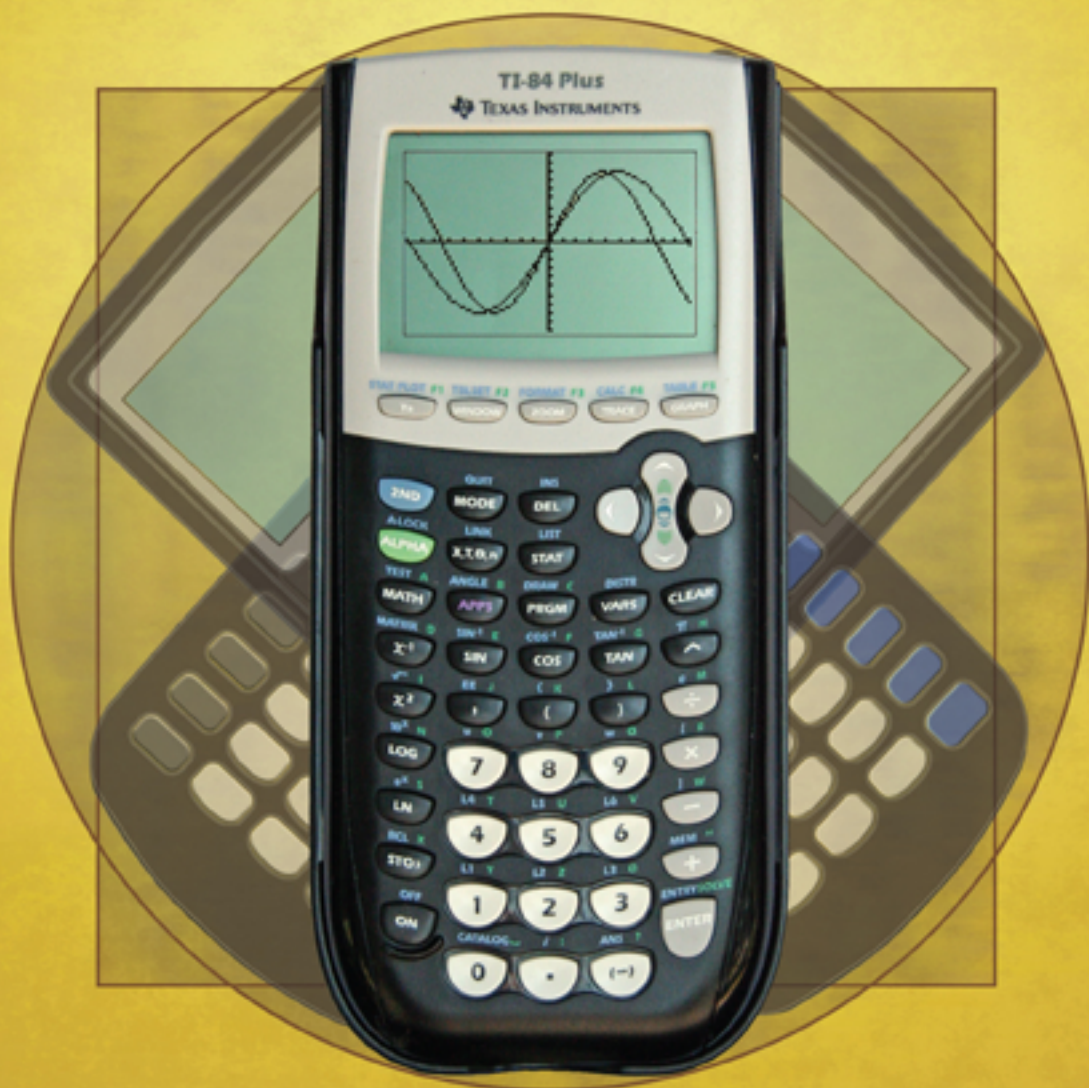# Programming *the*
TI-83 Plus/TI-84 Plus

Christopher R. Mitchell

Foreword by Brandon Wilson

MANNING

Programming the TI-83 Plus/TI-84 Plus

Programming the TI-83 Plus/ TI-84 Plus

CHRISTOPHER R. MITCHELL

MANNING

brief contents

contents

foreword

As a professional computer software developer, I can tell you that some of the greatest programmers start with the simplest of hardware and the most fervent determination.

Mastering a small computer system (such as the Texas Instruments graphing calculator) not only feels fantastic, but also teaches core programming concepts and solidifies ways of thinking that mediocre programmers seldom grasp. I have been in the TI graphing calculator community for well over a decade, as has Christopher (known among us as "Kerm Martian"). Throughout that time we have watched each other's humble beginnings, been amazed as our successful (and at times overly ambitious) projects flourished, and even watched others learn from us. I can think of no one more capable of teaching the basics of programming the TI-83 Plus series graphing calculators, and ensuring you have fun along the way, than Christopher "Kerm Martian" Mitchell.

My own fascination with graphing calculators, and particularly the TI-83 and TI-84 Plus series, began in the late 1990s, when only cripplingly slow dial-up and mailing lists bound us together. The TI-GRAPH LINK cable allowing connection between a computer and a graphing calculator had only recently made its debut, eliminating the need to painstakingly hand-type TI-BASIC games and utilities. At the time, I thought the program editor was merely for typing notes. It wasn't until I discovered the programming chapter of the thick, cryptic TI-83 manual that I realized it could do so much more.

I spent many days reading the entire manual over and over, striving to understand every command I could execute from within a TI-BASIC program. I became enamored with the concept of taking a limited set of instructions and transforming the calculator

into anything I could imagine. Before I knew it, I was spending 7th grade math class happily playing my own random number guessing game while others struggled to stay awake at their desks. The idea of sculpting complex applications (and let's face it, games) in my own mind and then pouring them out onto the calculator keys captivated me and pulled me into the world of software development.

Once I had mastered TI-BASIC, my curiosity did not cease. How does TI-BASIC work? What happens when the calculator executes a TI-BASIC command? What happens behind the scenes? How does the calculator know how to display graphs, or what to do when a key is pressed or a menu item chosen? I discovered that the answers lay within a mysterious second programming language that was all the rage—assembly language. This language consists of the raw instructions that the calculator's processor executes; it was used to write the calculator's OS and the interpreter that makes TI-BASIC possible.

For years, programmers had been writing in assembly language to create programs even more powerful and flexible than what TI-BASIC allowed. To share in the fun and understand all of the TI-83 Plus' inner workings, I knew I had to learn it. From that moment on, I made it my goal to learn everything there was to learn about the underlying software that makes the TI-83 Plus series tick. Even after years of reverse engineering the OS (and, on occasion, exploiting some of its more interesting security vulnerabilities), it remains an elusive goal.

For as long as I can remember, Christopher (or Kerm) has humbly granted himself the title of "world's most prolific calculator programmer," which as it turns out, is a well-deserved description. Despite TI-BASIC's reputation as a relatively limited language compared to the calculator's native assembly language, Christopher started cranking out programs soon after learning it and has never stopped—from TI-BASIC to assembly programs to FLASH applications.

Christopher's crowning achievement in the world of programming TI-83 Plus applications is Doors CS, a powerful, versatile calculator shell (a program that provides a user interface for running other programs). I can still recall the first version of Doors CS, written in pure TI-BASIC and requiring tedious manual configuration to overcome some (but not all) of TI-BASIC's shortcomings. As a calculator shell requires total access to the calculator's memory and the variables contained within for management and execution of programs, TI-BASIC is not a language conducive to producing a great shell. Despite this and a bit of negative criticism, Christopher persevered and strove to improve upon it no matter the cost, eventually implementing it in assembly language and adding many useful features and tight integration with the TI operating system. Some of my favorite memories are of staying up very late at night (and into the early morning) with Christopher, reverse engineering some of the more mysterious parts of the OS to diagnose lingering issues in Doors CS' interaction with existing calculator functionality.

Today, Christopher has produced one of the community's leading shells, software to allow using a calculator over the internet for chatting and playing calculator games with others, and other projects too numerous to mention. Cemetech.net has evolved

from his personal website into a haven for anyone interested in programming TI graphing calculators or receiving help in doing so.

Some of the greatest members of the calculator community—longtime developers like Dan Englender, Michael Vincent, Benjamin Moody, and countless others—made calculator programming the great learning experience and joy it is for so many, and it is safe to say that Christopher stands with them.

I have no doubt that Christopher's unique understanding of the TI-83 Plus series, the TI-BASIC language, and all that lies beyond will prove to be a valuable asset as you go through this book. It will teach you all there is to know about TI-BASIC, assembly language, and everything in between. And it will help you explore the wonder and awe that can be found in calculator programming.

Enjoy!

BRANDON WILSON
SENIOR SOFTWARE DEVELOPER
ADVANCED CALL CENTER TECHNOLOGIES (ACT)

preface

When I was 13 years old, I received my first graphing calculator. It was Christmas, and my biggest present under the tree was a TI-83. I was thrilled. I first used it just for math, but over several months, I became more curious and discovered that I could write programs directly on the calculator. The guidebook included with the calculator didn't really help with programming, other than demonstrating an interesting Sierpinski Triangle. Undeterred, I set off to teach myself calculator programming, although I never thought of it in such definite terms.

I first learned to display text on the screen and then to make simple animations. I discovered that I could also ask the user for input and thus make simple math programs to check my homework results. Soon classmates began passing around arcade games they had found for their calculators, so I dug into the source code for those games and found out how they worked, using my new skills to create games of my own. Over the years I grew more competent, including learning to write z80 assembly, a more complex but much more powerful language than TI-BASIC. I started an online community around graphing calculator programming called Cemetech (pronounced "KEH-meh-tek") that thrives as a hardware and software development haven to this day. I continued to pursue programming as well as my lifelong love of hardware and electronics. I earned two degrees in electrical engineering and one in computer science; I'm now working toward my doctorate. I credit much of my love of programming and engineering to those first faltering steps with my graphing calculator.

Having helped new calculator programmers to learn the tricks of the trade for close to 13 years on Cemetech's forum, I've heard countless variations on my story.

I've worked with students who got a calculator and started to play with its math features, only to discover it was programmable. I've helped others who downloaded games from their peers, then took the games apart to see what made them tick. I've seen like-minded students form small programming groups to make math programs and games for their friends. Many of these students are now in college or graduate school, studying engineering or computer science; others work in the industry as professional programmers or as teachers and professors. Almost all of them credit their first forays into calculator coding for their current love of technology and programming.

When I show off my latest projects, there's bound to be at least one person who asks, "why?" Why would I bother working with such a low-powered, primitive device, when I have the equipment and skills to write more complex software for vastly more capable systems? The answer is that I love the utility of graphing calculators as an introductory programming platform and I love a challenge. When I'm writing a TI-83+ program, every byte of the calculator's 24 K of RAM is important, and every cycle of its 6 MHz processor must be carefully rationed.

When I wear my other hats as an electrical engineer, a computer scientist, a webmaster, and a researcher, I work with systems that have many more capabilities. These systems provide their own performance and design challenges, but none are quite as simultaneously simple and complex as graphing calculators. From a teaching perspective, I believe calculators are an accessible platform on which to learn the problem-solving skills vital to becoming a good programmer. You can write and test code directly on a device that many students already own, and, with only the capabilities built into your $100 calculator, create surprisingly complex projects. In a very real sense, you're working with a full-fledged if slightly antiquated computer.

These dual attractions of graphing calculator programming have driven me to continue to pursue my own calculator projects and to build a community of like-minded coders and teachers. Throughout the years, I've sporadically hoped to document my extensive calculator programming experience in some way. In 2003, I wrote a rudimentary TI-BASIC tutorial. Two years later, I wrote and published a guide to advanced TI-BASIC optimizations with a fellow Cemetech administrator, which 14,000 coders have read to date. Between 2005 and 2006, I attempted to motivate the community to document their TI-BASIC knowledge in a wiki, a project that never gained much traction, but I continued to wish there was a way to write an exhaustive, thorough guide to TI-BASIC programming.

So it was with excitement that I received Manning's request that I write a book about graphing calculator programming. I've tried to transcribe as many of the lessons that I learned over the years onto these pages, from basic lessons to advanced tips and tricks. I've found that calculator programming has helped me to think more critically as a programmer and as an engineer and made it easier for me to pick up other languages. I've tried to pass along many of the general problem-solving lessons I've accumulated in these pages, and I hope that regardless of whether you are learning

calculator programming as its own goal or as a stepping-stone toward another language, you'll have as much fun reading the coming chapters as I had writing them.

Any good programmer, engineer, or scientist knows that there's always more to learn, so I hope to hear from many of you and find out what role calculators played in your life and how this book helped you. Perhaps you'll show off some tips and projects of your own on Cemetech or in the larger programming community. I hope in the future to continue to help you with your programming, through other books, Cemetech, or indirectly through the rest of the programming community.

Good luck, and enjoy!

acknowledgments

Thanks must first go to the friends, family, and loved ones who've supported my programming and engineering career throughout the years. I'd especially like to acknowledge my mother, Maria Mitchell, for getting me my first calculator, always supporting my education, and offering moral support during this book's creation. My friends and loved ones have been patient with my hobbies and projects and have always been ready with words of encouragement; my girlfriend, Sara Nodroff, was there for me throughout the many hours I spent on this project. I'm also grateful to teachers and advisers current and past who helped me get where I am today, especially to Jinyang Li, who was understanding of my threading the writing of this book around my PhD research.

Although my first forays into calculator programming took place on my own, the members of the worldwide graphing calculator enthusiast community have been my colleagues and friends for close to a decade. It's hard to name all of the individuals who have made a difference for me, so if I don't specifically acknowledge you, know that I treasure your help, inspiration, and camaraderie nonetheless. I must first tip my hat to my Cemetech administrators, staff, and friends, who have stood by me through my technical and personal struggles and achievements. Thomas "Elfprince13" Dickerson and Daniel "TIFreak8x" Thorneycroft have been with Cemetech since its early days and have encouraged my projects for more than seven years. Shaun "Merthsoft" Mcfall and Jon "Jonimus/TheStorm" Sturm, more recent additions to Cemetech, have become my valued friends and colleagues. Other Cemetech staff past and present have been my teachers, students, and friends, including Theodore Davis, Alex Glanville,

about this book

Graphing calculator programming is a rewarding way to get started in computer programming, to develop your existing skills, or just to have fun with the challenge of working with such a device. If you're a student or teacher, especially of math or science, the programs you write for your calculator can speed up annoying, repetitive calculations or help you check your work. You can enjoy the feeling of accomplishment from completing a useful utility or a fast-paced game for your calculator.

From this book, you'll learn everything you need to know to progress from a non-programmer to a TI-BASIC pro. If you have programming experience, or even TI-BASIC skills, it will teach you advanced tricks and hopefully help you see the language in a new way. The problem-solving skills in each chapter can be applied to almost any programming language that you might encounter.

If you're a beginner, I recommend that you read this book front to back, starting from the first chapter and working your way to the end. If you have some experience or are looking for answers to specific questions, you can skip to the relevant chapter. I assume beginning in chapter 2 that every reader has the same basic set of calculator skills and knows how to perform math, draw graphs, and use lists and matrices. If you're uncomfortable with any of those concepts, I strongly recommend that you read through appendix A before you get to chapter 2. In case you forget the syntax for any TI-BASIC command that you learn, you can look at appendix B, which is arranged to parallel the organization of the chapters. No programmer should have to code in a vacuum, so when you get stuck, be sure to visit the Author Online forum, Cemetech, or any of the other forums and websites listed in appendix C.

Throughout this book, you'll look at both educational and fun programs that test each new idea and cobble it together with the things that you've already learned. In many places, I'll talk about some program that you might want to write but don't yet know how to create and then introduce new concepts that will provide those skills.

Roadmap

This book consists of 13 chapters, divided into three parts. It also has three appendixes, which summarize skills, commands, and resources that any calculator programmer might need. Part 1 focuses on introducing programming skills that are important for TI-BASIC programming but apply to almost any language you might want to learn.

- Chapter 1 introduces graphing calculators and calculator programming, outlining why learning TI-BASIC is important and relevant. It presents your first three programs: a Hello World program, a guessing game, and a quadratic equation solver.
- Chapter 2 presents input and output on the homescreen, including displaying text and numbers and getting strings and values from the user.
- Chapter 3 covers conditionals and comparisons, the building blocks for creating programs that make decisions.
- Chapter 4 completes the picture of controlling program flow in TI-BASIC with labels, loops, menus, and subprograms, all of the structural features that you'll need to create arbitrarily complex programs.
- Chapter 5 steps back to detail the process of designing, creating, and debugging a program in any language. It illustrates each step with a running TI-BASIC example.

Part 2 takes the basic framework from part 1 and teaches additional commands and features necessary for more professional and complete programs. These include graphics, interactivity, and the proper use of the many data types your calculator understands, such as matrices, lists, strings, and pictures.

- Chapter 6 teaches you how to create fun, interactive programs and games with event loops. As with many other lessons, it wraps the TI-BASIC focus in skills you'll be able to bring to many other languages. This chapter culminates in a full Mouse and Cheese game for your edification.
- Chapter 7 discusses your first true graphics tools, presenting the concepts and commands for turning individual pixels on and off. It shows how to draw small and large text anywhere on the screen and reinforces the lessons of the chapter with two demo programs: a painting tool and a mouse cursor subprogram.
- Chapter 8 expands further on graphics and graphing, covering creating and manipulating graphs from inside programs, as well as drawing with points, lines, circles, and other shapes. It introduces the commands for storing and recalling pictures on the graphscreen.
- Chapter 9 rounds out part 2 with an overview of the many data types your calculator can handle and the important commands for manipulating each. It walks through strings, lists, matrices, real and complex numbers, and random numbers,

and it concludes with a complete framework for a role-playing game (RPG) that you can expand and enhance on your own.

Part 3 goes into advanced concepts and may be particularly engaging even if you have prior TI-BASIC or programming experience. It covers optimization, hybrid BASIC, and the rudiments of assembly.

- Chapter 10 details how to optimize your programs for speed and size, presenting TI-BASIC–specific tips without losing sight of the more general programming lessons for proper optimization.
- Chapter 11 shows hybrid TI-BASIC and the hybrid BASIC libraries and includes a discussion of the major libraries, where to find them, and how to use them.
- Chapter 12 introduces a new programming language, z80 assembly, giving you enough detail about binary, decimal, hexadecimal, and assembly commands and program flow to spur you to explore it more on your own.
- Chapter 13 concludes with ideas about where you can go with programming and calculator programming from here. It also discusses hardware development and hacking and how such a hobby ties into calculator programming.

The appendixes provide a quick reference to material supplementing and coalescing the contents of the chapters:

- Appendix A is a crash course in using your graphing calculator. Chapters 2 onward assume a basic set of general calculator skills, and appendix A reviews all of these skills in case you don't feel entirely comfortable with your device.
- Appendix B summarizes all of the commands found throughout the chapters and includes usage examples and syntax.
- Appendix C lists valuable resources for seeking programming help, finding additional programs for inspiration and source code examination, and tools to facilitate BASIC and assembly programming.

Who should read this book

Who are you? You might be a student who is getting a graphing calculator for the first time or recently started using one and wants to unlock your device's full potential. Perhaps you are a teacher, an engineer, a programmer, or just curious. If you've never before programmed anything, you have a whole world of amazing things that programming can enable you to do and learn in front of you, and I'll be honored to guide you forward. This book is primarily aimed at you, the budding programmer. I'll lead you through graphing calculator programming, but I'll also help you keep an eye on programming in general and teach you concepts you can apply to almost any language.

If you've toyed with programming before, for calculators, computers, or another platform, I hope this book can teach you how to learn more, to write and understand complete programs, and to have fun doing so. If you're an advanced programmer,

either for calculators or something else, I want to provide you with a great reference guide for calculator programming, advanced topics and optimization tricks, perhaps get you interested in z80 assembly programming, and give you another perspective on programming as a hobby and as a career.

I'll teach you everything you need to know to write complete programs for your graphing calculator (and everyone else's); I assume that you have no prior knowledge of calculator programming or any sort of programming. I'll teach you how to think like a programmer and how to apply problem-solving skills to take any program you might want to write, break it down into pieces, and code each one. If you have some prior programming skill, great; if you have previous graphing calculator programming skill, all the better. The chapters ahead are designed to teach you everything you need to know, from the basics up to the most advanced tricks for creating very fast, very small, very fancy programs. If you have some experience, you may end up skimming sections, but even if you feel like you know your way around a simple TI-BASIC program, you're likely to run across new tricks and features that you hadn't previously played with.

Why write calculator programs; why not just jump straight to programming a computer? The short answer: the opportunity to learn quickly, have fun, surmount the challenges of a programming platform, and get started right away. If you're reading this book, chances are you already have a graphing calculator. If you don't, then you can get one for less than $100. The TI-83+, TI-83+ Silver Edition, TI-84+, and TI-84+ Silver Edition covered in this book are all cheap, widely available, and widely owned graphing calculators and can all run each other's programs. The TI-83 can run very similar programs and is similarly inexpensive and ubiquitous. Although their programming languages are somewhat different, many of the same skills can be applied to programming other TI graphing calculators and to Casio calculators such as the color-screen Casio Prizm. Calculators are small and portable, great to carry around and whip out when you have some downtime to work on your programming but don't have or want to carry around a laptop. They last for months, not a few hours, on a single charge or set of batteries.

Typographic conventions and code

Looking at code examples as you learn is vital to a full understanding of a language. Examples large and small, along with occasional exercises, are scattered far and wide through this text. Full programs are often presented in listings, though shorter programs may be interspersed between paragraphs in `monospaced` font. Several other conventions are followed:

- All keypresses are enclosed in square brackets, such as [ENTER] or [2nd]. The text between the brackets is the text printed on your calculator's keys. Chapter 2 explains more about the key convention used and how to type key combinations.
- Commands and tokens are mentioned in the text by their name in monospaced font, like `Disp` or `For` or `Line`. Some commands have parentheses after them, such as `For(` and `Line(`, but for neatness in text, these parentheses are often

omitted. The requisite parentheses are shown in code examples and when each such command is first presented.

- All code herein applies to the TI-83+, TI-83+ Silver Edition, TI-84+, and TI-84+ Silver Edition. These calculator families are formally named TI-83 Plus and TI-84 Plus, respectively, but I'll call them the TI-83+ and TI-84+ throughout, because that has become accepted parlance in the programming community. I strongly recommend that you have one of these four calculators to accompany your adventure through this book. Much of the code and almost all of the concepts also apply to the TI-82 and TI-83 and to a lesser extent to the TI-85 and TI-86. The TI-89 uses a different, more complex variant of the TI-BASIC language.

The code for all of the programs presented in this book can be found on the publisher's website, www.manning.com/ProgrammingtheTI-83Plus/TI-84Plus. Each program can be tested on your calculator or emulator; a list of the top TI calculator emulation software packages is included in appendix C. You can also view the source of programs on your computer using SourceCoder, at http://sc.cemetech.net.

All screenshots in this book were taken with the Wabbitemu or jsTIfied emulators and adjusted and annotated in GIMP. All source code listings were generated from the original programs by SourceCoder or written in that IDE and checked in an emulator.

Online resources

The purchase of *Programming the TI-83 Plus/TI-84 Plus* includes free access to a private web forum run by Manning Publications, where you can make comments about this book, ask technical questions, and receive help from both the author and from other readers. The Author Online forum can be found at www.manning.com/ProgrammingtheTI-83Plus/TI-84Plus. This page contains information on how to register on and use the forum, what kind of help is available, and the rules of conduct.

Manning's commitment to our readers is to provide a venue where a meaningful dialogue between individual readers and between readers and the author can take place. It's not a commitment to any specific amount of participation on the part of the author, whose contribution to the forum remains voluntary (and unpaid). We suggest you try asking the author some challenging questions lest his interest stray!

The Author Online forum and the archives of previous discussions will be accessible from the publisher's website as long as the book is in print.

You can also ask technical questions on the author's forum, Cemetech, which has a special subforum for this book at www.cemetech.net/forum/f/70 (or http://cemete.ch/f70). Appendix C lists many more online resources, including places to download and publish programs, development tools, and emulators.

About the author

Christopher Mitchell is a graduate student of computer science and electrical engineering, a teacher, and a recognized leader in the TI and Casio graphing calculator programming communities. Christopher started programming Logo and QBasic

when he was seven years old, taught himself TI-BASIC at the age of 13, and has since branched out into hardware and software development for many platforms. He is the graphing calculator community's most prolific author, with well over 300 completed programs. Today, Christopher hosts discussions and collaboration on calculator programs and projects at his website, Cemetech. Christopher is proud to be a born-and-raised New Yorker. He has bachelor's and master's degrees in electrical engineering from Cooper Union and is now pursuing a PhD in computer science at the Courant Institute of NYU.

About the title

While we refer to the calculator by its shortened name TI-83+/TI-84+ throughout the book in order to save space and avoid repetition, the official name of the calculator is TI-83 Plus/TI-84 Plus and we do use this name written out in full in the title and in other official references to the book or the calculator.

About the cover

Manning has a tradition of using illustrations from 18th- and 19th-century collections of regional dress customs on their covers. After feedback from many students in this book's target audience, however, an alternative was created for this book, combining classical art with the instantly recognizable outline of a TI-83+ graphing calculator. The final design on the cover of this book, refined through many creative iterations, is inspired by Leonardo da Vinci's "Vitruvian Man," in which the human figure is replaced with calculators.

Getting started with programming

Graphing calculator programming is a great way to dive into the world of programming, to learn more about your calculator, and to enhance your academic prowess and problem-solving skills.

The five chapters in part 1 immerse you in everything you need to know to start writing full, powerful programs. It begins with your first three graphing calculator programs, shows you how to input and output numbers and text, and teaches you conditional and structural commands. By the end of this part, you'll be able to write programs and games that can interact with the user, make decisions, display menus, perform calculations, and call other programs. In each chapter, you'll see examples large and small that you can test and play with to cement your understanding of each new concept, and you'll occasionally be challenged to write your own application.

Chapter 1 introduces your calculator as a math and programming tool. You'll see how closely it resembles a full computer, and you'll explore a Hello World program, a quadratic equation solver, and a guessing game. Chapter 2 shows you a systematic approach to writing, editing, and running programs on your calculator, then teaches you commands to interact with the user. Chapters 3 and 4 show increasingly complex program-flow tools, from performing comparisons and using the results to make decisions, to jumping from place to place inside a program, to creating loops and calling subprograms.

If you have a vested interest in learning other programming languages besides TI-BASIC, you'll find chapter 5 particularly enlightening. It takes you through the process of imagining, designing, writing, and debugging a program from start to finish in any language. In the process, you'll learn to design your programs' interfaces, sketch flowcharts of program structure, and turn those diagrams into code.

If you're comfortable using a graphing calculator for math and graphing and have previously used lists and matrices, then you can jump directly from chapter 1 to chapter 2. If you want to make sure you understand the nonprogramming basics, you should review appendix A for a crash course in using your calculator as a math and science tool before beginning chapter 2.

Diving into calculator programming

1

This chapter covers

- Why you should program graphing calculators
- How calculator programming skills apply to computer coding
- Three sample programs so you can dive right in

In the past 40 years, programming has gone from being a highly specialized niche career to being a popular hobby and job. Today's programmers write applications and games for fun and profit, creating everything from the programs that run on your phone to the frameworks that underpin the entire internet. When you think of programming, however, you probably don't envision a graphing calculator. So why should you read this book, and why should you learn to program a graphing calculator?

Simply put, graphing calculators are a rewarding and easy way to immerse yourself in the world of programming. Graphing calculators like the ones in figure 1.1 can be found in almost every high school and college student's backpack, and though few of them know it, they're carrying around a full-fledged computer. Directly on your calculator, with nothing else required, you can write games, math programs that will help you check your work, and science programs to solve hard problems. You'll learn to think like a programmer, to apply problem-solving skills

to surmount obstacles, and to optimize and stream-
line your software. But you might be asking yourself
why you should bother learning calculator program-
ming instead of starting with a computer language
like Java or Python or C.

The answer is that besides offering a simple yet
powerful way to get started with programming and
besides being a portable computer you can slip into
your pocket, your calculator will make it much easier
for you to learn computer programming. To a large
extent, you'll be applying the same set of critical
thinking skills to any programming language that you
write, and the TI-BASIC calculator language you'll
learn throughout this book is a rewarding and easy
way to learn those skills. By the end of this chapter,
you'll have already written three programs, including
a game and a math program.

Figure 1.1   Common Texas
Instruments graphing calculators,
the TI-83+ (left) and TI-84+ Silver
Edition; the lessons in this book
apply to these calculators as well
as the TI-83+ Silver Edition, the
TI-84+, the TI-Nspire with a
TI-84+ keypad, and, to a large
extent, the TI-83.

Just from using your graphing calculator for math,
you already know some programming. The math operations in TI-BASIC programs are
identical to the math operations you type at the homescreen, and with many opera-
tions, such as manipulating graphs, you can build off the skills you've already learned
using your calculator for school or work. The programming commands have names
taken directly from English, such as `Input`, `Repeat`, and many others. The calculator
even makes it easy to track down your programming mistakes, taking you directly to
errors it finds so that you can correct them.

In this chapter, you'll take your first programming steps, diving right in with your first
three calculator programs. After we discuss how similar your calculator and a computer

Why program, and why program calculators?

Programming is a fun and rewarding career or hobby. It's great to hone problem-
solving skills and to learn to think more analytically. It's gratifying to develop an idea
for a program and, after planning and hard work, to successfully bring that idea to
fruition. You may find that you enjoy the satisfaction of surmounting challenges, of
learning to optimize your programs to make them small and fast, and of sharing your
finished work with friends and with users around the world.

Programming calculators is a great pursuit on its own and will teach you most of the
skills you'll need to easily pick up computer programming languages. Many of the past
and present graphing calculator programming stars started as bored or curious stu-
dents and now have advanced degrees or high-paying jobs in programming and engineer-
ing. This book will teach you everything you need to know to think like a programmer,
instilling an intuition for translating an idea into a program and thinking your way
around challenges that you'll find useful in a wide variety of technical pursuits.

are, you'll meet your calculator's ancestors and proceed to your first program. You'll learn to display the text "Hello, World" on your calculator's screen and then create a math program to solve the quadratic equation and a number-guessing game. Ready? Let's get started!

1.1 Your calculator: the pocket computer you already own

To understand how a graphing calculator is a small, handheld computer and can be programmed to do many of things that a computer can be made to do, you must look at what exactly a computer is. The traditional idea of a computer terminal with a tower, a monitor, a keyboard, and a mouse is your first clue. A computer has input and output devices, a processor, and long-term and short-term storage, as you can see in the top half of figure 1.2. The dashed line indicates the portion of the computer inside the box on your desk, while the input and output devices are usually attached via cables. The processor at the center of everything mediates communication between long-term memory, short-term memory, and input and output devices. The role of each component is summarized in table 1.1.

As you can see in the bottom half of figure 1.2, a graphing calculator also contains these major blocks. Table 1.1 compares each aspect of a calculator with its computer counterpart.
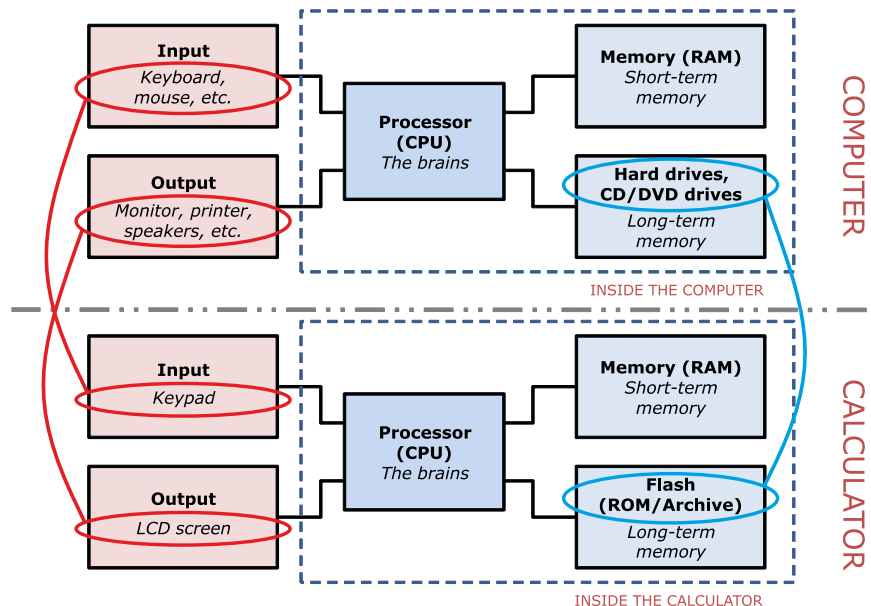
Figure 1.2 The basic building blocks of a computer and a calculator. Both have input and output; both have long-term and short-term storage. Both have a processor (CPU) that acts as the brains and mediates communication between the other pieces. The two types of devices are similar; the ovals highlight the main differences, such as that a calculator has flash memory for long-term storage instead of a hard drive.

Table 1.1   A side-by-side comparison of a calculator and computer

	Calculator	Computer
Input devices	Mouse/keyboard to control operating system and programs	Built-in keypad to control operating system and programs
Output devices	Monitor to display graphics and text	LCD screen to display graphs, text, and images
Inside the box	Power supply, processor, RAM (short-term storage), hard drive (long-term storage)	Batteries, processor, RAM (short-term storage), Flash/Archive (long-term storage)
When you run a program	Copied from hard drive to RAM, executed by processor from RAM	May be copied from Flash to RAM; run by processor from RAM

So why is a graphing calculator a computer, and a simple four-function calculator, like the cheap nongraphing calculator that you probably have in a desk drawer and that can only perform the simplest math, is not? The difference is that the simple calculator can only run its built-in software, which tells it how to do basic math. A graphing calculator also has a built-in OS that tells it how to do math, draw graphs, and store and recall variables but can accept brand-new programs that you or others create.

These programs can be loaded from other calculators, written on a computer using one of several applications designed for the purpose, or most conveniently and importantly, written by you, directly on the calculator itself. The programs you add to your calculator can do almost anything, including augmenting or supplementing the calculator's math and graphing tools and providing full suites for math, science, word processing, and more. Games can make graphing calculators a lot more fun: Arcade games, role-playing games (RPG), puzzle and board games, and thousands of others are possible with the calculator you have right now!

These are fun applications to write and use, but why bother writing calculator programs when you have more powerful computers available to you? Why learn to make programs that look good on a 96- by 64-pixel screen when even the lowest end modern laptop has 100 times as many pixels, or to fit a program into 20 KB of RAM when your computer has at least 50 million times as much? The answer is that calculators offer a much easier learning experience to budding programmers and a more rewarding challenge for seasoned coders. They'll give you a fun hobby, provide more control over your math and science tool, and can act as a stepping-stone to other programming languages. I'd like to introduce you to some of your calculator's extended family, including its shared ancestors with modern computers.

THE EVOLUTION OF THE MODERN GRAPHING CALCULATOR

The graphing calculator as a popular tool for math, science, and programming is now entering its fourth decade of widespread usage. Calculators for simple math gained public traction in the 1970s, and the first programmable calculators such as the TI-59 (programmed using punched cards) were produced in the late 1970s. But graphing

calculators are distinguished by having a much bigger screen, suitable for displaying graphs, and have much more powerful math and programming features than their nongraphing counterparts. Texas Instruments, currently leading Casio and HP in modern graphing calculator market share, released the TI-81 in 1990, with a 2 MHz processor and 2.4 KB (2400 bytes) of RAM. To put that in perspective, this paragraph up to the end of this sentence would already take 20% of a TI-81's memory. Other models were released in the following five years with gradually increasing capabilities. The TI-83+ (introduced in 1999) and TI-84+ (first available in 2004), were the predecessors to the TI-83+ Silver Edition and TI-84+ Silver Edition; I'll be focusing on these four models throughout the coming chapters.

The TI-83+, TI-83+ Silver Edition, TI-84+, and TI-84+ Silver Edition are similar calculators; their technical specifications are summarized in table 1.2. All four models run a Zilog z80 processor. They all have about 24 KB of RAM to store programs and data and between 163 KB and 1.5 MB of Archive, longer-term permanent storage. All four models have a 96- x 64-pixel monochrome (black-and-white) LCD screen. To put these sorts of technical specifications in perspective, a popular personal computer from 1982, the ZX Spectrum, had a 3.5 MHz z80 processor, between 16 KB and 128 KB of RAM, and used a TV as a 256 x 192 display. The Spectrum had about 20,000 software titles published for it, whereas over 38,000 programs and projects have been published for TI graphing calculators.

Table 1.2   **Specifications of the modern graphing calculators taught in the coming chapters. You'll need at least one of these to be able to follow along.**

	TI-83+	TI-83+ SE	TI-84+	TI-84+ SE
Zilog z80 processor	6 MHz	15 MHz	15 MHz	15 MHz
Screen	96- x 64-pixel monochrome passive-matrix liquid crystal display			
RAM	24 KB user/ 32 KB total	24 KB user/ 128 KB total	24 KB user/ 128 KB total	24 KB user/ 128 KB total
Archive/flash	163 KB user/ 512 KB total	1.5 MB user/ 2 MB total	480 KB user/1 MB total	1.5 MB user/ 2 MB total
Communication	9.6 Kbps serial	9.6 Kbps serial	9.6 Kbps serial, mini USB	9.6 Kbps serial, mini USB

> **TIP** You'll need at least one of the calculators in table 1.2 to work through the material in this book. It's recommended that you have a physical calculator, so you can work wherever the mood strikes you. But if you so choose, you could use one of the emulators listed in appendix C instead.

Specifications and numbers are all well and good, but they can't teach you nearly as much as getting your hands dirty with concrete examples. In the next sections, you'll work through your first programs: a Hello World program, a math program, and a game. You can type the code for each program directly into your calculator or read the descriptions and look at the screenshots to see some of the simplest (yet useful) programs your calculator can run. First up, Hello World.

1.2   Hello World: your first program

No instruction in a new language would be complete without plenty of well-annotated example programs to demonstrate each new concept learned. To jump directly into TI-BASIC programming, this section shows you the TI-BASIC version of the simplest program imaginable, universally called Hello World because it prints that phrase on the screen. I'll present an overview of the two major types of programming languages, interpreted languages and compiled languages, while showing you TI-BASIC, the language you'll learn in most of the coming chapters. You'll see the source code for the program, and I'll teach you how to test it on your own calculator. First, you need to know a few background details about the TI-BASIC language and how it compares to other languages you may know or have heard about.

1.2.1   Before you begin: notes on the TI-BASIC language

The programming language that's commonly known as TI-BASIC isn't officially called by any name by Texas Instruments itself and isn't technically a variant of the BASIC (Beginners All-Purpose Symbolic Instruction Code) language. But like BASIC, it's an interpreted language and shares many traits with that inspiration, so the name TI-BASIC has stuck.

Almost every language can be classified either as an interpreted or a compiled language; a high-level comparison of the two is provided in table 1.3 along with a few representative examples of each. See the sidebars "What's an interpreted language?" and "What's a compiled language?" for more details.

Table 1.3   Interpreted versus compiled programming languages

	Interpreted language	Compiled language
Execution speed	Slower	Faster
Preprocessing	None needed	Source code compilation
Syntax error checking	During execution	Before execution
Executed by	Interpreter program	Computer's processor
Examples	TI-BASIC, JavaScript, Java, Python	C, C++, Haskell, Fortran

For both types of languages, programmers type in the series of commands that will make up the program in a list of lines, a list called the program's source code. For both types, execution generally proceeds from the top of the program downward, although you'll see in section 1.4 and in later chapters how conditional commands, loops, and jumps can redirect execution.

> ### What's an interpreted language?
> A calculator or computer directly reads these programs, interpreting on the fly what the program will do. It reads each line of the program, figures out what that line is directing it to do, acts on it, and moves to the next line. If there are syntactical errors, such as sequences of commands that don't make sense, missing pieces of commands, and the like, the interpreter won't find these until it reaches the error while running the program. Interpreted programs are generally slower than compiled programs, because the interpreter must translate each line of the program into a form the computer's processor can understand and make sure the line has no errors before it gives that line of the program to the processor.

You'll now see the first of three TI-BASIC programs meant to immerse you in the basics of the language. I'll present the source code of a Hello World program and explain it. I'll walk you through the steps to type it and test it on your own calculator. If you have prior experience with TI-BASIC, some of the details in the coming examples may be extraneous, but you'll certainly still learn more about each command, its proper use, and special tricks and features of each as you read. Appendix A and the beginning of chapter 2 review using your calculator's menus and features, typing and editing programs, and other basic calculator skills, so don't worry if some of the concepts seem foreign. Let's jump into your first program: Hello World.

> ### What's a compiled language?
> A compiled program goes through an intermediate process called compilation before being run. A compiler's job is to take the code the programmer has typed and convert it into a program that can be run directly by the computer or calculator's processor before it's run. Because the compiler must examine a program for errors and translate it, much as an interpreter does, it can find some programming errors during the compilation process. After they're compiled, these programs generally run faster, because they're directly executed by the processor with no interpreter spending processor time.

1.2.2 *Displaying "Hello, World"*

The source code for the Hello World program in TI-BASIC is among the simplest programs you can write, consisting of a single line of code. In any language, Hello World

```
PrgmHIWORLD
HELLO, WORLD
                    Done
```

Figure 1.3   Output of Hello World program

is traditionally the first program presented, and it shows "Hello, World" or some variation thereof on the screen; our version of this program is shown in action in figure 1.3.

Even though it's a tiny toy program, it's useful for introducing the fundamentals of what a program is, how you create a program, and what happens when you run a program. Without further ado, here's the source code for Hello World.

```
PROGRAM:HIWORLD        ◁─────┤   Name of the program, not a line of code
:Disp "HELLO, WORLD"    ◁───────   A line of code, with a command (Disp)
                                   taking one argument ("HELLO, WORLD")
```

> **Why "HIWORLD", not "HELLO WORLD"?**
>
> In the example code, you can see that the Hello World program is named HIWORLD rather than HELLO WORLD, which seems a bit confusing. There's a good reason: calculator programs can have only uppercase names of at most eight characters, containing letters and numbers (but no spaces). Every program name must also start with a letter. Therefore, a name like HELLO or HELLOWOR or HIWORLD is allowed, but 1HELLO and HELLOWORLD and HELLO WORLD are all invalid.

The program shown consists of two pieces: the name of the program (the first line) and the source code for the program (in this case, the second line). Every command has a one-word name and takes zero, one, or more arguments. The command here is Disp, short for *display*, and instructs the calculator to display a line of text on the screen. I give it one argument here, the text to be displayed: "HELLO, WORLD." In programming parlance, a piece of text to be used or displayed is called a *string*. This line displays the string "HELLO, WORLD" on the screen. Notice that there's no explicit instruction telling the calculator to stop executing the program. Instead, whenever the interpreter reaches the end of a program, it takes that as an implicit command to end the program.

TYPING THE PROGRAM ON YOUR CALCULATOR

If you'd like to type this program into your calculator to try it, you'll first need to create a program named HIWORLD. Start at the homescreen of the calculator, the area where the cursor flashes, and where you can type math and perform the following steps:

1   Press [PRGM] to get to the Program menu, where you'll spend much of your time as you learn to program your calculator.

2   Press [▶][▶] (the right arrow key twice) to switch to the NEW tab, and press [ENTER].

3 The calculator will ask you for a name for your new program; you can type HIWORLD with the keys [^][x²][−][7][×][)][x⁻¹], the keys over which the letters H, I, W, O, R, L, D are written in green.

4 Press [ENTER] again to create a blank program with the name HIWORLD, as shown in figure 1.4.

You'll then be able to type lines into your program.

```
PROGRAM
Name=HIWORLD█
```

Figure 1.4   Creating a program named HIWORLD

Typing out tokens: Disp vs. "D" "i" "s" "p"

As I will remind you several times in your early experiences with TI-BASIC, commands are something called *tokens*, which means that the `Disp ` command is a single entity, not the series of characters "D," "I," "s," "p," and a following space. One important side effect of this is that you can't type out DISP as letters and expect it to work; the calculator won't understand what you're trying to do. You must use the tokens found in each of the menus.

To type the one line of code in this particular program after you created the new, blank program, continue to follow these steps:

5 Press [PRGM] from the program editor, which brings up a menu full of programming commands that you can use. It has three tabs, labeled CTL (Control), I/O (Input/Output), and EXEC (Execute program). You can press the left- and right-arrow keys to switch which of the three tabs is visible and the up and down arrows to scroll through each menu. You first need `Disp`, which is the third item in I/O. Press [▶] to go to the I/O tab; then press [▼][▼][ENTER] or just [3] to select 3:Disp. In every menu, you can either move the highlight over the number of the item you want and press [Enter] or press the number itself on the keypad, in this case [3]. This will paste the `Disp` command into your program.

6 After you have the `Disp` in the program editor, you need to type the string you want it to display. To type "HELLO, WORLD", you'll first need the quotation mark, [ALPHA][+]. HELLO is [ALPHA][^], [ALPHA][SIN], [ALPHA][)], [ALPHA][)], [ALPHA][7]. Notice that unlike a computer keyboard, you don't hold down [ALPHA] and tap the key from which you want a letter; instead, you press and release [ALPHA] and then press and release the other key. Chapter 2 will review typing and editing on your calculator if you're confused. The space character is [ALPHA][0]; see if you can find the letters for "WORLD" on your own, and don't forget the ending quotation mark. When you've finished, your program should look like figure 1.5.

```
PROGRAM:HIWORLD
:Disp "HELLO, WO
RLD"
```

Figure 1.5   The source of HIWORLD, on a calculator

Alpha and second functions of keys

To type out the letters in the string, you need to know that each key on the calculator has three functions, with a few exceptions. The normal function of each key is written on the key itself, such as [9] or [SIN] or [GRAPH]. Most also have a second function, accessed by pressing the [2nd] key followed by the key in question. Most have an alpha function too, found by pressing [ALPHA] and then the key. The second and alpha functions are labeled above each key and color-coded to match the modifier ([2nd] or [ALPHA]) key.

When you've finished, press [2nd][MODE] to quit the program editor and go back to the homescreen.

With the source code of the Hello World program entered into your calculator, you can now run it.

I made a typo! Now what?

If you make any errors while typing, press the [DEL] key, which will erase whatever is currently under the cursor (not before it, like Backspace on a computer keyboard). The normal mode of the program editor is Replace, which means if you move the cursor over a character or token and press something else, it will be replaced. If you instead want to insert at the current cursor position, press [2nd][DEL] to go into Insert mode. Once again, this will all be reviewed in chapter 2 as you begin to get more comfortable with entering programs and running them.

1.2.3   *Running the Hello World program*

Your experience with any programming language including TI-BASIC will be cycles of coding, testing, and fixing bugs and errors. If you're still in the program editor, press [2nd][MODE] to quit back to the calculator's homescreen so that you can run this program to test it. Press [PRGM] to open the Program menu, this time to the list of programs you have on your calculator. Notice that the function of the [PRGM] key depends on context. From the homescreen, it brings up a list of programs, whereas from the program editor, it shows a list of commands. A few of the other keys on the calculator have similar context-dependent functions.

Use the arrow keys to find HIWORLD in the list of programs, and press [ENTER]. This pastes `prgmHIWORLD` to the homescreen, a command to the calculator to run the program named HIWORLD. Press [ENTER] again to execute the command and run the program. You should see something like the screenshot back in figure 1.3, reproduced in figure 1.6.

When you run this or any other TI-BASIC program, the interpreter starts at the beginning of the program and executes each line sequentially unless instructed otherwise. For this program, it first sees the `Disp` command and knows the program wants to display something. You can display matrices, lists, numbers, or strings; the quotation mark immediately after `Disp` tells the calculator that you want to display a string, so it searches for the second, concluding quotation mark. Once the interpreter finds that second quotation mark, it knows

```
prgmHIWORLD
HELLO, WORLD
                    Done
```

Figure 1.6   The output of the Hello World program again, as in figure 1.3

what the string to be displayed is (HELLO, WORLD) and puts the string on the screen. It then goes to the next line of the program, but because there's no next line, the program ends. When a program ends and returns to the homescreen, the calculator almost always prints "Done," shown in the screenshot in figure 1.6.

LESSONS OF THE HELLO WORLD PROGRAM

The Hello World program is close to the simplest TI-BASIC program you can write, but it's a useful stepping-stone for becoming more familiar with entering and running programs, as well as getting a first glance at what happens when a program is run. You have now been introduced to the difference between compiled and interpreted programs; as you move into more complex programs in later chapters, you'll gain an insight into the strengths and weaknesses of TI-BASIC as an interpreted language.

