan, you are simply hoping for a lucky break, the equivalent of the randomly typing monkey producing one of the plays of Shakespeare. Lucky breaks are uncommon, and those that occur may still require a plan. Many people have heard the story of the discovery of penicillin: A researcher named Alexander Fleming forgot to close a petri dish one night and in the morning found that mold had inhibited the growth of the bacteria in the dish. But Fleming was not sitting around waiting for a lucky break; he had been experimenting in a thorough and controlled way and thus recognized the importance of what he saw in the petri dish. (If I found mold growing on something I left out the night before, this would not result in an important contribution to science.)

Planning also allows you to set intermediate goals and achieve them. Without a plan, you have only one goal: solve the whole problem. Until you have solved the problem, you won't feel you have accomplished anything. As you have probably experienced, many programs don't do anything useful until they are close to completion. Therefore, working only toward the primary goal inevitably leads to frustration, as there is no positive reinforcement from your efforts until the end. If instead, you create a plan with a series of minor goals, even if some seem tangential to the main problem, you will make measurable progress toward a solution and feel that your time has

been spent usefully. At the end of each work session, you'll be able to check off items from your plan, gaining confidence that you will find a solution instead of growing increasingly frustrated.

Restate the Problem

As demonstrated especially by the fox, goose, and corn problem, restating a problem can produce valuable results. In some cases, a problem that looks very difficult may seem easy when stated in a different way or using different terms. Restating a problem is like circling the base of a hill that you must climb; before starting your climb, why not check out the hill from every angle to see whether there's an easier way up?

Restatement sometimes shows us the goal was not what we thought it was. I once read about a grandmother who was watching over her baby granddaughter while knitting. In order to get her knitting done, the grandmother put the baby next to her in a portable play pen, but the baby didn't like being in the pen and kept crying. The grandmother tried all sorts of toys to make the pen more fun for the baby, until she realized that keeping the baby in the pen was just a means to an end. The goal was for the grandmother to be able to knit in peace. The solution: Let the baby play happily on the carpet, while the grandmother knits inside the pen. Restatement can be a powerful technique, but many programmers will skip it because it doesn't directly involve writing code or even designing a solution. This is another reason why having a plan is essential. Without a plan, your only goal is to have working code, and restatement is taking time away from writing code. With a plan, you can put "formally restate the problem" as your first step; therefore, completing the restatement officially counts as progress.

Even if a restatement doesn't lead to any immediate insight, it can help in other ways. For example, if a problem has been assigned to you (by a supervisor or an instructor), you can take your restatement to the person who assigned the problem and confirm your understanding. Also, restating the problem may be a necessary prerequisite step to using other common techniques, like reducing or dividing the problem.

More broadly, restatement can transform whole problem areas. The technique I employ for recursive solutions, which I share in a later chapter, is a method to restate recursive problems so that I can treat them the same as iterative problems.

Divide the Problem

Finding a way to divide a problem into steps or phases can make the problem much easier. If you can divide a problem into two pieces, you might think that each piece would be half as difficult to solve as the original whole, but usually, it's even easier than that.

Here's an analogy that will be familiar if you have already seen common sorting algorithms. Suppose you have 100 files you need to place in a box in alphabetical order, and your basic alphabetizing method is effectively what we call an insertion sort: You take one of the files at random, put it in the box,

then put the next file in the box in the correct relationship to the first file, and then continue, always putting the new file in its correct position relative to the other files, so that at any given time, the files in the box are alphabetized. Suppose someone initially separates the files into 4 groups of roughly equal size, A–F, G–M, N–S, and T–Z, and tells you to alphabetize the 4 groups individually and then drop them one after the other into the box.

If each of the groups contained about 25 files, then one might think that alphabetizing 4 groups of 25 is about the same amount of work as alphabetizing a single group of 100. But it's actually far *less* work because the work involved in inserting a single file grows as the number of files already filed grows—you have to look at each file in the box to know where the new file should be placed. (If you doubt this, think of a more extreme version—compare the thought of ordering 50 groups of 2 files, which you could probably do in under a minute, with ordering a single group of 100 files.)

In the same way, dividing a problem can often lower the difficulty by an order of magnitude. Combining programming techniques is much trickier than using techniques alone. For example, a section of code that employs a series of `if` statements inside a `while` loop that is itself inside a `for` loop will be more difficult to write—and to read—than a section of code that employs all those same control statements sequentially.

We'll discuss specific ways to divide problems in the chapters that follow, but you should always be alert to the possibility. Remember that some problems, like our sliding tile puzzle, often hide their potential subdivision. Sometimes the way to find a problem's divisions is to reduce the problem, as we'll discuss shortly.

Start with What You Know

First-time novelists are often given the advice "write what you know." This doesn't mean that novelists should try only to craft works around incidents and people they have directly observed in their own lives; if this were the case, we could never have fantasy novels, historical fiction, or many other popular genres. But it means that the further away a writer gets from his or her own experience, the more difficult writing may be.

