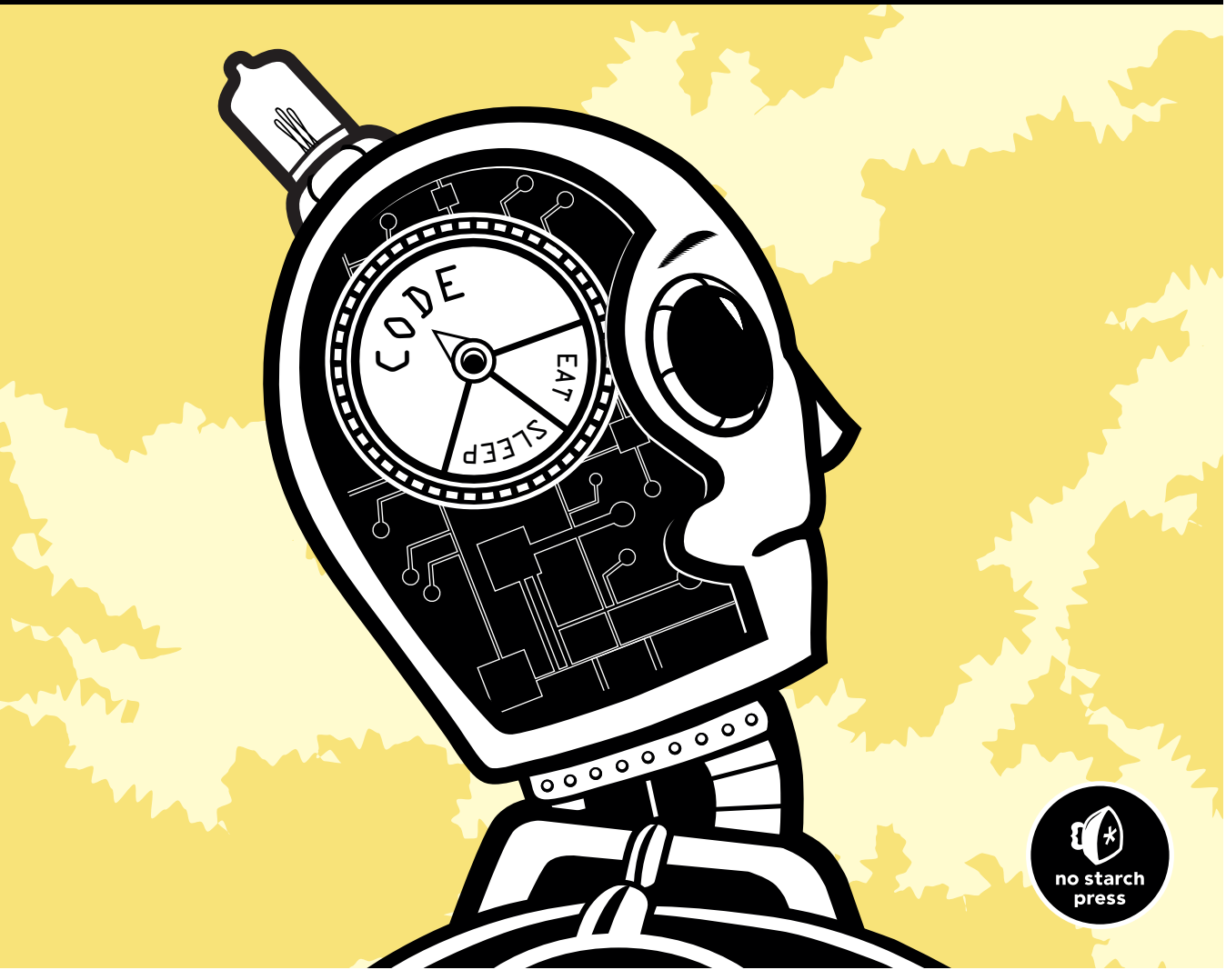
THINK LIKE A PROGRAMMER

AN INTRODUCTION TO
CREATIVE PROBLEM SOLVING

V. ANTON SPRAUL



no starch press

THINK LIKE A PROGRAMMER

THINK LIKE A PROGRAMMER

An Introduction to Creative Problem Solving

by V. Anton Spraul

no starch press

San Francisco

First printing

15 14 13 12 1 2 3 4 5 6 7 8 9

ISBN-10: 1-59327-424-6
ISBN-13: 978-1-59327-424-5

Publisher: William Pollock
Production Editor: Alison Law
Cover Design: Charlie Wylie
Interior Design: Octopod Studios
Developmental Editor: Keith Fancher
Technical Reviewer: Dan Randall
Copyeditor: Julianne Jigour
Compositor: Susan Glinert Stevens
Proofreader: Ward Webber

For information on book distributors or translations, please contact No Starch Press, Inc. directly:

No Starch Press, Inc.
38 Ringold Street, San Francisco, CA 94103
phone: 415.863.9900; fax: 415.863.9950; info@nostarch.com; www.nostarch.com

Library of Congress Cataloging-in-Publication Data
A catalog record of this book is available from the Library of Congress.