How about a program that might be useful to you in math class?

1.3 *Math programming: a quadratic solver*

A quadratic equation solver is a great first math program and is often the first math application that budding calculator programmers teach themselves. The equation is universally taught in algebra or geometry classes when many students first receive their graphing calculators, and the program itself is short and simple. Most important, when you finish typing what in our case is a nine-line program, you have a tool that you can use.

If you're unfamiliar with the quadratic equation, it's a method to find the two roots of an equation in the form $ax^2 + bx + c = 0$, or the values of x that make the equation true given values for a, b, and c. Letters a, b, and c represent the three parameters to the quadratic equation in math notation, corresponding to variables A, B, and C in TI-BASIC. The quadratic equation is written as shown in figure 1.7, along with two sample sets of a, b, and c values.

To solve the equation, a simple program should ask the user for values for a, b, and c, store them in A, B, and C, then plug them into the equation in figure 1.7, and print the values found for x to the user. Unfortunately, there are complications. Suppose that a = 1, b = 4, and c = 5. When you do the math, you'll need to take the square root of $4^2 - 4(1)(5) = 16 - 20 = -4$. As you may know, taking the square root of a negative

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

SOLUTION WITH REAL ROOTS
a = 1, b = 2, c = 1

$$x = \frac{-2 \pm \sqrt{2^2 - 4*1*1}}{2*1} = \frac{-2 \pm \sqrt{0}}{2}$$

$$x = -1$$

SOLUTION WITH IMAGINARY ROOTS
a = 1, b = 4, c = 5

$$x = \frac{-4 \pm \sqrt{4^2 - 4*1*5}}{2*1} = \frac{-4 \pm \sqrt{-4}}{2}$$

$$x = -2 \pm 1i$$

Figure 1.7   The quadratic equation for finding the roots of $0 = ax^2 + bx + c$ (top). The samples show a = 1, b = 2, c = 1 (left), which yields a single real root, and a = 1, b = 4, c = 5 (right), which yields distinct but imaginary roots.

number yields an imaginary result. Therefore, if $4ac > b^2$, then both roots are imaginary, because $b^2 - 4ac$ will be negative and the square root of a negative number is imaginary. If $b^2 = 4ac$, then the equation has a double root, and if $b^2 > 4ac$, making $b^2 - 4ac$ greater than zero, the quadratic equation will have two distinct (different) real roots. A competent quadratic equation solver would detect double roots and imaginary roots and adjust accordingly. For your first math program, you'll write a simple solver that doesn't try to determine imaginary roots (but does warn you about the imaginary roots) and doesn't check for double roots. Once you're a few chapters in, you'll know enough to write a version of this program that has both of these features on your own.

1.3.1   *Building the quadratic solver*

The TI-BASIC code for the simple quadratic equation solver is presented here:

```
PROGRAM:QUAD
:Prompt A,B,C
:If 4AC>B²
:Then
:Disp "IMAGINARY ROOTS"
:Else
:Disp (⁻B+√(B²-4AC))/(2A)
:Disp (⁻B-√(B²-4AC))/(2A)
:End
:Return
```

Let's work through this program line by line so you can understand how it works. When the calculator executes a program, it starts at the beginning and works its way down line by line, so you're reading the program just as the calculator does. I'll tell you the key sequences used to type each line of code, so that you can test this program on your own calculator. You can also download the source code for all the programs in this book from the book's website, www.manning.com/ProgrammingtheTI-83Plus/TI-84Plus. As you progress through the book, you'll learn many commands; if at any point you need a quick reference, appendix B summarizes every TI-BASIC command you'll learn in this book.

```
PROGRAM:QUAD
```

This first line isn't a piece of the source code; it's the name of the program. The TI-BASIC editor displays the name at the top of every program, so I'll adopt the same convention. For most programs it doesn't matter what you call the program, as long as it's at most eight uppercase letters and numbers and starts with a letter. For obvious reasons, this program will be called QUAD. If you'd like to type this program into your calculator to try it, you should start by creating a program named QUAD. From the homescreen, press [PRGM], then [▶][▶] to get to NEW, and press [ENTER]. You can type QUAD with [9][5][MATH][X⁻¹], and then press [ENTER] again. I won't walk you through typing this program key by key, but I'll give you plenty of hints to help you type it yourself.

[Line 1] :Prompt A,B,C

The first line of code of this sample program is a `Prompt` command. This instructs the calculator to prompt, or ask, the user for three variables, named A, B, and C. As mentioned with the Hello World program, commands, sometimes also called functions, consist of a name and take one or more arguments. These arguments can be inside parentheses or, as with `Prompt`, placed after the command. This particular `Prompt` has three arguments, A, B, and C. Users will be asked to enter a number at each of the three prompts that will appear, one each for A, B, and C. If users type anything that isn't a number, the calculator will produce an error message. After this command runs, the three values the user entered will be stored in three variables, or memory locations, labeled A, B, and C. This will allow you to use these values later in your program by referring to the variable names A, B, and C. You can type this line with [PRGM][▶][2] [ALPHA][MATH][,] [ALPHA][APPS][,] [ALPHA][PRGM][,] [ENTER].

[Line 2] :If 4AC>B^2

The second line of the program is a conditional statement, indicated by the `If` at the beginning of the line. Every `If` is followed by a statement that must evaluate to either logical true or false. Obviously, true indicates that the conditional statement that follows is correct, and false indicates that it's incorrect. If 4AC is greater than B^2, then the statement 4AC > B^2 is true. Otherwise, it's false. Conditional statements dictate which pieces of the program, or code, are executed. If this case, the section between `Then` and `Else` is run only if the condition evaluates to true, whereas the section from the `Else` to the `End` is run only if the condition is false. From here onward, I'll assume that you understand that [ALPHA][MATH] types the A character and that [ALPHA][anything] types the letter printed at the upper right of the key. I'll therefore represent [ALPHA][MATH] as ["A"] from now on. You can type this line using [PRGM][1][4]["A"]["C"][2nd][MATH][3]["B"][x^2].

[Line 3] :Then

`Then`, therefore, marks the beginning of the code to run if the conditional is true, or if the roots will be imaginary because B^2 – 4AC will be less than zero. `Then` can be found under [PRGM][2].

[Line 4] :Disp "IMAGINARY ROOTS"

The fourth line draws or prints a string (a sequence of text characters) onto the screen. In this case, the `Disp` command writes text onto the homescreen. The string to be displayed is offset by quotation marks, so that the calculator knows where the string starts and ends and is `"IMAGINARY ROOTS"`. `Disp` can be typed by pressing [PRGM][▶][3], and the quotation mark is [ALPHA][+]. For this line, it's worth noting that [2nd][ALPHA] will "lock" the calculator in ALPHA mode, letting you type letters without needing to prefix each with [ALPHA], until you press [ALPHA] again. I'll remind you of this in chapter 2 in case you happen to forget.

```
[Line 5] :Else
```

`Else` on the fifth line marks the boundary between the true and false sections of code for the conditional on line 2. If, when executing the program, the calculator finds the condition on line 2 to be true, it will skip directly from the `Else` to the `End` and continue executing downward. If the condition is false, it will skip directly from `Then` to `Else` and execute the code in between the `Else` and the `End` instead of skipping it. You can find `Else` in [PRGM][3].

```
[Line 6] :Disp (⁻B+√(B²-4AC))/(2A)
[Line 7] :Disp (⁻B-√(B²-4AC))/(2A)
```

Lines 6 and 7 perform the quadratic equation, calculating the two possible roots. The calculator needs two separate equations because it doesn't know what ± is; you must explicitly tell it to calculate the + case and the – case. An important lesson from typing these two lines, something that trips up many beginner programmers, is that the negative sign, such as -B, isn't the same as the subtraction sign, as in 2–1. The negative sign is written here as a superscript to distinguish it, and is also sometimes shown as (-) in tutorials. It's typed with the [(-)] key, between [.] and [ENTER]. The subtraction sign is typed with the subtraction key [−]. The radical or square root symbol is [2nd][x^2]; notice that just as [ALPHA][key] types the item shown at right above the key, [2nd][key] types the item shown at left above the key.

```
[Line 8] :End
[Line 9] :Return
```

The final two lines end the conditional statement and then end the program. The `End` marks the end of the conditional that started with the `If` on line 2 and continued with the `Then` and `Else`. The `Return` tells the calculator that the current program has completed and that it can return you to the homescreen (or, as you'll later learn, to another program that called this program). `End` is under [PRGM][7], and `Return` is near the bottom of [PRGM], at item E. To save time, you can press [PRGM] and then [▲] until you reach it.

     If you are following along and typing in this program, you can now press [2nd][MODE] to quit to the homescreen. Press [PRGM] and scroll down to QUAD and press [ENTER] twice, once to paste `prgmQUAD` to the homescreen, which is an instruction to the calculator to run QUAD, and the second time to start running it. If everything went

```
:█rompt A,B,C          :Disp (-B-√(B²-4
:If 4AC>B²             AC))/(2A)
:Then                  :End
:Disp "IMAGINARY       :Return
 ROOTS"
:Else
:Disp (-B+√(B²-4
AC))/(2A)
```

Figure 1.8   Source code for `prgmQUAD` typed into a calculator

well, you should see the prompt `A=?` on the screen. If you have problems, double-check that your program exactly matches the code shown. Later in this book, I'll teach you everything you need to know about understanding the calculator's error messages and how they can help you quickly pinpoint your (or the calculator's) error in a program.

In case you're having difficulty entering the QUAD program on your calculator or it's not working properly, figure 1.8 shows the source code as it should look typed into a calculator.

Next, we'll look at testing the program.

1.3.2   *Testing the solver*

When you test the program, it will prompt you for three separate numbers, the values for the variables A, B, and C that it will plug into the quadratic equation. At each prompt, you can use the number, decimal point, and negative sign keys to type a value; then press [ENTER]. After you provide a value for C, the program will either display the two roots of the equation $ax^2 + bx + c$ or tell you that the roots are imaginary. In chapter 9, I'll expand this simple quadratic equation solver to solve for the imaginary roots and to detect when the equation has a double root, where both roots are equal.

You can see the results of testing the program for two samples sets of A, B, and C values in figure 1.9. In your journey through the coming chapters, you'll run into

```
prgmQUAD              prgmQUAD
A=?2                  A=?5
B=?1                  B=?4
C=?-3                 C=?3
                  1   IMAGINARY ROOTS
               -1.5                 Done
               Done
█
```

Figure 1.9   Two tests of the quadratic solver for different values of A, B, and C. The left image shows the roots of the equation $2X^2 + X - 3 = 0$, which are 1 and −1.5. The right image shows the roots of the equation $5X^2 + 4X + 3 = 0$, which are imaginary roots.

many more useful program examples, and you'll likely discover problems of your own that you'll be able to translate into programs.

Graphing calculator programming without games would be no fun. I'll present a variety of increasingly complex games that you can write as your programming skills progress, but to get you started, let's jump right into a simple guessing game.

1.4 Game programming: a guessing game

Although graphing calculators are math devices, and they can be used to write complex and powerful math programs, game programming is a perennially popular reason why students and other users start exploring calculator programming. Among the vast range and variety of games that can be written are everything from simple text-based RPGs to complex arcade and 3D games. Here, we'll start with a simple example, a number-guessing game in which the calculator picks a random number and then asks users to guess numbers until they find the correct one. To make it more fun, the program tells users whether to guess higher or lower and challenges them to try to find the number in the fewest possible guesses. This program is nonlinear: execution doesn't simply flow from the top of the program to the bottom and then stop. First, you'll see the source code for the program, and I'll explain it piece by piece. You'll then have an opportunity to type the game into your own calculator and play it.

As in the previous example, if you'd like to try this program on your own calculator, you should start by creating a program, perhaps called GUESS. I'll provide slightly less key-by-key detail on how to type in this particular program, hopefully giving you a chance to exercise the knowledge you gained from the Hello World and quadratic solver programs.

1.4.1 Guessing game source and function

The source code of the guessing game is 13 lines of code and fairly straightforward. It introduces two commands that haven't been presented previously, namely `randInt` and `Repeat`; it also is the first of our three examples to demonstrate the store ($\rightarrow$) operator. Glance over the following code and try to get a general idea of how it works. Although the commands are probably new to you, their names should give you a vague intuition into their function.

```
PROGRAM:GUESS
:randInt(1,50)→N        ◁——  Generate the number
:0→M                          the player will guess
:Repeat G=N                          ◁——  Repeat the loop from here
:Prompt G          ◁——                    to the End until G = N
:If G>N
:Disp "TOO HIGH"        Ask the player
:If G<N                 to type a guess
:Disp "TOO LOW"
:M+1→M             ◁——   Increment the number
:End                     of guesses used
```

```
:Disp "CORRECT AFTER:"
:Disp M
:Disp "GUESSES"
```

UNDERSTANDING THE PROGRAM

To get a basic idea of the flow of this program, look at figure 1.10. Although you might not be familiar with reading flowcharts, I'll try to explain how this shows what the program does by putting you in the shoes of the calculator (specifically, its interpreter). As the calculator, you start at the top of the program at the box labeled START in figure 1.10 and keep executing the lines in order unless told otherwise. Let's follow the figure: First, you pick a number at random, the number the player will be trying to guess. Next, you "Ask player for guess." There, you wait for the player to input a number, their guess for the secret number. When the player enters a number, you can then take one of three paths. If the player guessed correctly, take the path labeled "Correct" and display how many guesses it took the player to get the number; then stop. If the number is wrong, then display either "TOO HIGH" or "TOO LOW," and ask the player for another guess.

The arrows that make you, the calculator, return to the same point in the program over and over are part of what is called a loop. Notice that the arrows from "Ask player for guess" to "TOO LOW" and "TOO HIGH" then lead back to "Ask player for guess" and that if the player keeps making guesses that are too high or too low, the program will continue to cycle back to the oval in figure 1.10. In the code, the `End` command returns to the `Repeat` command, closing the loop that starts with the section of code between the `Repeat` and the `End`. This program uses the loop to make the program keep running until the player's guess is correct, so that the player will be able to keep guessing. Keeping the flowchart in figure 1.10 in mind, we'll now take a more methodical look through the source code of the program.

GUESSING GAME SOURCE CODE: A WALKTHROUGH

As I take you through the code for this program line by line, you may wish to type it into your own calculator so that you can test it. For each line, I'll provide extra tips about where to find commands or symbols that you haven't seen before.
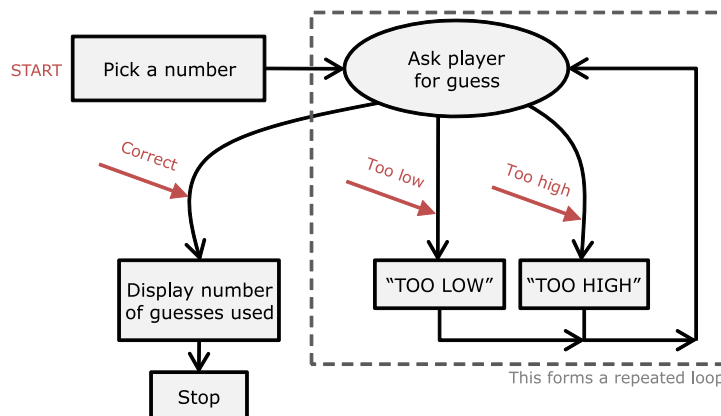
Figure 1.10 Guessing-game program flow. The program will keep displaying "TOO HIGH" or "TOO LOW" and returning to ask the player for another guess (a loop) until the player guesses correctly.

The first line of the program contains the `randInt` command, which takes two arguments. In this case, the two arguments are the minimum and maximum possible number that the function should generate.

```
:randInt(1,50)→N
```

As the name suggests, the `randInt` command generates a random integer (whole number) between and including 1 and 50. Figure 1.11 demonstrates that this random number is the number the player will guess.

Figure 1.11   The `randInt` command serves to pick a number to be guessed.

The small arrow indicates that the random number it returns should be stored into variable N, just as `Prompt N` would ask the user for a number and store it into N. In this case, the calculator generates the number that's stored into the variable, rather than asking the user to type it in. You can find `randInt` under [MATH]; then use the right- (or left-) arrow key to get to the PRB submenu, and choose `5: randInt(`. The closing parenthesis needed to complete the command is the key above the [9] key. You may also not know that the store operator ($\rightarrow$), used to update the contents of variables, is on the lower left of the keypad above the [ON] key, labeled the [STO>] key.

```
:0→M
```

Similarly, the second line stores a value to M, but this is simply a zero. The program will be using M to store the number of guesses that the user has made and N to store the target number that the user is trying to guess.

> **Choosing variables to use**
> It doesn't matter what variables a program uses to store numbers; in this example, the program could easily use A and B or S and T or C and Z instead of M and N. You have A–Z and many other variables available to you, and it's up to you as a programmer to choose variables that make sense to you. The only exception is the variable Y, which many programmers avoid because certain programming features change the value in Y as a side-effect.

```
:Repeat G=N
:Prompt G
:If G>N
:Disp "TOO HIGH"
:If G<N
:Disp "TOO LOW"
:M+1→M
:End
```

Repeat the loop from here to the `End` until G = N

The majority of the program is between the `Repeat` and `End` commands, which together enclose what is called a *repeat loop*. In a repeat loop, execution will repeatedly restart at the `Repeat` command every time the `End` command is reached, allowing the code inside

the loop to be run over and over again. The program eventually needs a way to get out of the loop; otherwise it'll be looping forever. The solution is a condition on the loop, just as If takes a condition. Notice the G = N condition on the Repeat; stated more verbosely, this Repeat command says "Repeat the loop from here to the End until G = N." That means that as long as G isn't equal to N, the loop will continue, but when G becomes N, which means that the user finally guessed the number correctly, the loop will end. You'll learn more about Repeat and its cousins While and For in chapter 4. The greater than, less than, and equals symbols are all in [2nd][MATH], the Test menu. Repeat is [PRGM][6].

```
:Prompt G
```

You have previously seen the Prompt command, which will ask the user to enter a value for G (their guess) and then store it into variable G.

```
:If G>N
:Disp "TOO HIGH"
:If G<N
:Disp "TOO LOW"
```

The next four lines will display a message to the user based on the guess that the user entered. If the value for G is larger than the target number (N), the program will display "TOO HIGH." If it's smaller than the target number, the program will display "TOO LOW." Figure 1.12 shows the parts of the diagram representing Prompt and these two conditional constructs.

Notice that if the user guessed the correct number, then both G > N and G < N are false, and nothing will be displayed.

```
:M+1→M
```

Execution will continue downward, taking the value in M, adding 1 to it, and storing this value back into M. As we discussed, M contains the number of guesses the user has made so far and is incremented each time the Repeat loop is run because the loop runs once per guess.

```
:End
```
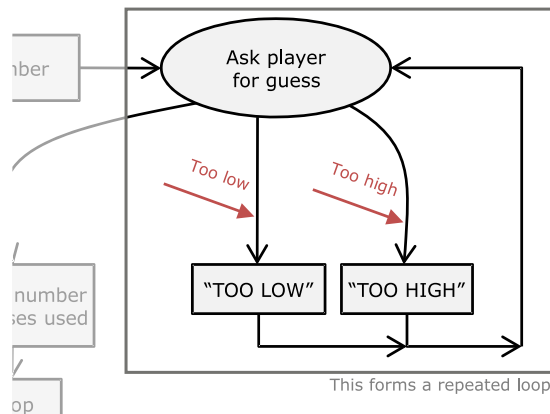

Figure 1.12   The loop that alternates between asking players for guesses and telling them whether the guesses are too high or too low

The next command read and executed is `End`. `End` triggers the interpreter to reexamine the `Repeat` condition to decide whether to go back and start executing at the `Repeat` again, prompting for another guess, or continue directly to the code under `End`, finishing the looping process. If the condition is true, that is, G = N, then the code after `End` will be executed. If the condition is false, or G isn't equal to N (written as G ≠ N), then the user has still not guessed the correct number, and execution will return to the `Repeat`, `Prompt` section of the code.

```
:Disp "CORRECT AFTER:"
:Disp M
:Disp "GUESSES"
```

The last three lines of the program tell the user that they guessed correctly and also display the number of guesses that the user made before finding the correct answer, as shown in figure 1.13.

Notice that although you have only seen `Disp` used with strings so far, `Disp` can also be used to check what number is currently stored inside a variable, in this case M. As with the Hello World program, you don't need to explicitly add the `Return` at the end of the program to tell the calculator that the program has reached its end; instead, the end of execution is signaled by the end of the program file.

With a full understanding of how the program works, you should now try out the program on your own calculator to see it in action.

Figure 1.13 **Displaying the notification that the player won and how many guesses they needed**

TYPING AND TESTING THE PROGRAM

If you haven't done so already and would like to type this program into your calculator and try it out, you already know where to find many of the commands, such as `Disp`, `End`, `Prompt`, and `If`. Throughout the discussion of the source code, I provided tips about where to find the commands you saw for the first time in this program. As with the two previous examples, you can run `prgmGUESS` by quitting to the homescreen, finding GUESS in the Program menu, and pressing [ENTER] twice. Figure 1.14 shows an example of a game where the player got the correct number in four guesses.

As I discussed in walking through the program's code, the program will pick a random number and repeatedly wait for the user to guess a number and tell them if it's higher or lower than the target number. Once the user guesses the correct number, the program will display the number of guesses made and exit.

```
Pr9mGUESS          G=?19
G=?25              TOO LOW
TOO HIGH           G=?22
G=?13              CORRECT AFTER:
TOO LOW                        4
G=?19              GUESSES
TOO LOW
G=?22                     Done
```

Figure 1.14 **Playing the guessing game to the correct answer of 22 in four guesses**

1.4.2 Lessons of the guessing game

One of the most important lessons of this particular example is that programs can have arbitrarily complex flows of execution depending on the user's or player's input. Put more simply, what happens inside the program and which pieces of the program are executed in which order are often based on user input. If the user guesses the correct number right away, the program will only go through the Repeat/End loop once and will never loop back to the Repeat. If the user guesses the same incorrect guess over and over, the program will keep looping over and over. Depending on whether each guess a user makes is higher or lower than the target value, one of two possible Disp commands is executed. Indeed, complex games and utilities can be built up from simple pieces such as loops and conditionals combined with concepts you'll learn later such as subprograms and jumps.

1.5 Summary

Perhaps by now, after seeing example programs in action, you've started to get a vague understanding of the general programming process, but if not, don't worry; I'll be covering all the lessons of this chapter in more depth in later chapters. Graphing calculator programs can be as complex or as simple as you want and are limited only by your imagination and problem-solving skills. As a temptation to motivate you to continue your calculator programming journey, enjoy the screenshots in figure 1.15 of various programs and games for the TI-83+/84+ series of graphing calculators that have been written in TI-BASIC, the main language you'll be learning, some created by the author and some written by other members of the graphing calculator enthusiast community.

Chapter 2 will introduce input and output commands, which you've seen briefly in the three examples in this chapter. Starting in chapter 2, you'll be diving right into programming, with the assumption that you're fairly comfortable with using a graphing calculator. If you're not, you should review general calculator features, including
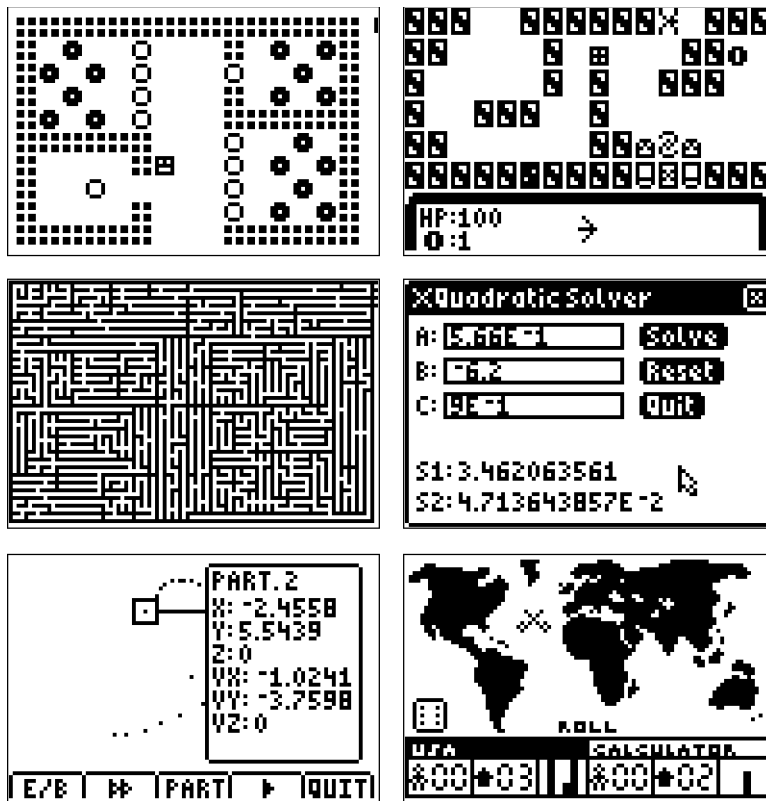
Figure 1.15   Screenshots from assorted examples of completed and freely available TI-BASIC calculator programs and games. Top row: "Donut Quest II" by Mikhail Lavrov, "Midnight" by Zachary Tuller. The middle and bottom rows both contain screenshots from programs by the author, Christopher "Kerm Martian" Mitchell. Middle row: "DFS Maze Generator" and "DCSQuad Solver"; bottom row: "ParSim Particle Simulator" and "World Domination I."

numeric functions and graphing skills and the types of variables and data that can be stored. These are discussed in appendix A. If you're shaky about the nonprogramming features of your calculator, I recommend that you peruse appendix A before getting too far into chapter 2. Onward to programming basics!

Communication: basic input and output

An important part of any program is interacting with the user. For almost any program or game you might want to create, you'll need to accept numbers, strings, or input from the user, display or draw output back to the user, or both. This chapter will teach you how to get strings and numbers from the user and write strings and numbers back.

As you learned in chapter 1, your calculator shares its architecture with any common computer. You may recall that calculators and computers both have connected devices called input and output devices, as shown in figure 2.1. For computers, things like the monitor, the speakers, and the printer are the output. They're devices the computer can instruct to communicate something to the user. Computers' input devices, such as the keyboard, the mouse, and a microphone, let the user talk back
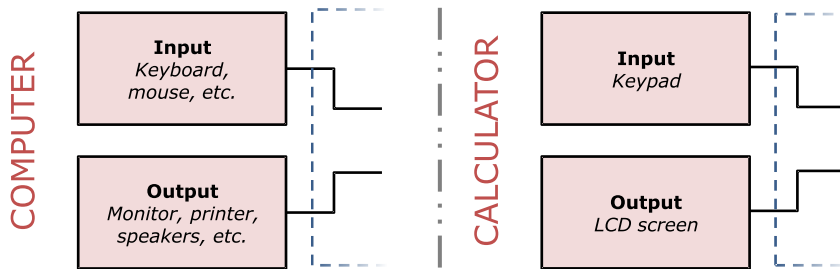
Figure 2.1   The input and output devices of a computer (left) and a calculator (right), excerpted from figure 1.2

to a computer. In much the same way, calculators have input and output devices for communication with the user: the keypad gives the user a way to instruct the calculator; the LCD screen lets the calculator show things back to the user.

In this chapter, you'll be learning the rudiments of reading input from the keyboard and writing output back to the screen. In later chapters, you'll learn more complex types of input and output, such as checking to see if specific keys were pressed or drawing a line on the screen. Imagine any simple math program, such as the quadratic solver that you created in chapter 1. It lets the user type in a few numbers, performs simple math on the values, and displays results to the user. In this chapter, you'll learn to construct similar applications. You'll write an animation, a program to raise a number to a power, and a program to calculate the slope of a line, among many others. You'll even learn about accepting strings from the user, which will give you the skills to write a program to converse with your calculator. In the hope that you'll start playing around with a few programs of your own as soon as possible, I'll teach you the basics of troubleshooting errors or bugs in your programs.

The title of this chapter refers to "input and output," a standard programming phrase, but you'll learn how to output text and numbers to the screen first. If you learned input first, your program would be able to accept input from the user but wouldn't be able to display anything back to the user to show that it had been able to do something useful with that input! Therefore, you'll learn output first, so that when you learn input, you can output something based on the numbers or text the user just provided as input.

Before any of this, you need to meet the program editor and the homescreen, the pieces of your calculator's built-in software that you'll be using in this chapter.

2.1   *Getting to know the program editor and homescreen*

As someone who has used a graphing calculator to some extent for math and graphing, you should have the vague impression that there are different areas of the calculator's built-in software, called its operating system, or OS. There's the portion where

you type math and get results, there's the screen where you type equations to graph, and then there's where you see your graphs drawn and can examine them. You may have even encountered menus such as the Statistics menu, the Window or Zoom menu, or the Memory menu. This and all the subsequent chapters will be largely spent in the program editor, where you type in the TI-BASIC source code for programs to run on your calculator.

We'll begin with a look at the program editor, the section of your calculator's OS you use to write software on the device itself. This section is about you, the programmer, typing source code into a program, which in a sense is input. But this isn't the section on the input named in the chapter's title; that input, where the user types data that your programs can then use, will be covered in section 2.3.

2.1.1 *The program editor: typing source code*

You first saw the Program menu and the program editor in chapter 1. You can access the Program menu by pressing the [PRGM] (program) key from the homescreen, the graph screen, or from almost anywhere else within the calculator's OS. The Program menu holds three tabs labeled EXEC, EDIT, and NEW, as shown in figure 2.2.

You'll find yourself using each of these three tabs often throughout the coming chapters; the function of each is summarized in table 2.1.

Figure 2.2 The three tabs of the TI-83+/84+ calculator's Program menu. The first tab, EXEC, lets you run programs. The second, EDIT, allows you to edit a program's source code. The third tab, NEW, provides a way to create a new program.

If you choose either to edit some existing program or create a program, you'll find yourself within the program editor.

Table 2.1 The functions of the EXEC, EDIT, and NEW tabs of the Program menu, along with what happens when you hit [ENTER] in each tab

Menu tab	Menu contains…	When you hit [ENTER] in it…
EXEC	A list of all programs on your calculator	Pastes the name of the currently selected program to the homescreen. Press [ENTER] again to run it.
EDIT	Same contents as EXEC	Opens the program editor so you edit the currently selected program. You must unarchive archived programs (marked with a *) to edit them.
NEW	A prompt to let you start a new program	You enter the name of your new program and then hit [ENTER] again to enter the program editor.
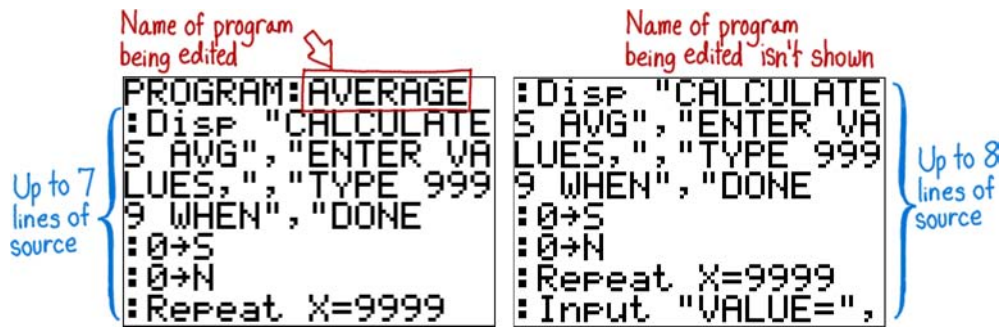
Figure 2.3   The view inside the program editor when you open `prgmAVERAGE` (see section 4.3.3) in the program editor. On the left is the standard view, with the program's name on the top and the source code in the bottom seven lines. If you install the Doors CS shell (see appendix C) and use it to edit programs, you get to use all eight lines to edit, as shown at right, but you don't get to see the name of the program.

INSIDE THE PROGRAM EDITOR

The program editor is the area you use to type in the source code for your program. It contains up to eight lines of text, which normally consist of one line showing the name of your program and seven lines of source code, as shown at the left side of figure 2.3. If your program is more than seven lines long, you can use the up- and down-arrow keys to move throughout the code.

As you type the programs in this and coming chapters into your calculator, and when you begin to make your own programs, you'll spend a great deal of time inside the program editor. To use it effectively, you'll need to know how exactly to type the different commands, letters, numbers, and symbols that make up the source code of any TI-BASIC program.

How do I save?

In text editors on your computer, you create or open a document, make changes, and then save. The program editor on your graphing calculator is different. Each change you make is instantly saved, so when you're finished, you just quit. You also can't undo any changes, so be careful using the [CLEAR] key.

UNDERSTANDING KEYS AND THEIR FUNCTIONS

The program editor is similar to any text editor you may have used on a computer. You use the keys on the calculator's keypad to type in numbers, letters, and symbols. As I explained in chapter 1, each key has up to three functions. When you press the key, you perform whatever function is printed directly on the key itself, which in some cases brings up a menu and in other cases types a token or a character. Table 2.2 shows how to access the (up to) three different functions for each key.

The keys and their functions are more or less the same throughout the entire calculator. You can use [2nd] and [ALPHA] the same way regardless of whether or not you're

Table 2.2   Accessing the different functions that each key on your calculator can perform inside the program editor. Notice that for the key sequence, you press and release the first key before pressing and releasing the second key. You don't hold the two keys together, as you might with the [Shift] or [Ctrl] key on a computer.

Key sequence	Example	Function
[KEY] KEY	[SIN] types the `sin(` token into your program. [STO>] pastes the → symbol into your program. Pressing [MATH] opens up the MATH menu full of more commands and symbols. [CLEAR] erases the entire line that you're typing.	Insert the symbol or number printed on the key, or open the menu named on the key, depending on the key.
[2nd][KEY] 2nd KEY	[2nd][(] types an opening curly brace, {. [2nd][^] produces the pi symbol, π.	Press the yellow or blue [2nd] key followed by another key to type the symbol or open the menu named in yellow or blue above the key.
[ALPHA][KEY] ALPHA KEY	[ALPHA][PRGM] types a C. [ALPHA][LN] types an S. [ALPHA][0] types a space.	You can type A–Z, theta (θ), ", :, ?, and space by pressing the [ALPHA] key followed by one of the other keys.
[2nd][ALPHA] 2nd ALPHA	[2nd][ALPHA][PRGM][MATH][)][PRGM] types the word CALC. Saves you typing [ALPHA][PRGM] for C, [ALPHA][MATH] for A, [ALPHA][)] for L, and finally [ALPHA][PRGM] for C.	Lock ALPHA mode until you press [ALPHA] or an arrow key. Lets you type a series of letters without pressing [ALPHA] before each one.

in the program editor, so what do you need to know that's specific to the program editor? You need to know where to find all of the commands used in programming.

> ### Tokens versus text: typing commands in the program editor
> As mentioned in chapter 1, you don't type out each command letter by letter. If I tell you to use the `Prompt` command in a program, you don't merrily turn on Alpha Lock and start typing P-R-O-…. Instead, you find the token that represents this command by pressing the [PRGM] key again while inside the program editor. Once you find the token you need, press [ENTER] to paste that token into your program. You can check if something in your program is a token by trying to move the cursor over it. If it jumps from the beginning of the command to the end (from the P in `Prompt` to the end of the word, for instance), it's a token. If you can instead put the cursor over each letter in the word, it's separate letters and won't work as a command.

Typing commands into programs

When you're in the program editor, the meaning of the [PRGM] key changes and won't show the set of three tabs, EXEC, EDIT, and NEW, that you see when you press

```
▮▮▮ I/O EXEC        CTL ▮▮▮ EXEC
1▮If                 1▮Input
2:Then               2:Prompt
3:Else               3:Disp
4:For(               4:DispGraph
5:While              5:DispTable
6:Repeat             6:Output(
7↓End                7↓getKey
```
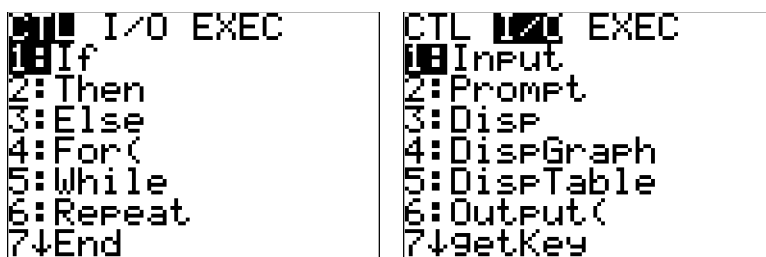
Figure 2.4   CTL and I/O, the first two tabs of the PRGM command menu, accessed by pressing [PRGM] from inside the program editor. The CTL tab contains control-flow commands, which you'll learn in chapter 3. This chapter will cover some of the input and output (I/O) commands shown in the I/O tab. I don't show you the EXEC tab because its contents are different on each calculator: it lists all of the programs your calculator holds, like the [EXEC] tab of the original Program menu.

the [PRGM] key from the homescreen. When you press [PRGM] from inside the program editor, it will instead show a different three-tab menu, the first two tabs of which, CTL and I/O, are displayed in figure 2.4. The EXEC tab contains a list of the programs on your calculator, which differs from calculator to calculator, so I don't show you that tab. In section 4.4, I'll show you how to use the contents of the EXEC tab.

You can scroll up and down each of the three tabs with the up- and down-arrow keys and move between the tabs using the left and right arrows. When you find the command you want, press [ENTER] to paste it into your program or press [CLEAR] to go back to the program editor without pasting any of the tokens. Table 2.3 explains what each tab is for.

CORRECTING MISTAKES

Typing out programs line by line would be straightforward if humans were perfect, but we're not. We need a way to correct our mistakes, to remove or add pieces to the

Table 2.3   The three tabs of the PRGM command menu and what each tab contains and is used for

PRGM command menu tab	Contains
CTL	The CTL tab contains control-flow commands, which you'll learn all about in chapter 3. These let your program run in loops, jump around your code, call other programs, and more.
I/O	The I/O tab contains input and output functions that let your program interact with users. You'll learn how to use some of these commands in this chapter.
EXEC	This tab lists all the programs on your calculator. Choosing a program named NAME from this menu pastes prgmNAME into your program; in section 4.4 you'll learn how this calls prgmNAME from inside your current program and how that's used.

program later, to insert text between existing text, and to create new lines or remove them. You might even find yourself wanting to paste the contents of one program into another. Luckily, the calculator's program editor has easy ways to perform all of these tasks.

The [DEL] key will delete whatever is directly under the cursor, be it a token or a letter or symbol. You'll notice that when you delete a token, the entire token is deleted at once, instead of letter by letter.

You can clear the contents of an entire line with [CLEAR], which you'll probably soon learn the frustration of doing by accident.

On the flip side, you can insert text in the middle of lines. Press [2nd][DEL] to go into Insert mode: the cursor will change from a solid black box to an underscore. Now, when you type, the things you type will be inserted instead of replacing what was already there. You can insert new lines in the middle of your program by entering Insert mode and pressing [ENTER]. To leave Insert mode, press any of the arrow keys, and the cursor will change back to a black rectangle.

COPYING CODE

Now you'll learn how to paste the contents of one program into another. It's a trick used to rename a program: create a program with the new name, paste the contents of the program with the old name into the new program, and delete the old program from the calculator's memory menu. You can also use it to prototype a piece of a program in a small program, test it out separately from the main program, then paste it into your program when you know it works without needing to type it out again. To paste a program into another program, follow these steps:

1. Switch your editing mode from the normal Replace mode to Insert mode by pressing [2nd][DEL] (remember, [2nd] and then [DEL], not [2nd] and [DEL] together).
2. Next, activate the Rcl (Recall) function by pressing [2nd][STO>].
3. Press [PRGM], move to the [EXEC] tab, and choose the name of the program that you want to paste in.
4. Hit [ENTER] to perform the Recall.

As well as adding code to programs and copying and pasting code between programs, you also occasionally need to delete old programs that you don't need any more or that you merely created as an experiment.

DELETING A PROGRAM

Although it's not strictly a task of the program editor, sooner or later you will want to know how to delete a program. After creating or editing programs, you may find your PRGM menu clogged with small test programs that you no longer need and that you'd like to clean off of your calculator. Deleting a program involves entering the Memory menu, which you may have encountered in your preprogramming use of your calculator:

1 Quit to the homescreen if you're not already there with [2nd][MODE].
2 Press [2nd][+] to open the Memory menu, since the [+] key has MEM listed as its [2nd] function.
3 Choose 2:Mem Mgmt/Del…, 7:Prgm…; move the arrow down to the program you want to delete and press [DEL].
4 Once you confirm that you do indeed want to delete that program, the calculator will delete it.

Programs that have been deleted can't be recovered, so make sure that you're sure you don't need a program any longer before you remove it. If your calculator is getting full, I recommend you back up old programs to your computer (see section A.6) before you delete them.

Don't worry if you didn't absorb all this information about using the program editor. As you type out the programs in this chapter and then start to write your own programs, you'll swiftly become accustomed to using the editor. It's intuitive, and even if you make errors, there's little you can do to damage your calculator. As you work through the many programs presented in every chapter and write your own programs, you'll spend a great deal of time inside the program editor, and you'll gain proficiency in typing more quickly, remembering where to find each command, and moving around your code.

Now that you've learned the basics of using the program editor to type programs, I'll introduce you to the homescreen. Just as the program editor is where you'll create programs, the homescreen is the area where your program will interact with users as it runs by displaying output and accepting input.

2.1.2 *The homescreen: your canvas for input and output*

You already know the homescreen as the place where you do math and, if you've ever run programs on your calculator, as the place to where you paste programs' names from the [PRGM] menu in order to run them. What you'll learn in this chapter is how to display text and numbers on the homescreen and how to get the user to type in text and numbers for your program to use. The homescreen itself is an array of characters, 8 rows tall and 16 columns wide, as shown in figure 2.5. You can use it one line at a time, such as when you perform math calculations; you'll also learn how to individually change each of the 128 spaces ($16 * 8 = 128$) in this chapter.

CLEARING THE HOMESCREEN

It's fine to display text, numbers, equations, and the like on the homescreen, but you'll occasionally want to start with a clean slate. To remove everything on the homescreen, there's a simple command called `ClrHome`. It will erase all 128 spaces in the homescreen and return the cursor to the top-left corner of the homescreen. If you then use functions to display text or ask for user input, those new contents will appear on this now-blank homescreen.

To demonstrate the effects of the `ClrHome` command, you can try the following program, CLRHOME. This may be the simplest TI-BASIC program you'll ever try, containing a
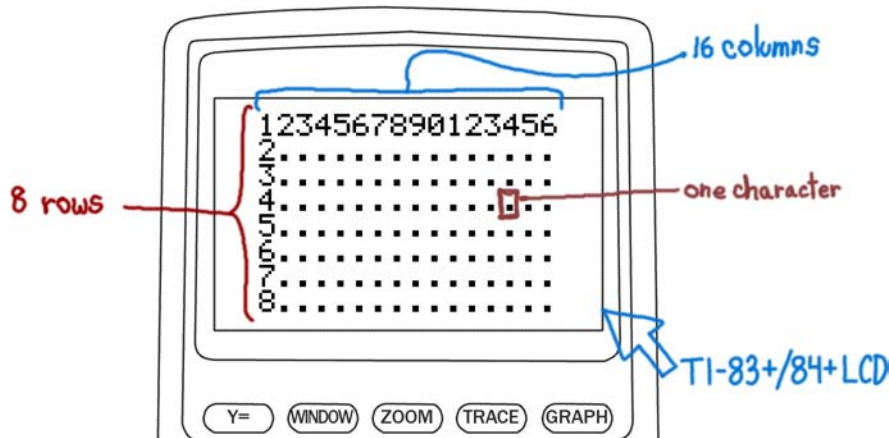
Figure 2.5   The 16-column, 8-row layout of the homescreen contains 128 total characters, each of which can be letters, numbers, symbols, or blank. In this chapter you'll learn to manipulate the contents of the homescreen.

single command with no arguments. When you run this program, the only thing it will do is clear the contents of the homescreen. To enter this into your calculator, you need to know that the ClrHome command is under [PRGM][▶][8] or [PRGM][▶] 8:ClrHome.

```
PROGRAM:CLRHOME
:ClrHome
```

Unfortunately, because every program ends by displaying "Done" on the homescreen, aligned at the right edge of the screen, the homescreen won't be completely blank after you run this program. Figure 2.6 shows two screenshots, the first taken before the CLRHOME program is run, the second after.

You now know what the homescreen is, that you'll be able to use it to communicate with the user in your programs, and how to clear it. The programs you will examine, write, and test in the remainder of this chapter, as well as chapters 3 and 4, will operate entirely on the homescreen, so you'll continue to build your familiarity with it as you
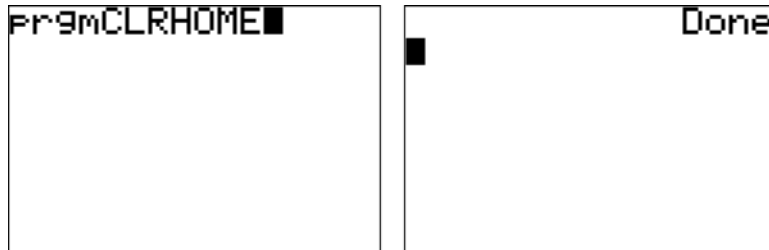
Figure 2.6   Before (left) and after running `prgmCLRHOME`. Notice that the program clears everything, including the line typed to make it run. Because "Done" is always displayed after any program finishes, that text remains on the screen when `prgmCLRHOME` ends.

read on. Now you need to learn your first real set of calculator programming skills: displaying text on the homescreen.

2.2   *Output: displaying text*

Almost any program needs to display something back to the user to provide feedback on what's happening inside the program. This might range from something as advanced as a graphical part of a game to something as simple as the numeric result of a calculation. I'll start by teaching you the simpler end of the spectrum, displaying numbers and strings on the homescreen; in chapter 7, I'll introduce programs with more game-like graphical abilities. In this chapter, you'll learn to display output on the homescreen in two stages:

1 I'll show you how to use the `Disp` (Display) command, which makes it easy to display text and numbers but doesn't let you fine-tune where the output appears. Your calculator will show each thing you display on a new line on the screen.
2 I'll explain the `Output` command, which adds the ability to place the output text or numbers at specific locations on the screen.

I'll teach you how you can use these newfound skills to make your own fun or useful programs, starting with `Disp`.

   This chapter has "input and output" in the title, and if you're familiar with computer hardware or software, you have heard the abbreviation I/O, which means the same thing. But I'm presenting the output concept first and input later, because without a way to produce output, your program has no way of showing the user what it has done with the input the user provided to the program. Without further ado, let's look at the `Disp` command.

2.2.1   *Displaying text and numbers on the homescreen*

The homescreen will be your primary palette for displaying text, numbers, and symbols over the next two chapters, and we must start somewhere: `Disp`. The `Disp` command can perform a surprising number of different tasks for you, including displaying a line of text, a number, a string, a list, a matrix, or even multiple such items together. You first saw `Disp` in action in the Hello World program in chapter 1. Recall from the GUESS game that you could put a line of text in quotes after the `Disp` command, and that line of text would be written to the screen:

```
:Disp "TOO LOW"                    ⟵──────────────      Displays "TOO LOW" on the screen
```

Each time you use `Disp`, the cursor advances down one line, even if you can't see the cursor itself; when the eight lines of the homescreen are full, the whole screen scrolls upward to make room for more contents. From the Quadratic Solver program, you also saw that you could put a number after the `Disp` command to make it display that number (or the results of a calculation):

```
:Disp (−B+√(B²−4AC))/(2A)          ⟵──────            The numeric result of this calculation will
                                                       be displayed, not the equation itself
```

With those two uses of the `Disp` function in mind, take a look at the HIWORLD2 program at the end of this paragraph. It uses the `ClrHome` command you just learned to move the cursor to the top of the screen and clear everything off the homescreen; then it displays three lines on the screen using three subsequent `Disp` commands:

```
PROGRAM:HIWORLD2
:ClrHome
:Disp "HELLO, WORLD"
:Disp 1337.42
:Disp 6*9
```

This demonstrates first a line of text ("HELLO, WORLD") being displayed, then a number (1337.42), and finally the result of a calculation (6 ∗ 9 = 54). The code for the HIWORLD2 program can be seen typed into the program editor at the left side of figure 2.7, whereas the output of the program is shown at the right side of figure 2.7.