In the same way, when programming, you should try to start with what you already know how to do and work outward from there. Once you have divided the problem up into pieces, for example, go ahead and complete any pieces you already know how to code. Having a working partial solution may spark ideas about the rest of the problem. Also, as you may have noticed, a common theme in problem solving is making useful progress to build confidence that you will ultimately complete the task. By starting with what you know, you build confidence and momentum toward the goal.

The "start with what you know" maxim also applies in cases where you haven't divided the problem. Imagine someone made a complete list of every skill in programming: writing a C++ class, sorting a list of numbers, finding the largest value in a linked list, and so on. At every point in your development as a programmer, there will be many skills on this list that you can do well, other skills you can use with effort, and then the other skills that you

don't yet know. A particular problem may be entirely solvable with the skills you already have or it may not, but you should fully investigate the problem using the skills already in your head before looking elsewhere. If we think of programming skills as tools and a programming problem as a home repair project, you should try to make the repair using the tools already in your garage before heading to the hardware store.

This technique follows the principles we have already discussed. It follows a plan and gives order to our efforts. When we begin our investigation of a problem by applying the skills we already have, we may learn more about the problem and its ultimate solution.

Reduce the Problem

With this technique, when faced with a problem you are unable to solve, you reduce the scope of the problem, by either adding or removing constraints, to produce a problem that you do know how to solve. We'll see this technique in action in later chapters, but here's a basic example. Suppose you are given a series of coordinates in three-dimensional space, and you must find the coordinates that are closest to each other. If you don't immediately know how to solve this, there are different ways you could reduce the problem to seek a solution. For example, what if the coordinates are in two-dimensional space, instead of three-dimensional space? If that doesn't help, what if the points lie along a single line so that the coordinates are just individual numbers (C++ doubles, let's say)? Now the question essentially becomes, in a list of numbers, find the two numbers with the minimum absolute difference.

Or you could reduce the problem by keeping the coordinates in three-dimensional space but have only three values, instead of an arbitrary-sized series. So instead of an algorithm to find the smallest distance between any two coordinates, it's just a question of comparing coordinate A to coordinate B, then B to C, and then A to C.

These reductions simplify the problem in different ways. The first reduction eliminates the need to compute the distance between three-dimensional points. Maybe we don't know how to do that yet, but until we figure that out, we can still make progress toward a solution. The second reduction, by contrast, focuses almost entirely on computing the distance between three-dimensional points but eliminates the problem of finding a minimal value in an arbitrary-sized series of values.

Of course, to solve the original problem, we will eventually need the skills involved in both reductions. Even so, reduction allows us to work on a simpler problem even when we can't find a way to divide the problem into steps. In effect, it's like a deliberate, but temporary, Kobayashi Maru. We know we're not working on the full problem, but the reduced problem has enough in common with the full problem that we will make progress toward the ultimate solution. Many times, programmers discover they have all the individual skills necessary to solve the problem, and by writing code to solve each individual aspect of the problem, they see how to combine the various pieces of code into a unified whole.

Reducing the problem also allows us to pinpoint exactly where the remaining difficulty lies. Beginning programmers often need to seek out experienced programmers for assistance, but this can be a frustrating experience for everyone involved if the struggling programmer is unable to accurately describe the help that is needed. One never wants to be reduced to saying, "Here's my program, and it doesn't work. Why not?" Using the problem-reduction technique, one can pinpoint the help needed, saying something like, "Here's some code I wrote. As you can see, I know how to find the distance between two three-dimensional coordinates, and I know how to check whether one distance is less than another. But I can't seem to find a general solution for finding the pair of coordinates with the minimum distance."

Look for Analogies

An *analogy*, for our purposes, is a similarity between a current problem and a problem already solved that can be exploited to help solve the current problem. The similarity may take many forms. Sometimes it means the two problems are really the same problem. This is the situation we had with the fox, goose, and corn problem and the Quarrasi lock problem.

Most analogies are not that direct. Sometimes the similarity concerns only part of the problems. For example, two number-processing problems might be different in all aspects except that both of them work with numbers requiring more precision than that given by built-in floating point data types; you won't be able to use this analogy to solve the whole problem, but if you've already figured out a way to handle the extra precision issue, you can handle that same issue the same way again.

Although recognizing analogies is the most important way you will improve your speed and skill at problem solving, it is also the most difficult skill to develop. The reason it is so difficult at first is that you can't look for analogies until you have a storehouse of previous solutions to reference.

This is where developing programmers often try to take a shortcut, finding code that is similar to the needed code and modifying from there. For several reasons, though, this is a mistake. First, if you don't complete a solution yourself, you won't have fully understood and internalized it. Put simply, it's very difficult to correctly modify a program that you don't fully understand. You don't need to have written code to fully understand, but if you could not have written the code, your understanding will be necessarily limited. Second, every successful program you write is more than a solution to a current problem; it's a potential source of analogies to solve future problems. The more you rely on other programmers' code now, the more you will have to rely on it in the future. We'll talk in depth about "good reuse" and "bad reuse" in Chapter 7.