BRIEF CONTENTS

CONTENTS IN DETAIL

ACKNOWLEDGMENTS

No book is truly the work of one author, and I've received lots of help on *Think Like a Programmer*.

I'm grateful to everyone at No Starch Press, especially Keith Fancher and Alison Law, who edited, shaped, and shepherded the book throughout its production. I must also thank Bill Pollock for his decision to sign me up in the first place—I hope he is as pleased with the result as I am. The folks at No Starch have been unfailingly kind and helpful in their correspondence with me. I hope one day to meet them in person and see to what degree they resemble their cartoon avatars on the company website.

Dan Randall did a wonderful job as technical editor. His numerous suggestions beyond the technical review helped me strengthen the manuscript in many areas.

On the home front, the most important people in my life, Mary Beth and Madeline, provided love, support, and enthusiasm—and, crucially, time to write.

Finally, to all the students of programming I've had over the years: Thank you for letting me be your teacher. The techniques and strategies described in this book were developed through our joint efforts. I hope we've made the journey easier for the next generation of programmers.

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

book as a "crib sheet" to get you past particular problems or focus on just the chapters that directly relate to your current struggles. Instead, I would work through the entire book, skipping material only if you lack the prerequisites needed to follow the discussion.

Programming Style

A quick note here about the programming style employed in this book: This book is not about high-performance programming or running the most compact, efficient code. The style I have chosen for the source code examples is intended to be readable above all other considerations. In some cases, I take multiple steps to accomplish something that could be done in one step, just so the principle I'm trying to demonstrate is made clear.

Some aspects of programming style will be covered in this book—but larger issues, like what should or should not be included in a class, not small issues, like how code should be indented. As a developing programmer, you will of course want to employ a consistent, readable style in all of the work you do.

Exercises

The book includes a number of programming exercises. This is not a textbook, and you won't find answers to any of the exercises in the back. The exercises provide opportunities for you to apply the concepts described in the chapters. Whether you choose to try any of the exercises is, of course, up to you, but it is essential that you put these concepts into practice. Simply reading through the book will accomplish nothing. Remember that this book is not going to tell you exactly what to do in each situation. In applying the techniques shown in this book, you will develop your own ability to discover what to do. Furthermore, growing your confidence, another primary goal of this book, requires success. In fact, that's a good way to know when you have worked through enough exercises in a given problem area: when you are confident that you can tackle other problems in the area. Lastly, programming exercises should be *fun*. While there may be moments where you'd rather be doing something else, working out a programming problem should be a rewarding challenge.

You should think of this book as an obstacle course for your brain. Obstacle courses build strength, stamina, and agility and give the trainer confidence. By reading through the chapters and applying the concepts to as many exercises as you can, you're going to build confidence and develop problem-solving skills that can be used in any programming situation. In the future, when you are faced with a difficult problem, you'll know whether you should try going over, under, or through it.

Why C++?

The programming examples in this text are coded using C++. Having said that, this book is about solving problems with programs, not specifically about C++. You won't find many tips and tricks specific to C++ here, and the general concepts taught throughout this book can be employed in any programming language. Nevertheless, you can't discuss programming without discussing programs, and a specific language had to be chosen.

C++ was selected for a number of reasons. First, it's popular in a variety of problem areas. Second, because of its origins in the strictly procedural C language, C++ code can be written using both the procedural and object-oriented paradigms. Object-oriented programming is so common now that it could not be omitted from a discussion on problem solving, but many fundamental problem-solving concepts can be discussed in strictly procedural programming terms, and doing so simplifies both the code and the discussion. Third, as a low-level language with high-level libraries, C++ allows us to discuss both levels of programming. The best programmers can "hand-wire" solutions when required and make use of high-level libraries and application programming interfaces to reduce development time. Lastly, and partly as a function of the other reasons listed, C++ is a great choice because once you have learned to solve problems in C++, you have learned to solve problems in any programming language. Many programmers have discovered how the skills learned in one language easily apply to other languages, but this is especially true for C++ because of its cross-paradigm approach and, frankly, because of its difficulty. C++ is the real deal—it's programming without training wheels. This is daunting at first, but once you start succeeding in C++, you'll know that you're not going to be someone who can do a little coding—you're going to be a programmer.

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

solvers are quick to recognize an *analogy*, an exploitable similarity between a solved problem and an unsolved problem. If we recognize that a feature of problem A is analogous to a feature of problem B and we have already solved problem B, we have a valuable insight into solving problem A.

In this section, we'll discuss classic problems from outside the world of programming that have lessons we can apply to programming problems.