One important item to note is that `Disp` performs slightly differently on strings and numbers. For anything that's a number, including a literal number like 1337.42 or the result of a calculation like 6 ∗ 9, that line is aligned to the right edge of the screen. For anything that's a string, the line is aligned to the left edge of the screen. In addition, there's no way to put a string together with a number on the same line with the `Disp` command, although the `Output` command in the next section will provide a solution.

`Disp` is easy to use and straightforward, but it still has a few tricks up its sleeve that you as a programmer can use to make your programs smaller and faster, such as displaying multiple numbers or strings with a single `Disp` command, which is the next concept I'll teach you.

DISPLAYING MULTIPLE ITEMS

I've previously discussed, and will continue to remind you throughout the coming chapters, that optimization is important on any platform. Whether you're writing a calculator program, a computer program, or a program to run on a smartphone, you should always be aware of how you can make your program be as small and as fast as possible. The `Disp` command gives you a way to optimize by allowing you to combine

Figure 2.7   **The source code (left) and output when the program is run (right) of a more complex Hello World program that clears the homescreen, displays "Hello, World," and then displays two numbers.** `Disp` **is capable of performing math on numbers before it displays them; note that** `Disp 6*9` **produces "54" rather than "6*9."**

> ## The homescreen and the MathPrint OSs
> In early 2010, Texas Instruments released an overhauled version of the TI-84+/Silver Edition operating system called 2.53 MP (MathPrint). It adds so-called pretty printing to the homescreen to make typed equations more closely resemble what you might see on a textbook page: exponents appear raised, fractions are displayed with a distinct numerator and denominator, and more. Unfortunately, the OS was poorly tested and is buggy and slow. A successor, 2.55MP, was also released, but it fails to fix many of the problems of 2.53MP. If you have a calculator with a MathPrint OS, your BASIC programs will run noticeably slower unless you use the `CLASSIC` command at the beginning of your programs to turn off the MathPrint features. If you want to turn them back on, use the `MATHPRINT` command. These two commands are only found on MathPrint OSs and can be pasted from the [MODE] menu or the Catalog, [2nd][0].

multiple `Disp` commands. You can use a single `Disp` command to display several lines of text, numbers, or some mix thereof by providing each item to display as an argument to the same `Disp` command and separating each argument with commas. The output of such a command might look something like figure 2.8.

To see this technique in action, consider the following program, which uses four `Disp` commands to display three strings and one number as in figure 2.8:

```
PROGRAM:DISPDISP
:Disp "WHEN YOU MULTIPL"
:Disp "Y SIX BY NINE,"
:Disp "YOU GET:"
:Disp 6*9
```


Figure 2.8   Adding multiple arguments separated by commas to a `Disp` command to display multiple lines. The DISPDISP and DISP4 programs produce the same output, shown in this screenshot.

Program DISPDISP contains four `Disp` commands. But this will produce precisely the same output as if you had used a single `Disp` command containing three strings and one mathematical expression separated by commas:

```
PROGRAM:DISP4
:Disp "WHEN YOU MULTIPL",
    "Y SIX BY NINE,",
    "YOU GET:",6*9
```

In either case, the program will produce the output seen in figure 2.8, consisting of four sequential lines, three of text at the left margin and one number at the right margin. You can mix any set of strings, numbers, and even lists and matrices within the same `Disp` command, in any order, to produce this sort of a result.

Now you know the basics of displaying text on the screen. But what if you have a lot to display? Consider a case where you want to display ten lines, but you can only fit

eight at a time. You could display five, stop in some way to let the user read, and then display the next five. Here's how you could pause the program.

PAUSE: TAKING A BREATH

Sometimes you want to display something and let the user stare at it for a while before your program continues. Luckily, there's a command for this, aptly named `Pause`. When the `Pause` command is active, the run indicator, that crawling line in the top-right corner of the screen, turns into dots that flash to indicate that the program is paused. The user can then take their time reading the screen and press [ENTER] when they're ready to continue the program. Figure 2.9 demonstrates just such a program.

The TRYPAUSE program shown here displays one line of text, uses a `Pause` command to wait for the user to press [ENTER], then displays a second line of text. It displays the results seen on the left side of figure 2.9 while the `Pause` command is active; then it displays the results shown on the right side of figure 2.9 when the user presses [ENTER] and the program ends:

```
PROGRAM:TRYPAUSE
:ClrHome
:Disp "PRESS [ENTER]"
:Pause
:Disp "WELL DONE"
```

In more complicated programs, you could use the same sort of technique to display pieces of a math program's results one by one, to wait between displaying steps of a solution, to wait after displaying the player's final score in the game, or any of thousands of similar tasks. I'll conclude this discussion of `Pause` with one final trick that the command can perform: display a line of text, a number, or a variable during the pause. You can use `Pause` in place of the `Disp` command in some cases, for example, in the program PAUSE2, shown next, which consolidates the first `Disp` with the `Pause`
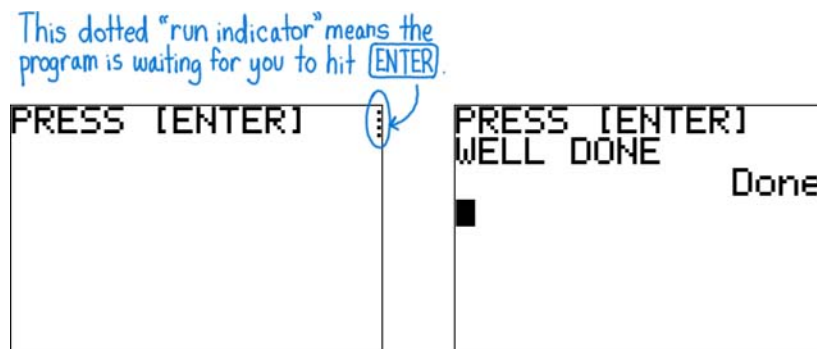
Figure 2.9   The TRYPAUSE program demonstrates the functionality of the `Pause` command. Of particular interest is the special dotted run indicator at the top right of the left screenshot that indicates the program is waiting for the user to press [ENTER]. The PAUSE2 program behaves exactly the same way.

command. This program behaves the same way and outputs the exact same things as the TRYPAUSE program:

```
PROGRAM:PAUSE2
:ClrHome
:Pause "PRESS [ENTER]"
:Disp "WELL DONE"
```

You will see many more uses of `Pause` throughout the rest of the chapter. But to expand the finesse with which you can display things on the homescreen, you'll now learn a command that lets your program display a letter, number, or string at any specific row and column of the homescreen.

2.2.2   *Positioning text with the Output command*

Sometimes it's not good enough to make the next number or string appear on the next line. Consider a program where you want to display eight different labels, such as "X=," "Y=," "VX=," and so on, for eight lines and then next to each of those display a number. With what you know so far, you can't do that. You'd have to display the first four labels and their associated numbers, pause, and then display the next four labels and numbers. Needless to say, this is a bit messy, so you need a better way to do this, something that would let you place the labels at the left edge of the screen and then go back and put the numbers at arbitrary locations on the screen without making the labels scroll away.

Luckily, you have just such a tool, the `Output` command! Remember, as shown in figure 2.5, the homescreen has 16 columns and 8 rows, each of which can contain a space (to make it blank) or a character such as a letter, number, or symbol. `Output` lets your program put any text at any location on the screen. It takes three arguments. The first is the row on which to display the text, from 1 to 8. The second is the column on which to start displaying the text, from 1 to 16; if the text or number is longer than 1 character, it displays the second character in the next column, moving across until it hits the right edge of the screen, at which point it stops. The third argument is the string or number to display. The following are four ways you could use `Output`:

```
:Output(<Row>,<Column>,"STRING")
:Output(<Row>,<Column>,<Number>)
:Output(<Row>,<Column>,<Mathematical expression>)
:Output(<Row>,<Column>,<Variable>)
```

The `Output` command is like `Disp` in that `Disp` can also display strings, numbers, variables, and the results of a math expression, but `Output` gives your program the power to put these at some precise location on the homescreen rather than whichever line is free next. When you use the `Output` command, nothing gets moved around and the screen doesn't scroll, so you can repeatedly use the `Output` command as much as you want to write things on the screen. You could even use it 128 times, displaying a one-character string at each position on the homescreen, to fill up the whole screen, although that might be inefficient.

For something more realistic, say you want to expand your Hello World program to center the text "HELLO, WORLD" on the screen, split between two lines. When run, this program will display "HELLO," starting at row 4, column 6 of the homescreen and the string "WORLD," starting at row 5, column 6. This roughly centers the lines on the screen, as you can see in figure 2.10.

How can you do this? With two simple `Output` commands, as demonstrated in the following source code. The program clears the screen, the first `Output` command displays the word "HELLO," and the second displays "WORLD"; the program pauses before exiting.

Figure 2.10   The output of the HIWORLD3 program, using the `Output()` command twice to center the "HELLO, WORLD" text on the homescreen

```
PROGRAM:HIWORLD3
:ClrHome
:Output(4,6,"HELLO,"
:Output(5,6,"WORLD"
:Pause
```

This is a step forward as far as the finesse you can use in displaying output to the user. I mentioned at the beginning of this section that you could use `Output` to display some text and a number together on one line, so let me show you how to do just that.

Omitting parentheses and quotes
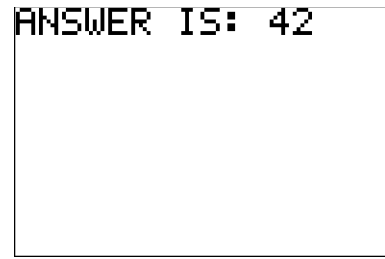
One confusing style change you might notice in the HIWORLD3 program is the missing parentheses at the end of the two `Output` commands. This isn't a mistake. As we've previously discussed, calculators have limited quantities of memory, so we try to save space wherever possible. Any parenthesis that's immediately followed by a new line can be removed; the calculator's interpreter will understand and not flag the line as having an error. The same trick can be used for string-ending quote marks that immediately precede a new line, as you'll see in the FRSTANIM program in this chapter.

PUTTING TEXT AND NUMBERS TOGETHER

Previously, you saw one of the shortcomings of `Disp` in its inability to put text and numbers together on the same line of the homescreen. `Output` solves that problem, because you can display some text at a given row and column and then a number at a later column of the same row. Consider the following program that displays the answer to an easy math problem:

```
PROGRAM:SAMELINE
:ClrHome
:Output(1,1,"ANSWER IS:"
:Output(1,12,42
```

The program will display the text "ANSWER IS: 42" on the top line of the homescreen. For this example, you could have put the 42 into the string, but in subsequent programs that solve an equation given numbers input by the user, you won't know what the number to be displayed is ahead of time. The output of the SAMELINE program is shown in figure 2.11.

Figure 2.11   Using `Output()` to display two different things, such as a string of text and a number, on the same line. This trick can be used for neatly formatted math solutions, games, and much more.

USING OUTPUT FOR A SIMPLE ANIMATION

The `Output` command can be used both to draw and, if made to output space characters, to erase. By repeatedly drawing and erasing, you can create crude animations on the homescreen. In this section, I'll show you a simple scrolling symbol; in chapters 4 and 6, you'll see more advanced animations and games made with `Output`.

Consider a program that performs complex math that takes a long time to complete. One thing you might want your program to do is display some sort of progress bar so that your users know the calculation is still running and the calculator hasn't simply frozen. I'll teach you in later chapters how to use fancy graphics to do this, but for now, here's a program that creates a progress bar out of equals signs and square brackets, FRSTANIM (First Animation):

```
PROGRAM:FRSTANIM
:ClrHome
:Output(1,1,"[    ]
:Output(1,2,"=
:Pause
:Output(1,2," =
:Pause
:Output(1,3," =
:Pause
:Output(1,4," =
:Pause
:Output(1,5," =
```

When examining and entering this program, notice that many of the `Output` commands display both a space and an equals sign. In each case, the space erases the equals sign from the previous `Output` while displaying a new equals sign that's shifted one column to the right. The first `Output` command creates a pair of brackets to bound the equals signs that will be created; if you type this program into your calculator, note that there are five spaces between the brackets.

```
:ClrHome
:Output(1,1,"[    ]
```

These two lines clear the screen and then display an opening bracket, five spaces, and a closing bracket.

```
:Output(1,2,"=
:Pause
```

These lines output an equals sign to the right of the opening bracket and then pause until the user presses [ENTER] to continue.

```
:Output(1,2," =
:Pause
```

Now the program displays a space and an equals sign at (1,2). This means that a space gets displayed at (1,2), placing it over the equals sign that the previous `Output` command wrote there and erasing that equals sign. It also means that an equals sign is displayed at (1,3), because `Output` moves left to right if you give it a string of more than one letter. To the user, it looks as though the equals sign has moved over from the second to the third column of the homescreen.

```
:Output(1,3," =
:Pause
```

This code does the same trick again, erasing the equals sign in the third column and drawing it in the fourth column.

```
:Output(1,4," =
:Pause
:Output(1,5," =
```

The last three lines of this program repeat the procedure to move the equals sign twice more by erasing it in the fourth column and redrawing it in the fifth column, pausing a final time, and then moving it to the sixth column.

One final item of interest is the missing quote marks (and parenthesis) at the end of each of the `Output` lines. The sidebar "Omitting parentheses and quotes" explains this. Once you type this program, FRSTANIM, and execute it, you'll see the results shown in figure 2.12 among the five frames of the animation.

The code for this program is somewhat repetitive; in chapter 4 you'll learn the `For` loop, which would let you turn this into two `Output` commands and a loop instead of six output commands executed one after the other.
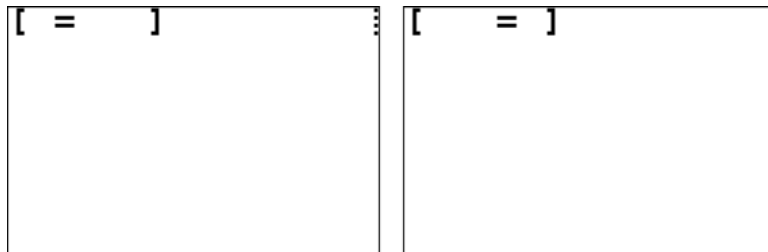
Figure 2.12   Some of the frames of the animation created by the FRSTANIM program, which scrolls an equals sign between the brackets. In chapter 4 you'll see a program with a similar concept that loops infinitely, flashing between two different screens, until stopped.

With the ability to display text and numbers to the user, you now need a way to get feedback from the user in the form of numbers, strings, and other types of data. The next section will teach you two TI-BASIC commands that let you do just that.

2.3   Input from users: the Prompt and Input commands

A program that can only display things to users isn't useful at all unless it also has a way of getting feedback, or input, from users. In computer programs, you provide input by moving the mouse, by clicking, and by typing on the keyboard. In much the same way, calculator users can type in numbers, strings, and other variables for programs to use. You've already seen this twice in chapter 1. In the QUAD quadratic equation solver, the `Prompt` command was used to ask the user for three numbers, A, B, and C, which the program then plugged into the quadratic equation. Similarly, the GUESS game used `Prompt` to ask the player for a numeric guess. This section will teach you more about the `Prompt` command and how it can be used to get numbers from the user. It will also introduce the `Input` command, which provides more control over what your program displays to a user when it asks them for input. I'll show you how you can also get nonnumerical input such as strings and how you can build a simple conversational program with these commands.

The simplest input task is using `Prompt` to get a number from the user and save it into a variable, so I'll present that first.

2.3.1   Prompting for numbers

Many programs, both games and utilities written for educational purposes, need a way to get numbers from the user. You already saw two such examples in chapter 1. Of the two commands that the calculator has for the purpose, we'll first examine `Prompt`, the command you've already seen. You can find `Prompt` under [PRGM][▶][2]. After the command, you must put at least one variable that `Prompt` will ask the user to type in. This can be a numeric ("real") variable such as B or R or N, which you'll see demonstrated in this section. It can also be other types such as strings, lists, or matrices, which I'll cover in chapter 9. Let's jump right into a simple two-line program that prompts the user to type in the variable X and then squares it and displays the resulting value. The source code for this program is as follows:

```
PROGRAM:PRMPTSQR
:Prompt X
:Disp X²
```

Recall from section 2.2.1 that you can perform math within a `Disp` statement, here squaring X before displaying it, rather than needing to square X, store the result back in X, and display that. You can see the output of this program for a sample input of X = 5 in figure 2.13.

There's not much to this command, as you can see. It displays the X=? prompt at the left margin of the calculator's screen, then flashes the cursor and waits for the user to type in a number. When the user does so and presses [ENTER], the program continues

at the next line. `Prompt` is a blocking input command, which means that your program can't continue to the next line of the program until the user types in their number and presses [ENTER].

`Prompt` has only one other feature that you'll find handy, because it's a simple command to use. Recall from chapter 1 that I used `Prompt` to get three numbers at once in `prgmQUAD`, the three coefficients needed for the quadratic formula. The naïve way to ask the user for variables A, B, and C might be as follows:

```
:Prompt A
:Prompt B
:Prompt C
```

Figure 2.13   Testing the program PRMPTSQR, which prompts the user for a value of X and then displays that value squared

Luckily, because every extra command is wasted space that makes a program bigger, we have a way to compress these three commands into a single command:

```
Prompt A,B,C
```

Notice that there are no spaces between the commas or the variables in that command. To see this technique being used for something slightly more useful, glance at `prgmPRMPTPWR`. This program asks the user for two numbers, P and Q, and then raises P to the Qth power.

```
PROGRAM:PRMPTPWR
:Prompt P,Q
:Disp P^Q
```

The PRMPTPWR program is demonstrated in figure 2.14. Here, 10 is raised to the fifth power, returning the value P^Q = 10^5 = 100000. Notice that the P=? and Q=? prompts appear on two lines, just as if two separate `Prompt` commands had been used to build the program.

For PRMPTPWR, you start to see a shortcoming of the `Prompt` command in action. Although the name of the program hints that it will deal with powers or exponents in some way, it's not clear from the program itself what P and Q represent. The best programs of any sort, including calculator programs, will be intuitive and understandable to their users even if the users haven't bothered to read the manuals for the programs. It would be better to add a way for the program to explain itself to the user. You could add the following line to the beginning of the program:

```
:Disp "RAISES P TO QTH","POWER
```

Figure 2.14   A program that prompts for two different variables, P and Q, and raises P to the Qth power

But you might want to instead give the user more explicit instructions to replace the text "P=?" and "Q=?." The Input command will give you exactly that power.

2.3.2   *Fancier Input for numbers and strings*

The Prompt command lets you ask the user to type in a value and then stores it in a variable, as you have seen, but it has one notable shortcoming. As a programmer, you can't control what text appears when a Prompt command occurs. If you're prompting for variable A, the prompt will always be A=?. This isn't always helpful. If, for example, you want the user to type in his or her age, the "A=?" text isn't going to provide many clues that's what the user is expected to type. If you've explored the commands in the three tabs of the PROGRAM ([PRGM]) menu shown in figure 2.4 and described in table 2.3, you may have noticed the Input command right before the Prompt command. For the sake of curiosity, try replacing the Prompt in my previous program PRMPTSQR (shown in figure 2.13 and again in figure 2.15) with the Input command, and see what happens:

```
PROGRAM:INPTSQR
:Input X
:Disp X²
```

You can see the output of the INPTSQR program in figure 2.15 in comparison to the similar PRMPTSQR program in section 2.3.1. As you may notice, the output of INPTSQR that it displays before asking the user to type a number isn't quite as descriptive as the output of the Prompt command, which at least lets users know the name of the variable into which the value they're typing will be stored. This may seem like a disadvantage in many cases, because your program is giving the users even less information about what it expects them to type in. You could use Disp to display a line of text explaining what the user needs to enter before the Input, but then you could still use the Prompt command, so what's the point?

The point of the Input command is that you can make it display anything you want on the same line as where the user types in input. Just as Output lets you display two
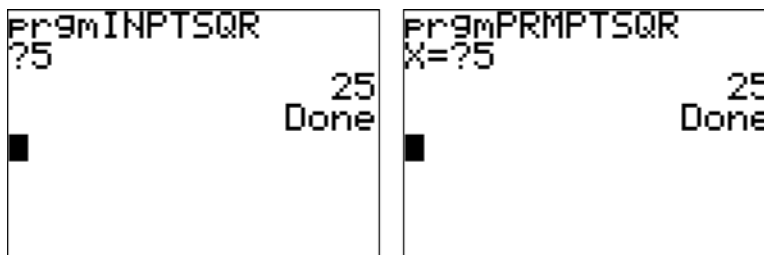
Figure 2.15   Using Input (left) versus Prompt (right) to ask the user for a number to be squared. The Prompt command always displays the name of the variable being stored, whereas Input displays a question mark and an optional string. The right screenshot is a copy of figure 2.13, the output of the PRMPTSQR program.

different things on the same line, `Input` lets you do output and input on the same line. Used with a string to display before the user types their input, `Input` takes the following arguments:

```
:Input "STRING",Variable
```

The calculator will display the string "STRING" and immediately wait for the user to type in a value that will be stored into the variable `Variable`. `Variable` can be a real (number) variable, like A, M, X, or θ, a string like `Str3`, or even a list like `L3` or a matrix such as [B]. Appendix A reviews the numeric and string variable concepts with which this and coming chapters assume you're at least marginally familiar. You'll learn more about using and manipulating strings, lists, and matrices in later chapters, especially chapter 9.

Besides the INPTSQR program you just saw that demonstrates how you can replace a `Prompt` command with an `Input` command, you'll see two more examples of `Input` in this section. First, I'll show you how to add a descriptive line to be displayed before the user can type in their input, which can be used to explain to the user what they should type. I'll end the section with a more elaborate program that calculates the slope of a line, using the `Input` command to get a pair of (X,Y) coordinates for two points on the line from the user.

To demonstrate what sort of things you can make `Input` do, I'll first show you a version of the INPTSQR program that asks "SQUARE OF?" before the user can type in a number, as shown in the following source code for INPTSQR2:

```
PROGRAM:INPTSQR2
:Input "SQUARE OF?",X
:Disp X²
```

When executed on your calculator, the program's output should resemble figure 2.16. You could put an extra space after the question mark on the first line of the program to separate the "SQUARE OF?" query and the number the user enters to make the program a little clearer, but you can see that we've already made a substantial improvement to the program as far as explaining to users what the program expects from them.

These smaller examples are a perfect way to introduce the `Input` command, indeed any command, because they show how the particular command works with little other code around it to confuse matters. In addition, throughout this and coming chapters, I'll occasionally stop in the midst of introducing commands and concepts to work through a complex, useful program or fun game that you might use day to day. Next, I'll present a program that uses the `Input`, `Disp`, and `ClrHome` commands to calculate and display the slope of a line.
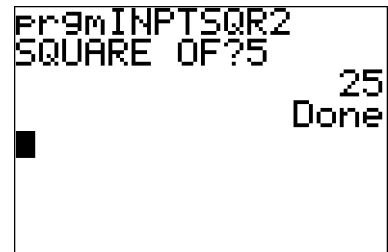
Figure 2.16   Using the `Input` command to display a line of text before the user can type something in. As you can see, `Input` allows you to customize the text displayed before the space where the user can type in their input, be it a number, string, or other data type. Here, that string is "SQUARE OF?," and the user has typed the number 5.

USING INPUT: CALCULATING SLOPE

The final math program of this section asks the user for the coordinates of a first point and the coordinates of a second point and displays the slope of the line that joins those two points. You'll see that I've used a few optimization tricks that you'll get used to as I go through more examples, including omitting a closing parenthesis that's immediately followed by the end of the line, as well as two quotation marks that also immediately precede the end of two lines. If you're not familiar with the slope of a line, it describes how steep it is; the larger the slope, the steeper the line. It's calculated from the quotient of how far the line rises over a certain horizontal distance divided by that horizontal distance, or in common math-class parlance, "rise over run." Figure 2.17 demonstrates the concept of a line's slope, calculated from any two points on the line.

In the following program, the (X, Y) coordinates of the first point are stored in variables (A, B), and the (X, Y) coordinates of the second point are in (C, D). The rise is the change in Y between the two points, or (D – B), and the run is the change in X between the two points, or (C – A). The slope is then (D – B)/(C – A), which you can see calculated on the last line of `prgmSLOPE` with the same formula as in figure 2.17:

```
PROGRAM:SLOPE
:ClrHome
:Disp "FIRST POINT
:Input "X1: ",A
:Input "Y1: ",B
:Disp "SECOND POINT
:Input "X2: ",C
:Input "Y2: ",D
:Disp "SLOPE:",(D-B)/(C-A
```

As always, you can run this by finding SLOPE in the [PRGM] menu, pasting `prgmSLOPE` to the homescreen by pressing [ENTER], and running it by pressing [ENTER] a second time. The horizontal line going through (1,1) and (5,1) will have a slope of 0, because

$$\text{Slope} = \frac{y_2 - y_1}{x_2 - x_1} = \frac{5-2}{6-1} = \frac{3}{5} = 0.4$$
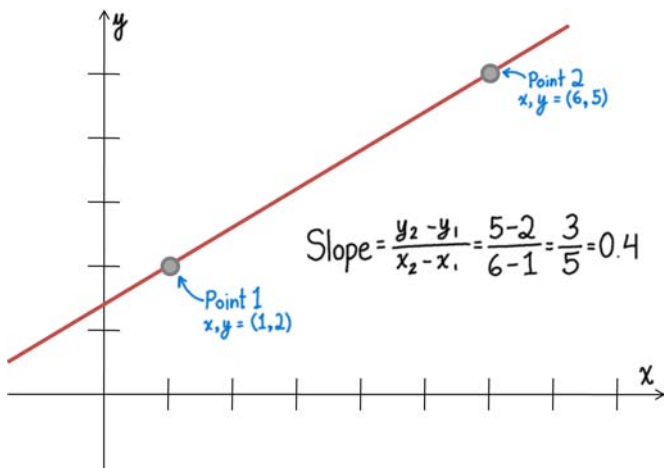
Figure 2.17   Calculating the slope of a line. Divide the vertical distance between the two points (the "rise") by the horizontal distance (the "run") to get a value for the slope. A horizontal line has a slope of 0, a diagonal line has a slope of 1, and a vertical line has a slope of infinity.
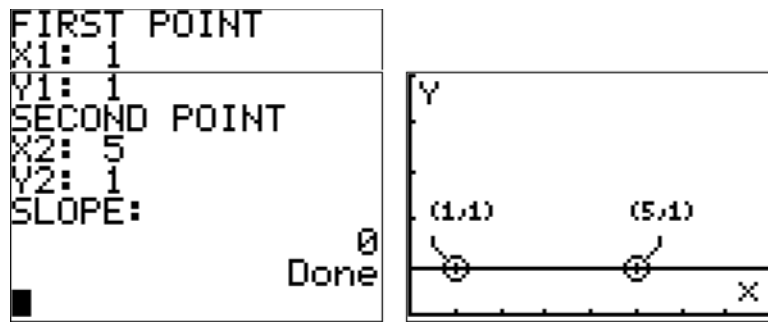
Figure 2.18 Testing `prgmSLOPE` on the line passing through points (1,1) and (5,1), shown on the left. The program correctly calculates a slope of 0 (a horizontal line), proven by the graph drawn at right.

its Y changes by 0 as it travels 4 horizontally; the results of the program for those two values are shown in figure 2.18. In this figure, a calculator-generated graph of the line in question is shown next to the output of the SLOPE program for your edification.

The program can easily calculate less trivial (obvious) slopes of tilted lines. The diagonal line passing through (-3,-3) and (4,4) rises 7 while it runs 7, so its slope should be $7/7 = 1$. You can see this accurate result from the SLOPE program in figure 2.19.

Once again, a graph is shown at the right side of figure 2.19 that demonstrates the results from the SLOPE program. You can see how this sort of combination of input, calculation, and output could be expanded into all sorts of mathematical and scientific solvers, such as the quadratic solver, `prgmQUAD`, from chapter 1.

Now that you've seen how `Input`, `Prompt`, `Output`, and `Disp` can be used for a few math programs, how about a simple "conversation" program that lets you talk to your calculator?
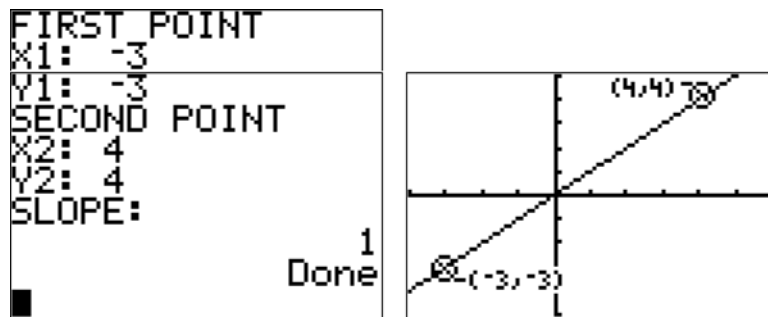
Figure 2.19 Calculating the slope of the line between (-3,-3) and (4,4) using the SLOPE program, at left. A graph of the line is at right.

2.3.3   *Exercise: making conversation*

Because you now know how to get input from the user and display that input (or some modified form of it, such as the input with additional math performed on it) on the screen, you can start using your knowledge to make more complex programs. In this exercise, you'll make a small program that pretends to chat with the user. It will ask users to type in their name and their age, and then it will greet them by name and repeat their age back to them. A transcript from a sample conversation with the program should look something like this:

```
HELLO, WHAT IS YOUR NAME?<User types name and presses [ENTER]>
AGE?<User types age and presses [ENTER]>
HELLO, <Name>
YOU ARE <Age> YEARS OLD
```

YOUR ASSIGNMENT

Your task is to write a program called CONVO that can hold a conversation like this. You'll definitely want to use `Input` in this program, and you'll also need `Output` and/or `Disp`. For the age, you can use a simple numeric variable such as A, X, D, and the like.

> **What's a string variable?**
>
> There are 10 string variables, `Str1` through `Str9` and `Str0`. Each one can hold any sequence of characters enclosed in quotes, like "HELLO" or "3.1415" or "THIS IS A VERY, VERY LONG SENTENCE THAT WASTES SPACE." Just as numbers can be stored into numeric variables and those variables can be used as if they were numbers, strings can be stored into `Str` variables, and the `Str` variables can then be used as if they were the strings themselves. For example, this code:
>
> ```
> :Disp "HELLO
> ```
>
> is equivalent to storing "HELLO" into a string variable and then using that instead:
>
> ```
> :"HELLO"→Str6:Disp Str6
> ```

Although we haven't discussed them much before, we'll use a string variable in the CONVO program, specifically `Str1` (short for String 1). In your program, `Str1` will be used with `Input`, so that once the user types a value to be stored into the string, it can in turn be used for display. You can type `Str1` with [VARS][7]1: Str1.

To give you a bit of a visual idea of how this program might look when run, take a look at figure 2.20. The user has typed in Kerm for his name and 24 as his age. Good luck!

SOLUTION

The source code for one possible solution CONVO program is shown here. It consists of six lines of code, all of which you should be able to easily type into your calculator

Figure 2.20 Running the CONVO program. Here, the user enters "KERM" as the string for his name and the number 24 as his age.

by now. As noted, the one thing you might not be easily able to find is the `Str1` token, which is under [VARS][7]1: Str1.

```
PROGRAM:CONVO
:ClrHome
:Disp "HELLO, WHAT IS"
:Input "YOUR NAME?",Str1
:Input "AGE?",A
:ClrHome
:Disp "HELLO, ",Str1,"YOU ARE",A,"YEARS OLD"
```

Once you type in this program and run it, you should see something resembling figure 2.20. The program will first ask you to type in your name; you should be able to type anything at this prompt. When you press [ENTER], it will ask you for your age, which must be a number. If you type anything other than a number, you'll get an error message from the calculator's OS. In that case, you can choose to quit the program, or if you select 2: Goto at the error screen, you'll have an opportunity to retype your age.

After you type your age and press [ENTER], the program will greet you by name and state your age, as shown at the right side of figure 2.20. As you can see, the output is correct, and the `Input`-based prompts are descriptive, showing the strings "YOUR NAME?" and "AGE?" right before the user has an opportunity to type in each of those items. If this program has one shortcoming, it's that the output shown in the right side of figure 2.20 is less than neat.

You could go further and try to neaten up the lines of text that the program outputs at the end. The most obvious change would be to put the user's age right next to the string "YOU ARE" rather than all the way over at the right edge of the screen one line down. But if you do that, you might even be able to fit the word "YEARS" on the same line. This fix is particularly easy because you know that the user's age probably has two digits in it and definitely has one, two, or three digits. Therefore, if you put four spaces between "ARE" and "YEARS", you know you'll have room for a space, then one or two digits, and then the word "YEARS." Let's try that out and see what happens.

experience a program you've written that produces an error message, or worse, doesn't generate an error but still doesn't function correctly. Maybe your program will display the wrong thing in the wrong place, or maybe it performs math incorrectly, but in some way it doesn't function the way you designed it to work. When this happens, you need not panic, because there are well-established guidelines to help you track down the problem. As you learn the general troubleshooting guidelines in this section, you'll likely start to think of your own methods of finding errors, which is excellent.

In sections 5.3 and 5.4, I'll present an in-depth guide to the various error messages that your calculator produces and how to track down solutions, as well as detailed guidelines for figuring out what part of a program is at fault when the program doesn't work as you planned. For now, I want to give you a few tips for debugging your own programs so that you can get started creating new programs as quickly as possible and so that you can have fun with it and not get frustrated.

2.4.1 *Easy-to-spot errors: TI-OS error messages*

Your calculator's operating system, the TI-OS, throws (generates) errors when your program has a mistake such as a typo, a command missing its arguments, a string where there should be a number, or any of a multitude of similar minor problems. Figure 2.22 shows an example of one of these messages, ERR:SYNTAX, which appears when you've made a typo in your program. Your calculator has 50 such error messages, about 25 of which may be generated by programs that you write, but there are 3 in particular I'd like to focus on here. With the programming skills you've been building so far, your own programs are likely to generate SYNTAX, BREAK, or INVALID DIM errors. Section 2.4 will explain the other types of error messages your programs might generate as they use a larger set of programming commands.

When you see any of these sorts of TI-OS errors shown in figure 2.22, you're generally invited either to Quit (which returns you to the calculator's homescreen) or, in some cases, to Goto the error. This latter option opens the program editor and brings you to the line in the program where the error occurred. You therefore have a chance to try to fix the error and then either continue to edit the program or quit the editor and try to run the program again.
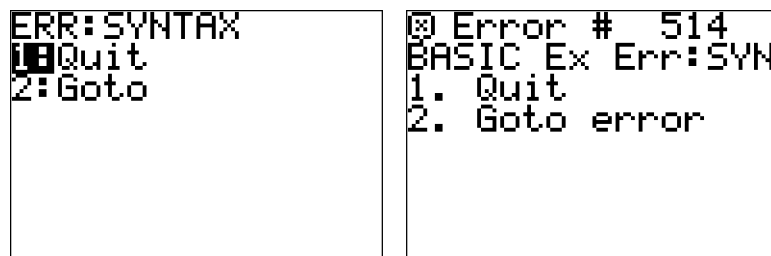
Figure 2.22   The error message shown when a TI-BASIC program produces a SYNTAX error, normally (at left) and when a shell is installed (at right)

Incidentally, these are the same error messages that appear when a user types something illegal at an Input or Prompt. In that case, the Goto option brings the user back to the Input or Prompt rather than into the source code of your program. Another error that users (rather than your program) may trigger is ERR:BREAK, which happens when they press the [ON] key to stop a program. In some cases, if you (or another programmer) have written a program that doesn't end, or that you don't want to wait to end, or even that you want to stop in its tracks and look at the source code for that portion of the program, you can press the [ON] key. This signals the interpreter to stop what it's doing and bring up a BREAK error similar to figure 2.22; just as with the SYNTAX error, you can choose to quit to the homescreen or to go to the line of code the interpreter was at in the program when the user pressed [ON].

TI-OS ERRORS: SYNTAX

As briefly discussed, the problems in your program that trigger a SYNTAX error might be as simple as a typo. You may have typed too many closing parentheses, forgotten to start a string with a quotation mark, or typed a token out letter by letter as, for example, "D" "i" "s" "p" instead of going to [PRGM][▶][3] for Disp. It might be something more complex, like forgetting one of the arguments to a command, in which case you'll have to refer to the command's relevant chapter herein or to appendix B. It might be something more obscure still, like using an imaginary or complex number where only a real number will work.

For the programs you've examined in this and the previous chapter and will continue to work with in chapters 3 and 4, your errors will be mostly limited to the simplest kinds. When we get to chapter 5, I'll introduce a systematic approach that addresses the many error messages the calculator can generate in response to typos and incorrectly typed commands.

TI-OS ERRORS: INVALID DIM

There's one final TI-OS error message that I'll address before moving on to troubleshooting problems in your program that don't produce messages yet still affect their functionality: the INVALID DIM error. ERR:INVALID DIM usually appears as shown in the left side of figure 2.23 when a list variable (which you'll learn to use in your programs in chapter 9) contains the wrong number of elements. Even nonprogrammers
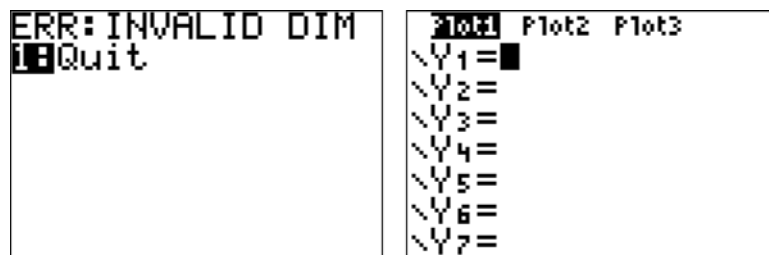
Figure 2.23   The INVALID DIM error (left) that results from accidentally turning on one of the statistics plots (right). The text describes the simple fix for this problem.

can become frustrated with this error, which frequently happens when one of the statistics plots (`Plot1`, `Plot2`, or `Plot3`) gets accidentally selected in the [Y=] screen, a mistake shown on the right side of figure 2.23. I'll mention this again when you start learning about using the graphscreen in chapter 7, but it's worthwhile to know the quick fix for this problem. Press [Y=], use the arrows to move the cursor up to whichever plot label is highlighted (white text on a black background), and press [ENTER] to deselect the plot label, turning it back to black on white. The ERR:INVALID DIM should now not appear when you press [GRAPH].
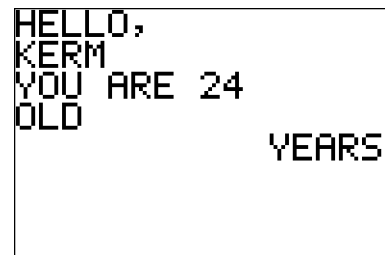
2.4.2 *The subtle errors: why isn't my program working the way I want?*

Unfortunately, not every problem is quite so obvious. There are many issues your programs will develop that won't advertise themselves with a big ERR message. Indeed, the vast majority of bugs in programs are far more subtle, including anything from small mistakes in the formatting of output to calculation mistakes that happen only in limited circumstances. Because these bugs don't come with bold messages advertising their presence, including a handy Goto option that takes you directly to the line of the program responsible, tracking down exactly what's causing them can often be as hard or harder than implementing the actual fix.

The most helpful thing you can do in such a circumstance is try to keep in mind the relation between the code that you wrote and the way the program works. Recall the program called CONVO2 from section 2.3, which made your calculator hold a rudimentary conversation; here I've made a slight change in it to introduce a bug:

```
PROGRAM:CONVO2
:ClrHome
:Disp "HELLO, WHAT IS"
:Input "YOUR NAME?",Str1
:Input "AGE?",A
:ClrHome
:Disp "HELLO, ",Str1
:Output(3,1,"YOU ARE
:Output(3,9,A
:Output(5,12,"YEARS
:Output(4,1,"OLD
```

If the text at the end of the program ended up formatted improperly, as shown in figure 2.24, you'd know that the line that printed the word "YEARS" on the screen was at fault. Not every bug is this obvious to spot. There are bugs where you may accidentally switch variables around, such as asking the user to input their age in A but then displaying B at coordinate (3,9). You might put parentheses in the wrong place in parts of programs that perform calculations or forget about Order of Operations (also known as PEMDAS). You might omit a line

Figure 2.24   A bug introduced into the conversational program CONVO2. Luckily, because the word "YEARS" is displayed by itself, it's easy to see where in the source code the mistake lies that caused "YEARS" to move down two lines.

you meant to type or insert another one by mistake. The worst bugs are caused by a fundamental confusion about the structure of your program, where you have designed it in a way that can't work unless you redesign it. Hopefully, you won't find yourself in such a quandary until the end of the next chapter, at which point you'll be ready to read chapter 5 and learn how to extricate yourself from such a mess.

You now know much of what there is to know about your graphing calculator's homescreen, displaying output and accepting input in your programs, and you've seen a few sample applications. I presented some lessons about the rudiments of debugging, so let's move on to learn more complex program structure.

2.5   *Summary*

You've now seen how to create basic programs that can use `Disp` and `Output` to write text and numbers on your calculator's screen and use `Input` and `Prompt` to get the user to enter values to be stored in variables. With these two sets of commands, you can write a great number of simple programs, especially those that ask a user for a set of values and then solve equations based on the numbers the user inputs. You already have enough information to make solvers for equations such as the quadratic equation, the Pythagorean Theorem, any of many physics equations such as those for kinematics, and even some engineering equations. You might be able to put together simple games.

But with the new programming skills you'll learn in chapter 3, you can go vastly further. In this chapter, all of your programs start at the first line of the program and continue straight through to the last line without any sort of detour. You have no way to skip any lines, to loop back up to another section of the program, or to skip over to a different part of the program if the user enters something that warrants it. Chapter 3 will introduce logic, comparisons, and conditional statements; by the end of chapter 3, you'll have the knowledge to put together the QUAD example from chapter 1 all by yourself, as well as many more programs.

Conditionals and Boolean logic 3

This chapter covers

- Representing rules and facts in a program
- How programs can make decisions
- Running code conditionally based on comparisons

Although you may have never thought about it before, almost every computer program you've used in your life *reacts* and makes decisions. Any program you run will execute different sections of code depending on what you type and click, what your files contain, and even what files on other computers contain. Calculator programs are no different, and most programs you'll want to write will need to have rules and the ability to make decisions. You've already seen a quadratic equation solver, QUAD, that displayed a warning about imaginary roots only if it determined that it would encounter imaginary roots. You also saw the GUESS guessing game's decision making in deciding whether to reloop (on an incorrect guess) or end the game (on a correct guess).

When I'm writing or testing or designing a program, I think of it in terms of flow, of pieces connected by arrows, through which the program's flow of execution proceeds. If a certain variable contains a specific value, then the flow proceeds

from one part of the program to another along an imaginary arrow. If it contains a different value, the flow proceeds down a different arrow to a different part of the program. As you get further into programming, you'll find it intuitive to think about programs the same way, as a flow of execution rather than code on a screen. When you think about a program that way, you think of it just as the computer sees it, and you'll remove the need to translate the design you imagine in your head into something your computer or calculator can understand.

In this chapter, you'll learn about comparisons and conditional statements, which empower you to create programs that make decisions. By the end of the chapter, you'll have explored examples that determine and display the sign of numbers, draw patterns on the calculator's screen, and more. I'll show you comparisons, the decision-making rules that your calculator reduces to true or false to decide how to proceed. We'll work with conditional statements, which give the program a way to execute different pieces of code based on the truth or falsehood of comparisons. I'll build on the commands you learned in chapter 2, including `Disp`, `Output`, `Input`, and `Prompt`.

Just as you make decisions given facts and rules ("I am hungry, so I will eat"; "I am not hungry, so I will not eat"), you need to be able to express rules in your programs that the calculator can plug facts into as it runs your programs. Let's get started with comparisons, your calculator's basis for making decisions.

3.1   Introduction to comparisons

The rules that programs follow to direct the flow of execution are called *conditions*. To simplify the terminology, I'll call an expression that compares two numbers or strings a *comparison*. Anything that's either true or false, such as a simple comparison or a logical combination of several comparisons, is a condition. When I combine a condition with the command called `If`, and optionally `Then`, `Else`, and/or `End`, I'll call that combination a *conditional statement* or *conditional*. If you've ever taken a class like Algebra I, you should be familiar with the concept of the comparison. A comparison consists of three things: a left side, a comparison operator in the middle, and a right side, as in figure 3.1.

Figure 3.1   The three pieces of a comparison: the left and right sides, which both have numerical values, and the comparison operator, which defines how the left and right sides will be compared

The sample comparison in figure 3.1 demonstrates comparing two numerical quantities, $X + 32$ (the contents of variable X plus 32) and Z (the contents of variable Z). If the value ($X + 32$) is indeed larger than the value of Z, then the comparison is true; if ($X + 32$) is equal to Z or smaller than Z (that is, ($X + 32$) is not greater than Z), then the comparison is false. Every comparison you'll see in TI-BASIC or any other programming language must evaluate (reduce) to true or false. Programming languages have no concept of fuzzy logic, wherein something might be sort of true, or kinda true.

Although the comparison in figure 3.1 is numerical, you can compare other things too. Depending on what language you're writing, you might be able to compare

strings, to see if they contain the same text; lists or arrays (sequences of numbers), in which case you compare respective elements of two lists; and many other data types. TI-BASIC lets you compare a number of different data types, although we'll discuss only numbers and strings in this chapter:

- *Numbers*—Two numeric expressions such as 32 or -5.4 or $X^2 + 9.3$ can be compared.
- *Strings*—Two strings can be compared. If they're the same length and contain the same letters, then they're equal; otherwise, they're not.

You can't compare two items of different types; for example, you can't compare a string and a number.

Numbers can be compared using one of six comparison operators, all of which can be found in the TEST menu, under [2nd][MATH]. All six should be familiar from any math or algebra class. Two of the operators test equality: = and ≠. Four of the operators test inequality: >, <, ≥, and ≤. For strings you use only the two equality operators, = and ≠. As you read on, keep in mind that every comparison is either true or false, depending on whether or not the comparison expression represents something factual.

TRUE AND FALSE IN TI-BASIC

In some languages, true and false are the two possible values of a data type called a Boolean or Boolean variable. Although a number can be -4 or 0 or 947,821.803, a Boolean can only indicate truth (or falsehood) and can only hold the values true or false. In TI-BASIC, there is no such thing as a Boolean variable; instead, the values true and false are themselves represented by numbers. The result of every comparison and every condition is therefore a number denoting true or false. In TI-BASIC, true is represented as 1 and false as 0. You can test this out yourself with a simple program, shown at the left side of figure 3.2 as TESTCMP.

```
PROGRAM:TESTCMP
:Disp 3=5
:Disp 3<5
```

Because I just said that the calculator uses the number 0 for false and the number 1 to represent true, this program should display 0 and then 1 on the homescreen. Lo and behold, as you can see in figure 3.2, it does exactly that. If you want to type this program to test it, remember that you can find the comparison operators under [2nd][MATH].
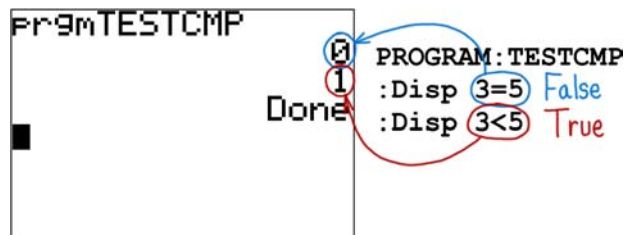
Figure 3.2 Displaying the values of the comparisons 3 = 5 and 3 < 5, demonstrating that 0 is false and 1 is true in TI-BASIC. Because the first comparison is clearly false (3 is not equal to 5) and the second is true (3 certainly is smaller than 5), the program displays the value 0 for false and then the value 1 for true.

Table 3.1 Five comparison operations, with their equivalents, their truth values, and the numerical values of those truth values. Comparisons between two numbers and comparisons between two strings are shown.

Comparison code	Equivalent to...	Boolean truth value	Numerical value
`:8→X` `:3X<20`	$24 < 20$	False	0
`:-3.2→X` `:X+5≥0`	$2.2 \geq 0$	True	1
`:"HELLO"="HI"`	"HELLO" = "HI"	False	0
`:-5→X` `:-X=5`	$5 = 5$	True	1
`:99→N` `:N≠6`	$99 \neq 6$	True	1

To get a further idea of what various equality and inequality operations return, take a look at table 3.1, which shows a few comparisons, their truth values, and their numerical values.

Table 3.1 demonstrates a number of interesting features about comparisons. In the leftmost column, BASIC code that you could try on your calculator is shown. The simplified equivalent of each is shown in the second column. In the first comparison, X is replaced with 8 and the multiplication $3 * 8$ is performed, leaving the equivalent comparison $24 < 20$. Because 24 is not less than 20, the comparison is false, and because false is represented as 0 in TI-BASIC, the comparison $24 < 20$ is equal to 0, shown in the last column.