Experiment

Sometimes the best way to make progress is to try things and observe the results. Note that experimentation is not the same as guessing. When you guess, you type some code and hope that it works, having no strong belief

that it will. An experiment is a controlled process. You hypothesize what will happen when certain code is executed, try it out, and see whether your hypothesis is correct. From these observations, you gain information that will help you solve the original problem.

Experimentation may be especially helpful when dealing with application programming interfaces or class libraries. Suppose you are writing a program that uses a library class representing a vector (in this context, a one-dimensional array that automatically grows as more items are added), but you've never used this vector class before, and you're not sure what happens when an item is deleted from the vector. Instead of forging ahead with solving the original problem while uncertainties swirl inside your head, you could create a short, separate program just to play around with the vector class and to specifically try out the situations that concern you. If you spend a little time on the "vector demonstrator" program, it might become a reference for future work with the class.

Other forms of experimentation are similar to debugging. Suppose a certain program is producing output that is backward from expectations—for example, if the output is numerical, the numbers are as expected, but in the reverse order. If you don't see why this is occurring after reviewing your code, as an experiment, you might try modifying the code to deliberately make the output backward (run a loop in the reverse direction, perhaps). The resulting change, or lack of change, in the output may reveal the problem in your original source code or may reveal a gap in your understanding. Either way, you're closer to a solution.

Don't Get Frustrated

The final technique isn't so much a technique, but a maxim: Don't get frustrated. When you are frustrated, you won't think as clearly, you won't work as efficiently, and everything will take longer and seem harder. Even worse, frustration tends to feed on itself, so that what begins as mild irritation ends as outright anger.

When I give this advice to new programmers, they often retort that while they agree with my point in principle, they have no control over their frustrations. Isn't asking a programmer not to get frustrated at lack of success like asking a little boy not to yell out if he steps on a tack? The answer is no. When someone steps on a tack, a strong signal is immediately sent through the central nervous system, where the lower depths of the brain respond. Unless you know you're about to step on the tack, it's impossible to react in time to countermand the automatic response from the brain. So we'll let the little boy off the hook for yelling out.

The programmer is not in the same boat. At the risk of sounding like a self-help guru, a frustrated programmer isn't responding to an external stimulus. The frustrated programmer isn't angry with the source code on the monitor, although the programmer may express the frustration in those terms. Instead, the frustrated programmer is angry at himself or herself. The source of the frustration is also the destination, the programmer's mind.

When you allow yourself to get frustrated—and I use the word "allow" deliberately—you are, in effect, giving yourself an excuse to continue to fail. Suppose you're working on a difficult problem and you feel your frustration rise. Hours later, you look back at an afternoon of gritted teeth and pencils snapped in anger and tell yourself that you would have made real progress if you had been able to calm down. In truth, you may have decided that giving in to your anger was easier than facing the difficult problem.

Ultimately, then, avoiding frustration is a decision you must make. However, there are some thoughts you can employ that will help. First of all, never forget the first rule, that you should always have a plan, and that while writing code that solves the original problem is the goal of that plan, it is not the only step of that plan. Thus, if you have a plan and you're following it, then you are making progress and you must believe this. If you've run through all the steps on your original plan and you're still not ready to start coding, then it's time to make another plan.

Also, when it comes down to getting frustrated or taking a break, you should take a break. One trick is to have more than one problem to work on so that if this one problem has you stymied, you can turn your efforts elsewhere. Note that if you successfully divide the problem, you can use this technique on a single problem; just block out the part of the problem that has you stuck, and work on something else. If you don't have another problem you can tackle, get out of your chair and do something else, something that keeps your blood flowing but doesn't make your brain hurt: Take a walk, do the laundry, go through your stretching routine (if you're signing up to be a programmer, sitting at a computer all day, I highly recommend developing a stretching routine!). Don't think about the problem until your break is over.

Exercises

Remember, to truly learn something you have to put it into practice, so work as many exercises as you can. In this first chapter, of course, we're not yet discussing programming, but even so, I encourage you to try some exercises out. Think of these questions as warm-ups for your fingers before we start playing the real music.

1-1. Try a medium-difficulty sudoku puzzle (you can find these all over the Web and probably in your local newspaper), experimenting with different strategies and taking note of the results. Can you write a general plan for solving a sudoku?

1-2. Consider a sliding tile puzzle variant where the tiles are covered with a picture instead of numbers. How much does this increase the difficulty, and why?

1-3. Find a strategy for sliding tile puzzles different from mine.

1-4. Search for old-fashioned puzzles of the fox, goose, and corn variety and try to solve them. Many of the great puzzles were originated or popularized by Sam Loyd, so you might search for his name. Furthermore, once you uncover (or give up and read) the solution, think of how you could make an easier version of the puzzle. What would you have to change? The constraints or just the wording?

1-5. Try to write some explicit strategies for other traditional pencil-and-paper games, like crosswords. Where should you start? What should you do when you're stuck? Even simple newspaper games, like "Jumble," are useful for contemplating strategy.