The Fox, the Goose, and the Corn

The first classic problem we will discuss is a riddle about a farmer who needs to cross a river. You have probably encountered it previously in one form or another.

PROBLEM: HOW TO CROSS THE RIVER?

A farmer with a fox, a goose, and a sack of corn needs to cross a river. The farmer has a rowboat, but there is room for only the farmer and one of his three items. Unfortunately, both the fox and the goose are hungry. The fox cannot be left alone with the goose, or the fox will eat the goose. Likewise, the goose cannot be left alone with the sack of corn, or the goose will eat the corn. How does the farmer get everything across the river?

The setup for this problem is shown in Figure 1-1. If you have never encountered this problem before, stop here and spend a few minutes trying to solve it. If you *have* heard this riddle before, try to remember the solution and whether you were able to solve the riddle on your own.

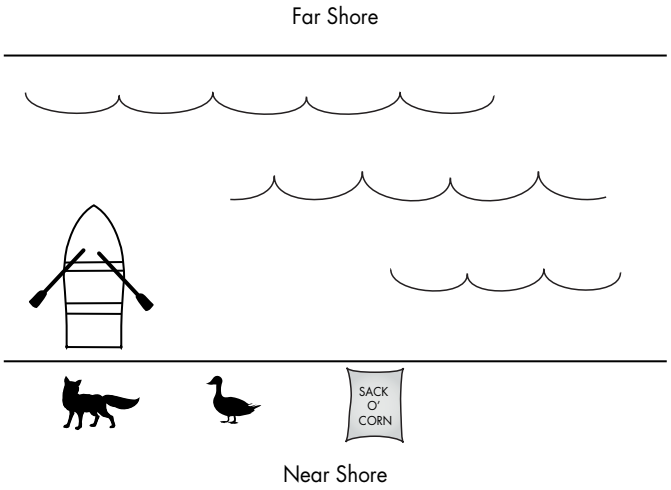


Figure 1-1: The fox, the goose, and the sack of corn. The boat can carry one item at a time. The fox cannot be left on the same shore as the goose, and the goose cannot be left on the same shore as the sack of corn.

Few people are able to solve this riddle, at least without a hint. I know I wasn't. Here's how the reasoning usually goes. Since the farmer can take only one thing at a time, he'll need multiple trips to take everything to the far shore. On the first trip, if the farmer takes the fox, the goose would be left with the sack of corn, and the goose would eat the corn. Likewise, if the farmer took the sack of corn on the first trip, the fox would be left with the goose, and the fox would eat the goose. Therefore, the farmer must take the goose on the first trip, resulting in the configuration shown in Figure 1-2.

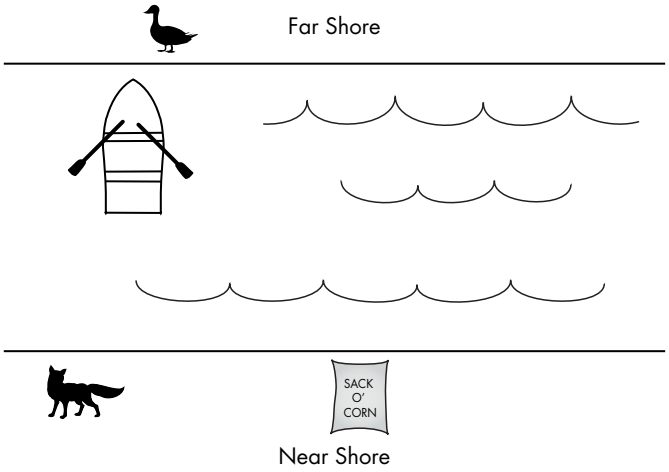


Figure 1-2: The required first step for solving the problem of the fox, the goose, and the sack of corn. From this step, however, all further steps appear to end in failure.

So far, so good. But on the second trip, the farmer must take the fox or the corn. Whatever the farmer takes, however, must be left on the far shore with the goose while the farmer returns to the near shore for the remaining item. This means that either the fox and goose will be left together or the goose and corn will be left together. Because neither of these situations is acceptable, the problem appears unsolvable.

Again, if you have seen this problem before, you probably remember the key element of the solution. The farmer has to take the goose on the first trip, as explained before. On the second trip, let's suppose the farmer takes the fox. Instead of leaving the fox with the goose, though, the farmer *takes the goose back* to the near shore. Then the farmer takes the sack of corn across, leaving the fox and the corn on the far shore, while returning for a fourth trip with the goose. The complete solution is shown in Figure 1-3.

This puzzle is difficult because most people never consider taking one of the items back from the far shore to the near shore. Some people will even suggest that the problem is unfair, saying something like, "You didn't say I could take something back!" This is true, but it's also true that nothing in the problem description suggests that taking something back is prohibited.