By the same token, the expression in the second row reduces to $2.2 \geq 0$. Because this is true, the value is 1.

The third comparison introduces string comparison. The two strings are not identical, so they're not equal, and the value of the comparison is false or 0.

The fourth comparison shows that the negative of -5, in other words just 5, is equal to 5, an expression that's true (or 1).

The fifth comparison demonstrates the not-equals symbol, which may take a second glance to understand. The comparison shown compares 99 and 6 and asserts that they're not equal. This is certainly true; 99 is not equal to 6. Therefore, the comparison is correct, true, and equal to 1. This may be initially confusing because the resulting Boolean value is true when the two numbers are unequal, but remember that the truth value of the statement only expresses whether the comparison operator as shown in figure 3.1 represents an accurate relationship between the left and right sides. Here, that correct relationship is that the two numbers are not equal.

Now that you know how the calculator evaluates a comparison and expresses the truth or falsehood of that comparison, you can move on to expand your comparisons

into conditional statements. These conditionals can be used to control the flow of execution in a program, whether certain commands get executed, and what parts of your program are run.

3.2 Conditional statements

With a knowledge of how to compare numbers and strings, you can write code that does different things based on the values of variables and the inputs that users enter. You already know how to, for example, display numbers that depend on a value that a user types in. Consider a simple program named DOUBLE that displays two times the number a user enters:

```
PROGRAM:DOUBLE
:Input "DOUBLE OF:",X
:Disp 2X
```

No matter what number the user enters, this program will double it and then display that value. What if you only want to display that doubled value if X is positive, though? In the GUESS game in chapter 1, the text "TOO HIGH" was displayed only if the guess was higher than the target number, and the text "TOO LOW" was displayed only if the guess was lower. To present a more complex but more useful case, remember that in the QUAD program in chapter 1 that solved the quadratic formula, the text "IMAGINARY ROOTS" was displayed if and only if the coefficients entered by the user yielded a quadratic formula with imaginary roots.

In this section, you'll learn increasingly complex ways to execute different commands based on whether a comparison statement is true or false. First, you'll learn how a conditional containing a comparison can control the execution of a single line of code (`If`). Next, I'll demonstrate conditionally executing a chunk of several lines of code (`If/Then/End`). Finally, you'll see how to run one piece of code if the condition is true and another if it's false (`If/Then/Else/End`). I'll start by showing you the simplest case, executing a single command if and only if a comparison is true.

3.2.1 The one-statement conditional: If

With TI-BASIC, you can conditionally execute a one-line command when a given comparison is true. As in the GUESS and the QUAD programs, a one-statement conditional uses the `If` command. Like `Prompt` and `Disp`, it takes one argument: a comparison statement. `If` can be found in the first tab of the [PRGM] menu containing programming commands and is the first item, 1: If. The general form of the simple one-line `If` conditional looks like this:

```
:If <condition>
:<statement to execute when condition is true>
```

The statement (the second line of this template) can be a simple command like `Disp`, a mathematical expression such as B+3→B, or one of the many commands you'll learn in later chapters. If you are still vague on the idea of a command that's conditionally executed, look at figure 3.3. On the left side of the figure are two lines of code. In this
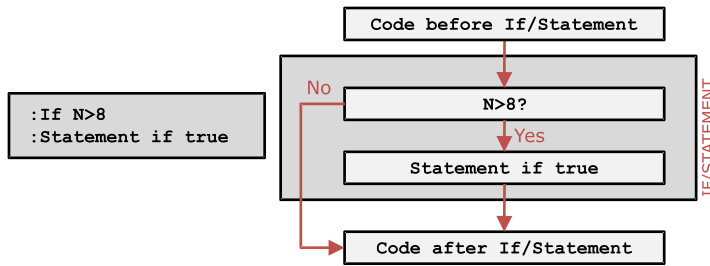
Figure 3.3   **The structure of a simple If/Statement construct. If the condition (here, N > 8) is true, the statement is executed. Otherwise, it is not. A more detailed explanation of this diagram is given in the text.**

example, the statement `Statement if true` is executed only if the variable N is indeed greater than 8, the comparison for the `If` command. The right side of the figure shows a flowchart representing the flow of execution when the calculator's TI-BASIC interpreter encounters these two lines of code. Hopefully flowcharts are becoming easier for you to read, but let's work through the chart piece by piece.

I assume either that there's more code before this chunk or that it's the first two lines of the program. In either case, the interpreter takes the first red arrow down into the `If/Statement` box, first examining the comparison expression $N > 8$. It checks the value of N, and if it's greater than 8, it knows the comparison is true. If it's true, and therefore $N > 8$, then it continues on to execute the `Statement if true` line (which is, of course, pseudocode; there's no actual command called Statement). If N is equal to or smaller than 8, then $N > 8$ is false, and the interpreter *skips* the `Statement if true` line and goes to the next line under that. In either case, the interpreter then continues by running whatever code immediately follows the two lines shown in figure 3.3.

With that in mind, let me show you the conditional statements from GUESS and QUAD again, and hopefully with your newfound knowledge you'll understand them better than from my brief chapter 1 explanations. First, from GUESS:

```
:If G>N
:Disp "TOO HIGH"
:If G<N
:Disp "TOO LOW"
```

This is a pair of conditional `If` commands with associated statements. The first one checks if $G > N$, and if so, executes the `Disp "TOO HIGH"` command. Similarly, in the second conditional, "TOO LOW" is displayed if and only if $G < N$. Notice that if $G = N$, neither of the comparisons will be true, and neither of the `Disp` commands will be executed, because $G < N$ is false and $G > N$ is also false. Recall that if $G = N$, the player guessed N correctly, so this behavior makes sense. Although these two `If/Statement` constructions are adjacent, they're two separate pairs of commands, and either `If/Statement` would work properly without the other.

Now to look at the code from QUAD, simpler in form but with a more complex comparison. Actually, the code in the original program used an `If/Then/Else/End` block, which you'll learn later in this section; a simplified version of the `If` statement would look like this:

```
:If 4AC>B²
:Disp "IMAGINARY ROOTS"
```

As you might expect, this displays the text "IMAGINARY ROOTS" only if the value of A times C times 4 is greater than B squared. Though both these examples happen to use `Disp` as the conditionally executed command, it's far from the only command you can use. You'll see two programs that use `If` to control a single statement, in one case to perform math and in the other to run a command, before we move on to more complex conditional constructs.

EXAMPLES: ABSOLUTES AND SIGNEDNESS

I will show you two examples of `If` in action, both of which happen to deal with the sign (positive or negative) of numbers. In both programs, the user will be asked to input a number. In the first program, an `If` will be used to generate the absolute value of the number, by negating it (switching its sign) if it's negative to convert it to positive or leaving it alone if it's already positive. In the second program, the word "POSITIVE" or "NEGATIVE" will be displayed depending on the sign of the number the user enters.

The first program is called IFABS, because it takes the absolute value of a number using the `If` command. It's a simple four-line program and will produce something like the results shown in figure 3.4. The code uses the `Input` command with which you're already familiar, then has an `If/Statement` construct, and finally has a `Disp` command to display the final absolute value.

```
PROGRAM:IFABS
:Input "ABS OF ",X
:If X<0
:⁻X→X
:Disp "IS",X
```

If you type this program into your calculator, make sure that in the third line you use the negative key, marked (-), rather than the subtraction operator. TI-BASIC differentiates between negation and subtraction, and if you type a subtraction symbol instead of a negate symbol, the interpreter will produce a SYNTAX error.

Although you've already seen one such example with the guessing game, GUESS, I'll show you another instance of using `If` to control whether one, both, or neither of
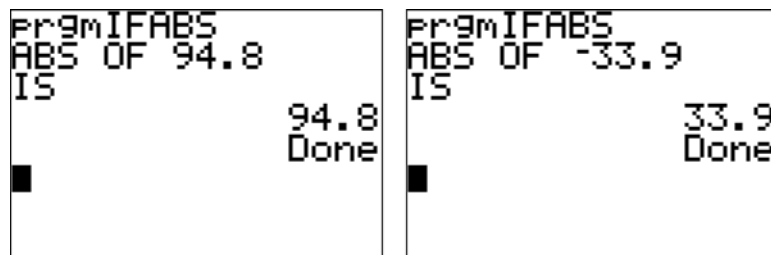
Figure 3.4   Two screenshots from the IFABS program that uses an `If` statement to produce the absolute value of a number. If the user inputs a negative number, it negates it to produce a positive; otherwise, it lets it be.
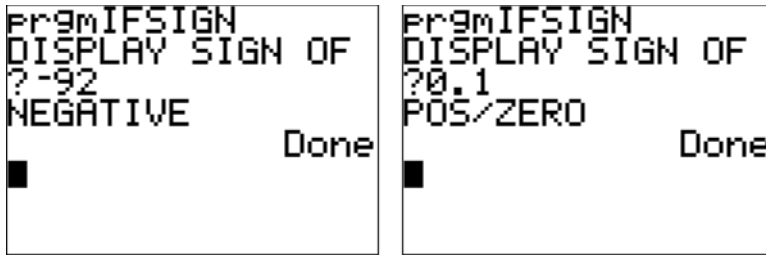
Figure 3.5   The IFSIGN program displaying the sign of a number using two `If`/`Statement` constructs

a pair of `Disp` commands are executed. In this case, the user will input a number, and the program will say if it's negative or positive/zero. The code might look like this:

```
PROGRAM:IFSIGN
:Disp "DISPLAY SIGN OF
:Input X
:If X<0
:Disp "NEGATIVE
:If X≥0
:Disp "POS/ZERO
```

The two `If` statements run independently, but the comparisons are mutually exclusive, so the program always displays exactly one of the strings. If X < 0, then it displays "NEGATIVE"; otherwise, it displays "POS/ZERO." There is no number that's both less than zero and greater than or equal to zero, nor is there any number that makes neither comparison true. You can see this program working in figure 3.5, for a negative input number on the left and a positive input number on the right.

Now you know how to execute a line of code (or not) based on whether a comparison is true or false, but what if you want to run more than one line of code conditionally? If you want to control whether a large block of code runs based on the truth of a comparison or conditional, you'd have to give each line its own identical `If` statement, which certainly seems like a waste of space. I've said several times thus far that making your programs as fast and as small as possible is important, and luckily there's a better solution to conditionally executing larger chunks of code.

3.2.2   *Conditional blocks: Then/End*

There are many cases when you'll want to conditionally execute more than one line of code. To drive the `Disp` examples farther into the ground, imagine that you might want to call the `Disp` command a few times, `Pause`, and then `ClrHome`, all based on a single conditional. For a more practical example, you might instead want to update several variables' contents in that conditional block. TI-BASIC provides a solution in the form of the `Then` and `End` commands. Instead of a single statement under an `If` line, you wrap many statements inside a `Then`/`End` block. The `Then` command is in [PRGM][2], and `End` is [PRGM][7]. When the TI-BASIC interpreter sees a `Then`, it

knows that every line after that `Then` until the corresponding `End` is part of the same conditional block. You can even nest conditional blocks inside of conditional blocks! First, the simpler case:

```
:If <condition>
:Then
:Statement 1
:Statement 2
:...
:Statement N
:End
```

When `<condition>` is true, all of the statements between `Then` and `End` are executed. If `<condition>` is instead false, *none* of the statements between `Then` and `End` are executed.

Check out figure 3.6 for the same concept again, in a more graphical flowchart format. As with single-statement `If` commands, you either have some code before the `If/Then/End` block or it falls at the beginning of the program. Either way, the `If/Then/End` code begins with an `If` statement. If the conditional statement is true, and the next line is a `Then`, the TI-BASIC interpreter executes all the lines after the `Then` until it reaches a corresponding `End` command. If it's instead false, the interpreter skips the `Then`, the `End`, and everything in between, continuing directly to the first line of code after the `End`, or ending the program if there are no more lines after `End`.

USING THEN AND END: BUILDING A PROGRAM TO PRINT AND MIRROR O
Let's build an example program that uses `If/Then/End` together. Consider a program that lets you type a pair of numbers, a row and column on the homescreen, which will then `Output` an O at those coordinates. The code for this might look this:

```
:Input "ROW=",A
:Input "COLUMN=",B
:ClrHome
:Output(A,B,"O
```

As you'll learn when I discuss defensive programming in chapter 6, this code has a problem. If the user types in 0 or -3 or 9999 for the column or the row, the program
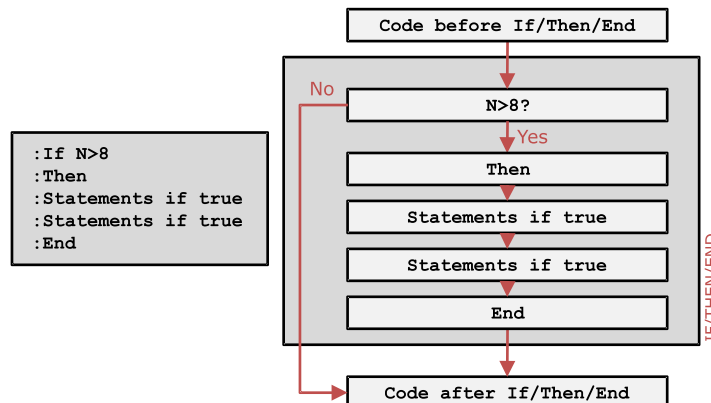
Figure 3.6   The structure of an `If/Then/End` construct. It differs from `If/Statement` only in that multiple statements can be conditionally executed at once.

will cause a TI-BASIC error, in this case ERR:DOMAIN, because the coordinates would be off the screen somewhere. We could improve this program as follows:

```
:Input "ROW=",A
:Input "COLUMN=",B
:If A<1
:1→A
:If A>8
:8→A
:If B<1
:1→B
:If B>16
:16→B
:ClrHome
:Output(A,B,"O
```

Now the program is coded defensively. If the user enters a row less than 1 or greater than 8, the program will correct the invalid row to something valid. Similarly, if the user enters a column less than 1 or greater than 16, the program will correct it. Great, but where does the Then/End come into play?

Suppose we add one more condition to this program. It will ask the user if they want to mirror the O four times around the screen, to turn it from something like the left side of figure 3.7 to something like the right side.

Now you can see where you could use an If/Then/End block: to contain three more Output commands to draw these extra Os! One of them will be at row A, column $(17 - B)$, another at row $(9 - A)$, column B, and the third at row $(9 - A)$, column $(17 - B)$. Where did these numbers come from? If the O is in the first row and column, then mirrored horizontally it should be in the first row (A) and the last column (16, or $17 - A = 17 - 1$). Mirrored vertically, it should be in the first column (B) and the last row (8, or $9 - B = 9 - 1$). The third copy should be in the last row and the last column, or $17 - A$ and $9 - B$. Here's the code for these three Output statements:

```
Output(A,17-B,"O
:Output(9-A,B,"O
:Output(9-A,17-B,"O
```
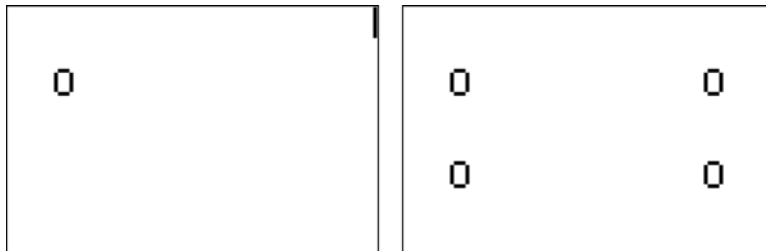

Figure 3.7 Taking a single O output at one point on the screen at left and mirroring it to three other locations on the screen. Conceptually, this is like first mirroring the screen horizontally and overlaying it on itself, producing two Os, then mirroring it vertically and overlaying it, producing four Os.

And now, the coup de grâce: you put this inside an `If/Then/End` block, add another `Input` command to ask the user whether or not they want to draw the mirrored Os, and get the full program. I call this program THENEND for obvious reasons; you can see the full source code in the next listing.

```
PROGRAM:THENEND
:Input "ROW=",A
:Input "COLUMN=",B
:If A<1
:1→A
:If A>8
:8→A
:If B<1
:1→B
:If B>16
:16→B
:Disp "MIRROR 4 TIMES?
:Input "(1=YES)?",M
:ClrHome
:Output(A,B,"O
:If M=1
:Then
:Output(A,17-B,"O
:Output(9-A,B,"O
:Output(9-A,17-B,"O
:End
:Pause
```

Change the row variable to be within the bounds of the screen if the user entered an offscreen value, namely less than 1 or greater than 8

Keep the column variable within the bounds of the screen

A simple type of "menu," where the user types 1 for yes and 0 (or something else) for no

Display three more Os in a single conditional block if the user requested this

Because the final `Pause` is outside the `Then/End` block, the program pauses whether or not the user types 1 at the `MIRROR?` prompt. The top two screenshots in figure 3.8 show a user trying out `prgmTHENEND`; the bottom two screenshots show the result of the values for row, column, and mirror that the user enters.

As you can see, the user enters a number for M; if that number is 1, then the program executes the body of the `Then/End` block and draws the three extra Os. If the user enters anything other than 1, such as 0 or 2 or 999, the program will skip the body of the `Then/End` loop. Following the usual steps for examples, you can type this program into your calculator and test it. You should be able to find all of the commands for this particular program, remembering that `If`, `Then`, and `End` are all in the first tab of the [PRGM] menu.

The `If` command has one final trick up its sleeve to help you write smaller, better programs. Up to this point, every `If`, whether controlling the execution of one or more statements, will run the associated code only if the condition is true. You'll now learn how to run one block of code if the condition is true and another if the condition is false.
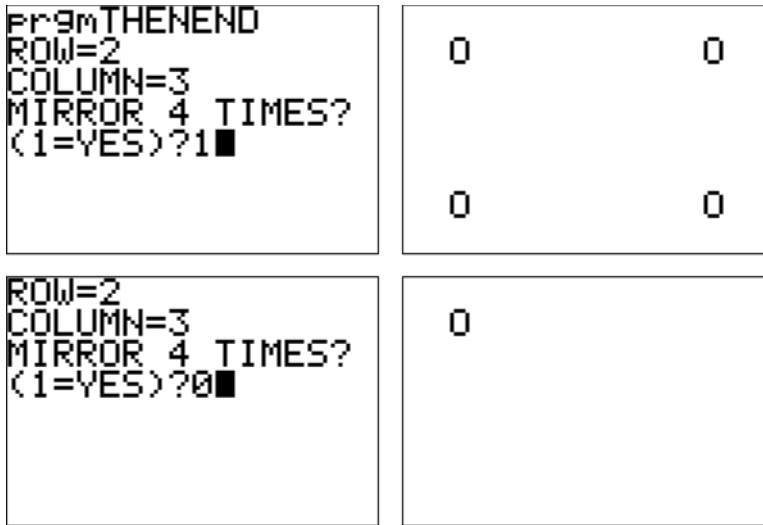
Figure 3.8   Running the THENEND program for a given row and column value, to draw an O onto the screen. An `If/Then/End` block controls whether the program also mirrors the O horizontally and vertically, as shown on the left side of the figure. The left side shows the user entering values (top) and the result (bottom) for mirroring turned on. The right side of the figure shows the same, but the user turns mirroring off.

3.2.3   *Conditionals with alternatives: Else*

Consider the IFSIGN program presented in section 3.2.1. It uses two conditional statements to display either "NEGATIVE" or "POS/ZERO" based on the sign of a number the user inputs. Here's the code again for review:

```
PROGRAM:IFSIGN
:Disp "DISPLAY SIGN OF
:Input X
:If X<0
:Disp "NEGATIVE
:If X≥0
:Disp "POS/ZERO
```

The two comparisons are mutually exclusive, so both lines will never be displayed at the same time for the same X

This program would be easier to read if we had some way to say, "If X < 0, display 'NEGATIVE'; *otherwise*, display 'POS/ZERO'." That would mean we'd need only one comparison rather than two. We already know that both comparisons won't be true and that both won't be false: if one is true, the other must be false. As you might expect because we're discussing this, TI-BASIC has just such a trick up its sleeve, the `Else` command.

Found under [PRGM][3], the `Else` command goes between `Then` and `End`. It separates a block of code above it that's executed when the associated conditional comparison is true and the block of code below it that's executed only when the comparison is false. You can't use `Else` without `Then` and `End`, so every `If/Then/Else/End` construct
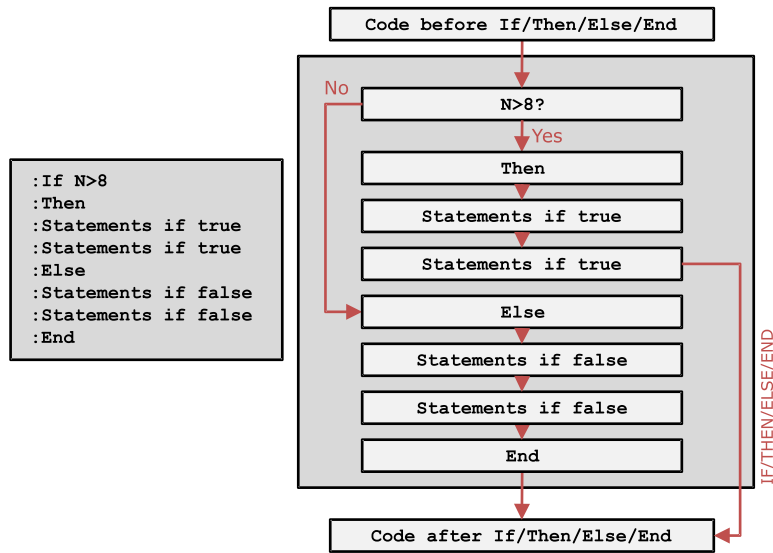
Figure 3.9   The flow of an `If/Then/Else/End` construct. When the condition is true, the statements between `Then` and `Else` are executed. When the condition is false, the statements between `Else` and `End` are executed instead.

must look something like the left side of figure 3.9. You can see the flow of execution inside the code on the right side of the figure, which certainly has gotten more complicated looking at first glance than the simple one-line `If` conditional you saw diagrammed in figure 3.3. If you take a second look, you'll soon see it's not quite as bad as it appears. If you compare if to figure 3.6, where I taught you about `Then/End`, it's quite similar. Instead of the code between `Then` and `End` getting executed when the condition (here, N > 8) is true, now it's the code between `Then` and `Else` that's run. The other difference is that when the condition is false, the TI-BASIC interpreter doesn't jump straight to the code after the `End`. Instead, it runs the code between `Else` and `End` and *then* continues with the code after the `End`.

You might already be starting to see how you can use this to improve the IFSIGN program, executing one of the `Disp` statements between `Then` and `Else` and the other `Disp` between `Else` and `End`. If you wrote the program out in full, it would look like this:

```
PROGRAM:IFSIGN2
:Disp "DISPLAY SIGN OF
:Input X
:If X<0
:Then
:Disp "NEGATIVE
:Else
:Disp "POS/ZERO
:End
```

Notice that even though there's only one statement executed if the condition is true and one statement if it's false, you can't omit the `Then` and the `End`. If you want to use `Else`, you have to use `Then` and `End` as well. Here, if X < 0, then "NEGATIVE" is displayed; otherwise (that is, if X ≥ 0), "POS/ZERO" is displayed. One of the interesting things about using `Else` is that if you reverse the conditions under which the comparison is true and swap the code between `Then` and `Else` and the code between `Else` and `End`, the program works exactly the same way. Here's the same program with that switch made:

```
PROGRAM:IFSIGN3
:Disp "DISPLAY SIGN OF
:Input X
:If X≥0
:Then
:Disp "POS/ZERO
:Else
:Disp "NEGATIVE
:End
```

The condition is now X ≥ 0 instead of X < 0, the code to be executed when the condition is true is `Disp "POS/ZERO`, and the code for false is `Disp "NEGATIVE`. Because both the comparison and the conditionally executed statements were reversed, the program performs exactly the same way. There's no right or wrong order to choose; it's entirely up to you.

You now know many things that you can do with conditions and the `If` command, executing code only when a given condition is true or false. But you'll find that in many cases, checking a single condition isn't good enough. You might want to run a section of code if only two different conditions are true or if any of three conditions are true. Maybe you even want to change the behavior of the simple one-line `If` command to execute its associated line of code when the condition is false instead of true. With Boolean logic, you can do all of this and more.

3.3   *Boolean logic*

Every conditional statement I've shown you up to this point has used a single comparison to decide whether to run a piece of code. What if you want to execute a piece of code if either of two conditions is true? Say you ask the user to type their name, and you want to tell them "THAT'S MY NAME TOO" if they type either "CALCULATOR" or "TI-83+." You could use the same `Disp` command twice, once a comparison against "CALCULATOR" and once with a comparison to "TI-83+." But this makes you repeat pieces of code, something you should avoid if you want to make your program as small as possible. What if instead you want to ask a user for a value and two bounding numbers and tell them if the number is between those two bounds? You would need to make sure that it was both below the upper bound and above the lower bound, but you don't have a good way to do that yet. You could put one `If/Then` block inside another block, so that the innermost code would only run if both `If` conditions were true, but again, there's a better way.

The solution is Boolean logic. Your TI-83+ or TI-84+ calculator provides four logical operators: and, or, xor, and not. The first three operators join two separate conditions into one large condition, whereas the not operator reverses the truth of a single condition. In this section, I'll first show you how each of the four logical operators works. I'll continue by introducing grouping parentheses as used for logic and conclude with a practical Boolean logic application, performing bounds checking.

I'll begin by showing you each of the logical operators and showing examples of them in action, along with *truth tables* for each operator.

3.3.1 *Truth of logical operators*

Every logical operator takes either one or two operands, which have already been evaluated as either true or false, and produces either true or false. In the following discussion, A and B both represent comparisons that produce true or false. These could be something like X<5, or A+3B=99, or even Str3="MATH". They don't represent the literal variables A and B.

The functions of the and and or operators are fairly intuitive and are summarized in table 3.2. First, an explanation of each of the four operators:

- and—If some comparison A is true, and another comparison B is also true, then A and B is also true. If either A or B is false, the expression A and B is false. In other words, both A and B must be true for A and B to be true.
- or—By the same token, A or B is true if either one of A or B is true. If A is false and B is true, A or B is still true. The expression is false only if A is false and B is false.
- xor—The xor operator is more confusing to most new programmers; it's true when exactly one of its two comparisons is true. If A is true and B is true, or A is false and B is false, then A xor B is false. If exactly one of the comparisons is true and the other is false, then A xor B is true.
- not—The not operator takes only one comparison and reverses its truth. If A is true, then not(A) is false; if A is false, not(A) is true.

To see how the logical operators work more clearly, listing 3.2 shows a program called TRUTH that will let you enter values for two variables, C and D, and produces

Table 3.2 The truth tables for and, or, xor, and not. Given some Boolean value for A and another Boolean value for B (shown in the left column), each of the four logical operators produces either true or false, as shown in the remaining columns.

Truth of A/B	A and B	A or B	A xor B	not(A)
A false, B false	False	False	False	True
A true, B false	False	True	True	False
A false, B true	False	True	True	True
A true, B true	True	True	False	False

the values of the four logical operations on those values. Here, you (as the program's user) will be asked to enter 0 or 1 for C and D, representing the logical values false and true.

Listing 3.2   Testing the truth values of the four Boolean logical operators

```
PROGRAM:TRUTH
:ClrHome
:Disp "ENTER 0 OR 1 FOR","C AND D
:Prompt C,D
:Output(5,1,"C and D=
:Output(5,9,C and D
:Output(6,1,"C or D=
:Output(6,8,C or D
:Output(7,1,"C xor D=
:Output(7,9,C xor D
:Output(8,1,"not(C)=
:Output(8,8,not(C
:Pause
```

Display the string "**C and D=**" followed by the actual value (0 or I, false or true) of that logical operation)

Repeat for or, then xor and not

You can find the four logical operators in the LOGIC menu, under [2nd][MATH][▶]. As with all of the commands I've shown you thus far, you can't type in the words *and, or, xor,* and *not* letter by letter; you have to paste them from the LOGIC menu as mentioned in the sidebar "Tokens versus text" in section 2.1.1. The TRUTH program in listing 3.2 displays the four logical operators on the bottom four lines of the homescreen and then shows the value of each operator on A and B (or just A for not). To see more clearly what I mean, look at figure 3.10. If any of the values produced in figure 3.10 don't make sense to you, or indeed if any of the source code of TRUTH is confusing, take a look back at table 3.2.

You can even join multiple operations. With and and or, you might want to either require all three of a set of conditions to be true or allow one of three true comparisons to make the entire statement true:

```
:If A=1 and B=3 and C=6
:<Command>
:If K=12 or K=13 or K=14
:<Command>
```
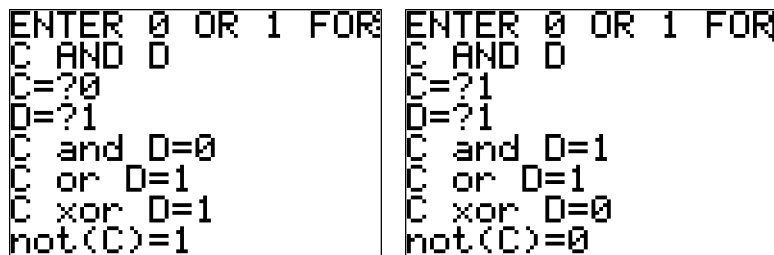

Figure 3.10   Two sample runs of the TRUTH program to generate truth values for Boolean logical operators applied to two logical values. The left screenshot shows C = false, D = true; the right screenshot shows both C and D equal to true.

If you start mixing and, or, and xor operations together, though, you might get unexpected results. The designers of TI-BASIC anticipated this, so after I show you details of the problem, I'll show you the logical solution.

3.3.2 *Using logical grouping parentheses*

Imagine a game where the player controls a ship fighting in space and wants to fire a special weapon. Say that the amount of ammo the player has for that particular weapon is in variable A and the player's level is in variable L. You want the player to be able to shoot the weapon as long as they have ammo and are also level 2 or level 3. Before you knew and and or, you might have written the code like this:

```
:If A>0                                    Only check the level L if
:Then                                      the player has ammo
:If L=2
:Then                      Putting If/Then/End
:<Shoot weapon>            statements inside If/Then/End
:End                       statements is called nesting
:If L=3
:Then
:<Shoot weapon>
:End                                       End the outer ammo
:End                                       check conditional
```

Now that you know and and or, you might realize that you can condense the three If statements into a single conditional:

```
:If A>0 and L=2 or L=3
:Then
:<Shoot weapon>
:End
```

This is neat and tidy code, but it has a problem. What exactly is it trying to express? Can you fire the weapon if L = 3 but A = 0, for example? TI-BASIC lets you use grouping parentheses to control the order of operations, which is used to clarify this sort of code. Because you want L=2 or L=3 to be evaluated first and then anded with A > 0, you can group L=2 or L=3 with parentheses:

```
:If A>0 and (L=2 or L=3)
:Then
:<Shoot weapon>
:End
```

The parentheses for grouping logical operations work just like parentheses in mathematical equations: the contents of the parentheses are evaluated before the rest of the equation.

What if you instead wanted the player to have at least two pieces of ammo if they're level 2 in order to fire but let one piece of ammo be sufficient for level 3? You could join two grouped and expressions with an or. The resulting code would look like the following:

```
:If (A>1 and L=2) or (A>0 and L=3)
:Then
```

```
:<Shoot weapon>
:End
```

In chapter 10, you'll see how you could reduce that from four to two comparisons; for now, the preceding statement is compact enough.

Most concepts make more sense when used in full programs, so let me show you Boolean logical operators used to make the THENEND character-mirroring program from section 3.2.2 more compact.

3.3.3   *Applying Boolean logic: bounds checking*

If you refer back to listing 3.1, you'll see that I presented a program called THENEND that lets the user pick a row and column location on the homescreen and then displays an O there. If you tell the program to do so, it also mirrors the O horizontally and vertically on the homescreen. In order to make the program more *defensively* programmed, a concept I'll touch on in more detail in chapter 5, the code checks that the row and column that the user enters are actually valid homescreen coordinates. That excerpt from `prgmTHENEND` is as follows:

```
:Input "ROW=",A
:Input "COLUMN=",B
:If A<1
:1→A
:If A>8
:8→A
:If B<1
:1→B
:If B>16
:16→B
```

Of course, this makes the character appear at some point on the screen no matter what numbers the user enters. If they type "-999" for the row and "-999" for the column, the conditional statements will modify the row to 1 and the column to 1. (Refer to the four conditional statements just shown if you don't immediately see why that is.) That might not be the desired behavior for your program, though. Instead, in this section I'll make it reject the coordinates if they're invalid and not draw anything.

A valid set of homescreen coordinates must consist of a row from 1 to 8 and a column from 1 to 16. You could combine the four comparisons in the code fragment from the THENEND mirror program to create a conditional that's true when the coordinate pair (row,column) = (A,B) is invalid:

```
:If A<1 or A>8 or B<1 or B>16
:Then
:Disp "OFF-SCREEN
:Else
:<Draw the "O"s>
:End
```

The conditional statement on the first line of this code is true if at least one of the individual comparisons is true, because they're all joined with `or` logic. This is what

you want to happen, because any of the four comparisons being true means one of the coordinates is offscreen. An important skill is to be able to flip around a conditional statement to reverse when it's true and false. One possible solution is the not operator, which I haven't used much so far. not would make the code look like this instead:

```
:If not(A<1 or A>8 or B<1 or B>16)
:Then
:<Draw the "O"s>
:Else
:Disp "OFF-SCREEN
:End
```

There is another solution that doesn't involve not. If you want the condition to be true only when all of the coordinates are valid and false otherwise, you'll use four comparisons joined with and. Each of the comparisons refers to one edge of the screen, a concept shown in figure 3.11.

If you want a conditional checking the character coordinates against each edge of the screen to be true when the character is onscreen and false if the coordinates are past any of the edges of the screen, you can and together four comparisons that are each true if the character is on the correct side of one edge. In figure 3.11, the character is above the bottom edge of the homescreen if the comparison $A \leq 8$ is true. Because you want the full conditional statement (indicating whether the coordinates are valid) to be false if any of the four individual comparisons is false, you'll use the and command to join them:

```
:If A≥1 and A≤8 and B≥1 and B≤16
:Then
:<Draw the "O"s>
:Else
:Disp "OFF-SCREEN
:End
```

Keep in mind that all three of these formulations to determine if row and column A and B are on- or offscreen function identically and that none is any more correct that the others.

With the third of the three formulations, we can put together the full program, which we'll call MIRROR2 (although it could also be THENEND2). It will ask the user for coordinates and whether to mirror the character, draw the character(s) if they're
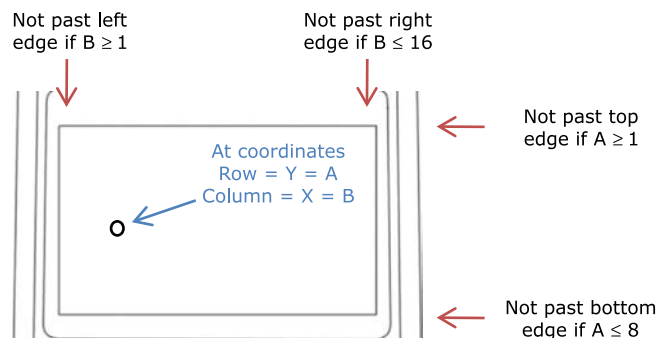
Figure 3.11 Performing a bounds check on the coordinates of a character on the homescreen. If all four of the comparisons shown are true, then the character is on the screen. If any one of them is false, the character is past at least one edge of the screen and should not be displayed.
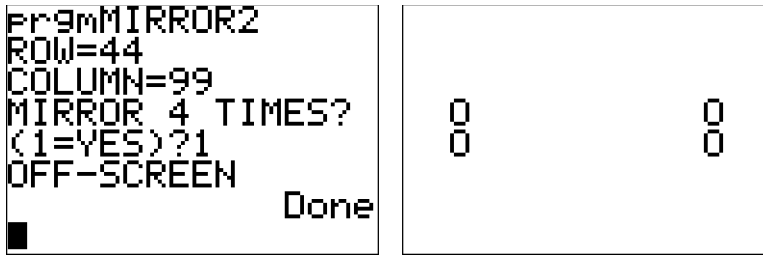
Figure 3.12   In the left screenshot, the bounds checking for the MIRROR2 program has detected that both the row and the column are invalid homescreen coordinates. For the right screenshot, a row of 5 and column of 3 were entered, and 1 (yes) was chosen for `mirror`. Because the row and column are valid, the proper output image shown was generated.

onscreen, or display an error message if they would be off the screen. The code for this program is in listing 3.3, and when run it resembles figure 3.12 with invalid (left) and valid (right) coordinates.

Listing 3.3   MIRROR2, demonstrating bounds checking added to THENEND

```
PROGRAM:MIRROR2
:Input "ROW=",A
:Input "COLUMN=",B
:Disp "MIRROR 4 TIMES?
:Input "(1=YES)?",M
:If A≥1 and A≤8 and B≥1 and B≤16        If the coordinates are within all
:Then                                    four screen edges, draw the Os
:ClrHome
:Output(A,B,"O
:If M=1
:Then
:Output(A,17-B,"O
:Output(9-A,B,"O          Otherwise, warn the user
:Output(9-A,17-B,"O       that the coordinates
:End                      entered were invalid
:Pause
:Else                                    Close the If/Then/
:Disp "OFF-SCREEN                        Else/End structure
:End
```

As you can see, the drawing code runs if the four comparisons that check for the edges of the screen are all true. If any one of them is false, then the conditional is false, and the code between the `Else` and the `End` at the end of the program runs instead.

You now have a toolset to skip pieces of a program, to jump forward across pieces of code without executing all of it line by line. What if you want to arbitrarily jump forward through a program or even jump backward to a previous line? This, too, you can do in TI-BASIC, and the next chapter will teach you all about how to do this.

3.4 *Summary*

You have now learned how rules for decisions are expressed in TI-BASIC, in the form of comparisons that are part of conditional structures. In any programming language, such comparisons are used to control the flow of execution and to dictate which pieces of code are executed and which are skipped. I discussed storing truth values as numbers in TI-BASIC, the different comparison operators, and the types of logical operators used to combine Boolean truth values. You also saw three increasingly complex ways to conditionally execute code, starting from a single line of code controlled by an `If` statement. The second method was the `Then` and `End` commands to conditionally execute a chunk of several lines of code. I concluded with the `If/Then/Else/End` structure that runs one of two chunks of code based on whether a comparison is true or false.

In chapter 4, you'll learn about more ways to control the flow of execution. The chapter you've just read introduced ways to skip forward past code; in chapter 4, you'll learn to jump backward, forward, and to arbitrary locations in your programs. You'll see how to generate loops that will cycle until some condition is reached. I'll show the value of executing programs from inside other programs and how you can add spiffy menus to your programs. The material in the next chapter depends on understanding conditional flow and comparisons, so with that material under your belt, onward.

Control structures 4
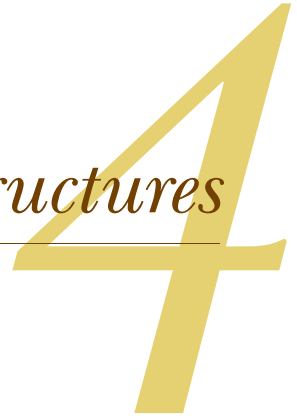
Conditional statements allow your program to make decisions given rules and facts, as you now know from chapter 3. If your programs could use these decisions to jump around within their source code and execute different sections, they could become vastly more powerful. With such commands, you could create whole new classes of programs and games. You'd be able to write your own programs like the GUESS guessing game in chapter 1 that we touched on briefly again in chapters 2 and 3. With loop and jump commands, you'd be able to write a science program that simulates the solar system, repeatedly stepping the planets along their orbits around the sun until told to stop. A math program could offer a menu of possible types of equations it could help you solve, and it could use jumps to go to different parts of the program to solve that equation depending on which option the user chooses.

In this chapter, you'll learn a plethora of control-flow commands that let your programs change their flow based on the value of variables. You'll learn how to make the program jump from one part to another with `Lbl` and `Goto`, skipping chunks of code or even going backward. I'll present the `Menu`, a shortcut to give users a visually attractive choice for jumping to several different parts of your program. I'll devote a large chunk of the chapter to showing you the three types of loops: `For`, `While`, and `Repeat`, which you may recall from chapters 1 and 3. Finally, I'll teach you the concept of recursion and termination using subprograms. By the end of this chapter, you'll have created several programs that use each of these techniques, including programs to calculate Fibonacci numbers and factorials, math tools that take averages and check if a number is prime, and a "screensaver" program that generates a virtual ownership sticker, demonstrated in section 4.4.1. More importantly, you'll have gained the skills to create your own such programs.

The first control structure we'll discuss is `Lbl` and `Goto`, which together allow you to create unconditional loops or, when combined with conditionals, loops that can choose to recycle or to end after each cycle.

4.1 Labels and Goto

As your programs get longer and more complicated, you'll find that different areas of your program are responsible for different functions. You might create a math program that solves a number of different equations for its users. You might have a game with a help section, the game itself, and a third bit of code that runs when the game exits. In both cases, you need a way for your program to jump around from area to area of the code. `Lbl` and `Goto` offer such a system, respectively a way to name a line in your code and to jump to a named line of code. In this section, I'll teach you how to use these two commands and demonstrate them with an exercise in which you make a conditional loop using `Lbl` and `Goto`.

Let's begin with an explanation of the two commands, how to use them, and what they can enable your program to do.

4.1.1 Understanding Lbl and Goto

TI-BASIC handles jumps using two paired commands, `Lbl` and `Goto`. `Lbl`, short for Label, creates a named location in your program's source code to which the program can jump. `Goto` correspondingly tells the program to jump to a particular `Lbl` and resume executing at that point. The `Lbl` and `Goto` commands are both in the [PRGM] menu, in the first tab, CTL. `Lbl` is found at [PRGM][9] and `Goto` at [PRGM][0]. The two commands each take a single argument, the name of the current label or the name of the label to which the program should jump. To keep programs small, label names have strict limits. They should be:

- *Short*—One or two uppercase letters and/or numbers.
- *Meaningful*—I have a pattern of `Lbl` names that I use in many of my programs, including `Lbl AA` at the beginning of the program for the main menu, `Lbl H` at

the start of the Help section, and `Lbl Q` at the code that prints credits before the program quits. You can use this system or create your own.

- *Unique*—You shouldn't have two labels with the same name in the same program. You can have two labels with the same name only if they're in different programs.

That last bullet point in the label name guidelines brings up a limitation of `Lbl/Goto`. You can only use the commands to jump to a point in the current program; you can't jump into any other program. But there are techniques to run programs from other programs, which I'll explain before the end of this chapter.

Figure 4.1 provides a graphical view of using `Lbl` and `Goto` in a program. When the TI-BASIC interpreter reaches a `Goto` command, it searches for the corresponding `Lbl` with the name the `Goto` specifies, here `Lbl A`. If it can't find the `Lbl`, it produces an error message. The `Lbl` can be either before or after the `Goto`; here, it happens to be before.

Figure 4.1   A label or `Lbl` is like a flag stuck at a point in your program, a marker that the TI-BASIC interpreter jumps to when it sees a `Goto` command. It knows which `Lbl` to go to from the matching label name (here, A) on the `Goto` and `Lbl`.

Another important feature to note is that if the `Goto` command runs, execution will jump to the corresponding `Lbl`, but executing a `Lbl` has no effect on the flow of the program. In figure 4.1, after the first line of the program is run, the `Lbl` and the third line will also be run, but passing through `Lbl A` won't affect the flow of execution in any way.

One silly and occasionally useful thing to do with `Lbl` and `Goto` is create an infinite loop, a loop that never stops on its own. The following program, when run, will produce an infinite series of the phrase "INFINITE LOOP" alternating with the number 42 down the calculator's screen. You can stop it only by pressing [ON], the emergency interrupt key that you can use if a TI-BASIC program isn't ending on its own. The left side of figure 4.2 shows this program running.

```
PROGRAM:INFLOOP
:Lbl S
:Disp "INFINITE LOOP",42
:Goto S
```

Every time this program reaches the last line, `Goto S`, it searches for `Lbl S`, which it finds at the beginning of the program. It starts the program over, eventually reaches `Goto S` again, and starts over, ad infinitum.

Figure 4.2   Two uses of `Lbl`/`Goto` to create loops. At left, the uncontrolled infinite loop of the INFLOOP program. The right side shows a conditional loop that ends when X = 0, `prgmGOTOTIL0`.

You can also use `Goto` in a more controlled manner by combining it with `If`. The following simple program GOTOTIL0 (`Goto` Until Zero) asks the user for a number and loops as long as the number is not zero:

```
PROGRAM:GOTOTIL0
:Lbl SO
:Prompt X
:If X≠0
:Goto SO
:Disp "TERMINATED
```

**Loop back to Lbl SO
if X is not zero**

Each time this program goes through the code between the `Lbl` and the `Goto`, it asks the user for a value for X. If the value the user enters is not zero, then the `Goto SO` (named thusly to stand for "Start Over") command is executed. If X is zero, the `Goto` is skipped, the `Disp` runs, and the program ends. You can see it in action at the right side of figure 4.2.

Lbl, Goto, and memory leaks

The most common problem new programmers encounter when starting to use `Lbl` and `Goto` is memory leaks. As their programs run for a few minutes, they get slower and slower and eventually stop with an ERR:MEMORY error. When you first encounter this error, it can be quite baffling. The culprit is using `Goto` inside any of the control structures that include an `End` command, such as `Then/End`, `Then/Else/End`, `For/End`, `While/End`, and `Repeat/End`. If you put `Goto` inside any of those programming constructs, the calculator will be keeping track of an `End` it will never see. Every time the interpreter runs across a `For`, `Then`, or `While`, it knows it will eventually need to find the corresponding `End` and will make a note of where the `For`, `Then`, or `While` was. If the program jumps out of one of these, loops around, and jumps out of that same construct again, it will then be searching for *two* `Ends` that it will never see. As it loops repeatedly, it will be looking for more and more `End` commands that it will never find, eventually running out of memory to list them all. Luckily, when such a program finally ends, normally or with an error, you get the memory back, but you still should fix the program so that the offending `Goto` is paired with an `If` in the one-line `If/Statement` format introduced in section 3.2.1.

These two new commands can be used to replace the loop in the guessing game from chapter 1, creating your first full program with control-flow commands that you'll write yourself.

4.1.2   *Exercise: convert the guessing game to use Lbl/Goto*

One nice way to demonstrate the parallels between `Lbl`/`Goto` and looping constructs like the `Repeat` loop, mentioned in chapter 1 and covered thoroughly in section 4.3.3, is to convert a program with a loop to use a `Lbl` and a `Goto` instead. Recall from chapter 1 that the code for GUESS, the guessing game, looked like this:

```
PROGRAM:GUESS
:randInt(1,50)→N
:0→M
:Repeat G=N                      ◁        Repeat the loop from here
:Prompt G                                 to the End until G = N.
:If G>N
:Disp "TOO HIGH
:If G<N
:Disp "TOO LOW
:M+1→M
:End
:Disp "CORRECT AFTER:
:Disp M
:Disp "GUESSES
```

My challenge to you is to take this code and remove both the `Repeat` and the `End`. You'll need to use a condition with an `If`, and you'll need one `Lbl` and one `Goto`. You'll need to figure out how you can reconfigure the program to work exactly as it did before, asking the player to guess an unknown number and telling them higher or lower until a correct guess is made. Remember that every `Repeat` loop runs at least once; the condition with the `Repeat` doesn't get checked until the calculator encounters the corresponding `End` command. A `Repeat` loop is equivalent to writing "Repeat until <condition> is true"; this will help you avoid frustration when you're working with the GUESS source.

   Once you think you have the correct solution, read on to the next section and see if your solution matches mine.

LBL/GOTO GUESSING GAME: THE SOLUTION

Predictably, the necessary changes belong where the `Repeat` and the `End` commands were in the original program. As I reiterated, the condition on `Repeat` is checked when the corresponding `End` is encountered. Therefore, the `If` with its conditional and the `Goto` will be placed where the `End` was before, so that the `End` will cause the program to loop if no correct guess has been made yet. It will need somewhere to loop to: just as `Goto` tells the calculator to jump to a location in the program, recall that `Lbl` gives a specific location a name that `Goto` can jump to. Because you know that the `End` caused the TI-BASIC interpreter to jump back up to the `Repeat` if the condition wasn't true yet, here you can make the conditional `If` jump back up to a `Lbl` where the `Repeat` used to be if the condition is still false.

