1 one

What This Book Is About

It is impossible to know things of this world
unless you know mathematics.
Roger Bacon, *Opus Majus*

This book is about programming, but it is different from most programming books. Along with algorithms and code, you'll find mathematical proofs and historical notes about mathematical discoveries from ancient times to the 20th century.

More specifically, the book is about *generic programming*, an approach to programming that was introduced in the 1980s and started to become popular following the creation of the C++ Standard Template Library (STL) in the 1990s. We might define it like this:

Definition 1.1. Generic programming is an approach to programming that focuses on designing algorithms and data structures so that they work in the most general setting without loss of efficiency.

If you've used STL, at this point you may be thinking, "Wait a minute, that's all there is to generic programming? What about all that stuff about templates and iterator traits?" Those are tools that enable the language to support generic programming, and it's important to know how to use them effectively. But generic programming itself is more of an *attitude* toward programming than a particular set of tools.

We believe that this attitude—trying to write code in this general way—is one that all programmers should embrace. The components of a well-written generic program are easier to use and modify than those of a program whose data structures, algorithms, and interfaces hardcode unnecessary assumptions about

a specific application. Making a program more generic renders it simultaneously both simpler and more powerful.

1.1 Programming and Mathematics

So where does this generic programming attitude come from, and how do you learn it? It comes from mathematics, and especially from a branch of mathematics called *abstract algebra*. To help you understand the approach, this book will introduce you to a little bit of abstract algebra, which focuses on how to reason about objects in terms of abstract properties of operations on them. It's a topic normally studied only by university students majoring in math, but we believe it's critical in understanding generic programming.

In fact, it turns out that many of the fundamental ideas in programming came from mathematics. Learning how these ideas came into being and evolved over time can help you think about software design. For example, Euclid's *Elements*, a book written more than 2000 years ago, is still one of the best examples of how to build up a complex system from small, easily understood pieces.

Although the essence of generic programming is abstraction, abstractions do not spring into existence fully formed. To see how to make something more general, you need to start with something concrete. In particular, you need to understand the specifics of a particular domain to discover the right abstractions.

The abstractions that appear in abstract algebra largely come from concrete results in one of the oldest branches of mathematics, called *number theory*. For this reason, we will also introduce some key ideas from number theory, which deals with properties of integers, especially divisibility.

The thought process you'll go through in learning this math can improve your programming skills. But we'll also show how some of the mathematical results themselves turn out to be crucial to some modern software applications. In particular, by the end of the book we'll show how some of these results are used in cryptographic protocols underlying online privacy and online commerce.

The book will move back and forth between talking about math and talking about programming. In particular, we'll interweave important ideas in mathematics with a discussion of both specific algorithms and general programming techniques. We'll mention some algorithms only briefly, while others will be refined and generalized throughout the book. A couple of chapters will contain only mathematical material, and a couple will contain only programming material, but most have a mixture of both.

1.2 A Historical Perspective

We've always found that it's easier and more interesting to learn something if it's part of a story. What was going on at the time? Who were the people involved,

and how did they come to have these ideas? Was one person's work an attempt to build on another's—or an attempt to reject what came before? So as we introduce the mathematical ideas in this book, we'll try to tell you the story of those ideas and of the people who came up with them. In many cases, we've provided short biographical sketches of the mathematicians who are the main characters in our story. These aren't comprehensive encyclopedia entries, but rather an attempt to give you some context for who these people were.

Although we take a historical perspective, that doesn't mean that the book is intended as a history of mathematics or even that all the ideas are presented in the order in which they were discovered. We'll jump around in space and time when necessary, but we'll try to give a historical context for each of the ideas.

1.3 Prerequisites

Since a lot of the book is about mathematics, you may be concerned that you need to have taken a lot of math classes to understand it. While you'll need to be able to think logically (something you should already be good at as a programmer), we don't assume any specific mathematical knowledge beyond high school algebra and geometry. In a couple of sections, we show some applications that use a little linear algebra (vectors and matrices), but you can safely skip these if you haven't been exposed to the background material before. If you're unfamiliar with any of the notation we use, it's explained in Appendix A.

An important part of mathematics is being able to prove something formally. This book contains quite a few proofs. You'll find the book easier to understand if you've done some proofs before, whether in high school geometry, in a computer science class on automata theory, or in logic. We've described some of the common proof techniques we use, along with examples, in Appendix B.

We assume that if you're reading this book, you're already a programmer. In particular, you should be reasonably proficient in a typical imperative programming language like C, C++, or Java. Our examples will use C++, but we expect you'll be able to understand them even if you've never programmed in that language before. When we make use of a construct unique to C++, we explain it in Appendix C. Irrespective of our use of C++, we believe that the principles discussed in this book apply to programming in general.

Many of the programming topics in this book are also covered from a different perspective, and more formally, in *Elements of Programming* by Stepanov and McJones. Readers interested in additional depth may find that book to be a useful companion to this one. Throughout this book, we occasionally refer interested readers to a relevant section of *Elements of Programming*.

1.4 Roadmap

Before diving into the details, it's useful to see a brief overview of where we're headed:

- Chapter 2 tells the story of an ancient algorithm for multiplication, and how to improve it.
- Chapter 3 looks at some early observations about properties of numbers, and an efficient implementation of an algorithm for finding primes.
- Chapter 4 introduces an algorithm for finding the greatest common divisor (GCD), which will be the basis for some of our abstractions and applications later on.
- Chapter 5 focuses on mathematical results, introducing a couple of important theorems that will play a critical role by the end of the book.
- Chapter 6 introduces the mathematical field of abstract algebra, which provides the core idea for generic programming.
- Chapter 7 shows how these mathematical ideas allow us to generalize our multiplication algorithm beyond simple arithmetic to a variety of practical programming applications.
- Chapter 8 introduces new abstract mathematical structures, and shows some new applications they enable.
- Chapter 9 talks about axiom systems, theories, and models, which are all building blocks of generic programming.
- Chapter 10 introduces concepts in generic programming, and examines the subtleties of some apparently simple programming tasks.
- Chapter 11 continues the exploration of some fundamental programming tasks, examining how different practical implementations can exploit theoretical knowledge of the problem.
- Chapter 12 looks at how hardware constraints can lead to a new approach for an old algorithm, and shows new applications of GCD.
- Chapter 13 puts the mathematical and algorithmic results together to build an important cryptography application.
- Chapter 14 is a summary of some of the principal ideas in the book.

The strands of programming and mathematics are interwoven throughout, though one or the other may lie hidden for a chapter or two. But every chapter plays a part in the overall chain of reasoning that summarizes the entire book:

> *To be a good programmer, you need to understand the principles of generic programming. To understand the principles of generic programming, you need to understand abstraction. To understand abstraction, you need to understand the mathematics on which it's based.*

That's the story we're hoping to tell.