To summarize: the `End` command becomes an `If`/`Goto` pair, and the `Repeat` becomes a `Lbl`. This means that the loop structure remains intact, because after `Goto` jumps to the `Lbl`, execution will continue at the `Prompt`, flow downward, and eventually return to the `If`/`Goto`. If the `Repeat` condition is still false, the `Goto` will jump again, and the loop continues once again. The loop ends only when the condition becomes true. So far so good, presumably? There is, however, one gotcha. Because `Repeat` repeats a loop until the condition is true, I phrased the `Repeat` condition as G = N. The loop continues until G = N or as long as G ≠ N. Because `If` executes the statement directly underneath it only when its condition is true, and our statement to be conditionally executed is the `Goto` that jumps when no correct guess has yet been made, then the condition on `If` must be *true* if the guess is wrong. The comparison G ≠ N is true when the guess is not equal to the actual number (which would make the `Goto` jump to reloop) and false when the guess is equal to the actual number (which would skip the `Goto` and make the program `Disp` the guess statistics and end).

With these pieces of information in mind, and without further ado, the solution to this exercise is presented in listing 4.1. There's the `Lbl A`, a label name chosen to be short and simple, to which the `Goto` jumps after an incorrect guess. There's also the `If` G≠N/`Goto A` that reloops back to `Lbl A` as long as each guess is wrong and lets the program complete when the guess is correct.

Listing 4.1   GUESSLBL, the GUESS program with `Lbl`/`Goto` instead of `Repeat`/`End`

```
PROGRAM:GUESSLBL
:randInt(1,50)→N
:0→M
:Lbl A                          ◁──────────   Return here after an incorrect
:Prompt G                                     guess; the Repeat is here
:If G>N
:Disp "TOO HIGH
:If G<N
:Disp "TOO LOW
:M+1→M
:If G≠N                                       If this guess is wrong, return to
:Goto A                                       Lbl A to reloop; the End is here
:Disp "CORRECT AFTER:
:Disp M
:Disp "GUESSES"
```

Comparing the code for GUESSLBL to GUESS, you can see that a `Repeat`/`End` loop is equivalent to a `Lbl` and a conditionally executed `Goto`. When you get to section 4.3 and learn `For`, `While`, and `Repeat`, you'll see that each of these looping constructs is equivalent to a conditionally executed jump combined with a label that makes one of three unique sorts of loops. Before we get to the three types of loops, I'd like to show you a feature of TI-BASIC related to `Lbl` and `Goto`, the `Menu` command.

4.2   Menus

In many programs, you'll want to give the user a choice. Perhaps you want to list several areas of your program that the user can choose to jump to, such as one of several math or physics equation solvers, or to different levels of a game. Perhaps you want to ask the user or player for a yes/no answer or to choose a difficulty level for a game. In each of these cases, you could display a set of items with corresponding numbers and ask the user to type a number corresponding to their choice, an alternative shown at the left side of figure 4.3. But a more elegant solution would be a menu of choices from which they can make a selection by pressing [ENTER].

The solution is the `Menu` command in TI-BASIC, which lets you create simple and attractive menus that operate like the TI-OS's own menus. I'll show you how to use the `Menu` command, then move on to expand the now-classic GUESS game with a main menu.

4.2.1   Using the Menu command

The TI-BASIC `Menu` command offers a fast and visually attractive solution to giving users a choice. An example is shown to the right of its more rudimentary cousin in figure 4.3.

Observe that the `Menu`-based menu at the right side of figure 4.3 looks a lot like the calculator's own menus. You can use the up- and down-arrow keys to move the highlight up and down the menu, you press [ENTER] to select an item, and you can even press one of the number keys to pick an option faster. For example, pressing the [4] key would choose the QUIT option in this menu. There are two things that distinguish TI-BASIC menus from the menus in the rest of the calculator's OS. First, unlike menus such as the [PRGM] or [MATH] menus, TI-BASIC menus can have only one tab, so that you can't move left and right to other menus. Second, they can have at most only seven items and therefore can't scroll the entire list up or down.

The `Menu` command is in the first tab of the [PRGM] menu, the twelfth item in the list. You can scroll down to it or press [PRGM][ALPHA][PRGM] ([PRGM]"C") to paste it into your program. The `Menu` command takes at least 3 arguments and at most 15
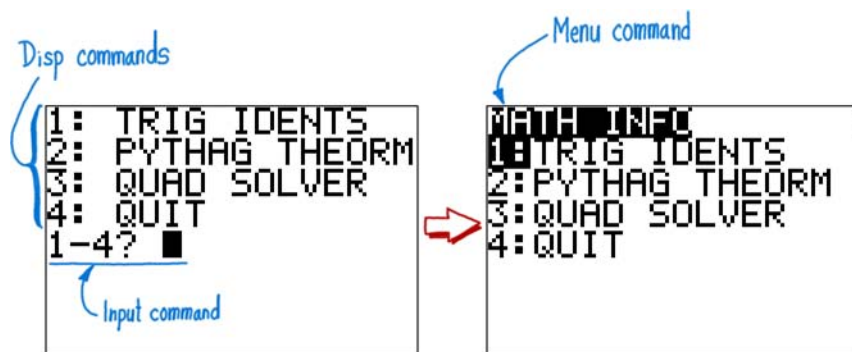
Figure 4.3   A crude menu that asks users to type a number corresponding to their choice, at left, with its fancier `Menu` equivalent at right

arguments. The first argument is always the title displayed at the top of the menu. The remaining arguments are in pairs, containing the text to display for an option and the Lbl to jump to if that option is chosen. Here are a few examples:

```
Menu("TITLE OF MENU","ONLY OPTION",AA
Menu("MENU TITLE","FIRST OPTION",T6,"SECOND ONE",OO
Menu("PICK A LETTER","A",0,"B",1,"C",2,"D",3,"E",4,"F",5,"G",6
```

The first example Menu line has one option, the second has two options, and the third has the maximum of seven options. The one- or two-character item after each option is the name of the corresponding Lbl for that option.

   I needed to teach you Goto/Lbl before you could learn about the Menu command because selecting any item in a menu is like having the program run a Goto: it will jump to a Lbl in your program. To show you what I mean, here are the code segments side by side to produce the two screenshots in figure 4.3:

```
PROGRAM:MENUA1
:ClrHome
:Lbl A
:Disp "1: TRIG IDENTS","2: PYTHAG
➡ THEORM","3: QUAD SOLVER",
➡ "4: QUIT
:Input "1-4? ",X
:If X=1
:Goto L1
:If X=2
:Goto L2
:If X=3
:Goto L3
:If X=4
:Return            ← This means "end the program"
:Goto A
:Lbl L1
:<...Trig identities...>
:Goto A
:Lbl L2
:<...Pythagorean Theorem solver...>
:Goto A
:Lbl L3
:<...Quadratic equation solver...>
:Goto A
```

If the user typed something other than l–4, jump back to the menu

```
PROGRAM:MENUA2
:Lbl A
:Menu("MATH INFO","TRIG IDENTS",
➡ L1,"PYTHAG THEORM",L2,"QUAD
➡ SOLVER",L3,"QUIT",Q
:Lbl Q
:Return            ← This means "end the program"
:Lbl L1
:<...Trig identities...>
:Goto A
:Lbl L2
:<...Pythagorean Theorem solver...>
:Goto A
:Lbl L3
:<...Quadratic equation solver...>
:Goto A
```

At the left side, the code for the menu that doesn't use the Menu command must check the value of X that the user types and jump to one of several Lbls based on its value. If it's not 1, 2, 3, or 4, then the program loops back and displays the menu again. It also displays the menu again after running any of the three math-related functions of the program. The Menu-based code on the right side, by contrast, uses the Menu itself to specify where to jump. Notice that each item in the menu such as TRIG IDENTS is followed by another argument such as L1, the name of the Lbl to jump to if that option is selected.

4.2.2   *Example: add a menu to the guessing game*

As the name implies, the `Menu` command works well for the so-called main menu, the area of a program where you choose what particular feature you want to go to. For a calculus program, this might be whether you want to take an integral or a derivative or look at reference tables. For a physics program, it might be choosing what class of function you want to solve. For a game, it could be whether to play the game, view a help file, or quit. I'll show you how you can expand the modified guessing game, GUESSLBL, to have a main menu. I'll title the menu "GUESS A NUMBER," which is luckily less than the 16-character limit for a `Menu` title, and it will have three options. PLAY will go to `Lbl G`, the beginning of the game itself, where it picks a random integer to store into N. HELP will go to `Lbl H`, which will display help, pause, and then return to the main menu. QUIT will go to `Lbl Q`, which will display some ending text and then terminate the program. Since both PLAY and HELP will eventually lead back to the main menu, we need the main menu to have its own label; let's call it `AA`. This menu will resemble the screenshot in figure 4.4.

Figure 4.4   **The main menu of the GUESSMNU program, created by adding a `Menu` command and a few other pieces to the GUESSLBL program**

To get an idea of what this program will look like, here it is with only the `Lbl`s, `Goto`s, and `Menu` in place. Don't forget that you can omit parentheses that are immediately followed by the end of a line!

```
:Lbl AA
:Menu("GUESS A NUMBER","PLAY",G,"HELP",H,"QUIT",Q
:Lbl H
:<display help information and Pause>
:Goto AA
:Lbl G
:<the GUESSLBL program>
:Goto AA
:Lbl Q
:<display credits>
```

After the user plays a game, or the Help section ends, the program uses `Goto AA` to return to `Lbl AA`, where the main menu is displayed once again. If the user chooses Quit instead, then the program jumps to `Lbl Q`, the ending credits are displayed, and the end of the program file is reached, causing the program to terminate.

Now I'll show you the same program again with the three omitted sections included. The Help section is a simple `Disp` and `Pause`, the credits at the end reiterate the name of the program in this case (although for your own programs and games they might include your name), and the GUESSLBL program will be inserted from the code presented a short time ago. If you're typing this into your calculator, remember that you can paste the contents of one program into another using the Rcl (recall)

feature found by pressing [2nd][STO>]; details on this were in section 2.1.1. At any rate, the full source of the GUESSMNU (GUESS with main menu) program is in shown in the following listing.

```
PROGRAM:GUESSMNU
:Lbl AA
:Menu("GUESS A NUMBER","PLAY",G,"HELP",H,"QUIT",Q
:Lbl H
:ClrHome
:Disp "GUESS A NUMBER","BETWEEN 1 AND 50","GO FOR FEWEST","GUESSES
:Pause
:Goto AA
:Lbl G
:randInt(1,50)→N
:0→M
:Lbl A
:Prompt G
:If G>N
:Disp "TOO HIGH
:If G<N
:Disp "TOO LOW
:M+1→M
:If G≠N
:Goto A
:Disp "CORRECT AFTER:
:Disp M
:Pause "GUESSES
:Goto AA
:Lbl Q
:ClrHome
:Disp "THANKS FOR","PLAYING.
```

Label AA is the main menu; control returns here when a game ends

The game jumps here when the user chooses the HELP menu option

Return to the main menu after displaying Help information

The game starts here when the PLAY menu option is chosen

This is the inner loop; control returns here after each wrong guess

Loop again if the guess was wrong (or continue downward if it was correct)

After a completed game, go back to the main menu

Control comes here if the QUIT menu option is chosen

Figure 4.4 showed what the `Menu` command in this program will produce; for completeness, figure 4.5 shows the results from choosing the Help (left) and Quit (right) options from the main menu in GUESSMNU, because those sections are also new.

The `Menu` command is a good tool for many programs you'll write and is handy in making your programs both more professional looking and easier for your users to use.
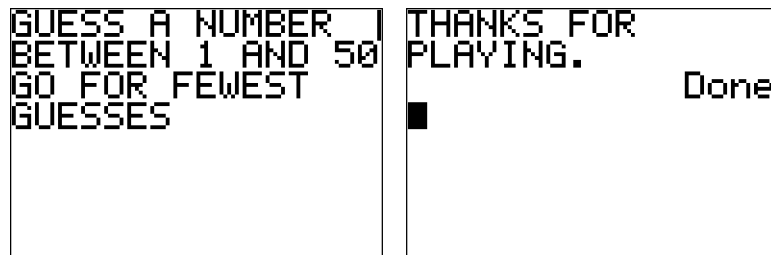
Figure 4.5   The Help section at `Lbl H` in the GUESSMNU program (left) and the Quit section at `Lbl Q` (right)

The `Lbl` and `Goto` concepts that underlie the `Menu` command can be used to make unconditional and conditional jumps and loops, as I've shown you. But using `Goto` can be a slow proposal; refer to the "Speed of Loops and Gotos" sidebar for more information. What if you want to create faster loops? What if you want to easily loop a specific number of times? How can you use a loop to do some sort of counting or iterating over a pattern of numbers? You'll now learn the `For`, `While`, and `Repeat` loops; loops are one of the last pieces controlling program flow that you need to know to create powerful, dynamic programs.

4.3   For, While, and Repeat

Your first exposure to loops in chapter 1 showed you how the `Repeat` command could create a looping guessing game that would continue to ask players for guesses until they guessed the correct number. In this chapter, you learned how you could create looping flow in your program with `Lbl` and `Goto`, which let you return to an earlier point in your program. TI-BASIC has three other ways to make loops besides using `Goto`, and you'll learn all three in this section. In almost every programming language, there are similar loop commands; for example, the C and Java languages both have `for` and `while` that match their TI-BASIC equivalents and something called `do/while` that's similar to `Repeat`.

The three TI-BASIC loop commands, `For`, `While`, and `Repeat`, all have distinct tasks that they're good at. `For` is good for loops that should execute a specific number of times and for counting up or down from one value to another. `While` is used to make loops that continue while a specific condition stays true. `Repeat` is used for loops that must continue until a condition becomes true. Each of the three types of loops begins with its respective command and some arguments, continues with the commands to be run in the body of the loop, and ends with the `End` command.

I'll begin with the `For` loop, which runs a predictable number of times based on the arguments provided to it; it's used for repetition and counting.

4.3.1   Repetition with For loops

Imagine that you want to make a program that can count up from 1 to 10, displaying all the numbers in between. Consider trying to write a program that iterates (cycles) through the numbers from -50 to 50, taking the multiples of 5 along the way and running them through some equation. Now that you've learned `Goto` and `Lbl`, you might be able to cobble these two programs together, but `Lbl` and `Goto` aren't the best solution in this case. The `For` command offers a simple way to create any sort of loop that either needs to count up or down or needs to run a procedure a fixed number of times. `For` loops always start with a `For` command, contain one or more lines of commands and statements, and conclude with an `End` command that tells the program to loop back up to the `For` command. It can be found under [PRGM][4], and as you already know, its corresponding `End` command can be typed with the keys [PRGM][7].

The `For` command takes three or four arguments: a variable to use to store the current counting value, the number to start counting at, the number to finish counting

The speed of loops and Gotos

As long as you manage to avoid creating memory leaks (see "Lbl, Goto, and memory leaks" earlier in this chapter), `Goto` and `Lbl` can be powerful and versatile tools for redirecting the flow of execution through your program. Unfortunately, with them comes the possibility of abuse. The `Goto` command is good but in many cases isn't the best tool for the job. Just because you can use a pair of tweezers to unscrew a screw doesn't mean you should use them instead of a screwdriver, and by the same token, `Goto` is often a slow choice compared with `For`, `While`, or `Repeat`.

When your program executes a `Goto` command, it saves its place, rewinds to the beginning of the program, and starts searching for the requested `Lbl`. If that `Lbl` is near the beginning of the program, the search will end quickly and the jump will occur. If the `Lbl` is after many lines of code, the program will have to spend a long time searching. Since your loop probably runs many times, and the calculator doesn't try to remember where it found the `Lbl`, it will go back to the beginning of the program and start searching for the `Lbl` again every time it reaches a `Goto` command.

When your program begins a `Repeat`, `For`, or `While` loop, on the other hand, it makes a note to itself that it may need to return to that point. When it reaches the corresponding `End`, it already knows exactly to where to jump in order to restart the loop, no matter where that `Repeat/For/While` is and how far it is from the beginning of the program or the `End` command. When you create a loop that you want to run often and fast, you should use one of the three looping constructs you'll learn in section 4.3. For jumps that are taken infrequently or aren't really loops, such as the menu examples shown in sections 4.1 and 4.2, `Lbl` and `Goto` are a good tool for the job.

at, and optionally, how much to add to the variable each time the loop runs through a full iteration (or pass). If the fourth argument is omitted, 1 is added to the variable after each iteration. The `Add` argument is more commonly called the *step* in programming parlance, as it defines how big of a "step" to take between `Start` and `End` after each iteration. The syntax of `For` looks like this, including its loop body and `End`:

```
:For(<Variable>,<Start>,<End>[,<Add>])
:Commands forming body of loop
:More commands
:End
```

To help you understand what happens inside the program when a `For` loop runs, I've provided the flowchart of a simple `For` loop in figure 4.6. This particular loop starts at 1 and ends at 5, with an increment (or add) argument of 2. The variable to be used for the loop, specified as the first argument to `For`, is X. This means that the first time the code between the `For` and the `End` runs, X will contain 1. When the `End` is reached the first time, the calculator will add 2 to X, so it will now contain 3. It will run the loop again, add 2 so that X contains 5, and run the loop a third time. When it finishes the third loop, it will see that X is already equal to the ending value for the loop, and will continue with the code after the `End` rather than running the loop a fourth time.
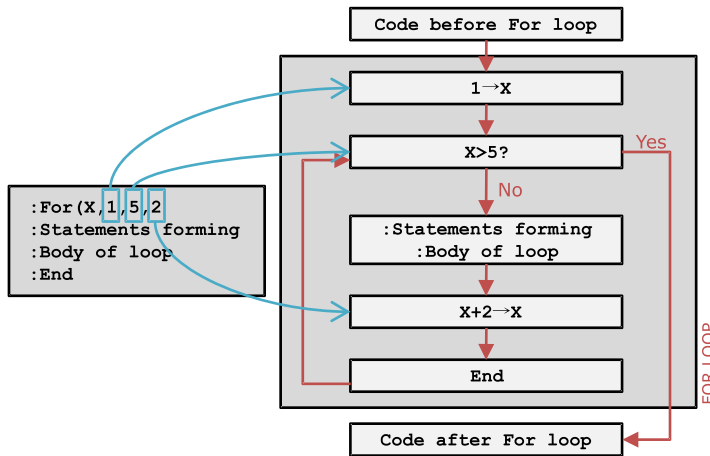
Figure 4.6   The structure of a `For` loop. The loop variable, X, is initialized to the start value (1) before the loop begins. Each time through the loop, the increment (2) is added to X. The loop ends when the loop variable X exceeds the termination value (5).

`For` loops can count either up or down; if the end argument is smaller than the start argument and the increment or add argument is negative, then `For` will decrease the variable each time through the loop. To start you off with a simple example, I'll show you how `For` can be used to simply count, displaying the value of the `For` loop's variable on each iteration through the loop.

EXAMPLE: USING FOR LOOPS TO COUNT

Let's jump right into a simple counting program that will display the numbers from 1 to 10 on the calculator's screen. It will use the variable X for the loop, a common choice, start from 1, and end at 10. Notice that in the first line of the COUNTUP program shown, only three arguments are given; as I mentioned, omitting the fourth argument, variously called the step, increment, or add value, makes the calculator assume you want it to be equal to 1. This little optimization lets you save some space in your program.

```
PROGRAM:COUNTUP
:For(X,1,10
:Disp X
:End
```

As you might expect from the description of `prgmCOUNTUP` and from reading the three lines of its source code, the output when the program finishes executing looks like the screenshot in figure 4.7.

How about if you want to make the program ever so slightly more advanced to test out more of `For`'s features? Let's say you decide to start counting at -10 instead of 1, and you want to step by 2 instead of by 1. This will make the program set X to -10, -8, -6, …, 8, 10, a total of 11 numbers; the source would be as follows:

Figure 4.7   The simple COUNTUP program that counts from 1 to 10 using a `For` loop. Because the output is longer than seven lines, the first four numbers have scrolled off the top of the screen.

```
PROGRAM:COUNTUP2
:For(X,-10,10,2
:Disp X
:End
```

If you decide to test this program, remember that the negative sign in front of the 10 that makes -10 is the [(-)] key in the bottom row of the calculator's keypad, not the minus or subtraction sign. Predictably, the middle of the output from running this program looks like figure 4.8.

Figure 4.8   A screenshot from the middle of running program COUNTUP2, counting from -10 to 10 in steps of 2

   Now that you've seen two examples where the program decides the values to use in the For loop, how about something more flexible that instead asks the user for each of the values? The code for such a program, COUNTASK, is shown here. It asks the user for A, B, and C, which respectively hold the start, end, and step values, and then uses those variables as the second, third, and fourth arguments to For. If you entered 1, 10, and 1, you'd replicate the output from COUNTUP; if you entered -10, 10, and 2, you'd reproduce the function of the COUNTUP2 program.

```
PROGRAM:COUNTASK
:Input "START=",A
:Input "STOP=",B
:Input "STEP=",C
:For(X,A,B,C
:Disp X
:End
```

If you haven't already tried it on your own, you could also enter something like 10, 0, -1 to watch the program count down from 10 to 0, or 0, -100, -4 to count down from 0 to -100 in multiples of 4. An example for running the loop from 8 to 128 in increments of 16 is shown in figure 4.9. As an interesting experiment, try giving values that don't make sense, such as Start = 40, Stop = 10, Step = 1. You'll see that the calculator responds by running the loop zero times, since it will never reach 10 from 40 adding 1 at each iteration. On a related note, if you enter 0 for the step, the calculator will produce

Figure 4.9   Running the COUNTASK program. The user enters a start value of 8, a stop value of 128, and a step of 16, so the For loop iterates from X = 8 to X = 128, adding 16 to X at each iteration.

an INCREMENT error, indicating that it will get stuck in an infinite loop trying to add 0 each time the loop restarts.

These smaller programs are nifty for demonstrating what happens to the For loop's specified variable as it runs, and they're great for leading to more complex and useful examples, as in previous chapters. I mentioned that For loops can be used for running a loop a set number of times as well as counting, so I'll show you a For loop used to repeat a chunk of code a precise number of times with a Fibonacci number solver.

EXAMPLE: FINDING FIBONACCI NUMBERS

The Fibonacci numbers are a series. For this particular series, each number is the sum of the two previous numbers of the series. The first two Fibonacci numbers are 1, which means the third number is 1 + 1 = 2, the fourth is 1 + 2 = 3, and the fifth is 2 + 3 = 5. The series continues infinitely. Fibonacci numbers appear in nature in the patterns at the center of sunflowers and as the areas of the squares forming the Golden Spiral (from the Golden Ratio). You can easily calculate the Nth Fibonacci number in the series with a For loop, as I'll show you.

This program will ask the user which Fibonacci number they want to calculate. For the first or the second number, the program displays 1, because the first two Fibonacci numbers are equal to 1. Otherwise, it stores 1 to A and B; A will represent the Fibonacci number two cycles ago, and B will hold the Fibonacci number one cycle ago. The current Fibonacci number will be in F. The program then runs N − 2 cycles for the Nth Fibonacci number; for the third number, it runs once, for the fourth, twice, and so on. Each iteration through the For loop, it calculates the current Fibonacci number from the sum of the previous two, which it stores in F. It then updates A and B to get ready for the next cycle. The purpose of the For loop here is to count exactly how many times it should calculate Fibonacci numbers before stopping and displaying the result.

The code for this FIBONACC program is shown in listing 4.3. As discussed, it first asks the user which number to calculate and then calculates and displays the result using a For loop. If you have any uncertainties either about how to calculate Fibonacci numbers or how this program does so, review what the limits on a For loop mean and how many cycles a loop from 3 to N will run for different values of N, and try following (in your mind) the flow of execution around and around the For loop. How defensive is this program? Will it work properly if the user enters N = 999? How about N = 0 or N = -3? The results of running the program for N = 6 and N = 20 are shown in figure 4.10.

Listing 4.3   Calculating Fibonacci numbers with a For loop

```
PROGRAM:FIBONACC
:Disp "CALCULATES NTH","FIBONACCI NUM
:Prompt N
:If N=1 or N=2
:Then
:Pause 1
```

Equivalent to Disp 1 followed by Pause

```
:Else
:1→A
:1→B
:For(X,3,N
:A+B→F
:B→A
:F→B
:End
:Pause F
:End
```

A will be the (X – 2)th Fibonacci number, and B will be the (X – l)th

F is the Xth Fibonacci number

Stop when X = N, when F holds the Xth = Nth Fibonacci number

The `For` loop is a powerful tool for counting, for controlled repetition, and as you may even discover on your own, for short pauses in your program. Unfortunately, if you don't know when you're writing your program how many times a loop should run, then `For` isn't a good choice. Instead, you should consider one of two loops that are controlled by a condition instead of a count, `While` or `Repeat`. Executed at least zero times and repeated as long as its governing condition remains true, `While` is a versatile option for such loops.

4.3.2   *Using While to loop*

The `While` loop is simpler than the `For` loop: instead of start and end values, a variable to modify, and a per-iteration increment value, the only argument a `While` command takes is a simple condition, which can be a plain comparison, several comparisons combined with logical (Boolean) operators, or even just a variable by itself. Simply put, each time an iteration of a `While` loop begins, it tests the condition given as its only argument. If the statement is true, the body of the loop runs and then returns to the `While` once it reaches its `End`. If the statement is false, however, it jumps directly to the first line following the `End` command. Because a `While` loop checks its condition at the beginning of each iteration, if the condition is false (zero) the first time it checks, the body of the loop will never run at all. This means that if the condition involves a value that you calculate inside the loop, you must *initialize* the variable used in the condition to a value designed to make the loop run its first iteration. It will then be able to calculate a "real" value for the variable inside the loop body and check the condition at the beginning of the second iteration.
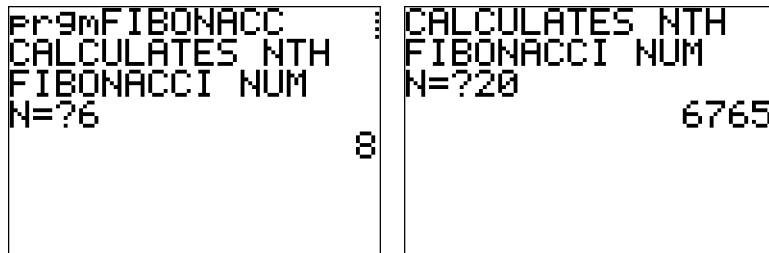
Figure 4.10   Calculating the 6th and 20th Fibonacci numbers using a `For` loop

The `While` command can be found with `For` and `Repeat` in the [PRGM] menu, the fifth item in the first tab; to quickly type it, press [PRGM][5]. As with the `For` loop, other than a quick three-line template for using the `While` command, I think the clearest explanation of what exactly happens in a `While` loop is the flow diagram in figure 4.11. First, the syntax:

```
:While <condition that will continue the loop until it is false>
:Statements and code forming body of loop
:End
```

Figure 4.11 shows how `While` loops are processed by your calculator's TI-BASIC interpreter. As with each of the loop and conditional diagrams I've shown you thus far, execution either starts in the code preceding the `While` loop or starts at the `While` command if it's the first line of the program. The first thing the loop does is check the condition attached to the `While`: if it's false, the program jumps directly to the code after the loop's `End` command, skipping the contents of the `While` loop. If the condition is true, the calculator executes the body of the loop, and once it reaches the `End`, it returns to the `While` with its associated condition to check if it needs to run the loop again. It will continue looping around until the condition becomes false.

You'll learn starting in chapter 5 that both `While` and `Repeat` loops are frequently used in games, where your program will want to keep checking if the player pressed a key but has no way of knowing when the user is going to press anything. They're also used for any sort of math or science equation where you need to run an *iterative* algorithm, an equation or set of equations that you have to run repeatedly until they produce numbers indicating they have *converged*, or found a solution.

One such equation checks if a given number is prime. It starts with 2 and checks whether each integer up to the square root of the given number is a factor of that number. If any such integer is a factor, then the user-supplied number is not prime. If the program finds no integer factors, then the number is prime. A `While` loop is perfect for this; it can be made to end prematurely if a factor is found or to stop when it reaches the end of the possible factors of the number and declares the number to be prime.
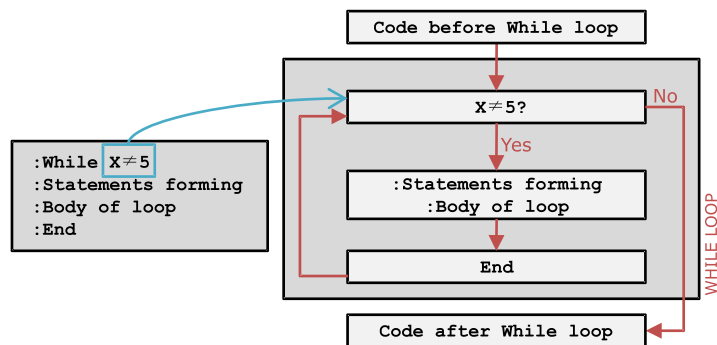
Figure 4.11   The structure of a `While` loop. The statements inside the loop are run *while* the condition is true. If the condition is already false when the loop first starts, the statements inside won't get executed at all.

EXAMPLE: CHECKING PRIME NUMBERS

In this example, I'll show you how to determine if a number is prime using a `While` loop. It uses a naïve algorithm that tests each possible integer factor of the number and continues until it either runs out of possible factors or finds a number that's indeed a factor. An integer factor is an integer that when multiplied by some other integer will produce the number in question. Prime numbers have only 1 and themselves as integer factors. Nonprimes have at least one other integer factor; the integer factors of 6 are 1, 2, 3, and 6, because by the commutative property $3 * 2 = 6$ and $2 * 3 = 6$.

The structure of the ISPRIME program, shown in listing 4.4, follows a common pattern that you've seen a few times before. It first displays instructions and asks the user to enter a value, initializes other values it will need, and runs a loop to execute the desired algorithm. When the loop terminates, it displays the results to the user in some readable format. This particular program uses three variables: X, T, and P.

The variable X holds the number that the user wishes to check for primality (whether it is prime). This number doesn't change after the user enters it.

The program uses T to hold the current test factor, the number in the sequence from 2 to the square root of X that are possible factors; each time it goes through the `While` loop, it adds 1 to T. Why only test up to the square root of X instead of up to X? Because if there's an integer factor of X larger than $\sqrt{X}$, then it must be multiplied by some value less than $\sqrt{X}$ to get X, and so the program would have already found that factor between 2 and $\sqrt{X}$. As always, you want your programs to be as small and fast as possible, so reducing the number of possible factors tested to a bare minimum is a good way to make the program fast.

The third variable, P, is used as a *flag* to represent whether or not the user-supplied number X is prime. The program starts with the assumption that X is in fact prime, so it initializes P to 1. If it runs through every possible factor and doesn't find any integer factors, then P is still 1, which means the number is prime. If it finds a valid integer factor while it's searching through factors, it sets P to 0, indicating X is not prime. As you might have guessed, 0 and 1 are chosen because 0 is false and 1 is true, so the contents of P are the truth value of the assertion "X is prime." To save further time when running this program, as soon as it finds one factor that means X is not prime, it can stop the `While` loop and not check any further potential factors. To add this condition to the `While` loop, ISPRIME introduces a new trick that I haven't shown you in any previous programs.

The loop in the ISPRIME program should end immediately if P becomes zero. Therefore, the condition on the loop could be written as follows:

```
:While P=1 and T≤√(X
```

Alternatively, since the program sets P to zero when a factor T is found that proves X is not prime, the condition could be rewritten as follows:

```
:While P≠0 and T≤√(X
```

In other words, continue the loop as long as P is not equal to zero and T is less than or equal to the square root of X. If P becomes equal to zero, or T becomes larger than √(X), end the loop immediately. But if you think carefully, the comparison P≠0 is false (that is, 0) when P is 0 and true (1) when P is 1. Therefore, since P≠0 is the same thing as P when P is 0 or 1, there's no point adding the ≠0 at all, and P can be used directly as a Boolean value.

Take a look at the ISPRIME program in listing 4.4, test it, and try to understand how it works. There are two tokens here that you might not yet know how to type. Although you have hopefully run across the square root symbol on your calculator at least in your nonprogramming use or perhaps in the QUAD program, you can type the symbol with [2nd][x^2]. The other token you may not know is int, which returns its input rounded down to the nearest integer; it's in the second tab of the [MATH] menu, at [MATH][▶][5].

Listing 4.4   The ISPRIME program for checking if a number is prime

```
PROGRAM:ISPRIME
:Disp "CHECKS WHETHER X","IS PRIME
:Prompt X
:2→T
:1→P
:While P and T≤√(X
:If X/T=int(X/T
:0→P
:T+1→T
:End
:If P=0
:Then
:Disp "IS NOT PRIME
:Else
:Disp "IS PRIME
:End
```

Continue while the number might still be prime and there are potential factors left to check

This comparison is true if X/T is an integer; int() removes anything after the decimal point, so if the number was an integer, it doesn't change it

When this loop ends, the number is either definitely prime or definitely not prime

Based on the description I've already provided, the code outside the While loop should be clear, including getting the value of X from the user, initializing P and T, then displaying whether or not X is prime with an If/Then/Else/End construct after the loop. The five lines of the While loop itself should also be straightforward. The loop continues either until the possible factor being tested is greater than the square root of X or until the loop determines that the number is not prime. If X/T is an integer, then X is the product of two integers other than 1 and itself: because the loop starts at 2 and ends at √(X), 1 and X are never tested as possible factors. T is then incremented so that the next potential factor can be tested on the next iteration of the loop. The results of the ISPRIME program for X = 50 and X = 101 are shown in figure 4.12.

Like most of the programs I've shown you so far, there are plenty of additions and changes that you could make to this program to make it more powerful or more correct. What if the user enters 0, 1, 2, or 3 as X into this program? Since T starts at 2, T will never

```
prgmISPRIME
CHECKS WHETHER X
IS PRIME
X=?50
IS NOT PRIME
              Done
█
```
```
prgmISPRIME
CHECKS WHETHER X
IS PRIME
X=?101
IS PRIME
              Done
█
```
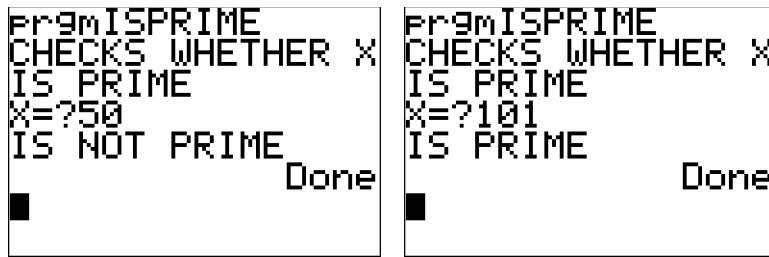
Figure 4.12   Testing the ISPRIME prime-number tester program with a nonprime number (50) and a prime number (101). This program uses a `While` loop to test possible factors.

be smaller than the square root of 0, 1, 2, or 3, so the program will correctly declare them all prime. But what about negative numbers? This program will fail with a NONREAL ANS error, because taking the square root of a negative number yields an imaginary number, one of the cases you need to worry about for a quadratic solver like QUAD. Because a negative number is prime if the negated (positive) equivalent is prime, you could solve this by taking the absolute value of X on the line after the `Prompt` command:

```
:Prompt X
:abs(X→X
```

The `abs` command is also in the MATH menu with the `int` command, at [MATH][▶][1]. I'm sure that you could find other things that could be done to this program, such as making it print all the factors of nonprime numbers, which would require changing the `While` loop's condition to not stop immediately when it finds the first integer factor.

　　`While` loops are the best choice of program flow construct when you want to create a loop that ends when a certain condition stops being true and you don't know in advance how many times you'll need to loop. If you still don't know the number of iterations you'll need but instead want to wait for a condition to become true that's initially false, then you should instead use a `Repeat` loop.

4.3.3   *The Repeat loop*

The `Repeat` loop is the final of the four ways you can create loops in TI-BASIC. I showed you the slow but flexible `Lbl`/`Goto` loop, which lets you jump arbitrarily forward or backward. I continued with the `For` loop, good for generating patterns of numbers or looping a set number of times. I just presented the `While` loop, which continues running indefinitely as long as a specified condition remains true. Like the `While` Loop, the `Repeat` loop starts with a command taking a single condition argument, continues with a loop body, and concludes with an `End`. Then what distinguishes it from a `While` loop? Two things:

- A `Repeat` loop continues *until* its governing condition becomes true. In other words, it loops only as long as the condition is false.

- The condition on a `Repeat` loop is checked *after* each iteration instead of before. This means that no matter what the condition is, even if it's the obviously true comparison 1 = 1, the loop will run at least once.

Once again, a diagram such as figure 4.13 is the best way to explain the concept of the `Repeat` loop. As you might expect from the two points I just mentioned, it strongly resembles the flow diagram of the `While` loop shown in figure 4.11. The main difference is that the condition, X≠5, is checked at the conclusion of each iteration of the loop rather than at the beginning. In addition, the "Yes" (true) case causes the loop to exit, because a `Repeat` loop runs *until* its condition is true.

EXAMPLE: AVERAGING AN ARBITRARY SET OF NUMBERS

One simple task well suited to a `Repeat` loop is getting a series of numbers from a user without having to ask them in advance how many numbers they'll enter. The program shown in listing 4.5, AVERAGE, surprisingly enough calculates the average of several numbers. Because the definition of an average is the sum of the numbers divided by the number of values, this program tracks the sum collected thus far in variable S and the number of values entered in N. Each time its loop repeats, it adds the newest value to S and increments N by one. The loop needs to end sometime, so the program defines 9999 as a special value. When the user enters the number 9999, it doesn't get added to the average. Instead, the `Repeat` loop ends, and the quotient S/N, the average, is displayed.

Notice that on the fourth line of AVERAGE in listing 4.5, the `Repeat` loop is defined to continue until X = 9999, but what if X is already 9999 before the AVERAGE program even begins executing? If this was a `While` loop, and the condition was `While` X≠9999, the loop wouldn't run a single time. To make matters worse, because S and N would both be zero, the last line of the program would try to calculate the undefined value 0/0, which would make the calculator produce a Division by Zero (DIVBY0) error. Luckily, `Repeat` saves the day. Because every `Repeat` loop runs at least once, X will be initialized during the `Input` command; there's no need for the program to set X equal to anything before the loop begins.
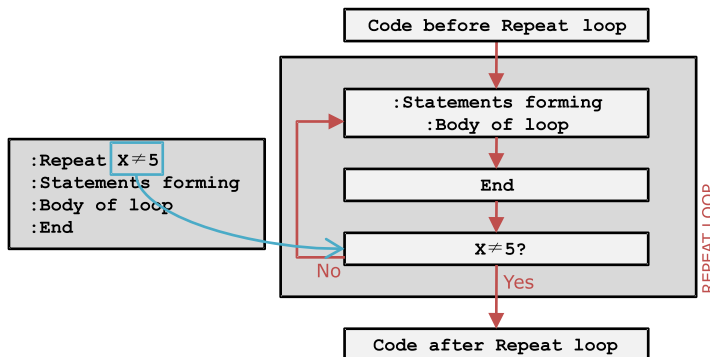
Figure 4.13   The structure of a `Repeat` loop. It's similar to a `While` loop, except that the condition is checked after the loop, not before, so the statements that form the body of the loop always run at least once. Also, it repeats *until* the condition becomes true.

Figure 4.14   Using a `Repeat` loop to calculate averages. The `Repeat` loop lets the user enter a long list of numbers and stops only when the user enters a "magic" number, 9999, that indicates all the numbers to be averaged have been entered.

Listing 4.5   The AVERAGE program that demonstrates a `Repeat` loop

```
PROGRAM:AVERAGE
:Disp "CALCULATES AVG","ENTER VALUES,","TYPE 9999 WHEN","DONE
:0→S
:0→N
:Repeat X=9999                                    Repeat this loop until
:Input "VALUE=",X                                 the user enters 9999;
:If X≠9999                                        it always runs at
:Then                    Don't average the 9999   least once
:X+S→S                   that marks the end of
:N+1→N                   the numbers to average   This End is part of
:End                                              the If/Then/End
:End
:Disp "AVERAGE=",S/N     This End loops back to the Repeat statement if X is not equal to 9999
```

The sequence of two screenshots in figure 4.14 shows `prgmAVERAGE` calculating the average of three numbers. The program carefully does not average the special "I am done entering numbers" value 9999 into the final average. Although avoiding initializing X saves some space in this program, there's still a danger of a divide-by-zero error if the user enters 9999 as the first value. Adding `:If N≠0` as the second-to-last line of the program would fix this easily. Finally, there's also the obvious problem that if the user wants to enter the value 9999 as part of an average, it's impossible with this program, but that's of minor concern.

CONVERTING BETWEEN WHILE AND REPEAT LOOPS

It's important to realize that most programs written using a `Repeat` loop could easily be switched to use a `While` loop instead, and vice versa. The AVERAGE program just presented could be changed to have a `While X≠9999` statement replacing the `Repeat` line. But because `While` conditions are checked before the first iteration through the loop, you would have to be careful what value X had before the loop started. If X was 9999 before your program even started, then the program wouldn't work. You'd therefore need to initialize X to some value other than 9999, such as 0:

```
PROGRAM:AVERAGE
:Disp "CALCULATES AVG","ENTER VALUES,","TYPE 9999 WHEN","DONE
:0→S
:0→N
:0→X
:While X≠9999
:Input "VALUE=",X
:<remainder of program>
```

Following the same logic, the ISPRIME prime number tester from section 4.3.2 could be modified to use a `Repeat` instead of a `While` loop. Its original `While` loop runs while P and T≤√(X), equivalent to `While P≠0 and T≤√(X)`; here's the code again:

```
:While P and T≤√(X
:If X/T=int(X/T
:0→P
:T+1→T
:End
```

In other words, the loop needs to repeat until either P = 0 or T > √(X). As you might imagine, that's exactly the condition to be used with the `Repeat` form of this loop:

```
:Repeat P=0 or T≤√(X
:If X/T=int(X/T
:0→P
:T+1→T
:End
```

There's no initialization code that can be removed from making this change, and the extra `=0` takes up more space, so unfortunately this program is better with a `While` loop. You'll find cases in your own programs where even though most `Repeat` and `While` loops can be converted into the opposite type, one or the other is better suited for the specific situation. Indeed, the different types of loop structures in every programming language each have places where they're the most elegant solution out of their siblings, as you'll learn if you pursue programming past calculators.

What if instead of running the same piece of code over and over with no breaks in between, you want to occasionally run a repeated chunk of code in the middle of otherwise linear code? Perhaps you've considered how you might make your program end in the middle, without having to let it get to the end of the source code to stop. You'll now learn how these two related concepts fit together, the final major tool in your arsenal of flow pieces you can fit into full, powerful programs.

4.4   *Subprograms and termination*

Every program that you've written so far has contained its entire code within itself, which may seem like an odd statement. Each program contains itself; where else would it be contained? If you're familiar with any other languages besides TI-BASIC, you might know about the concepts of libraries and subprograms. If not, both terms refer to putting pieces of your program's code into other reusable programs; these new programs are called libraries or subprograms. This lets you reuse the same code

in several programs without having to type out that code all over again each time you need it. Instead, you tell your program to *call* the program that has the function you need, meaning that it pauses its own execution, runs the contents of the other program, and resumes where it left off. Subprograms can also be used for a technique known as recursion, where a program calls itself repeatedly.

In order to use subprograms, you need to know two main concepts: how to have a program call another program and how to have any program (including a subprogram) terminate before the TI-BASIC interpreter reaches the last line of the program. In this section, you'll learn both of these skills.

4.4.1 *Putting repeated code in subprograms*

Though you may not know it yet, you've been running the command to call a program from another program since chapter 1. Every time you paste `"prgmNAME"` to the homescreen from the [PRGM] menu and press [ENTER] to execute that command, you're running exactly the command that your programs will use to call other programs, including themselves. When the name of a program appears in another program on a line by itself, prefixed with the `prgm` token, it signals the TI-BASIC interpreter to run that named program and return to the next line of code after the call in the program that caused the subprogram to be executed. The program that runs or calls the subprogram is called the caller, and the subprogram that's executed is termed the callee. The command in the caller program to execute a callee named MYPROG might look like this:

```
:prgmMYPROG
```

To type this on your calculator, you'll need to be able to type `prgmMYPROG` into your caller program. You can either go to the third tab of the [PRGM] menu, the EXEC tab, and choose MYPROG (assuming it already exists on your calculator), or you can type the `prgm` token with [PRGM]D (the 13th item), then type the letters of the callee program's name.

One of the primary uses for subprograms is to hold sections of repeated code that would be wasteful to type over and over again. As a first demonstration of subprograms, I'll show you a routine that draws a border of zeroes around the edges of the homescreen so that you can output text of your choosing in the center of the homescreen.

SUBPROGRAMS FOR CONCISE CODE

Because a subprogram can contain a single copy of a frequently used section of code, and the programs that would otherwise have many copies of that code can instead simply call the subprogram, subprograms save space. Here, I'll show you a subprogram that uses two `For` loops to draw zeroes along the edges of the homescreen. I'll also present a demo program that calls the subprogram several times, then writes text inside the border the program draws, displaying your name as your calculator's owner. I'll aim for results that look like figure 4.15.
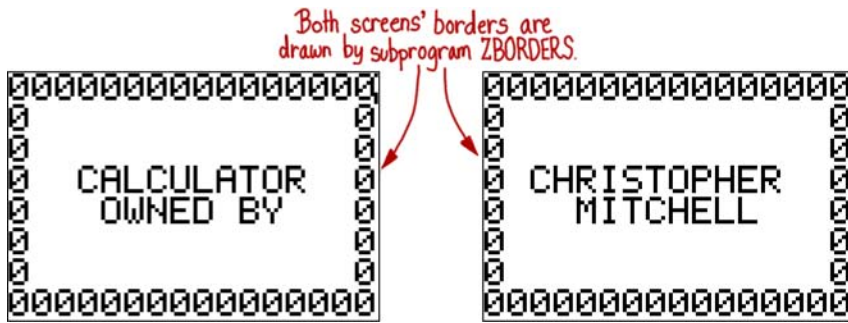
Figure 4.15   The OWNEDBY program demonstrates the use of a subprogram, ZBORDERS, to draw the borders of these screens, as well as an infinite `While` loop and variable delays created by `For` loops. The [ON] key can be used to terminate this program.

First, let's look at the subprogram. This program will draw borders around the edges of the screen, filling up columns 1 and 16 and rows 1 and 8 of the homescreen with zeroes. But it will want to erase it first. How shall we fill the edges? Because we know in advance exactly how many zeroes we need to draw, a pair of `For` loops would work perfectly. The first loop will draw horizontally along the rows, and the second will draw vertically along the columns. We'll call this program ZBORDERS:

```
PROGRAM:ZBORDERS
:ClrHome
:For(C,2,15
:Output(1,C,"0
:Output(8,C,"0
:End
:For(R,1,8
:Output(R,1,"0
:Output(R,16,"0
:End
```

Draw 0s along the top and bottom edges of the screen

Draw 0s along the left and right edges of the screen

Why does C go from 2 to 15 instead of 1 to 16? Because the second loop fills in the four extreme corners of the screen as it draws the vertical edges, there's no point writing over those zeroes again. This small optimization will save a few dozen milliseconds; in a larger program, the savings would be more substantial.

Now you need a main program that will take advantage of this subprogram, ZBORDERS. We'll call this program OWNEDBY, and it will alternately flash the text "CALCULATOR OWNED BY" and your name. I'll fill in my own name, and I'll leave it up to you to modify the program to include your own name.

```
PROGRAM:OWNEDBY
:While 1
:prgmZBORDERS
:Output(4,4,"CALCULATOR
:Output(5,5,"OWNED BY
:For(X,1,500
:End
:prgmZBORDERS
```

Erase the screen and draw borders along all the edges

Erase the screen and draw the borders again

```
:Output(4,3,"CHRISTOPHER
:Output(5,5,"MITCHELL
:For(X,1,500
:End
:End
```

Other than calling the ZBORDERS subprogram twice to erase the screen and redraw the border at the edge of the screen, this program demonstrates two relatively new concepts. The first is the condition on the `While` loop, namely the plain number 1. Recall that a `While` loop continues as long as the condition on the loop is true and that 1 represents true in TI-BASIC. This means that the condition on this loop, a constant true, can never be false, and the loop can never end: it's an infinite loop. The only way to end this program is to press the [ON] key. The second new concept is the two empty `For` loops. Both count from 1 to 500, but neither has anything inside the body of the `For` loop, so you might be questioning why the loops are there at all. I added them to generate a slight pause in the program by wasting time, but they end without requiring the user to press [ENTER], which is why I'm using this technique instead of the `Pause` command. If you experiment with the ending value for the cycle, you'll find that the amount of time the program pauses before updating the screen changes as well. Change the 500 to a 1000, and each screen will be displayed for twice as long. In a later chapter, I'll teach you a better and smaller way to create this sort of a delay. A final note: the outer `While` loop is an infinite loop, so you must use [ON] to terminate the program (which triggers an ERR:BREAK message). You can stop any program at any point with [ON], which will be particularly useful when we discuss debugging.

Assuming that using subprograms to save space and your own time by removing the need to type out the same piece of code over and over again seems reasonable, I want to show you another trick that subprograms can be used for. Recursion is a technique in computer programming where a function or program calls itself repeatedly, and it's used for calculating certain types of numbers and mathematical series.

SUBPROGRAMS FOR RECURSION

One trick that subprograms can be used for is to create recursion. In a recursive algorithm, a routine calls itself, getting deeper and deeper into copies of itself calling copies of itself. Eventually, it reaches some condition, called a termination condition or base case, that tells it to stop calling itself and instead return back up through all its calls, finishing each call to itself and returning some final answer. A typical example algorithm to demonstrate recursion is the factorial function, represented with the exclamation point (!). Fibonacci numbers can also be calculated with a recursive program, although I demonstrated using a loop to determine such numbers.
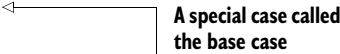
Returning to the factorial function, the value of N! (pronounced "N factorial") of any positive integer N is equal to the product of every integer from 1 to N, inclusive:

$$N! = N * (N - 1) * (N - 2) * \ldots * 2 * 1$$

You can calculate this value with recursion, because N! is also equal to N * (N – 1)!, or the product of N and the factorial of N – 1. If you write a routine that can calculate N!,

you can call it from itself for N – 1, multiply that number by N, and read out the answer. This seems like a confusing concept: if the function calls itself to perform most of the math, when do you actually do the calculation? Let me show you the algorithm in pseudocode first, and then I'll explain it in TI-BASIC:

```
Factorial(N):
    If N=1:
        Return 1
    Else:
        Return N*Factorial(N-1)
```

A special case called
the base case

This function has two possible cases. The termination condition is N = 1, because that case does not and should not call further down into more copies of the Fibonacci function. If N is anything else, then the function calls itself with N – 1 and multiplies that by N before returning upward. Recursion is a powerful technique for solving several different types of programs, which you'll likely encounter as you learn about algorithms in calculator or computer programming.

Return 1? What's that?

Thus far, all of our programs simply stop when they're finished. But the Factorial pseudocode program, as well as the ZFACT program we'll write in TI-BASIC, are functions. Functions in most languages can have *return values*. Consider the `randInt` function you explored earlier in the guessing game: it returns a random integer that the program then stores into a variable with the assignment or store (→) operator. Although TI-BASIC doesn't have a good way to express return values, one variable commonly used by convention to hold the return value from a program is the special variable `Ans` (or Answer). Chapter 10 will teach you more tricks with `Ans`.

With that background, let's create the full program now. We're going to need two programs here. The first program is called the driver program. It's the main function that the user runs; it sets up variables and other parameters for the recursive function, calls the second program that contains the recursive function itself, and then handles postrecursion tasks like displaying the results from the recursive algorithm. The driver function, which I'll call `FACTORL` (short for factorial) should look like this:

```
PROGRAM:FACTORL
:Input "FACTORIAL OF :",X
:prgmZFACT
:Disp "FACTORIAL IS",F
```

This program introduces a naming convention often used for subprograms that are pieces of other programs but shouldn't generally be run themselves by the user. The names of such programs should start with Z or θ; because the calculator displays the programs in the [PRGM] menu in ascending alphabetic order, these subprograms will fall at the bottom of the program list.

To guide you toward the contents of the subprogram that performs the factorial function itself, `prgmZFACT`, I'll start with figure 4.16; refer to the figure as you read this

paragraph. Each of the boxes in this diagram represents one program being run. Since programs can call themselves, conceptually pausing one copy of themselves before diving into another copy, the same program can appear multiple times in this diagram. Here, the `FACTORL` function is the outermost function, because it's run first. It asks the user for X, and for simplicity, I've written the diagram as if X is always 5. It then calls ZFACT, which subtracts 1 from X to get X = X − 1, and then calls (another copy of) ZFACT. This repeats, with each ZFACT calling under ZFACT (comments A, B, and C in figure 4.16), until in the fifth ZFACT down with four other ZFACTs still waiting, X is equal to 1. This innermost ZFACT sets F, the output number, to 1, because 1! = 1. It then ends, and the last few lines of the fourth ZFACT down run. These add 1 to X again, because this ZFACT corresponds to X = 2 and multiplies 1! by 2 to produce 2! = 2 (comment D in figure 4.16). This ZFACT ends, returning to the third ZFACT, which was patiently waiting. The third ZFACT sets X = X + 1 = 3 and ends with F = 3! = 3 ∗ 2! = 6 (comment E in figure 4.16). The second ZFACT resumes and repeats the same task, concluding with X = 4 and F = 24. Finally, the outermost ZFACT returns with X = 5 and F = 120. With the `prgmZFACT` it called finished, `prgmFACTORL` concludes by displaying F = X! = 5! = 120.
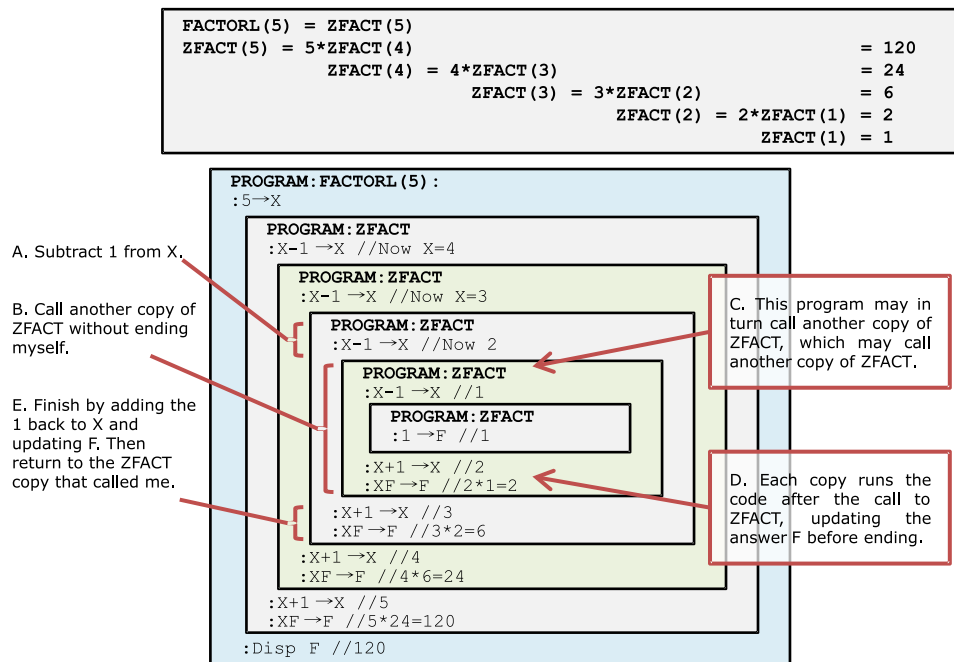
Figure 4.16   A diagram of what happens when `prgmFACTORL` is run to find the factorial of 5 (or 5!). `FACTORL` calls ZFACT, which keeps calling additional copies of itself and subtracting one from X until X = 1. It then sets the output value F to 1 and starts returning up through the copies of ZFACT, multiplying the current X by F and saving the result in F before ending that copy of ZFACT. Eventually, the last copy of ZFACT ends, and `FACTORL` regains control, displaying the value of F. The items after the // marks are comments, which aren't part of the code that you type onto your calculator, and are there for your edification.

From this description and figure 4.16, the code inside the ZFACT program might be written as follows. Many languages have something called variable scope, which means that if `prgmFACTORL` has a variable F and calls `prgmZFACT`, ZFACT will have its own version of variable F that doesn't change the value of F that `prgmFACTORL` sees. In TI-BASIC, every program sees the same versions of the variables, so you'll need to do some tricks to avoid accidentally erasing the numbers you're trying to calculate. Try this on for size:

```
PROGRAM:ZFACT
:If X=1
:Then
:1→F
:Else
:X-1→X
:prgmZFACT
:X+1→X
:XF→F
:End
```

As you can see, the input to this function is X, the number to calculate the factorial of. If X = 1, it returns 1! = 1 stored in F. Otherwise, it calls itself on X − 1 to find F = (X − 1)!, then multiplies X by F to produce F = X!

To type this on your calculator, you'll need to be able to type `prgmZFACT` into your two programs. As I mentioned, you can type out the program name yourself prefixed with the `prgm` token from the first tab of the [PRGM] menu or find `prgmZFACT` in the third tab if you already created it.

This recursion discussion was likely one of the most challenging concepts to understand that I've covered thus far, so I encourage you to review it and make sure you understand it before you continue. As an exercise, see if you can translate the Fibonacci solver into a simple recursive program or, conversely, calculate the factorial function iteratively with a `For` or `While` loop. Both can be done efficiently and cleanly. If you've absorbed the several roles of subprograms in TI-BASIC, join me for the final concept in the chapter, termination, which will be a breeze compared with recursion.

All programs need to end sometime, and although you're now familiar with programs that stop after they reach the last line of code, there are ways to make programs end sooner. The `Return` and `Stop` commands are cousins with similar behavior that you can use to prematurely stop a program from executing.

4.4.2   *Termination: Return and Stop*

For quite a few sections, you've learned increasingly elaborate ways to make execution proceed through your programs, to loop and jump around, to iterate, and to recurse. All good things must come to an end, and sooner or later you'll want your programs to relinquish control and cease running. For the most part, the programs I've been showing you have been crafted to read the last line of the program, at which point the calculator realizes that there's no more code in the program that can be run and promptly terminates it. But you'll want other ways to make your programs end.

For one thing, you may want to be able to end a program without making it flow to or jump to the last line of the program. In the two programs MENUA1 and MENUA2 in section 4.2, I briefly presented the `Return` command for the first time, which is the sole command at `Lbl Q` and which is run when the user chooses the QUIT menu option. That command is one of the two I'll explain in slightly more detail here. The other reason you might want to make a program stop is if something unexpected or unwanted happens, or your program reaches a condition that should make it stop before it gets to the last line of code.

Two similar commands will give you the power to do this: `Return` and `Stop`. Both of these commands are in the first tab of the [PRGM] menu, items E and F respectively. From a cursory examination, both programs do the same thing: they stop the program in its tracks, making it immediately terminate. But they have one difference. If you use it in a subprogram, `Return` will only make the current program end, letting whichever program called the subprogram continue unimpeded. On the other hand, `Stop` will immediately halt the current program, any program that called it, and any program that called that program, all the way up the chain. If you added a `Stop` in the innermost `prgmZFACT` in figure 4.16, all of the programs within programs would end immediately, without running the remainder of their code.

To demonstrate the differences between `Return` and `Stop` more distinctly, look at the following pair of programs, TESTRET and TESTSTOP. The main driver programs, TESTRET and TESTSTOP, have corresponding subprograms, ZRETURN and ZSTOP, respectively. ZRETURN has a `Disp` command at either side of a `Return` command, and ZSTOP has the same structure with a `Stop` command replacing the `Return`. Here are the four programs; the bottom program in each of the two columns is called by the program directly above it:

```
PROGRAM:TESTRET              PROGRAM:TESTSTOP
:Disp "BEFORE PRGM CALL      :Disp "BEFORE PRGM CALL
:prgmZRETURN                 :prgmZSTOP
:Disp "AFTER PRGM CALL       :Disp "AFTER PRGM CALL

PROGRAM:ZRETURN              PROGRAM:ZSTOP
:Disp "INSIDE CALL           :Disp "INSIDE CALL
:Return                      :Stop
:Disp "AFTER RETURN          :Disp "AFTER STOP
```

Observe what happens when `prgmTESTRET` and `prgmTESTSTOP` are run, as shown in figure 4.17. In both pairs of programs, the last line of the subprogram doesn't run, so neither "AFTER RETURN" nor "AFTER STOP" is displayed. But because `Return` only ends the current program, "AFTER PRGM CALL" is still displayed in TESTRET. I said that `Stop` ends every program running, even ones that called subprograms, so "INSIDE CALL" is the last line displayed when TESTSTOP is run, and the `Disp "AFTER PRGM CALL` in TESTSTOP is never executed.

I strongly recommend that you use `Return` whenever possible and try to avoid using `Stop`. Although the Doors CS shell for the TI-83+/84+ (see appendix C) works properly with TI-BASIC programs that use `Stop`, the older MirageOS shell crashes

**Figure 4.17   Two programs that call subprograms. The left program uses a
`Return` command to end the subprogram; the right program uses a `Stop`. Notice
that the `Stop` command quits directly to the homescreen without allowing the
caller program to complete.**

when it encounters a `Stop` token in a BASIC program. In some cases only the `Stop`
command will produce the desired effect, but when subprograms aren't involved,
always use `Return`.

4.5   *Summary*

In the past few sections, I've introduced the various ways to repeat sections of code,
from loops to subprograms, and shown you jumping around programs with labels,
`Goto`, and menus. The examples provided go a long way toward making the concepts
more concrete and understandable, but there's no substitute for designing, writing,
and testing your own programs. You'll absolutely find things that don't work or that
you don't understand, work through the problems, and learn more about trouble-
shooting your own programs.

Before you move on to the next chapter, I hope that you've tried running most of
the programs in this chapter, and I encourage you to think of some games or math or
science programs that you can try to write with the concepts you learned in this chap-
ter. If you don't understand the difference between a `While` and a `Repeat` loop, make
a four-line program with each command and see how they behave differently. Stick
`Disp` commands in so you can see the contents of variables. Once you're satisfied, let's
continue to a set of methodical guidelines for imagining and creating your own pro-
grams and projects. In the next chapter, I'll give you a short break from learning new
programming commands and concepts in lieu of discussing tips and tricks for design-
ing, writing, and troubleshooting your own programs.

5

Theory interlude: problem solving and debugging

This chapter covers

- Confidently building your own programs
- Bringing ideas all the way from concept to finished program
- Testing and debugging techniques to find and fix errors
- The program-building progression in a full running program

In the past three chapters, you've been immersed in the essentials of TI-BASIC. From input and output commands to conditional statements and flow control, you now know the essential building blocks that go into any calculator (or computer) program. But if knowing the commands is like knowing the vocabulary of a language, and knowing how to use each command is like learning sentence structure, there's one more step that's key: you need to know how to put your sentences together into a coherent, flowing essay or novel. In addition, there's a world of difference between a story that makes sense and a story that is action-packed and full of detail without wasting a single word. In this chapter, you'll learn to construct the TI-BASIC equivalent of an enthralling story: a program that is fast, small, well-planned,

and well-written. You'll learn to track down the programming equivalents of spelling and grammatical errors: typos and mistakes in how your program works; I'll also show you how to cut out unnecessary filler to optimize your programs.

This chapter will teach all the generalities you need to write good programs for any platform, including calculators. You'll learn to go from idea to design to diagramming program flow and sketching interfaces to coding and testing. After I discuss general skills you can apply to any program in section 5.1, I'll take you through the process applied to a specific example in sections 5.2 and 5.3. The example I'll show you is a math program that solves for Pythagorean Triplets (A, B, C) that satisfy the constraint A + B + C = N, given a value for N that the user enters. I'll walk you through planning, diagramming, coding, and testing this calculator program, as well as the steps any good programmer goes through after completing a program: trying to think how it could be made better, faster, or smaller. The remainder of the chapter will be spent on tracking down and solving errors in your code, both things like typos and missing function arguments that make the calculator produce an error message and the more difficult, subtler errors that come from numerical, structural, or planning mistakes.

Let's begin with an overview of taking any program idea from the conceptual phases through to a complete, polished program.

5.1   *Introduction: idea to program*

Creating great programs that are fun, useful, small, and fast isn't magic that happens in an instant, nor is it a mysterious task that only years of training can perfect. Although simple programs can be created easily with moderate experience and skill, and the best programmers and engineers spend many years and thousands of hours practicing their art, good programs can be brought from idea to completion by following a short series of simple steps.

In this section, I'll describe these steps as they might apply to writing any program or game. Figure 5.1 shows how this discussion will proceed. First, I'll discuss the four
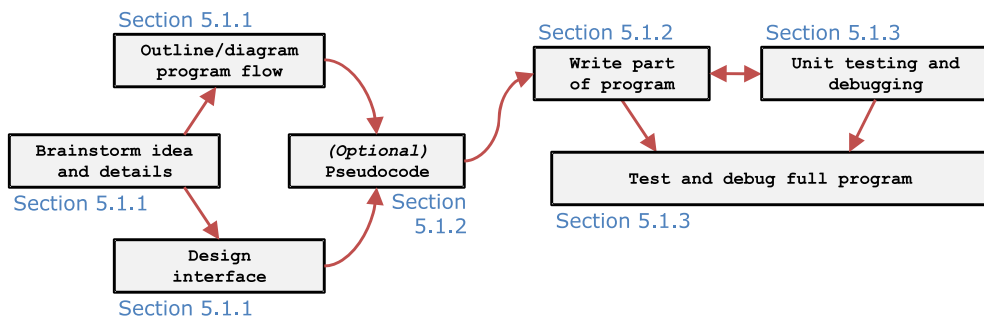
Figure 5.1   Taking a program from idea to completion, as covered in sections 5.1.1 through 5.1.3. After you brainstorm a good idea for a program or game and work out the details, you should diagram and outline as much as you need to have a firm idea of how the program will be constructed. If necessary, you can turn these plans into pseudocode before you write the actual program. You should alternate writing and testing rather than writing an entire large program in one go, so you can catch errors early. When you finish, thorough testing and debugging will ensure you've made a fast, reliable program.

possible pieces of the planning progress in section 5.1.1. Section 5.1.2 will discuss how to turn that plan into code, including choosing when to use loops like `For`, `While`, and `Repeat`, where `Lbl` and `Goto` might be appropriate, and how the different conditional constructs and input and output commands could be used to realize your plan. Section 5.1.3 will cover the important step of testing your program piece by piece as you write it and then as a complete whole once you have a completed preliminary program.

Before you begin writing any code, you should start with a clear plan of how your program will be structured, what features you want it to have, and how the user or player will interact with it.

5.1.1   *High-level design: features and interface*

The first stage in creating a great program is deciding on the features, flow, and interface for your program. Your planning process should follow this basic outline:

1 Decide what your program or game is or will do.
2 Design further details: decide the specific features set or game mechanics.
3 If applicable, sketch out the interface. If you're making a math program, for instance, decide how you'll present answers to the user.
4 Use diagrams and pseudocode to design the structure and flow of your program.

Once you've completed these steps, you can move on to turning your plans into code. In this section, I'll be discussing each of the four items in this outline for any general program you might wish to write, so that you'll know how to bring a program idea from conception through detailed diagrams and pseudocode.

At the beginning of your planning, the first thing you need to do is figure out what you want your program or game to do. It might sound obvious, but if you don't have a clear vision about what your program is going to offer to players or users, you'll find yourself floundering as you try to complete the rest of the planning and coding of your project. There's no hard-and-fast rule about how much detail you need to go into before you can start diagramming, designing, or coding; you should do as much planning as you feel you need to have a clear vision of how you'll proceed. What you want the program to be or do can be as easy as "a Pong game" or "a program to solve a system of equations." On the flip side, it could be as complex as figuring out what power-ups you want your Pong game to have and how the scoring would work, or designing exactly what sort of interface into which your users will be entering numbers for the equation solver. When in doubt, I recommend you err toward more detail, because you'll have left that much less to decide when you're in the middle of writing your code.

Between deciding details about your program idea and starting to design the flow of the program's code drawn as flowcharts or written as pseudocode, you may want to sketch out the interface for the program, how the screens where the user interacts with your program will look. This will become more important once you learn to draw complex interfaces, graphs, and games on the graphscreen, but it's still important

when you're dealing solely with the calculator's homescreen. With a basic idea of how your program will look to the person using it, you can move on to design the flow of your program's logic and code.

If you chose to draw diagrams, the diagrams should show enough detail about how your program works as a whole or how a specific piece works that you'll be able to use the diagram to help you create the program's code. For very simple programs, the diagram can be simpler, and as you become an experienced programmer, you may decide to do this stage in your head. No matter your skill level, if you have time to quickly sketch out the highest-level overview of your program in some sort of diagram such as figure 5.2 or figure 5.3, your programming will go much faster and involve less frustrating rewriting and reconfiguration of features.

If you compare figures 5.2 and 5.3, you'll notice that figure 5.2 describes all of the different pieces of the program, such as the help screen, the ending credits, and the in-game portion, but is vague on the actual gameplay. Figure 5.3, which refers to the system-of-equations solver, is much more specific but focuses entirely on the mechanics of solving the system of equations. This particular application uses a matrix-based method to solve a system of equations, putting the coefficients in a matrix and using the `rref` solution (you'll learn more about matrices in chapter 9). It's up to you to work out a good balance between the number and detail of your diagrams; you'll likely discover over time when you haven't planned enough and learn to fine-tune how much of your time is worth spending on planning.

Besides drawing interfaces and diagramming how the program will work, you may wish to also (or instead) write pseudocode, especially if you're not a visual person and diagrams like figures 5.2 and 5.3 confuse you rather than help you. As I've mentioned once or twice in previous chapters, pseudocode is a way of writing something that looks like code, describes code, but omits some of the annoying details of actual code. Pseudocode style varies widely from person to person, and there's no single right way
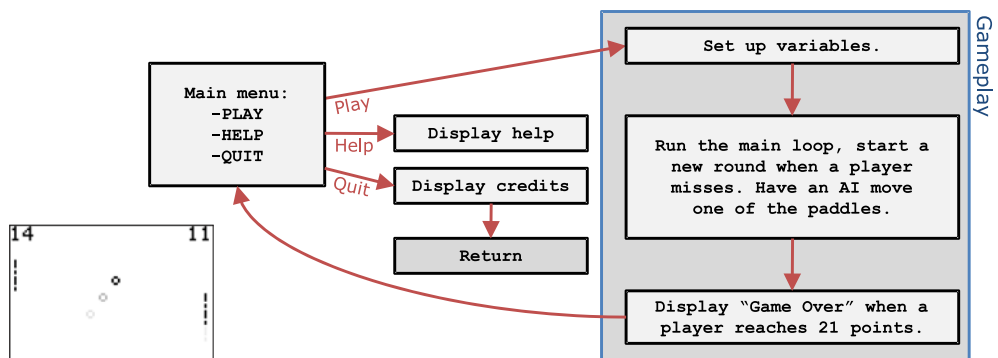
Figure 5.2   A high-level diagram of a theoretical Pong game, perhaps too high-level. In this particular diagram, you can see that the program will have a game section (reached from Play), a help section, and a section that displays credits and quits. If you're a beginner programmer, it would be good to also diagram more specifics of the boxes describing the gameplay, here surrounded by the large box at right.
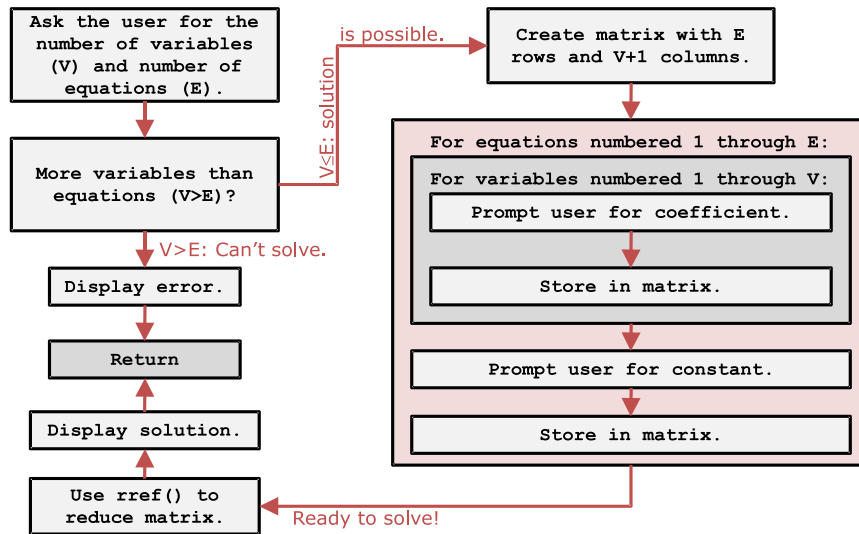
Figure 5.3   This diagram for a hypothetical system-of-equations solver describes how the program will get variables from the user, store coefficients in a matrix, and use the `rref` command to solve the system but omits information about things like menus, how the solution will be displayed, and whether the program can also solve other types of problems. Compare this to the amount of detail in figure 5.2.

to create pseudocode. I'll show you a style that resembles TI-BASIC. Consider this chunk of real TI-BASIC code:

```
:Input "NO. OF VALUES=",V
:0→T
:For(X,1,V
:Input "VALUE=",N
:N+T→T
:End
:Disp "TOTAL IS",T
```

You could express this in pseudocode as

```
:Ask user for number of values, store in V
:Ask the user for V values, and sum them in T
:Display the total
```

Your own pseudocode could more closely resemble TI-BASIC, look like another language if you know a language like C or Python or Java already, or be your own personal style. The most important attribute of the pseudocode you write and the diagrams and sketches that you make is that they make sense to you. They should give you a clear mental picture of how you'll have to proceed to turn your plan into code that will form a working program.

I'll now move on to taking your fleshed-out plan and turning it into a program or game, which although a hard task should be eased by the preplanning that you've done.

5.1.2 Structuring your code: diagrams to commands

There are no hard-and-fast rules to teach you how to properly take any planned idea and turn it into code that will work. I can't possibly list even a tiny fraction of the possible programs you might want to make or the possible pieces you might design to combine into a finished project. But I can give you some general guidelines for the commands you learned in chapters 2, 3, and 4 and how those might map to diagrams and pseudocode you've created. From your diagrams, you might work with one rectangle at a time being converted to code; with pseudocode, you'd likely translate line by line.

INPUT AND OUTPUT COMMANDS

Input and output commands, as I explained in chapter 2, are among the easiest to reason about:

- When you want to revert to a blank homescreen, you use `ClrHome`.
- If you need to display a single line of text and then pause, you could use `Pause "TEXT HERE."` If you need to display a lot of text, or text and numbers, and then pause, you can use `Disp` with several items separated by commas and then a lone `Pause` command.
- To display numbers or text when you don't care about placement, use `Disp`.
- If you want a neater display, you'll have to specify the position as well as the number or string to display as arguments to the `Output` command.
- To get a number or string from the user, `Prompt` is easy, but `Input` lets you specify the text displayed right before the program pauses to wait for the user's value.

In chapters 6, 7, and 8, you'll learn other input and output commands, dealing with directly reading keys and drawing and graphing on the graphscreen.

COMPARISONS AND CONDITIONALS

When your program needs to make a decision, it will probably need to use a comparison, as you learned in chapter 3. These comparisons can be used with `If`, `If/Then`, and `If/Then/Else` constructs or with `While` and `Repeat` loops.

- If you need a comparison that compares the numeric values of two variables, or a variable and a number, put one of the six inequality operators from the TEST menu between them.
- You can also use the equals sign (=) to check if two strings match.
- If your program specifies that all of several conditions must be true, you should join the different comparisons with the `and` Boolean logic operator.
- If only one of several conditions needs to be true, use `or`.
- If a condition needs to be false instead of true, wrap it in the `not` Boolean operator or reverse the inequality symbol (for example, turn > into ≤).
- If you have a complex combination of `and`, `or`, `not`, and `xor`, you'll probably need to use grouping parentheses.

Recall the lessons about the three different types of `If` constructs from chapter 3:

- If only one extra line of code needs to be executed when a condition is true, pair that line with a lone `If` statement.
- If several lines need to be run only when a condition is true, add a `Then` and `End`.
- The cleanest way to run one chunk of code when the condition is true and another when it is false is to use `If/Then/Else/End`.

FLOW CONTROL: LOOPS, JUMPS, SUBPROGRAMS, AND MENUS

Whenever your program needs to jump to one of several areas, needs to repeat a chunk of code over and over, has to reuse the same piece of code in several places, or needs to present the user with choices, you'll need the flow-control commands from chapter 4.

- If you need a menu, the easiest solution is the `Menu` command paired with several `Lbl` commands. You could also make your own menu with `Disp/Output/Input` if the `Menu` command isn't flexible enough for your purposes.
- If your plan specifies that you'll need to jump from one place to one of several places in your program, use several `If/Goto` statements with corresponding `Lbl`s.
- If you need to go from any of several places in the program to one place, then use `Goto`s all pointing to the same `Lbl`.
- Any jump in your program can be formed with a `Lbl` and a `Goto`, but be sure to avoid memory leaks by not putting a `Goto` inside loops, `If/Then`, or `If/Then/Else` statements.
- If you need to run one section of code repeatedly, you should use a loop. If you can count the number of times the loop will run, or you want a variable to take each of several predictable, evenly spaced values, use a `For` loop.
- If the loop should run until a certain condition becomes true, or if you want a loop that always runs at least once, use a `Repeat` loop.
- If the loop should run only while a condition stays true, or you want a loop that can run zero times, use a `While` loop.

Terminating your program and properly using subprograms follow more cut-and-dried guidelines than many of the other commands just mentioned:

- If you want a program that could be called from another program to immediately make every program quit, including itself and its caller(s), use the `Stop` command.
- In any other circumstances, use `Return`. If you want a subprogram to stop and return to its caller, use `Return`. If you want a standalone program to stop, use `Return`.
- A subprogram should be used when you would instead be rewriting the same section of code in several places in your program. It should be code that stands on its own and doesn't need to jump anywhere else in the program. If it's the

same code in several places, use a subprogram, but if it's one piece of code in one place to be run several times, use a loop.

FINAL THOUGHTS

As with most of the material in this chapter, these are general rules for the proper use of the programming commands and techniques you've learned up to this chapter. They aren't unbreakable rules, and you'll hopefully discover many other clever things you can do with the various commands as you continue to program. But they'll be a good guide for you as you dive ever deeper into programming, with calculators or with any language that uses similar commands and constructs such as C, C++, Java, Python, and PHP.

As you work through converting your ideas into code, you should always test your program piece by piece, and when you have a complete program, test again. I'll present this vital step to you as the third piece of developing a polished program.

5.1.3   Testing and debugging

A good coder will thoroughly test a program when it's finished to make sure it will work well for users. A great coder will write the program a piece at a time, testing each piece as it's added in case it contains errors. This latter technique is part of an approach called unit testing, and it helps you avoid writing a lot of code, finding that it doesn't work, and being unable to find the problem. When you build a complex program, it's likely you'll make a mistake, so in some cases you create other programs that will individually test the pieces of the first program. This isn't generally feasible for your TI-BASIC programs, but you can get the same effect by alternating coding and testing. When you test as you create, you catch errors close to where you make them, and you can usually solve them easily. If the error isn't easily solved, and you need to rearrange your original program plan, you at least have the opportunity to do so before you exert yourself completing the rest of the code. If you wait until you finish, you risk pain and suffering as you try to narrow down a pesky bug.

How you decide to test your program also depends a lot on how you write your program. A common approach is to write the overall structure first, such as the main menu and labels for each section. You can then work from the easiest bits, like the high-score display, the help section, the credits, and the settings, to the harder bits, usually the core gameplay or features of the particular program. This way, you can leave a dummy `Lbl` for the hardest portion that `Goto`s right back to the main menu but test out the menu and other areas before diving into writing the hardest part.

Alternatively, if writing pieces of the program out of order is likely to cause you to make errors, lose your place, or accidentally omit bits of the program, you can write your code from top to bottom. You can and should still test your program as you go, so that you don't reach the end and discover that somewhere deep within your code you made an error. But if you try writing your programs feature by feature rather than top to bottom, you'll find that it's easier and makes more sense, because you'll

be completing conceptual chunks one by one rather than working top to bottom, which generally only makes sense if the program doesn't loop or jump.

If as you're unit testing you do find problems, or if you find errors when you've completed the program, refer to sections 5.4 and 5.5 of this chapter. You'll find tips on tracking down the simple mistakes that throw TI-OS errors and the subtle blunders that break your program in harder-to-trace ways. Instead of trying to describe the decision-making process behind deciding the order in which to write your program's components, I'll dive right into this chapter's big running example, and you'll see how it works.

I'll begin with planning the structure of the Pythagorean Triplet solver using flow diagrams, interface sketches, and pseudocode.

5.2 Planning a program's structure

The previous section described the general steps to take to build a program easily and without getting lost in throwing away and rewriting wrong and confusing code. This section will follow the same outline and progression but will follow a specific running math example. I'll show you the planning process for a real program, a tool to search for certain sets of numbers called Pythagorean Triplets. Figure 5.4 shows how this discussion will proceed; notice that this is the same diagram as figure 5.1, with the section numbers modified to help you navigate through the material.

I'll first discuss the problem we'll be solving; explaining to yourself what you want your program to do or be is in some ways just as important as typing out the program's commands. I'll then show you two different diagrams of how the program will work: one a general overview and the other a specific look at how the program will be looking for acceptable triplets. I'll conclude with pseudocode and an interface sketch.

Let's begin with the process of coming up with a concrete idea and basic plan for the Pythagorean Triplet solver.
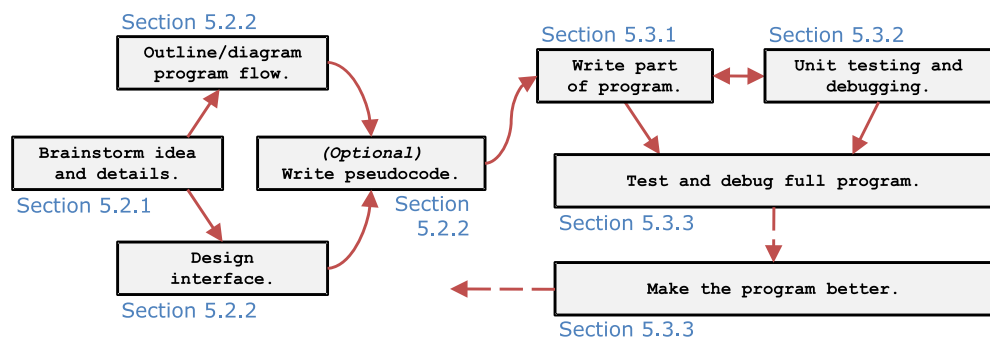

Figure 5.4 Planning, creating, and testing the Pythagorean Triplet solver, which will be known as PYTHTRIP. The subsections of section 5.2 walk you through the planning process for this program, whereas section 5.3 works through writing and testing the program piece by piece and then optimizing it.

5.2.1   *Idea and details: first steps*

For the project I'll carry through this chapter, we'll be solving a subset of the problem known as Pythagorean triples. As you may know, the Pythagorean Theorem describes the relationship between the lengths of the legs of a right triangle, as shown in figure 5.5. A triangle is called a right triangle when it has a 90-degree right angle between two of the edges, here, between A and B.
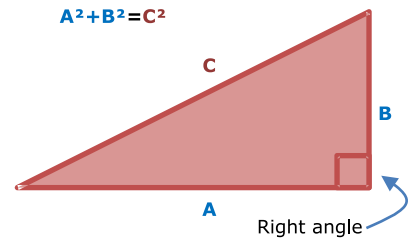
$A^2+B^2=C^2$

Figure 5.5   A right triangle with legs A and B and hypotenuse C, where the lengths of the legs and hypotenuse fulfill the Pythagorean Theorem, $A^2 + B^2 = C^2$

The two legs of the right triangle with lengths A and B are related to the length of the hypotenuse C by the Pythagorean Theorem:

$$A^2 + B^2 = C^2$$

For the program I'll show you how to write in this and the next sections, we want to find Pythagorean Triplets, sets of (A, B, C) values that fulfill the Pythagorean Theorem. We want this program to find triplets only where A + B + C = N, for any N that the user enters. Two valid values of N that we might try are N = 12, which works for the triplet (3, 4, 5), and N = 198, which can be formed from the triplet (36, 77, 85). If the program finds values for A, B, and C that sum to N, it will show the user the first solution it finds and quit. If it instead finds no such values, it will alert the user that it was unsuccessful before exiting. The values A, B, and C will be subjected to the additional constraint that 0 < A < B < C. That is, all of the values are positive, all are distinct, and all are integers.

My design therefore will need to have the following pieces:

- My program will need to get a value for N from the user.
- The program can then look for values of A and B that create a Pythagorean triple and check each such set of numbers to see if they add up to N.
- It should tell the user if it finds one. It should also detect if no such numbers were found and tell the user when that happens.

I'll use the homescreen for input and output. For this program, I'm choosing to save the details of how exactly to solve for A, B, and C for the next planning stage, where I'll create diagrams and pseudocode to plan the program. Let's therefore move on to discuss creating the appropriate diagrams, pseudocode, and sketches to help me write this program to be both correct and well written.

5.2.2   *Diagrams and pseudocode*

Diagrams and pseudocode to help you plan your program can take many forms, as I introduced in section 5.1.1. I'll show you several ways I could choose to plan the Pythagorean Triplet solver. If this was your program, you could choose to use all of

these to plan the program, use none of them, or pick which planning methods are most helpful to you. Remember that the goal is to help you write a good program as quickly as possible but not to weigh you down with so much preplanning that it takes you hours to get to writing the program. The amount of planning you do should generally be proportional to the complexity of the program you're writing. In addition, as you become a more experienced programmer, you'll discover that it takes less and less planning to form a clear mental picture of what your completed program will contain.

I'll show you a diagram that provides a general overview of the entire Pythagorean Triplet solver, which I'll call PYTHTRIP to fit into the 8-character limit on TI-83+/84+ program names. I'll show you the pseudocode that you could write to match that diagram and then a figure and pseudocode that provide a more detailed look at the core math functionality of the solver. Finally, I'll show you how you could sketch out the interface for this program.

A FULL-PROGRAM STRUCTURE DIAGRAM AND PSEUDOCODE

First, you could decide that you want to get a general idea of how the program as a whole will be structured and that you don't need to worry about the specifics until you start writing the program. For this particular program, you know that you need to get a value for N from the user, solve for A, B, and C, and report the results (or failure) of the program to the user. If you tried to draw a diagram for this in the general flowchart form that I've been showing you throughout this book, you might end up with something like figure 5.6.

Alternatively, you may decide to write the structure for this program in pseudocode. Pseudocode isn't a real programming language; instead, it's a way of representing program flow in something that looks like but isn't code. Everyone's personal pseudocode style differs, and there's no right or wrong way to write pseudocode. For the sake of this book, I'll write pseudocode that resembles TI-BASIC, and I recommend you do the same. Remember, I'm not trying to write actual TI-BASIC here but to sketch out what a simplified version of the program might look like, omitting details like how to solve for A, B, and C. Try to compare this to figure 5.6 and see how it almost directly corresponds.
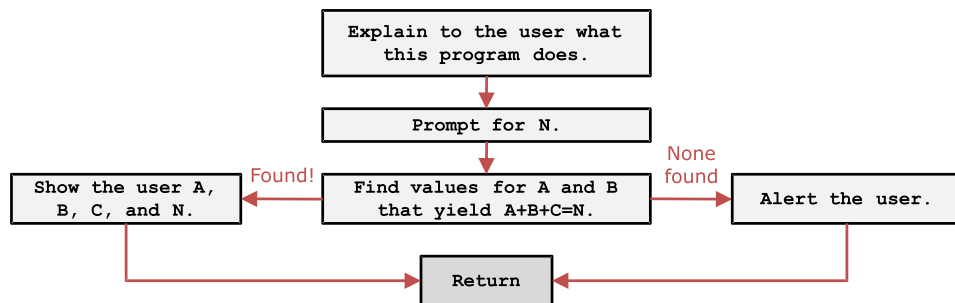

Figure 5.6 A simple block overview of the Pythagorean Triplet program. It explains the program to the user, prompts for a value for N, and solves for A, B, and C. It tells the user the values if it finds them or warns them it failed and then quits.

```
:Disp instructions
:Prompt N
:Search for A,B,C where A+B+C=N
:If found
:Then
:Disp A,B,C,N
:Else
:Disp "NO RESULTS"
:End
```

As you can see, this pseudocode resembles TI-BASIC but obviously would never run. It expresses the flow of the program, in this case first showing the user information about how the program works and prompting for a value for N, then searching for a valid triplet, and finally informing the user of the program's results.

A DETAILED DIAGRAM AND PSEUDOCODE

After you write your highest-level plan for your program, which makes lots of generalizations, you might decide to add more specificity and detail. For this program, the vagueness comes from the `Search for A, B, C` line, which compresses the entire mathematical process of searching for Pythagorean Triplets into one line without describing how it's done. If as a beginner programmer you tried to write the program from the diagram in figure 5.6 or the pseudocode just shown, you might get lost and frustrated. Indeed, I'd argue that the details of how exactly the program will search for value for A and B (and therefore C) are more important than writing out what this particular program will `Input` and `Disp`, because the math to search for Pythagorean Triplets is the core of the program's useful functionality. For each new program you write, you should balance how much (and which parts) you want to decide beforehand and what you want to leave to figure out while you write the program itself.

The more detailed examination of the `Look for values` box in figure 5.6 expands to the contents of figure 5.7, which also includes the two ending display boxes and the `Return` box. To understand the contents of the large dark box labeled `Look for values` in this diagram, you must consider what we want this program to do. It should find, if they exist, values for A, B, and C that make the equation $A + B + C = N$ true and must also be a Pythagorean triple. If you pick values for A and B, then you already have C as well, because A, B, and C follow the equation $A^2 + B^2 = C^2$; in other words, if you pick A and B, then C must be $\sqrt{(A^2 + B^2)}$. Therefore, we now know one fact: we must pick A and B and use these to calculate C.

With our valid Pythagorean triple, we'll have to check if $A + B + C = N$, and if not, keep trying more values for A and B. The logical next question then is which values of A and B to test. I specified the constraint $0 < A < B < C$. This means A, B, and C are all positive, and none is equal to any of the others. We could make A vary between 1 (the smallest possible value larger than zero) and N. Why stop at N? Because if B and C are positive, then A can't possibly be larger than N to make $A + B + C = N$ be true. We can stop at N – 2, because neither B nor C can be smaller than 1.

We can follow a similar process to pick the bounds on B. We'll say that it can go as high as N – 2 as with A, but for the lower bound, I'll pick A + 1, because we know that
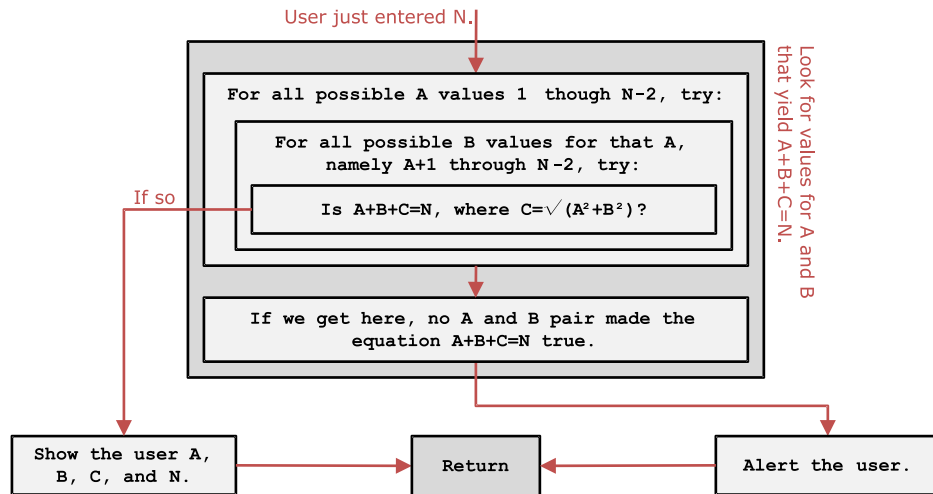
Figure 5.7   Expanding the outline of the core math of the PYTHTRIP program. Instead of simply a "black box" that finds if there's a solution triplet (A,B,C) to A + B + C = N for a given N, we now expand the program plan to think about how to test each possible A and B value to see if they form a Pythagorean Triplet that satisfies the given desired target N value.

A < B and that A and B must both be integers. Therefore, A can be 1 through N – 2, and B can be A + 1 through N – 2. This leaves us with this search process:

- Try varying A from 1 to N – 2.
- For each A, try a range of B values, from A + 1 through N – 2.
- With each pair of A and B values, calculate C.
- If A + B + C = N, display and stop; otherwise, try another pair of values.

Figure 5.7 shows this written out as two nested `For` loops. The outer `For` loop will run its body for each possible A value. In the body, an inner `For` loop will try a range of B values for each A. In the inner loop's body, the program calculates C and tests if A + B + C = N. You could also write this out in pseudocode:

```
:For every A in the range 1 to N-2:
:Vary B for that A between A+1 and N-2:
:Now for A and B, set √(A²+B²)→C
:Display results and quit if A+B+C=N
:End B loop
:End A loop
```

This pseudocode in particular will be helpful when we're ready to turn the plan into actual TI-BASIC code.

As a final short lesson, I'll review the task of sketching out program interfaces.

DESIGNING AN INTERFACE

If your program will be simple, you might not need to think about the interface. If it has several different features, is a game, or is designed to look especially professional

Figure 5.8   Designing the UI. The starting screen is at left; depending whether or not the program finds a solution, one of the screens at the right is shown.
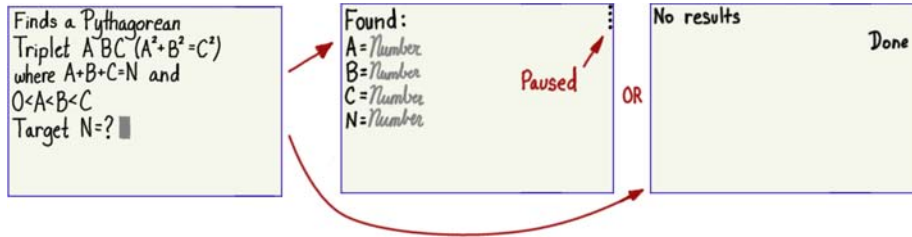
and spiffy, designing the interface is a helpful part of the planning process. You'll have a chance to see how your program will look before you write it, and you'll be able to see if the way your program presents itself to and interacts with the user might be confusing or frustrating. The Pythagorean Triplet solver is a relatively simple program and so the interface is fairly straightforward: some input commands and some output commands. But to give you an idea of what parts you might want to sketch for more complex programs and games, take a look at figure 5.8.

With a solid plan for how this program will look and be structured written down, we can move on to turn the plan into code.

5.3   *Headache-free coding and testing*

After considering different ways to formulate a program and have a complete design drawn out in diagrams and/or pseudocode, you're ready to begin to write real code. Ideally, this should be one of the easiest phases. If you simply dove into writing code, you'd likely have to deal with rethinking pieces as you went, redesigning the flow of the program, deleting and re-creating pieces, and spending time frustrated. With a well-thought-out plan, all you need to be able to do is take your diagrams and/or pseudocode and turn it into actual code. You'll still need to test and debug your program as you go, but the process will have been made vastly easier by your initial planning and brainstorming.

In this section, I'll first teach you how to turn the flowcharts, interface sketches, and pseudocode for PYTHTRIP into actual code. I'll show how this program can be unit tested and how final testing will be performed. Finally, I'll show the finished program and take it one step further. But first, although you may have already gained some insight into the process from previous chapters' examples, I'll show you how to turn this program's plan into code.

5.3.1   *Flowchart to code chunks*

This section will draw from figures 5.6, 5.7, and 5.8 as well as the list of concept-to-command mappings in section 5.1.2 to turn the PYTHTRIP plan into code. As mentioned in this chapter, it's generally preferable to work in chunks, coding one feature before adding more features around it. For this program, the core is calculating an A,

B, and C for a given N. Therefore, we'll start there and then add extra code around that core to create the program's UI.

If you review section 5.2, you'll see that figure 5.7 details the math necessary to calculate an A, B, and C that sum to a given N and also form a valid Pythagorean Triplet. That figure was used to create a fragment of pseudocode that describes the same concept, reproduced here:

```
:For every A in the range 1 to N-2:
:Vary B for that A between A+1 and N-2:
:Now for A and B, set √(A²+B²)→C
:Display results and quit if A+B+C=N
:End B loop
:End A loop
```

We can turn this pseudocode into true TI-BASIC code. First, let's construct the loops that will vary A and B. In this case, we want to use every integer between 1 and N – 2 for A and every integer between A + 1 and N – 2 for B. The logical choice here is a pair of `For` loops: `For` loops are good when you need to count or examine each element in a regularly spaced series. The two `For` loops can be written as follows:

```
:For(A,1,N-2
:For(B,A+1,N-2
:End
:End
```

Vary A between 1 and N – 2, with an implied (omitted) increment of 1

For each A, try B values between A + 1 and N – 2, with increment 1. The ending parentheses are omitted to save space.

End the For loop for B

End the For loop for A

Inside the inner `For` loop, A and B will take a full range of possible values that could form acceptable triplets. To complete the triplet, we need the third number, C. We know from the pseudocode that we could write this as $\sqrt{(A^2 + B^2)} \to C$ and then check if A + B + C = N. But we can save ourselves a variable by creating a conditional that checks instead if $A + B + \sqrt{(A^2 + B^2)} = N$ and if so displays A and B and stops:

```
:If A+B+√(A²+B²)=N
:Then
:Disp A,B
:Return
:End
```

Equivalent to If A + B + C = N

Don't check any more A, B pairs because a solution has been found

This `If/Then` conditional executes its code block only if A, B, and C sum to N, which means this is a valid triplet that correctly completes A + B + C = N. For now, we make the program display A and B and immediately return if a solution is found.

At this point, we have all of the code necessary to solve the problem specified for this program. We'll do some unit testing to make sure it works right before adding UI components to it and testing the full PYTHTRIG program.

5.3.2 *Performing unit and full testing*

The two chunks of code just presented form a full (if rudimentary) Pythagorean Triplet solver. We can combine them into a preliminary PYTHTRIP program, as shown in the following listing. The program is shown as PYTHPREL, but if you're following

along on your calculator, name it PYTHTRIP so you'll be able to add the final components to it to complete the program.

```
PROGRAM:PYTHPREL
:For(A,1,N-2              Vary A and B over all
:For(B,A+1,N-2            possible valid values
:If A+B+√(A²+B²)=N        ←——————        Check if they create a Pythagorean
:Then                                    Triplet summing to N
:Disp A,B
:Return
:End                     End the conditional block
:End                     and the two For loops
:End
```

To test this program on your calculator, you'll have to do something I haven't previously shown you: manually set up a variable. Figure 5.9 shows how to set up variable N for this program and then execute the simple program (note that it's PYTHPREL here but can be PYTHTRIP on your calculator). I already said that (3, 4, 5) is a valid triplet; because $3^2 + 4^2 = 5^2$ and because $3 + 4 + 5 = 12$, this program should display 3 and 4 for N = 12. Lo and behold, that's precisely what it does.

　　If you try N = 198, another sum that can be formed with a triplet, the program should work for about two minutes before spitting out A = 36 and B = 77.

　　Because this core code works, we can proceed to wrap it in an interface. The interface diagram in figure 5.9 indicates that the program should display a short explanation before prompting for N. We can clear the screen with `ClrHome` and write out the explanation and prompt with a multipart `Disp` command and an `Input` command. Figure 5.10 shows how this will look.

　　Notice that I've cut up the explanation into pieces based on the fact that the homescreen is 16 characters wide:

```
:ClrHome
:Disp "FINDS A PYTHAGOR","EAN TRIPLET","A B C (A²+B²=C²)",
➡ "WHERE A+B+C=N","AND 0<A<B<C
:Input "TARGET N=",N
```


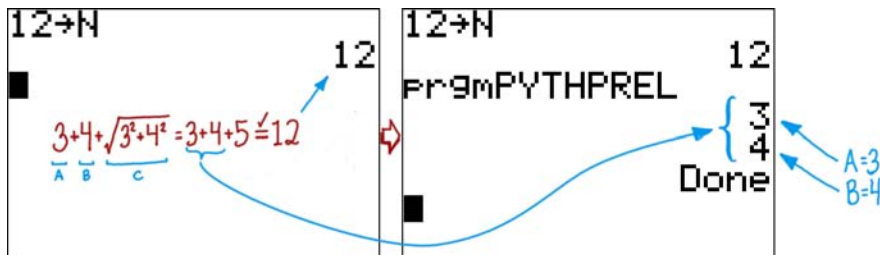**Figure 5.9**　**Unit testing the core of the Pythagorean Triplet solver. At left, setting up N = 12; then at right, `prgmPYTHPREL` correctly finds that A = 3 and B = 4 (and C = 5) form a valid triplet that sums to 12.**
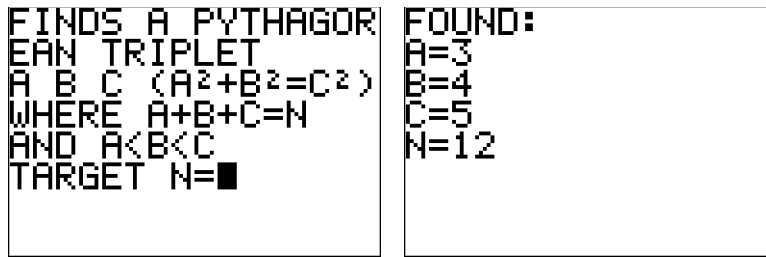
Figure 5.10   The completed interface for displaying instructions (left) and results (right) for PYTHTRIP

Next, we need to display "NO RESULTS" at the end if we didn't already `Return` after showing a valid solution. That's a simple `Disp` command with a `ClrHome`:

```
:ClrHome
:Disp "NO RESULTS"
```

Finally, the slightly fancy display of A, B, C, and N shown in figure 5.10 can be made with a `Disp` command and four output commands:

```
:Disp "FOUND:","A=","B=","C=","N=
:Output(2,3,A
:Output(3,3,B
:Output(4,3,√(A²+B²
:Output(5,3,N
```

Notice that all four output commands start on the third column of their respective lines, making the values appear neatly after the four equals signs as in figure 5.10.

Now that we've created all the pieces we need for the final program, we can glue them all together into `prgmPYTHTRIP`.

5.3.3   *The final Pythagorean Triplet solver*

Now that we've designed the different pieces of our PYTHTRIP program, we can finally put it all together, as shown in listing 5.2 and demonstrated in figure 5.10. This program will ask the user for a value for the sum of A + B + C, called N, and use the two nested `For` loops we designed to loop for Pythagorean Triplets. If it finds a solution, it informs the user and quits; if it doesn't find any solution, it also tells the user about that before quitting.

Listing 5.2   Final PYTHTRIP program

```
PROGRAM:PYTHTRIP
:ClrHome
:Disp "FINDS A PYTHAGOR","EAN TRIPLET","A B C (A²+B²=C²)",
      "WHERE A+B+C=N","AND 0<A<B<C
:Input "TARGET N=",N
:For(A,1,N-2
:For(B,A+1,N-2
```

> Variable A can be at least 1 and at most N (or more like N – 2)

> A < B, so don't bother testing any B that is less than or equal to A

```
:If A+B+√(A²+B²)=N
:Then
:ClrHome
:Disp "FOUND:","A=","B=","C=","N=
:Output(2,3,A
:Output(3,3,B
:Output(4,3,√(A²+B²
:Output(5,3,N
:Pause
:Return
:End
:End
:End
:ClrHome
:Disp "NO RESULTS
```

The Pythagorean formula says $A^2 + B^2 = C^2$, so $C = \sqrt{(A^2 + B^2)}$

Quit without running the rest of the loop; we found the solution

Close the two For loops

End the If/Then statement

MAKING YOUR PROGRAMS BETTER

With rare exceptions, you'll find that no program is ever truly finished. You or your users will find new bugs to fix, users and players will suggest features to add, or you'll think of new ways to optimize or improve the program. Just as it's best to consider a few ways you could construct a program before you begin to write code, you should avoid convincing yourself that any program is in the best form it could ever take once you finish putting it together. The program PYTHTRIP in listing 5.2 is a short, correct, and optimized implementation of the original specification to find a Pythagorean Triplet (A,B,C) such that A + B + C = N. But you should always, even for this program, try to think how you could make it faster, smaller, more user friendly, and more fun. Chapter 10 will take you through all kinds of optimization tips and tricks, but let's quickly look at a handful of them now.

Here, you might start at improving the speed of the program. Though optimized, it still takes four times as long to run if you double N, because the two For loops now cover twice as many values for A and B, respectively. Ideally, you'd be able to come up with a way to reduce this: what if there was a way you could use a single For loop to solve the problem? That might at first seem impossible: you have two independent variables, A and B, one dependent variable, C, and one known quantity, N. It would seem that you would need to try all possible values for A and B. But if you think carefully, you can express A in terms of B or B in terms of A. When you pick values for A, for example, then the only unknown left in the equation A + B + C = A + B + $\sqrt{(A^2 + B^2)}$ = N is B, and hence there's only a single value of B that completes the equation. This means that you'd only have to test a single B for each A value, reducing the two For loops to a single For loop!

The following algebra is a little convoluted if you're not used to this sort of manipulation, but I've annotated it to help clarify some of the more complicated steps. Remember, the goal is to get either A or B all by itself.

$A + B + C = N$
$A + B + \sqrt{(A^2 + B^2)} = N$
$\sqrt{(A^2 + B^2)} = N - A - B$

Replace C with $\sqrt{(A^2 + B^2)}$, because $A^2 + B^2 = C^2$

$$A^2 + B^2 = (N - A - B)^2$$

Square both sides

$$A^2 + B^2 = N^2 - NA - NB - NA + A^2 + AB - NB + AB + B^2$$

Expand the right side of the equation

$$A^2 - A^2 + B^2 - B^2 = N^2 - 2NA - 2NB + 2AB$$

Move and combine terms

$$0 = N^2 - 2NA - 2NB + 2AB$$

$$2NB - 2AB = N^2 - 2NA$$

$$2B\,(N - A) = N^2 - 2NA$$

Factor out 2B from the left side

$$2B\,(N - A)/[2(N - A)] = (N^2 - 2NA)/[2(N - A)]$$

$$B = (N^2 - 2NA)/[2(N - A)]$$

Divide both sides by 2(N + A) to get B by itself

What's with all the algebra?

Sometimes rethinking your program involves rethinking the structure; other times it's rearranging the math. Here, we're realizing that each value of A has only one possible B value that makes A + B + C = N for the Pythagorean Triplet (A,B,C), not a range of possible values between A + 1 and N as we assume in `prgmPYTHTRIP`. By rearranging the equation that relates A, B, and N, we're able to solve for the B that matches each A in a single step. We can then see if A and B form a valid Pythagorean Triplet (A,B,C).

With this final equation, we have an expression for B in terms of A and N. Now for each A value between 1 and N, there's only a single corresponding B value that could possibly make the equation A + B + C = N correct.

So why aren't there N different solutions? The majority of the values for B that the equation will generate won't be integers; they'll instead have decimal parts. The problem statement requires that A, B, C, and N are all integers, so we can discard all possible values for B that aren't integers. Using a form you've seen once or twice before, the following comparison is true only when B is an integer:

```
int(B)=B
```

Now that we can extract a value for B from each possible A value without running through every possible B, we can rewrite PYTHTRIP with a single `For` loop, as in the following listing.

Listing 5.3   A faster PYTHTRIP program, called PYTHFAST

```
PROGRAM:PYTHFAST
:ClrHome
:Disp "FINDS A PYTHAGOR","EAN TRIPLET","A B C (A²+B²=C²)",
➡  "WHERE A+B+C=N","AND 0<A<B<C
:Input "TARGET N=",N
:For(A,1,N-2
:(N²-2NA)/(2(N-A))→B
:If int(B)=B and A<B
:Then
:ClrHome
:Disp "FOUND:","A=","B=","C=","N=
:Output(2,3,A
```

For each A, only a single B value satisfies B = $(N^2 - 2NA)/(2(N + A))$

Calculate and store that B value

Check if B is an integer and 0 < A < B. It's unnecessary to check B < C because if either A > 0 or B > 1 (both necessary for 0 < A < B), then $A^2 + B^2 > B$.

```
:Output(3,3,B
:Output(4,3,√(A²+B²
:Output(5,3,N
:Pause
:Return
:End
:End
:ClrHome
:Disp "NO RESULTS
```

As before, quit early if a solution is found

Why is this better? The fewer loops, or the shorter the loops, the faster the program will run. If you're clever, you can even reduce the number of A values that PYTHFAST tries, because there are fewer possible values for A that make A + B + C = N and 0 < A < B < C than A = [1,N − 2]. As I've mentioned several times, the best code is both fast and short. Therefore, let's check that this new version is either faster or shorter, or ideally both!

One approximate way to measure the length of programs is the number of lines of code. PYTHFAST is 19 lines, and PYTHTRIP is 20 lines. If you look in your calculator's Memory ⇒ Management ⇒ Programs menu ([2nd][+][2][7]), you'll see that as in figure 5.11, PYTHFAST is only 5 bytes larger than PYTHTRIP, a good tradeoff if, as we expect, PYTHFAST is much faster. Table 5.1 illustrates the extreme speed advantages of the optimized PYTHFAST program.

Table 5.1   Timing the PYTHTRIP and PYTHFAST programs for some target N values

Target N = A + B + C value	Time for PYTHTRIP	Time for PYTHFAST
N = 12 (A = 3, B = 4, C = 5)	0.9 seconds	0.4 seconds
N = 198 (A = 36, B = 77, C = 85)	113.1 seconds	1.1 seconds
N = 1000 (A = 200, B = 375, C = 425)	5728 seconds	5.1 seconds

For N = 12, the simplest Pythagorean Triplet where 0 < A < B < C and A, B, and C are integers, the PYTHFAST version runs only twice as fast as the slower PYTHTRIP version. For N = 198, the faster version goes about one hundred times faster than the original implementation. For N = 1000, the PYTHFAST program beats out PYTHTRIP by more than a thousand times, completing in 5.1 seconds instead of over an hour and a half.

As you can see, there can be huge rewards from constantly considering how a good program might be made better, even if the first, second, or tenth version of the program
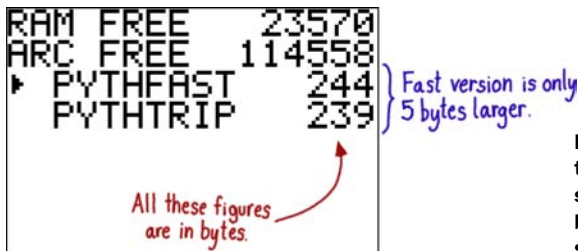

Figure 5.11   The Memory menu shows that the faster version of the Pythagorean Triplet solver programs is only 5 bytes larger (244 bytes as opposed to 239 bytes) than the slower PYTHTRIP program.

is good or even great. You can almost always find ways to make your math and science programs run faster or show the user more information, your utilities to have more features while taking less space, or your games to be faster and more fun. Here, we tried to see where the program might be wasting time, specifically trying to reduce the number of B values we try for each A. We looked carefully at the equation and saw that there's only one B value possible for each A value; we used a simple check can determine if that B is an integer and thus forms a Pythagorean Triplet. We could have derived an equation to get A for a given B instead; either way, we'd have gotten the impressive speedup in table 5.1. Reconsidering the mathematical or structural design of a program even when it's complete is key to creating fast, small, reliable programs.

Things don't always work out quite this well, and more often than not you'll find that when you run your programs, even if you've carefully planned and unit tested, things don't work right. There are two major types of errors your programs can have: errors that produce an error message and those that are more subtle, making your program work differently than you intended without revealing what exactly has gone wrong. I'll first show you TI-OS BASIC errors, introduce the most important types, and explain general approaches to resolving such errors.

5.4   *Understanding TI-BASIC errors*

If every program worked correctly, programming would be a wonderfully stress-free job and hobby, though there would be little challenge. For better or worse, errors are inevitable with any programming language; as you become more experienced, you'll encounter fewer errors. The easiest errors to track down are those that come with a clear warning sign to tell you what went wrong. TI-BASIC programs can throw many such errors.

When you do something that your calculator can tell is obviously wrong, such as putting too many closing parentheses, the wrong number of arguments to a command, or trying to add 3 + "HELLO", your calculator will notify you with errors such as those listed in table 5.2.

Table 5.2   The TI-OS error codes you're most likely to see when writing TI-BASIC programs. The table shows each TI-OS error, the equivalent Doors CS code, and an explanation of each.

TI-OS Error Code	Doors CS Error	Meaning
ERR: OVERFLOW	Error 507: OVR	Attempted to store a number larger than ±10^{99} in a variable.
ERR: DIVBY0	Error 508: DV0	A math expression divided by zero.
ERR: DOMAIN	Error 510: DOM	Either you tried to use a math function undefined for that value (such as log of a nonpositive number) or you specified an offscreen coordinate for `Output`, `Text`, etc.
ERR: INCREMENT	Error 511: INC	The increment for a `For` loop or a `seq` command was zero.
ERR: BREAK	Error 512: BRK	The user pressed [ON] to stop a TI-BASIC program.

Table 5.2   The TI-OS error codes you're most likely to see when writing TI-BASIC programs. The table shows each TI-OS error, the equivalent Doors CS code, and an explanation of each. *(continued)*

TI-OS Error Code	Doors CS Error	Meaning
ERR: NONREAL	Error 513: NRL	A calculation was made in Real mode but returned a complex number as an answer.
ERR: SYNTAX	Error 514: SYN	Generic typo in a program, such as missing quotes, colons, parentheses, etc. Also can occur when using the wrong variable type in a function (like a `Str` in a `For` loop).
ERR: DATATYPE	Error 515: TYP	A literal number, string, matrix, picture, list, etc. is used where another data type must be used instead.
ERR: ARGUMENT	Error 516: ARG	Most likely, you specified the wrong number of arguments to a function.
ERR: MISMATCH	Error 517: MSM	You tried to perform math on two lists or two matrices that have incompatible dimensions.
ERR: INVALID DIM	Error 518: DIM	You tried to get or set an element outside the size of a matrix or list or tried to create a list or matrix of an invalid size.
ERR: UNDEFINED	Error 519: UND	You referenced a numeric, list, string, picture, matrix, etc. variable that hasn't been given a value or doesn't exist.
ERR: MEMORY	Error 520: MEM	Your calculator is out of memory, either because of lots of programs/files/data or because of a memory leak (see the "Lbl, Goto, and memory leaks" sidebar in chapter 4).
ERR: ILLEGAL NEST	Error 522: ILN	You used a `seq` inside a `seq` command, a `fnInt` inside a `fnInt` command, or an `expr` inside an `expr` command.
ERR: LABEL	Error 526: LBL	A `Goto` command tried to jump, but the target `Lbl` it specifies doesn't exist in this program.

When you receive one of these errors, as shown in figure 5.12, you can choose to immediately quit or to go to where in the source the error occurred. This latter option lets you see exactly where you made your mistake and attempt to correct it. Most errors are fairly self-explanatory if you compare your code with the errors in table 5.2.
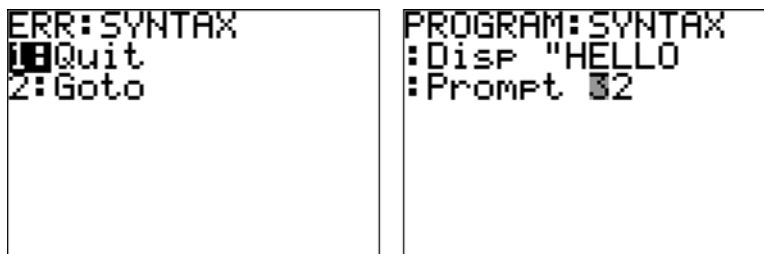
Figure 5.12   A SYNTAX error. If you choose 2: Goto in the error message at left, the calculator shows you the source code of your program with the cursor placed over the error; here you need the name of a variable instead of a number after `Prompt`.

The least-clear errors are those that stem from the values of variables, rather than directly from your code. If you read a DIVBY0 error, the TI-OS will take you to a line with division; you could quit to the homescreen, type in the name of the variable in the denominator of the division, and hit [ENTER] to see that it's zero. For a MEMORY error, you could check if the Memory menu ([2nd][+][2]) shows you to be low on RAM, and if not, check if you've created a memory leak. On the other hand, errors like ERR: LABEL are easy to solve, because choosing `Goto` at the error message will take you directly to the offending `Goto` or `Menu` command, from which you'll know the name of the missing `Lbl`.

Unfortunately, not every mistake you make will produce an error message; the majority of your debugging time will be spent tracing code that doesn't do what it's meant to do but doesn't produce any obvious error messages.

5.5   *Tracing malfunctioning code*

When you don't see any error messages, but your program is still failing to display the correct output, do math properly, or run your game as you intended, you must fall back on a utility belt full of debugging skills. Don't despair! Practically no programming error is insurmountable. You'll develop your own particular techniques as you make coding mistakes and learn to find them, but most methods fall into one or more of three categories:

- Add code to narrow down the line or section in error.
- Pluck out a section of code into a separate program for analysis or rewriting.
- Analyze the values of variables at different points in the program.

Many techniques fall into more than one category. You can strategically stop the program with the [ON] key and check the value of variables, or you can insert `Disp/Pause` statements in code you suspect to be broken to trace the value of that code's variables as it executes. You can even combine all three, by pulling out a malfunctioning section, adding code to display values and help you understand what isn't working, then return to the original program and fix the problem.

To debug a program, you must first recognize that it has a bug. Ideally, you'll discover problems during testing. You should always thoroughly test your programs and games to make sure they work well for you, and if you can, give them to your friends to test as well. Try to think like a user, not like a programmer. Enter invalid numbers at prompts. Press random keys during a game. Enter values into a math program that you know should produce certain output values, and make sure that they do, as we did for our unit testing of PYTHTRIP via the intermediary PYTHPREL program. Once you've determined that an error exists, your remaining tasks are twofold, as will be covered in this section. First, you need to determine where the problem lies. Second, you need to figure out how to fix the issue.

I'll begin by outlining techniques to find malfunctioning code.

TRACING: WHICH LINE IS IT, ANYWAY?

If you made a good set of plans describing your program, they'll come in handy to help you debug. You'll usually be able to tell in what portion of your program an error occurs from how the error manifests itself. If the player isn't getting points for shooting down enemy spaceships in a game, then you'll probably need to look at the code that handles the destruction of the enemy ships and see if you forgot to make it update the score variable. If a math program displays a correct solution for one of two variables but not the second, you can probably focus on the equations that solve for the second variable. You may remember enough of your code to be able to start editing the program immediately and find the error and perhaps even fix it. If not, you'll need to narrow down the problem area to determine if you can just fix one or two lines or if the fundamental program design is flawed.

Once you know the general area of the program where the error occurs, you can apply tracing techniques to determine where the problem occurs:

- If the issue appears to be a mathematical error, examine your equations, and if possible, compare them as written in your program to your reference materials, either books or your notes. Make sure that you didn't omit a step in the solution.
- If code that should run isn't running, or code that shouldn't run is, check your conditional statements, particularly your comparisons. Make sure you used the proper variables, the proper values for comparison, and the correct operators. If you're using logical operators such as `and` and `or`, did you omit necessary grouping parentheses?
- If loops aren't running properly, check their conditions and variables. Do your `For` loops have incorrect arguments? Did you remember to initialize the variables for your `While` loops? If your loops are running forever, did you add proper code inside the loops to update variables used in loop conditions so that the loops will know when to stop?
- If code takes much longer to run than expected, make sure that it isn't stuck in an infinite loop, and if not, try to rethink your solution to see where it might be optimized, as we did earlier in this chapter.
- If variables suddenly have unexpected values, look for lines of code where you might have mixed up variable names.

These are a small subset of the many possible issues you may encounter. For any error, tracing the error should involve first understanding the symptoms of the problem and then using those symptoms to identify what portion of the code might be wrong.

For errors where you find the problem and the fix isn't an easy addition or removal of a few numbers or commands, you may also have to think hard about your solution.

RESOLVING BUGS: BREADCRUMBS AND OTHER TECHNIQUES

Fixing bugs and mistakes may be as simple as fixing a mistyped number or correcting the target label specified in a `Goto`. In many cases, even once you've identified the malfunctioning code, the solution may not be immediately apparent. If the techniques in

the previous section have failed to pinpoint the problem, you can use techniques such as the addition of breadcrumbs, adding extra output to your program to trace problems. From your planning, diagrams, pseudocode, and the ideas for your program that you have in your head, you know how it should work, and you can presumably tell when it deviates from that plan. Therefore, you can add `Disp` commands, `Pause` instructions, or both to your program and examine the values of variables as the program proceeds through suspect code. You could start with widely spaced debugging code, check when your variables get unexpected values, and then add more `Disp/Pause` commands to further narrow down the problem.

You may also find that when you reach a certain `Pause`, you want to stop the program with `BREAK` and either examine the code at that point or return to the homescreen, where you can manually check the contents of your program's variables. TI-BASIC is easier than many languages to debug in that variables retain their contents even after the program ends, which makes performing a "postmortem" investigation of problems simpler. On the flip side, applications written specifically to help you debug problematic programs in languages such as C can step line by line through your code, showing you the contents of the program's variables and memory at each step, whereas with TI-BASIC you need to add extra commands to your program to achieve the same effect.

If all else fails, your last resort is to try rewriting that particular section of code or to confer with other programmers. If you decide to rewrite the code, it's important that you keep the original code, in case you suddenly realize what was wrong with the first implementation while you write the new one. Such sudden revelations are more common than you might think, so retaining your original is invaluable. If you need other programmers to talk to, feel free to refer to the forums listed in appendix C.

Once you successfully track down and solve any problems in your program, you'll find that you almost always learn some new tricks that will help you complete your next project that much more easily. Remember, virtually every coding error is solvable with a bit of patience and careful examination.

5.6 *Summary*

In this chapter, I reviewed the skills necessary for you to become confident writing your own calculator programs, skills you can apply to any programming language. I introduced you to the steps for taking a program all the way from idea to a finished product and the pitfalls you might stumble upon along your journey. I then made these lessons more concrete with a Pythagorean Triplet solver, showing how to deduce the necessary math to solve the problem, create corresponding diagrams and pseudocode, and finally translate those into code. I demonstrated how a bit of consideration and optimizing can make a fast, small program even better. This chapter presented the skills to create programs from scratch and fix any problems that you need in order to boldly experiment with your own great programs. Learning programming isn't a passive activity, and I strongly urge you to start toying with program ideas of your own,

to start fine-tuning the lessons I've presented here into intuitive understanding of writing and debugging programs.

With these skills, you can start writing your own programs and games in earnest, exploring the features your calculator has to offer to programmers. In the next chapter, you'll learn to write dynamic, interactive programs and games that can directly read the calculator's keypad.

Part 2

Becoming a TI-BASIC master

Part 1 of this book teaches you skills that can easily be applied to any language, part 2 focuses on commands available specifically in TI-BASIC. Although event loops are useful for any programmer writing interactive programs, in TI-BASIC they'll let you create fun, real-time games. You'll learn about drawing and graphing from within programs and how to use real and complex numbers, randomization, lists, and matrices to build more advanced utilities and games.

In chapter 6, you'll learn about the event loop, a way to simultaneously check if the user has pressed keys and to update graphics and program state. Together with the skills you developed in part 1, you'll be able to build and test a full, interactive Mouse and Cheese game. Chapters 7 and 8 teach you about the many graphical tools available to TI-BASIC programs, including manipulating points and pixels; drawing lines, shapes, and text on the screen; and working with graphed equations. Examples include moving text and a mouse cursor around the screen, creating a painting program, and rendering polygons.

Chapter 9, the final chapter of part 2, provides a broad overview of strings, matrices, and lists as well as real, complex, and random numbers. These data types can be used in many different ways in your own program; the chapter presents a simple RPG (role-playing game) that you can expand with your own ideas and challenges you to write some of your own complete games.

6

Advanced input and events

This chapter covers

- Monitoring for events with event loops
- Getting keypresses directly with `getKey`
- Moving characters around the homescreen
- Building a fun, interactive game with event loops

One day, you pick up your calculator, having made yourself several games and programs. You've shared them with your friends, and they think you've done a good job. You're frustrated, though, because the games aren't very interactive. You want something more immersive, where you move a hungry mouse around the screen, trying to collect pieces of cheese. You want to make the game challenging, so you need to add a way to lose: you add a hunger bar. You decide that the hunger bar will gradually fill and that the player will lose if the hunger bar fills up. In sketching out your game, you envision something like the screenshot in figure 6.1, with a moving mouse (M) chasing a piece of cheese (square).

Excitedly, you create a file on your calculator and get ready to create this fast-paced game, only to suddenly discover that you have no idea how to start. You realize that you don't know any way to check if keys on the keyboard have been

pressed, other than using the `Input` or `Prompt` command to ask the user for a letter or number. For your game, you need a way to simultaneously check if the user pressed keys to move the mouse around and to fill up the hunger bar. If the player inputs nothing, you still want the hunger bar to continue filling up.

Needless to say, your adventures in programming don't stop there, leaving you unable to turn the game in your mind into a program. Like many other programming languages, TI-BASIC doesn't limit programmers to programs where the user types in input and gets the output in the form of a `Disp` or math programs that can only calculate results and display them in boring numerical form.

Figure 6.1   You imagine making a Mouse and Cheese game.

Sure, you could use `Input` and `Prompt`, but those would limit you to a static program, one that would stop completely every time you want to get input from the user. If you want to create something more fluid, such as an arcade-style Mouse and Cheese game or a math program where the user can move a cursor along a graphed function to examine properties of the function, you need something better. The solution is the `getKey` function, which allows you to check if any of the keys on the keyboard are being pressed without stopping the program, even if nothing is being pressed.

In this chapter, you'll learn about `getKey`, the command you use to check for key-presses, and you'll see how to move a letter around the screen with the arrow keys. With those skills, you'll move on to build the full Mouse and Cheese game with your new `getKey` knowledge and then look at ways to expand the game. The chapter will conclude with useful facts about `getKey`. But first, before you can effectively use `getKey`, you need to learn about an important programming concept: event loops, their purpose and construction, and how you can use them to create programs that can respond to events such as keypresses.
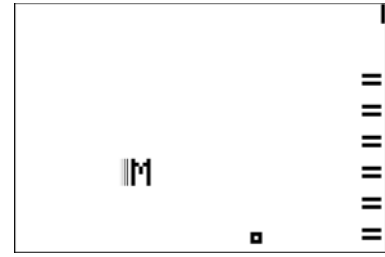
6.1   *Event loop concepts*

In its simplest form, the event loop lets your program do two things at the same time: occasionally check for and react to *asynchronous events* and execute other code repeatedly at the same time. As you'll see in our Mouse and Cheese game at the end of the chapter, the program can wait for the player to press a key and gradually increase the mouse's hunger at the same time.

An asynchronous event is something that happens that the program can't control, in this case the user pressing keys. The user chooses when to press a key, so a keypress is an event that occurs asynchronously. In other sorts of programs, asynchronous events might also include data arriving from a network.

Fortunately, you can catch asynchronous keypresses in your TI-BASIC programs. You use a construct called the event loop, the structure of which is shown in figure 6.2,
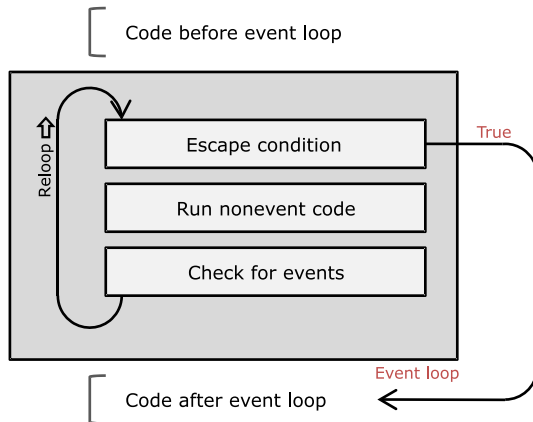
Figure 6.2   The structure of an event loop

which repeatedly checks for certain events to occur while running other code at the same time.

The ideal event loop contains as little code as possible, as you'll see in the Mouse and Cheese game that concludes this chapter. The less code in the event loop, the more often the loop will check for events (in this case, keypresses), and the faster your program will be able to respond to the events. If you try to cram a lot of code into the event loop, it will take your program longer to run each iteration of the loop, proportional to the amount of extra code you're putting in the loop, and the user will notice a delay between pressing a button and your program responding to the key.

A note on asynchronicity and computers

On some platforms, computers running OSs such as Linux, Windows, or Mac OS, you can divide your program into several threads or use related concepts called signals or interrupts, which let you handle asynchronous events completely separately from the rest of your program. One thread executes a loop looking for events to happen, while another thread might be responsible for updating the contents of the screen. With signals or interrupts, the OS will temporarily pause whatever part of your program is running and execute a different part that understands what to do with whatever asynchronous event just arrived. For example, when a TI-BASIC program stops with ERR:BREAK because the user pressed [ON], an asynchronous event has occurred that triggers an interrupt, stopping your program from executing and letting the calculator's own software take over again.

Unfortunately, TI graphing calculators and TI-BASIC in particular aren't quite as complex as computers, despite their power, and with TI-BASIC you can't use your own threads, signals, or interrupts. You can't tell the TI-OS that you want your own program to be notified when [ON] is pressed, which is part of the reason why [ON] has no keycode for `getKey` in figure 6.4 (more on that later).

Every event loop contains at least two basic parts: the section of code that looks for events and handles them and an escape condition. Most event loops have a third part: a section of code that executes other commands that are independent of events that are occurring. In figure 6.2, you can see the basic structure of the event loop; for the loops that only check for events, the "Do nonevent work" step is omitted.

I'll show you an example of each type of event loop in the next section, and later in this chapter, when you've learned to read events in the form of keypresses, you'll code your own event loops.

EVENT LOOP SKELETON

All event loops must check for events and act on events that occur. They must also have a certain set of conditions that end the repetition of the event loop; otherwise, the loop would run forever. Nearly all event loops use the `Repeat` command. As you learned two chapters ago, `Repeat [condition]` will repeat the code between the `Repeat` and its associated `End` command until the `[condition]` becomes true. `Repeat` loops are always guaranteed to execute at least once, which saves you the trouble of initializing the variables used in the condition. Because you know that the loop won't end unless the condition becomes true, it stands to reason that you need to modify at least one of the variables used in the condition inside the event loop. If written in pseudocode, the simplest event loop is structured like this:

```
:Repeat A=1
:Check for events
:If some event
:1→A
:End
```

If you're instead willing to escape the event loop if either of two events happens, you could expand that pseudocode example as follows:

```
:Repeat A=1 or B=1
:Check for events
:If some event
:1→A
:If some other event
:1→B
:End
```

As you can see, this will keep checking for events until one of the events that the program is looking for happens. At that point A or B will be set to 1, the `Repeat` condition will become true, and the program will continue onward, executing whatever code is after the `End` command.

Event loops can be more powerful than this. They don't need to only check for events; because the program doesn't stop at the `Check for events` line, continuing even if no event has happened (unlike `Input` or `Prompt`), the program can do other things inside the loop. This addition will expand the pseudocode event loop to look something like this:

```
:Repeat A=1 or B=1
:Run some non-event-related code
:Check for events
:If some event
:1→A
:If some other event
:1→B
:End
```

This piece of code could be made more realistic with a loop that will count upward until the [CLEAR] key is pressed. You don't yet know exactly how to check for the [CLEAR] key, so I'll write that part of the program in pseudocode:

```
:0→X                              ◁──────────────┐   The variable used
:ClrHome                                             for counting
:Repeat KEY=[CLEAR]
:X+1→X
:Output(1,1,X
:Check for keypresses
:Store any keys pressed into KEY
:End
```

This code will repeatedly loop through, adding 1 to X and displaying this new value on the screen, until the event loop notices that [CLEAR] was pressed. When this happens, the event loop ends, and the program continues with the code after the End command.

A REAL EVENT LOOP

I'll now take this pseudocode one step further so that you can try running an actual event loop, although one piece of it will be unfamiliar. The following chunk of code, the program EVNTLOOP, uses the getKey command, even though you haven't seen it yet. It returns a number indicating what key, if any, has been pressed, which it stores in K. Take my word that it works properly for now; in section 6.2 I'll go in depth into the details of cajoling getKey to do your bidding. Here getKey is used to check for the [CLEAR] key. The Repeat condition has become Repeat K = 45; you'll see later that the [CLEAR] key corresponds to a value of 45 returned from getKey. The event loop now looks like this:

```
PROGRAM:EVNTLOOP
:0→X
:ClrHome
:Repeat K=45
:X+1→X
:Output(1,1,X
:getKey→K
:End
```

This is the simplest instance of an event loop that both looks for events and performs other tasks. Once you see how to use getKey, you'll be able to write more interactive programs than any other examples you've explored thus far. Later in this chapter, you'll create such a game.

Now that you've seen the basic structure of an event loop, you need to find out how you can check for events, which for most graphing calculator programming will

be keys being pressed on the keyboard. Armed with that knowledge, you'll be able to write event loops that can read the keyboard, and then you'll expand your event loops to do other things while dealing with keypresses. This is a necessary detour from event loops themselves to give you the tools you need to construct your own event loops that can respond to almost any key on the calculator's keypad.

6.2   *getKey*

Say that you want to make a simple game where you control a character on the home-screen. You want to be able to move the character around the homescreen to any position, say to get to items it can pick up or to run away from some enemy that's trying to attack it. You know how to use the `Output` command to draw the player's character, the items to pick up, and the enemy and even how to make the enemy move around the screen by erasing it and updating its position variables. But what if you want to let the player move their own character around the screen? From what you know already, you could use `Input` or `Prompt`, perhaps to make the user type a letter (U, D, L, or R) to move the player's character up, down, left, or right. Perhaps the player could instead type a number, such as 2, 4, 6, or 8. But neither of these solutions is good for a fun, interactive, real-time game. The game would end up being more like a turn-based game, pausing every time it needs to ask the user where (or if!) they want to move their character. A better solution would be to read the calculator's keypad and be able to see if the user is pressing any keys without having to stop the rest of the program.

In this section, you'll first learn what `getKey` is and how to use it to check for keys pressed on the keyboard; then we'll move on to a few example programs that let you move characters around the screen of the calculator to get some experience using `getKey`.

6.2.1   *Using getKey for nonblocking input*

In this perfect solution, the enemy would be able to continue to move toward the player while the program does other things. At the same time as the calculator repeatedly checks for pressed keys, items could appear or disappear, a time limit could count down, and anything else the program can do could run. In common computer terminology, this type of input is called *nonblocking* input, which means that the device checks for input but doesn't wait if there's no input, continuing regardless and allowing it to work on other things while it waits. By contrast, `Input` and `Prompt` are *blocking* input, which means that the program stops in its tracks while waiting for the user to do something, such as enter a number or a string and press [ENTER]. If the user does nothing, blocking input commands like `Prompt` and `Input` continue to wait. Luckily the TI-BASIC language has a nonblocking input command that you can use to make the game work perfectly, a command called `getKey`. When run, this command checks to see if any keys on the keyboard have been pressed; if so, it returns a number indicating which key was pressed. If no key has been pressed since the last time that `getKey` was called, it returns zero. It's a nonblocking function because it returns zero when no key is pressed instead of waiting for a key.
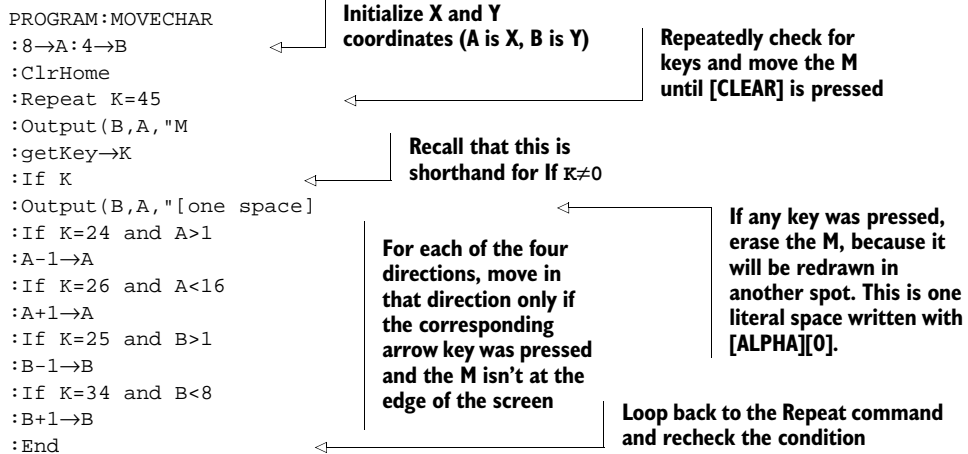
A SAMPLE GETKEY PROGRAM: MOVING AN M

To see the getKey command in action, I'll start with the solution to the player-movement game. You'll begin by creating a small program to move a single letter M around the screen. It will start the M at an initial position near the middle of the screen and then move the letter around as the user presses the [up], [down], [left], and [right] arrows. [CLEAR] is the exit key for this particular program. Your program will prevent the character from moving outside the edges of the screen. Should you wish to try this program, the only command you might not yet know how to find is getKey, which is under [PRGM], I/O, 7: getKey ([PRGM][RIGHT][7]).

This program, shown in listing 6.1, will introduce three new concepts that you haven't seen before or that we've only briefly discussed:

- Using getKey to read keys from the keyboard. Figure 6.4 shows the codes for the arrow keys used in this program.
- Outputting a space character over another character to erase the previous character.
- Performing bounds checking to ensure nothing goes off the edge of the screen.

After you have a chance to look at the program in the following listing and try it out, I'll describe each of the pieces in detail to help you understand how it works.

Listing 6.1 Four-directional movement of an M around the homescreen

```
PROGRAM:MOVECHAR
:8→A:4→B          Initialize X and Y coordinates (A is X, B is Y)
:ClrHome
:Repeat K=45       Repeatedly check for keys and move the M until [CLEAR] is pressed
:Output(B,A,"M
:getKey→K
:If K              Recall that this is shorthand for If K≠0
:Output(B,A,"[one space]
:If K=24 and A>1
:A-1→A
:If K=26 and A<16
:A+1→A
:If K=25 and B>1
:B-1→B
:If K=34 and B<8
:B+1→B
:End
```

For each of the four directions, move in that direction only if the corresponding arrow key was pressed and the M isn't at the edge of the screen

If any key was pressed, erase the M, because it will be redrawn in another spot. This is one literal space written with [ALPHA][0].

Loop back to the Repeat command and recheck the condition

NOTE If you're an intermediate or experienced programmer, the use of A and B for the coordinates of the M in the code in listing 6.1 may seem weird to you. You may be asking why I didn't use the more obvious variables X and Y, so that I could write things like Output(Y,X,"M. Unfortunately, as I'll reiterate in chapters 7 and 8 when I introduce the graphscreen, the calculator's OS has a bug that sets Y to zero every time you clear the screen, which, although it won't break the MOVECHAR program, will be something you'll need to be aware of. Better to start training yourself not to use Y in graphical programs now and set good habits.
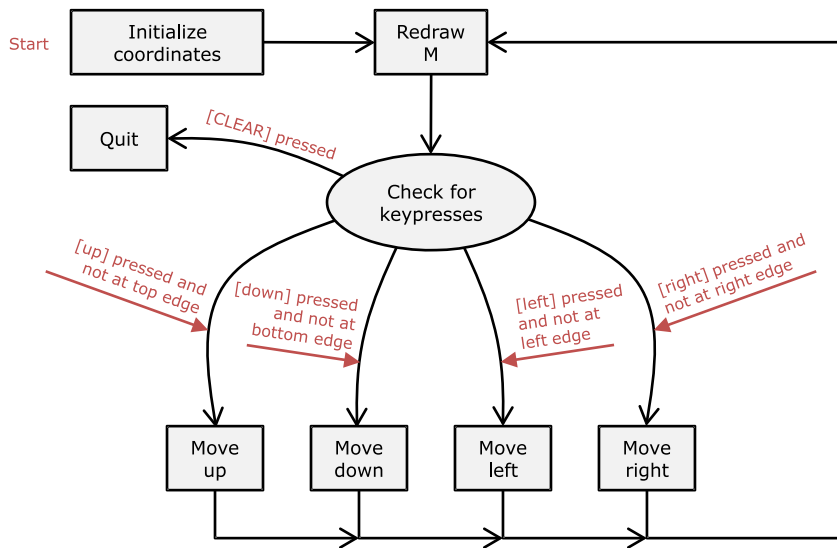
Figure 6.3   Flowchart of functionality for the four-direction M-moving program, MOVECHAR

The flow of the program should be fairly clear. If not, look at figure 6.3, and try to match the pieces in the diagram to the source code in listing 6.1. It initializes A and B, which you'll use as your X and Y coordinates on the homescreen (column and row) respectively, and then begins a `Repeat` loop. The `Repeat` loop continues until K = 45; you can see that K is assigned from the output of `getKey`, so the loop continues until `getKey` returns 45. If you take a look at the specification above the program, you'll see that the program should exit, in this case leave the program loop and reach the end of the program, when [CLEAR] is pressed. And indeed, 45 is the number that `getKey` returns when [CLEAR] is pressed. Inside the loop, the program displays an M character at the current row and column and then checks the keyboard, moving the M accordingly if any keys were pressed. Remember that `If K` is the same as saying `If K≠0`: the `Output` command immediately after the `getKey` erases the character just drawn if any key was pressed. We do this so the program doesn't leave a trail of Ms behind the character as it moves. You could clear the screen instead of outputting a space character to overwrite the M, but you'll see in more complex examples that it's usually better to erase a single character rather than erase everything on the screen only to draw most of it back in.

The four sets of conditionals each cover one of the four directions in which the M can move: left, right, up, and down, in order. Each conditional has an `and` because it can only move if both the key for that direction was pressed and moving in that direction isn't going to make the M fall off the edge of the screen. The relationship between the keys and their conditionals and edge conditions is shown in figure 6.4.
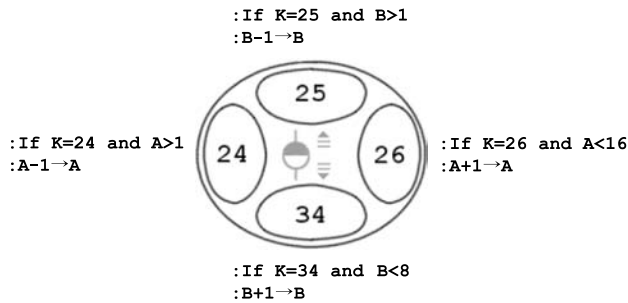
```
:If K=25 and B>1
:B-1→B
```



```
:If K=24 and A>1
:A-1→A
```

```
:If K=26 and A<16
:A+1→A
```

```
:If K=34 and B<8
:B+1→B
```

Figure 6.4   Arrow keys and their respective conditional blocks for the MOVECHAR program

If both conditions are true for a given direction, then the coordinates stored in the column (A) or row (B) variable change accordingly; the program then loops back to the Repeat, where the M will be redrawn.

DEFENSIVE PROGRAMMING

If the program didn't have that check for the edge of the screen, and you tried to use the Output command with coordinates outside the edges of the screen, the calculator would throw an ERR:DOMAIN error. This safety checking is an example of defensive programming and is good practice to get used to for any programming language. The main concept of defensive programming is to not trust the *sanity* of any input from outside the program. A programmer should assume that any input to their program is potentially wrong, unexpected, or could break the program. An example of sanity checking to program defensively is checking that input typed into a Prompt where you expect a number is not instead a symbol, a string, or even a blank line. Luckily, the TI-OS performs sanity checking with Prompt and Input for you and will display errors such as ERR:SYNTAX if the user types non-numbers into a numeric input field. But the OS can't catch every error: if you ask the user for a number between 1 and 10, it's up to your program to check the value the user has typed and make sure that it is between 1 and 10. A defensively written program would continue asking the user for a number until the user typed a value between 1 and 10. A program lacking these defenses might break if the user typed in a bad value and it didn't check the sanity of the input. A great example would be our ISPRIME prime number checker from chapter 4, which originally broke when the user entered a negative number.

Because TI-BASIC is a language written for both beginners and advanced users, it does a lot of error checking on its own and is thus itself defensively programmed. It will produce an error instead of crashing if you try to draw off the edge of the screen. Fortunately, this way you won't crash your calculator every time you make a programming error, which could be frustrating for new programmers. Unfortunately, unless you defend against and handle the errors yourself, your program will be stopped by the TI-OS when one of these errors occurs, meaning your user or player will have to start over. Errors in your program also make it appear less professional and polished to the user or player. Programming defensively keeps your program in control of the calculator unless the user presses [ON] to interrupt it and ensures a much better user experience.

With your first `getKey`-based movement program under your belt, we can move on to a more systematic explanation of the values that `getKey` uses to represent each key.

6.2.2   *Learning getKey keycodes: the chart and the memorization*

As you may have seen from the sample program in listing 6.1, the keycodes that `getKey` returns, indicating which (if any) key was pressed, are all integers, whole numbers above zero. `getKey` also can return 0, which means that no keys were pressed since the last time your program called `getKey`. The keycodes are all two-digit or three-digit numbers and follow a simple pattern. It's so simple that I'll first give you a diagram of the TI-83+ calculator showing the codes for all the keys, so that you can try to see the pattern for yourself, and then I'll tell you what the pattern is. Notice that the only key that's missing a keycode is [ON]; it has no corresponding code. Figure 6.5 displays the keycode for each key on top of the key.

As you can see, each key has only one code. You might ask how you can check for modifier keys, such as [2nd][key] or [ALPHA][key]: I'll discuss this in more detail in section 6.4.2. For now, assume that you can only detect unmodified keys.

Looking at the calculator's keypad in figure 6.5, and working on the programs that you've tried that include `getKey` thus far, you may have started to notice the pattern in the keycodes. Disregarding the arrow keys, you'll notice that every keycode for the first column ends in 1, every keycode in the second column ends in 2, and so on. Going down the rows, every graph key starts with a 1; the second row

Figure 6.5   `getKey` keycodes for each of the keys on a TI-83+ or TI-84+ keypad. [ON] has no keycode.

including [2nd], [MODE], and [DEL] all begin with 2; and the keys on the last row of the keypad all begin with 10. To calculate the keycode for any key, count which number row it's in from the top, multiply by 10, and add the number of the column that it's in, starting from the left. For example, to find the keycode for [SIN] you see that it's in the fifth row and second column of the keypad. Because $(5 * 10) + 2 = 52$, the keycode for [SIN] should be 52, and indeed it is. The only exceptions, other than the missing keycode for [ON], are the arrow keys, which are 24 and 26 for [left] and [right] and 25 and 34 for [up] and [down]. Because the arrow keys don't follow the same obvious pattern as the other keys, you may decide to memorize the four arrow keys' codes, but once you write a few games, you'll find the codes to be second nature. If you look closely at the diagram, you may notice that the [down] arrow has a first
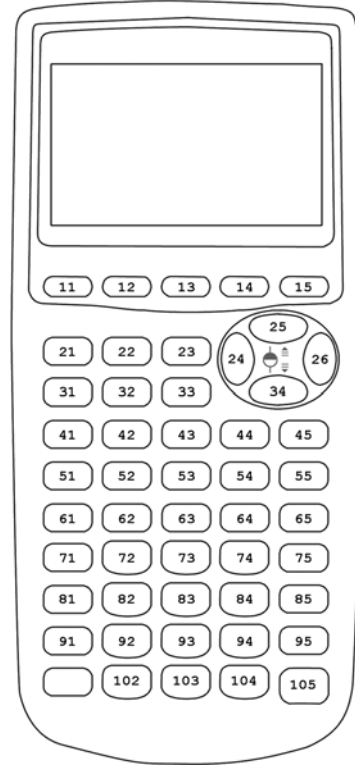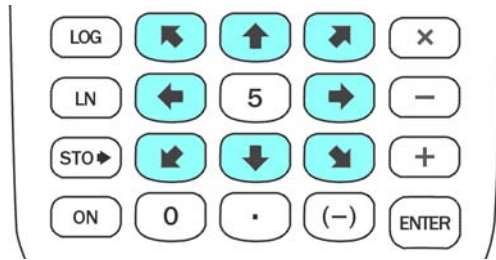
Figure 6.6   Using the number keys for diagonal movement, as shown on the bottom four rows of a TI-83+ calculator's keypad

digit of 3, as if it was in the row with [STAT]. The [left], [up], and [right] arrows are numbered as if they follow the [DEL] key, so even with the arrows some pattern is maintained.

Another question you might ask is how you can check for two keys at once, (holding down [left] and [up] at the same time to make a character move diagonally across the screen). Sadly, without advanced functions that I'll discuss in chapter 11, `getKey` can only tell the program calling it about one key being pressed. There are still clever ways to allow the user to move diagonally. One sly solution uses eight of the nine number keys of the calculator as if they were arrow keys. [8] is used as up, [2] as down, [4] as left, and [6] as right. The corner keys ([1], [3], [7], and [9]) are then used as the diagonal movement keys. Figure 6.6 shows the layout of keys for diagonal movement.

I'll finish with a simple program that will tell you the keycode for any key on the keyboard. Run it, press the key that you want, and it will display the code for that key:

```
PROGRAM:KEYCODE
:Repeat Ans
:getKey
:End
:Disp Ans
```

This program saves a variable by using the special `Ans` variable, typed with [2nd][(-)]. When `getKey` is run without an assignment (→Variable), it stores the code of the key returned into `Ans`. A `Repeat` loop is wrapped around this, forcing `getKey` to continue to run and store keycodes in `Ans` until Ans≠0. At this point, the user must have pressed a key, so the program displays `Ans`, thus displaying the keycode of whichever key was pressed as reported by `getKey`. Chapter 10 will teach you more about using `Ans`.

With the essentials of event loops and `getKey` laid out before you, I'll give you a problem to solve yourself. Based on figure 6.6 and your newfound knowledge of the `getKey` keycodes (if you must, you may refer to figure 6.5), the first exercise of this chapter will have you take the MOVECHAR program and modify it for eight-directional movement.

6.2.3   *Exercise: eight-directional movement*

This exercise asks you to modify the code given for the MOVECHAR program in section 6.2.1 to allow for eight-directional movement of the M character around the

homescreen using the number keys as arrow keys, as shown in figure 6.6. Remember to think about what should happen if the character reaches the edge of the screen, especially because each diagonal movement requires checking two edges. For example, if the character is at the top edge of the screen and the player presses [7] to move up and left, think about whether you should let the player just move left or not move at all. There are two main approaches to solve this problem. One involves using a set of eight conditional statements (one for each of the possible directions), whereas the second possible solution integrates the tests for diagonal keys into the four conditional statements already given in the MOVECHAR program. Remember also to change the four cardinal directions (up, down, left, and right) to use number keys instead of the arrow keys.

SOLUTION 1: FOUR CONDITIONALS

As usual, there are many possible ways that this exercise could be solved, which fall into one of two major approaches. You'll see these two possible approaches presented here, the first of which uses four conditional statements and the second of which uses eight different statements. You'll see that the first of the programs is much shorter and cleaner; the second is longer. The two programs work slightly differently when the player presses a diagonal movement key at the edge of the screen. In the first, shorter program, if the player is at the left edge and presses [7] (up-left), the character will move up (but not left off the edge). In the second program, the character won't move unless it can go both up and left.

The first program functions based on the fact that three keys can be used to go in each direction. That sounds confusing, so let's look at what I mean. The three up keys (up-left, up, and up-right), [7], [8], and [9], are all used to move the player's character upward. For now, we won't worry about the fact that [7] also needs to move it left and [9] also needs to move it right. Because the keycodes for [7], [8], and [9] are 72, 73, and 74, you could write

```
:If K=72 or K=73 or K=74
:B-1→B
```

Recall that B is the variable holding the Y-coordinate, so decreasing it moves the character up the screen. That set of two lines of code will let the player move off the top edge of the screen, so you need to add a condition checking if the character is already at the top edge. One possible solution could be

```
:If K=72 or K=73 or K=74 and B>1
```

Why B > 1? Because the character should move up only if the M isn't already at the top of the screen, which has Y-coordinate B = 1. Unfortunately, this full line of code has the flaw that the B > 1 condition might be anded with K = 74 and thus only apply to moving up and right, whereas you want it to be applied to all three possible keys. You can solve the problem by using grouping parentheses to make sure the three ored key conditions are checked before being anded with the edge-detection condition:

```
:If (K=72 or K=73 or K=74) and B>1
```
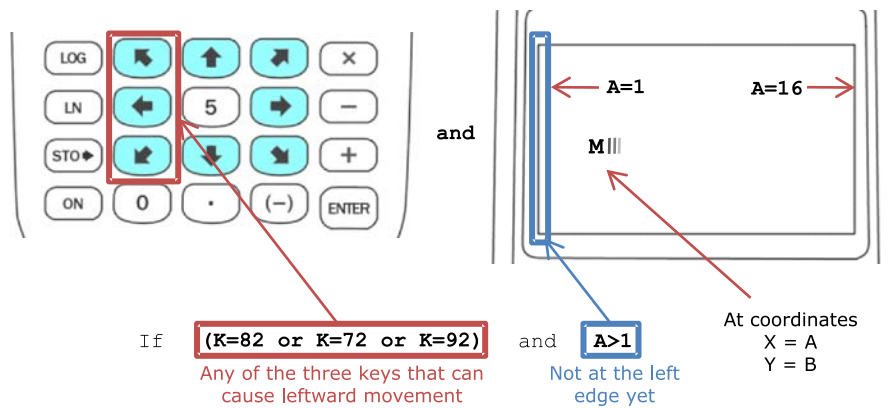
Figure 6.7 Conditional checks for arrow key pressed and M not at the edge of the screen

A visual explanation of this concept is shown in figure 6.7, for leftward movement of the M character. You can see this solution in action in the source code of the program MOVE8D1 in listing 6.2, used four different times for the four cardinal directions. Each of the diagonal keys appears in two of the statements, so that the up and left conditions both include, for example, the keycode for up-left (K = 72).

Because the two directions for each diagonal are checked separately, each with separate edge checking, the player can move along an edge by pressing one of the diagonal keys.

Listing 6.2   One possible 8-directional movement solution

```
PROGRAM:MOVE8D1
:8→A:4→B
:ClrHome
:Repeat K=45
:Output(B,A,"M
:getKey→K
:If K
:Output(B,A,"[one space]
:If (K=82 or K=72 or K=92) and A>1        Keys [l], [4], or [7] and
:A-1→A                                    not at the left edge
:If (K=84 or K=74 or K=94) and A<16
:A+1→A
:If (K=72 or K=73 or K=74) and B>1        Keys [7], [8], or [9] and
:B-1→B                                    not at the top edge
:If (K=92 or K=93 or K=94) and B<8
:B+1→B
:End
```

ADDING A SPECIFICATION CONSTRAINT

This solution works well but reveals an omission in the original exercise that I assigned. I never specified what should happen when the character reaches an edge and the user tries to press an arrow key. In the code in listing 6.2, it's easiest to allow

the character to slide along edges, but when you design programs like this, it's much better to define exactly what will happen in cases such as this. You need to pick whether you want a diagonal key at an edge to let the character slide along that edge or to force it to stay in the same place. For the sake of the second example solution, I'll add the additional constraint that if the character can't move diagonally when a diagonal key is pressed, it should not move at all. After I go through this solution, I'll switch the constraint back to allowing the character to slide along edges with diagonal keys, and you'll see what changes.

SOLUTION 2: EIGHT CONDITIONALS

The second possible solution program is longer and less elegant, but it follows this extra new requirement more closely. The code for this second solution is shown in listing 6.3. It includes eight separate sets of conditionals, one for each of the eight movement keys, and checks all the necessary edges with each key. For the down key, it checks that the player isn't at the bottom edge, whereas for the down-right key, it checks that the player is at neither the right edge nor the bottom edge. For the diagonal keys, both the X and Y variables (A and B) are updated. As you learned in chapter 3, when a conditional controls more than one statement being executed, you can't use the short form of If and must instead wrap the statements in Then and End. As with the original example and the first solution, a Repeat loop is used to continue running the program until the player presses [CLEAR] or breaks the program with [ON].

Listing 6.3   MOVE8D2, another possible eight-directional movement solution

```
PROGRAM:MOVE8D2
:8→A:4→B
:ClrHome
:Repeat K=45
:Output(B,A,"M
:getKey→K
:If K
:Output(B,A,"[one space]
:If K=72 and A>1 and B>1
:Then
:A-1→A:B-1→B
:End
:If K=82 and A>1
:A-1→A
:If K=92 and A>1 and B<8
:Then
:A-1→A:B+1→B
:End
:If K=73 and B>1
:B-1→B
:If K=93 and B<8
:B+1→B
:If K=74 and A<16 and B>1
:Then
:A+1→A:B-1→B
:End
```

Separating commands with a colon is the same as hitting [ENTER] in between

Left, right, up, and down movement only needs to update one variable

Diagonal movement updates both X and Y, so it needs a Then/End

```
:If K=84 and A<16
:A+1→A
:If K=94 and A<16 and B<8
:Then
:A+1→A:B+1→B
:End
:End
```

As promised, I'll conclude this exercise by removing the constraint that diagonal keys must cause diagonal movement or nothing at all. You'll see that this rearranges the second solution in program MOVE8D2 in listing 6.3, moving the conditional edge checks for the diagonal keys inside the conditional block for that particular key. As expected, because I'm only changing the requirements for diagonal movement, the code for vertical and horizontal movement doesn't change. This code as presented omits the larger loop structure, showing only the variable update section (or event-handling code, if you will):

```
:If K=72:Then
:If A>1:A-1→A
:If B>1:B-1→B
:End
:If K=74:Then
:If A<16:A+1→A
:If B>1:B-1→B
:End
:If K=92:Then
:If A>1:A-1→A
:If B<8:B+1→B
:End
:If K=94:Then
:If A<16:A+1→A
:If B<8:B+1→B
:End
:If K=73 and B>1
:B-1→B
:if K=93 and B<8
:B+1→B
:If K=82 and A>1
:A-1→A
:If K=84 and A<16
:A+1→A
```

Here, I keep the four conditional blocks that I had in MOVE8D2, one per diagonal key, but the edge conditions are no longer anded with the checks for the keys themselves. Instead, they're moved inside the block and used by themselves to conditionally control movement horizontally and vertically. For example, if the up-left diagonal key is pressed, then the character may move up, left, both, or neither, depending on the values of A and B. You'll notice that several redundant conditional variable updates remain, such as decrementing B as long as B > 1 for both K = 72 and K = 74. Imagine what would happen if I rearranged these pieces of code to remove the redundancies:

```
:If (K=72 or K=74) and B>1
:B-1→B
```

```
:If (K=92 or K=94) and B<8
:B+1→B
:If (K=72 or K=92) and A>1
:A-1→A
:If (K=74 or K=94) and A<16
:A+1→A
:If K=73 and B>1
:B-1→B
:if K=93 and B<8
:B+1→B
:If K=82 and A>1
:A-1→A
:If K=84 and A<16
:A+1→A
```

Noticing that the two halves of this code block do the same thing for different key codes, I can consolidate once more. I can add the K = 73 to the first conditional, the K = 93 to the second, and so on, to combine eight conditional updates into four conditional updates. But if you look carefully, you'll see that this program is now back at exactly the first solution to the exercise, which means I came up with the same solution again, albeit in a different form.

```
:If (K=72 or K=73 or K=74) and B>1
:B-1→B
:If (K=92 or K=93 or K=94) and B<8
:B+1→B
:If (K=72 or K=82 or K=92) and A>1
:A-1→A
:If (K=74 or K=84 or K=94) and A<16
:A+1→A
```

Notice that this exactly matches the key-handling code in listing 6.2, with the conditionals rearranged. Because the four separate conditional statements don't depend on each other, the rearrangement makes no difference to how the code works.

PROGRAMMING FLEXIBLY

Ideally, as a programmer writing any language, you'll begin to see different ways to think about the same problem and to solve it given whatever constraints the problem might have. This will also help you learn to see things in conditionals, logic, and program flow that can be simplified, combined, or rearranged to save commands and thus reduce size and increase program speed. As discussed previously, it's best to think about two or three ways to solve the problem and pick the one that seems like it will be the fastest and most efficient. If you find yourself halfway into writing something like the second example just shown, and you suddenly think of how to write the program the first way, it would be better to either switch to the new method or try out both than to become enamored of your first attempt and unwilling to try alternative solutions. Although you'll spend more time writing the program, in the end you'll have a better product, your users will be happier, and you'll feel more pride as a programmer.

In this particular example, the second program, in enforcing that diagonal keys move the M diagonally or not at all, has to be longer. Longer code is generally bad, because it usually takes longer for the calculator to run. But if the longer code is necessary to fulfill additional constraints, as happened here, writing more code to satisfy the program's requirements can be unavoidable. In addition, if the longer version is easier to understand, it will be easier for you to remember what you did if you need to go back into your code to expand it, solve a bug, add a new feature, or tweak the way it works. For the second solution, it's clear how the program is checking for the keys corresponding to each of the eight directions, while carefully checking if the M is at the edges that each movement direction could cause the M to cross. As you become a more experienced programmer and start to get a more intuitive feel for reading even complex code, you'll find yourself leaning more toward the shorter, more-cryptic solutions to things. You'll see some extreme examples of efficient, somewhat obfuscated code in part 3, especially in chapter 10.

Because you've seen how to use `getKey` for interactive, responsive programs, you're ready to create a more complex event loop that utilizes `getKey` as part of a full game. You'll combine your new skills with event loops and `getKey` with other things you now know about using loops, conditionals, and output to write a fun, if *cheesy* (groan), game.

6.3 The Mouse and Cheese game

It's time to combine the concepts you've been learning for moving a letter around the homescreen with the discussion of event loops to create a game. In this game, the player controls a mouse, represented by the trusty capital M. The player will use the arrow keys to move the M around the screen, chasing after pieces of cheese, represented by a small square. To make the game competitive, the mouse's hunger will increase as time passes, regardless of whether or not keys are pressed, which is where your event-loop knowledge will become necessary. To display the hunger, the program will draw a bar at the right edge of the screen made of equals (=) characters. When the hunger bar fills all the way up, the game will end and will tell the player how many pieces of cheese they were able to eat before the game ended. You can see a game in progress and the end of a game in figure 6.8.
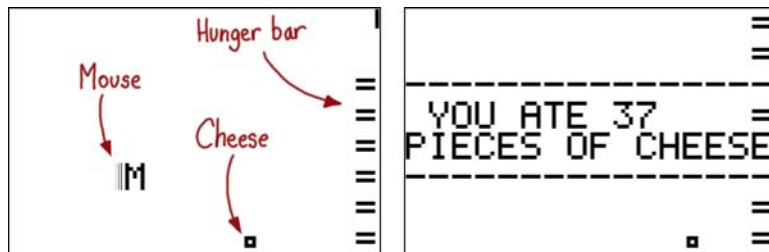

Figure 6.8   Screenshots of the Mouse and Cheese game. The mouse (M) is moved by the player to chase after the pieces of cheese (the square). The player loses if the hunger bar (right edge of the screen) fills up, at which point the game tells the player how many pieces of cheese the mouse ate (right).

The program will be structured with two nested `Repeat` loops; the following pseudo-code outlines the structure of the program that you'll create:

```
:Repeat until hunger bar fills or [CLEAR] pressed
:[initialize cheese and hunger bar]
:Repeat until hunger bar fills or [CLEAR] pressed or mouse reaches cheese
:[move mouse and increase hunger]
:End
:[handle eating cheese]
:End
```

At the beginning, the mouse's hunger and coordinates will be initialized, and a starting score of zero will be set. At the end, the final score will be displayed to the player. The outer loop will run once per piece of cheese, so it will first create a piece of cheese at random coordinates and display it, then draw the hunger bar, and then start the inner loop. At the end of the outer loop, the player's score will be increased and the mouse's hunger removed if it reaches the piece of cheese. The outer loop ends if the hunger bar is full or the user presses [CLEAR]. The inner `Repeat` loop runs the heart of the game. On every iteration, it checks for keys and moves the mouse accordingly, increases the mouse's hunger, and, if necessary, draws another = sign on top of the hunger bar.

With this basic skeleton in mind as a guide, you can move on to looking at the full TI-BASIC source code for the game.

6.3.1 *Writing and running the game*

You know where to find `getKey` now, and all of the other commands in the code in listing 6.4 should be familiar to you as well. The square symbol on the seventh line of the code might be unfamiliar; you can find it under the MARK menu, [2nd][Y=][▶][▶]. You should type this game into your calculator so that you can try it out and better understand the discussion of how the code works. You'll also want to use it as a base to experiment with tweaks and changes. As with all other programs you've worked with so far, run the program from the homescreen (or your favorite shell). You can press the arrow keys to move the mouse around and press [CLEAR] to end a game early.

Listing 6.4   The Mouse and Cheese game

```
PROGRAM:CHEESE
:7→A:4→B
:0→H:0→S
:Repeat H≥8 or K=45
:randInt(1,15→C
:randInt(1,8→D
:ClrHome
:Output(D,C,"□
:For(X,1,int(H
:Output(8-X,16,"=
:End
```

X and Y coordinates of the mouse

Hunger (H) ranges from 0 to 8; score (S) increases for each piece of cheese eaten

Initialize coordinates of piece of cheese to a random spot on the screen

Outer loop: one iteration per piece of cheese, ends when game ends

Draw the current hunger bar

Only display the cheese once (this symbol is under the MARK tab of [2nd][Y=])

```
:Repeat H≥8 or K=45 or (C=A and B=D
:If H=int(H
:Output(8-H,16,"=
:Output(B,A,"M
:getKey→K
:If K
:Output(B,A,"[one space]
:If K=24 and A>1
:A-1→A
:If K=26 and A<15
:A+1→A
:If K=25 and B>1
:B-1→B
:If K=34 and B<8
:B+1→B
:H+.1→H
:End
:If C=A and B=D
:Then
:0→H
:S+1→S
:End:End
:Output(3,1,"---------------
:Output(4,1," YOU ATE
:Output(4,10,S
:Output(5,1,"PIECES OF CHEESE
:Output(6,1,"---------------
:Pause
:ClrHome
```

Inner loop: runs repeatedly until hunger bar fills, [CLEAR] is pressed, or mouse reaches cheese

Increase hunger on every loop, whether or not a key is pressed

If mouse is at same coordinates as cheese, increase score and remove hunger

End outer loop and finish the game by displaying score

6.3.2 Understanding the game

The Mouse and Cheese game draws on many of the skills that I've discussed thus far, including conditionals, nested Repeat loops, using getKey in asynchronous event loops, using Output to display strings, characters, and numbers, and dealing with integer and decimal numbers. I'll start a detailed look at the code with the initialization at the beginning of the program, work my way through the outer loop and the inner loop, and conclude with the game-ending score display.

The Mouse and Cheese game uses six major variables. The player's mouse is at some (X,Y) coordinates, which are stored in the variable pair (A,B), and each piece of cheese is at coordinates (C,D). Because only one piece of cheese is displayed at a time, it doesn't need to keep track of multiple pieces of cheese. It does need to keep track of the mouse's hunger, which it puts in H, and the player's score, in S.

```
:7→A:4→B
:0→H:0→S

:randInt(1,15→C
:randInt(1,8→D
```

Because the homescreen is 8 characters tall, I decided to make H = 0 mean no hunger, and H = 8 mean full hunger. The score S will increase by 1 with each piece of cheese

eaten, but you could easily change it to some other value. You could even give the player more points for getting the cheese faster, or some other more complicated scoring scheme.

INITIALIZATION AND PROGRAM STRUCTURE

You might start writing this game by figuring out exactly how the screen will be laid out. Because the hunger bar will be the last column of the homescreen, the 16th column, the mouse and cheese can both be anywhere in the 15-column by 8-row area starting at the left edge of the screen. The mouse's position can be initialized near the center of that area, at (X,Y) = (A,B) = (7,4). The hunger H should start at 0, and the score S will also be initialized to 0. Next come the two nested `Repeat` loops that form the body of the game.

One excellent question would be why the program uses two nested loops instead of one giant loop. A single giant loop could indeed be used instead, in which you'd have a conditional that would generate new coordinates for the cheese (creating a new piece) every time the mouse reached the cheese, but this would be slow, because the program would have to jump over that piece of code every time through the `getKey` loop, regardless of whether or not the mouse reached the cheese. The left side of figure 6.9 shows this incorrect structure, where all of the main game code including creating new pieces of cheese is inside the event loop. The right side of the diagram shows the proper structure, where only updating the hunger bar and updating the mouse's position are inside the event loop. Creating and destroying the cheese are part of the outer loop.
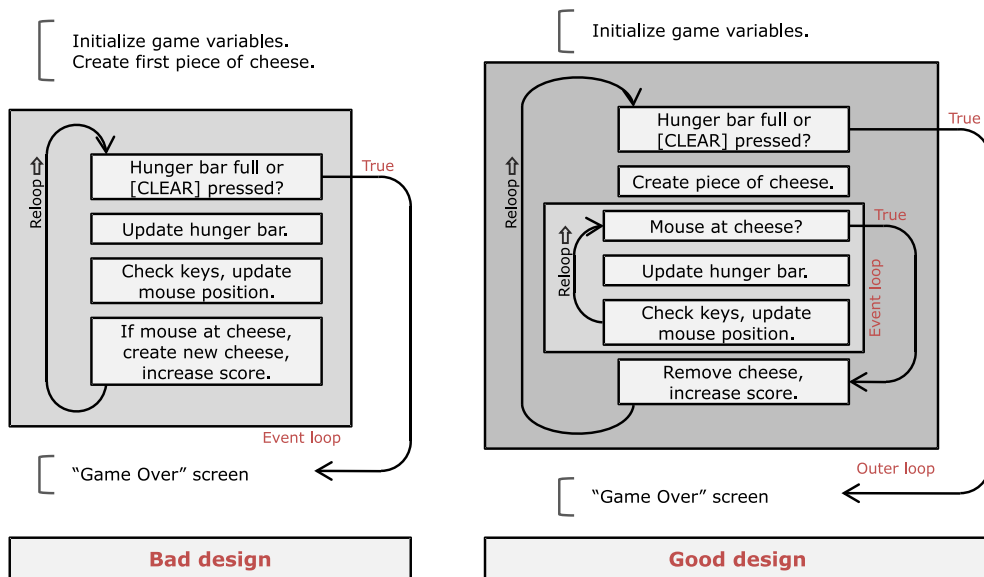
Figure 6.9   Putting the full game loop including cheese management inside the event loop (left) and properly reducing the amount of work in the inner event loop (right)

The outer loop starts by creating the cheese and ends by handling the mouse reaching the cheese; the inner loop only handles moving the mouse around and adjusting the hunger bar. The inner loop terminates when the mouse reaches the cheese to let the outer loop handle the mouse eating the cheese. This is shown in the following pseudocode:

```
:Repeat H≥8 or K=45
:[initialize cheese and hunger bar]
:Repeat H≥8 or K=45 or (C=A and B=D
:[move mouse and increase hunger]
:End
:[handle eating cheese]
:End
```

Look at the conditionals on each of the loops first. You know that `Repeat [condition]` is like telling the program "repeatedly run this loop until [condition] is true" from chapter 4. For the outer loop, the program keeps creating pieces of cheese and letting the mouse chase them until the hunger bar is full or the user presses [CLEAR]. Because the hunger bar will get full or the [CLEAR] key will be pressed when the program is inside the inner loop, both the inner and outer loops need to have these two conditions. Therefore, when the hunger bar fills or [CLEAR] is pressed, the inner loop will end, the end of the outer loop will run, and then the outer loop will end too.

THE OUTER LOOP: CHEESE MAINTENANCE

Because one new piece of cheese should be created every time the outer loop starts, the program generates the cheese at random coordinates between the start of the outer loop and the beginning of the inner loop. The `randInt` command can be used to do this:

```
:randInt(1,15→C
:randInt(1,8→D
```

Notice that because the mouse and cheese can only be between X = 1 and X = 15, and Y = 1 and Y = 8, you use these values as the limits for the `randInt` command. After the program generates a new piece of cheese, it should clear the screen, draw the cheese, and draw the hunger bar. Why do you clear the screen instead of just erasing the old piece of cheese using the `Output(D,C,"[one space]` trick? Because when the mouse eats a piece of cheese, the hunger bar becomes empty, and it might have been nearly full. It's much easier in this case to clear the screen, then draw the cheese and empty hunger bar. The space trick would work fine in this case; it would be more complicated. In many of your programs, just as in the eight-direction program, you'll run into several possible ways to do something. In many cases, one of the options will be fast, one will use a small amount of code, and the smallest version may not necessarily be the fastest.

   Clearing the screen is easy enough with `ClrHome`, but how shall the hunger bar be drawn? Because I decided hunger can go from 0 through 8, the program could loop X from 0 to H, drawing an equals sign at (X,Y) = (15,X + 1), which would make the hunger

bar grow from the top of the screen downward as the mouse's hunger increases (here, X is being used as a throwaway looping variable; the program doesn't use it after the hunger-bar-drawing loop ends). The program needs to display at row X + 1 instead of row X because 0 isn't a valid Y coordinate on the homescreen: the first row is 1, not 0. But to be extra fancy, we'll make the hunger bar grow from the bottom of the screen to the top. The coordinates can be flipped over from the top to the bottom of the screen by subtracting them from 8. Therefore, 8 – X will be 8 when X = 0, then 7 when X = 1, and so on, so it will grow upward from the last row. Because the hunger always gets reset to H = 0 when the mouse eats a piece of cheese, making the hunger bar nearly empty, the program doesn't need a loop here to redraw the hunger. It could instead have a single `Output(8,16,"=` command. In case you decide to modify the game, to only decrease the mouse's hunger rather than removing it completely when the mouse gets a piece of cheese, this loop will be able to correctly handle drawing any hunger level.

THE EVENT LOOP: HUNGER AND THE SCURRY OF THE MOUSE

Continuing through the program, next comes the inner loop, which is the event loop of the game. This loop will continue to execute, checking the keyboard for activity and moving the mouse accordingly, while simultaneously updating the mouse's hunger and filling the hunger bar displayed onscreen. The hunger bar update is the "other code" section of the event loop, whereas the keypresses that cause the mouse to move are the events that it acts upon. For the sake of analysis, take a look at the body of the inner loop by itself:

```
:Repeat H≥8 or K=45 or (C=A and B=D
:If H=int(H
:Output(8-H,16,"=
:Output(B,A,"M
:getKey→K
:If K
:Output(B,A,"[one space]
:If K=24 and A>1
:A-1→A
:If K=26 and A<15
:A+1→A
:If K=25 and B>1
:B-1→B
:If K=34 and B<8
:B+1→B
:H+.1→H
:End
```

Annotations: "Repeat until hunger bar is full, [CLEAR] is pressed, or mouse reaches cheese"; "Add another notch to hunger bar if hunger has increased enough"; "Only erase mouse if it might move".

As you can see, the event loop will terminate on any of three conditions. First, if H is at least 8, the mouse's hunger bar is full, and the game is over. Because the outer and inner loop share this condition, the outer loop will also end if this is true. The second condition is K = 45, which means, as shown in figure 6.10 (an excerpt of figure 6.5), that the [CLEAR] key has been pressed.

This will end the outer loop as well once the program reaches the outer loop's `End`, because the outer loop has K = 45 as one of its `or`ed termination conditions. The third
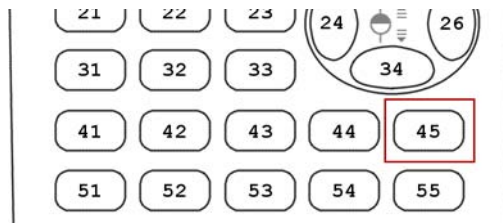
Figure 6.10   An excerpt of the keycode chart showing that the [CLEAR] key is code 45

condition is unique to the inner loop, namely C=A and B=D. This expression gets put inside parentheses so that it's evaluated as a whole, which means that the event loop will only end if both C = A, the X coordinates of the mouse and the cheese match, and B = D, the Y coordinates of the mouse and the cheese match. If these grouping parentheses were omitted, or instead the program ored A = C or B = D with the other termination conditions, the event loop would end if the mouse moved into the same column of the screen as the cheese, even if it wasn't also in the same row, occupying the same spot as the cheese.

Next is the somewhat mysterious conditional If H=int(H. I said previously that the valid range for the hunger variable H is from 0 to 8. The screen is conveniently 8 rows of characters tall, so H = 0 is the bottommost row, H = 1 is the second to bottom, H = 7 is the top row, and you never have to worry about drawing the hunger bar for H = 8, because that means the game is already over. These mappings of homescreen row to hunger value are shown in figure 6.11.
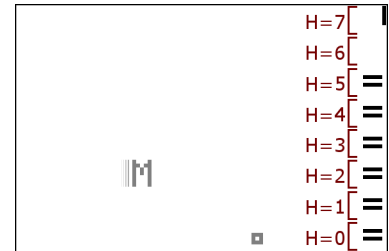
Figure 6.11   The Mouse and Cheese hunger bar and its relationship to the value of variable H

The conditional is there to ensure the program only draws the next notch upward on the hunger bar when the value of H reaches another integer. Because hunger ranges from 0 to 8, if the program added 1 to H every time it went through the event loop, the player would lose quickly, because it would take only 8 iterations of the event loop for H to reach 8. To give the player more of a sporting chance, the program only increases H by 0.1 each time through the event loop; it takes 80 individual 0.1s to make 8 (8/0.1 = 80 or 0.1 ∗ 80 = 8), so this gives the player 80 iterations of the event loop to reach the cheese. One thing you can play with, which I'll discuss later, is that if you change H+.1→H to H+.2→H, for example, the user will only have 40 iterations of the event loop to get to the cheese, and the game will be harder.

Anyway, because the program increases H by these small values, there's no point in displaying the highest notch in the hunger bar every time the calculator executes the event loop, because it will be redrawing the same equals sign 10 times before H reaches another integer. In addition, it would have to use Output(8-int(H),16,"=, because the calculator can't output at decimal locations on the screen, such as (3.4,16). If you don't believe me, try it: you'll get a Domain error. Therefore, the program only

displays a new notch on the hunger bar when H reaches an integer. Because int(3.4) = 3, int(5.8) = 5, and int(4) = 4, the only time int(H) = H will be true is when H is an integer already.

After displaying the hunger bar, the program executes what should by now be a familiar set of event-handling commands. It reads the current key being pressed into variable K, erases the mouse if K is nonzero (meaning that a key is being pressed), then handles the arrow keys. As usual, it checks the four arrow keys for the four major directions and also performs defensive boundary checks so that the mouse doesn't move off the edge of the screen. The body of the event loop ends with the update to the hunger value that I've been discussing.

ENDING THE OUTER LOOP AND ENDING THE GAME

The last piece of the outer loop runs after the inner loop ends. Recall that the inner loop can end as a result of three possible conditions:

- The hunger bar is full, meaning that H = 8.
- The user pressed [CLEAR], so K = 45.
- The mouse and the cheese are at the same coordinates, so A = C and B = D.

You only want to award the player another point and reset the mouse's hunger if the inner loop ended because of the third condition. As a reminder, the end of the outer loop and the end of the program look like this:

```
:If C=A and B=D
:Then
:0→H
:S+1→S
:End:End
:Output(3,1,"----------------
:Output(4,1," YOU ATE
:Output(4,10,S
:Output(5,1,"PIECES OF CHEESE
:Output(6,1,"----------------
:Pause
:ClrHome
```

As you can see, the program only resets H to 0 and increments S by 1 if C = A and B = D. If, on the other hand, K = 45 or H ≥ 8, which means that the inner event loop didn't end because the mouse reached the cheese, this conditional code doesn't execute.

The next line of code is a pair of `End` commands. The first `End` closes the `If/Then` statement, whereas the second `End` completes the outer `Repeat` loop. If K = 45 or H ≥ 8, then the outer loop will also end. If neither of those statements is true, which means the inner loop ended because A = C and B = D, then the outer `Repeat` loop will start again, generating a new piece of cheese and redrawing the game screen. If either K = 45 or H ≥ 8, the last seven lines of the program will run. These first output the number of pieces of cheese eaten during the game and then pause until the player hits [ENTER]. The game concludes by clearing the screen with `ClrHome`, and because the end of the program file is reached, the program ends.

6.3.3 *Tweaking the game*

Play with this program as much as you want, to try to change the way it works, make it have more features, or even adapt it into a whole new game of your own. The most important lesson of the Mouse and Cheese game is for you to feel comfortable about `getKey` and event loops and create your own games and programs, but a good starting point to fully grasp the concepts here is to first experiment with an existing program like CHEESE. Don't worry about breaking things; you can always start with a fresh copy of CHEESE if you irreparably break the original in your experimentation, or you could exercise and refine your debugging skills to try to track down what happened.

Among the many possible things you could do to expand the program with new features or adjust the existing gameplay, the following stand out:

- Use the lessons of section 6.2.3 to convert the Mouse and Cheese program to do eight-directional movement. You could lift the key-processing sections of code almost directly from program MOVE8D1 or MOVE8D2; you'd only need to adjust it to stop at column 15 instead of column 16, because the hunger bar is in the rightmost column of the homescreen now.
- Make the game have multiple pieces of cheese on the screen at the same time. How would you go about doing this? You could use (E,F) in addition to (C,D) for a second piece of cheese, and then you'd have to change both conditional checks for the mouse and the cheese being at the same coordinates (`A=C and B=D`) to also check for this second piece of cheese. What if you wanted the player to be able to specify the number of pieces of cheese?
- Make the hunger bar run out faster (or slower). There's a single value in the program that you need to adjust to make this happen. Hint: it's the line where the hunger variable H gets set to a larger value. What sort of values could you use? What if the values you use don't add up exactly to 8? Why does the program crash with an ERR:DOMAIN in that case? Or why does the hunger bar not get updated properly?
- Instead of adding more cheese, add a piece of poison or a mouse trap that the mouse must not touch. How could you implement this? What if even being in one of the squares next to the trap could harm the mouse? How about if you made it only slow down the mouse instead of ending the game?
- Make the cheese move randomly around to make it harder to get.
- Make the cheese move specifically away from the mouse at all times. How could you adjust this so that it's still always possible to get the cheese? Among other things, you'd have to add bounds checking for the cheese, too, so that it wouldn't move off the screen in trying to avoid the mouse.
- Make the screen wrap around, so that when the mouse reaches the right edge and goes right, it reappears at the left edge, or it goes to the top edge when it crosses the bottom edge.

The game as presented could be adapted to other themes besides a mouse and pieces of cheese; you could easily make it any sort of game where a character of some sort needs to collect (or avoid) some sort of object.

6.3.4   *Exercise: going further by moving the cheese*

As a final exercise for this chapter, you'll try to specifically implement one of the suggestions in the preceding list to see how modifying this program can be done. I'll pick the feature of making the cheese move randomly around the screen to make it harder for the mouse to reach it, as demonstrated in figure 6.12. As with all programs you write, you should briefly decide how you'll implement this feature before you dive into the code. Once you know what you want to change, you can then write the new pieces of code. In general,

Figure 6.12   By allowing the cheese to move randomly around the screen, the game starts to get more challenging.

you'll want to directly modify a program when you add new features, but for the sake of this example, I'll first show you the new pieces of code you're adding on their own and then the whole program again with the changes inserted.

The program still deals with one piece of cheese, so it will continue to use (C,D) for the coordinates of the cheese. A new piece of cheese should still appear when the mouse and the cheese reach the same place on the homescreen, so the inner loop can still end when (A=C and B=D). The main change that you need to make is to have the coordinates C and D get randomly modified. Because this must happen at the same time that the mouse is being moved around by the player, these coordinate updates must be in the inner loop. If you put them in the outer loop somewhere, the cheese wouldn't move around until after the player caught it, in which case the feature would be pointless. You know that you want to insert a piece of code somewhere in the inner loop. You should reexamine the original piece of pseudocode for the Mouse and Cheese game to see where you'll need to insert code:

```
:Repeat until hunger bar fills or [CLEAR] pressed
:[initialize cheese and hunger bar]
:Repeat until hunger bar fills or [CLEAR] pressed or mouse reaches cheese
:[move mouse and increase hunger]
:[move cheese]
:End
:[handle eating cheese]
:End
```

As you can see, the cheese must be moved around in the innermost loop at `[move cheese]`, where the program moves the mouse in response to keypresses. Because the inner loop continues to run whether or not the player presses keys, thanks to the non-blocking quality of `getKey` that I discussed earlier, you can make the cheese continuously move by updating its coordinates in the inner loop.

How can you make it move? I specified that it should move randomly, so you know that you need to use one of the commands that generate randomness, such as `rand` or `randInt`. One possible solution is to add a random integer between -1 and 1 to C and another random integer between -1 and 1 to D. This will make the cheese move in any of the possible eight directions from its starting point or stay in the same place if both random numbers are 0. That should work well, but you still need to make sure that the cheese doesn't go off the screen. Because it's moving around randomly, it could easily get to one of the edges and try to cross the edge, which would yield an exciting ERR:DOMAIN error.

You'll therefore need to do bounds checking. In the previous examples in this chapter, the programs perform bounds checking *before* updating the position coordinates. In the original four-direction `getKey` test program, the M only moved to the left (by subtracting 1 from A) if the user pressed the left-arrow key and the M was not already at the left edge:

```
:If K=24 and A≠1
:A-1→A
```

PRELIMINARY MOVEMENT SOLUTION

Your program could do that here, by temporarily putting the two random values into separate variables and then only updating C and D if the updates would keep the cheese on the screen. This particular implementation might look like as follows (remember, in this case the -1 uses the negative sign, the [(-)] key, not the subtraction sign, the [−] key):

```
:randInt(-1,1→E
:randInt(-1,1→F
:If E+C≥1 and E+C≤15
:C+E→C
:If F+D≥1 and F+D≤8
:D+F→D
```

Here the random updates are stored into E and F, and then the program checks if adding those values to C and D will still keep the cheese on the screen. If they will, then the program updates the coordinates with the random changes in E and F; otherwise they're discarded. But for argument's sake, I'll show you an alternative, more efficient method.

EFFICIENT MOVEMENT SOLUTION

Here you'll try a different approach, where C and D are blindly updated with random adjustments and fixed afterward if the random updates accidentally made the cheese go off the edge of the screen. This method saves the use of variables E and F, because you don't need to worry about temporarily storing the results of the two `randInt` commands. But it's a longer segment of code, so it might run more slowly than the previous option. In part 3 of this book, you'll learn optimization tricks that let you combine the best parts of both solutions into a fast and small piece of code that also doesn't require using extra variables. Before we get to that, here's the piece of code that fixes

C and D after potentially setting them to values outside the edges of the game space, as just described:

```
:C+randInt(-1,1→C
:D+randInt(-1,1→D
:If C<1:1→C
:If C>15:15→C
:If D<1:1→D
:If D>8:8→D
```

Conditional and conditionally executed command pairs combined on each line

To reduce vertical scrolling while editing the source code, you put the conditional `If` statements and their conditionally executed command (the stores to C and D) on the same line, separated by a colon, rather than hitting [ENTER] in between. As discussed, this is a stylistic choice, and it doesn't change the size or speed of the program. It also has two advantages. First, it saves vertical scrolling, because the set of four direction conditionals are four lines instead of the eight lines they'd take if the program had [ENTER]s instead of colons. This means that this chunk of code is six lines in total instead of ten, so you can see the entire thing on one screen without scrolling at all. Second, it's a way of thinking like a programmer: you know that each `If` statement and the statement that it controls (here, the stores to C and D) are closely related, and you get a visual hint from the two statements being together on the same line of the program.

You can see in the code that C and D are first updated with random values, which might put them outside the game area. The program then runs four separate checks to correct the value of C and D, one for each of the four edges of the area. If C is less than 1, which means the cheese is farther left than the left edge of the screen, the program fixes the problem by setting C equal to 1, the leftmost column of the screen. If C is greater than 15, which means it's either in the column reserved for the hunger bar or off the right edge of the screen, the program sets it to 15 to move it back to the rightmost useable column of the homescreen. The program performs similar checks on D.

REDRAWING THE CHEESE

You need to add one final item: erasing and redrawing the cheese. That's easy enough; you know you need to erase the cheese before updating its coordinates; otherwise the program will be erasing in the wrong place. It then needs to redraw the cheese after updating its coordinates. That leaves the following section of code:

```
:Output(D,C,"[one space]
:C+randInt(-1,1→C
:D+randInt(-1,1→D
:If C<1:1→C
:If C>15:15→C
:If D<1:1→D
:If D>8:8→D
:Output(D,C,"□
```

And that's all the code you need to erase the cheese, randomly move it, and redraw it. You can insert this into the inner loop of the CHEESE program, which we'll now call

CHEESE2. You can either put this new block of code before the `getKey`, after all the key-update conditionals, or even after the update to the hunger variable. For the sake of keeping related pieces of code together, we'll put it after all the keypresses have been processed but before the hunger variable H is updated. The final CHEESE2 program will look like the following listing.

Listing 6.5   The Mouse and Cheese game with randomly moving cheese

```
PROGRAM:CHEESE2
:7→A:4→B
:0→H:0→S
:Repeat H≥8 or K=45
:randInt(1,15→C
:randInt(1,8→D
:ClrHome
:Output(D,C,"□
:For(X,1,int(H
:Output(8-X,16,"=
:End
:Repeat H≥8 or K=45 or (C=A and B=D
:If H=int(H
:Output(8-H,16,"=
:Output(B,A,"M
:getKey→K
:If K
:Output(B,A,"[one space]
:If K=24 and A>1
:A-1→A
:If K=26 and A<15
:A+1→A
:If K=25 and B>1
:B-1→B
:If K=34 and B<8
:B+1→B
:Output(D,C,"[one space]
:C+randInt(-1,1→C
:D+randInt(-1,1→D
:If C<1:1→C
:If C>15:15→C
:If D<1:1→D
:If D>8:8→D
:Output(D,C,"□
:H+.1→H
:End
:If C=A and B=D
:Then
:0→H
:S+1→S
:End:End
:Output(3,1,"----------------
:Output(4,1," YOU ATE
:Output(4,10,S
:Output(5,1,"PIECES OF CHEESE
```

The newly inserted cheese-moving code

```
:Output(6,1,"----------------
:Pause
:ClrHome
```

If you'd like to experiment further with the idea of the randomly moving cheese, you could try the other section of random cheese movement code, where I showed you how to provide preemptive protection against the cheese going out of bounds. You could also try making the game slightly easier by running the random updates only sometimes or changing only one of the two coordinates in each inner loop.

With your newfound knowledge about getKey, event loops, and more interactive TI-BASIC games, have fun with this example! If you're stuck, try looking at it from a different perspective, or put it down and come back to it later. If you're having trouble seeing how it all fits together, try looking at it in smaller pieces: how the mouse's hunger, score, and coordinates get initialized; how the cheese is placed; how the inner getKey loop works; what happens when the mouse gets the cheese; and what happens when the game ends. In particular, look at why this program is written as two nested loops rather than a single big loop with conditional statements. When you write your own getKey-based programs, you'll want to be careful to put only the things that must be run repeatedly in your event loop, as discussed in section 6.1.

6.4 *getKey odds and ends*

getKey is a powerful function, but as I touched briefly on at several earlier places in this chapter, it has a handful of limitations. In the two following brief sections, we'll explore these limitations in more depth. I'll show you a programmatic solution to one of them, the lack of an ability to read the [2nd] and [ALPHA] modifier keys used along with one of the other keys. In chapter 12, you'll see a solution for another of the problems, the missing ability to read more than one key pressed at the same time.

6.4.1 *Quirks and limitations of getKey*

Although getKey can do many things, and it will enable you to make many new programs and games, there are a few obvious things that it can't do, two of which are technical limitations that can't be worked around. A third is a software decision on Texas Instruments' part that programmers have found ways to change.

First, getKey can't tell when the user presses the [ON] key on the calculator's keypad. If the user presses [ON], the program will stop with an ERR:BREAK error, even if you use getKey.

Second, getKey remembers only the last key that the user pressed. If your program executes the getKey command, then the user presses [1], [2], and [3], and finally your program runs getKey again, it will only return the keycode for the [3] key and will have forgotten about the [1] and the [2].

Third is that the getKey command can't tell when the user presses two keys at the same time. In many programs and games, you might like your users to be able to move a game character or a cursor diagonally across the screen; the obvious way to do this

would be holding down two of the arrow keys at the same time, such as [up] and [right] to move diagonally toward the top-right corner of the screen. Unfortunately, just as getKey can't remember two different keys pressed one after the other, it has no way of telling your program that two keys were pressed at the same time.

In sections 6.2.2 and 6.2.3, I discussed one solution, in which you use the number keys as an eight-directional pad. The second solution, which lets you read multiple arrow keys at the same time, uses a hybrid BASIC function from the library xLIB, written in z80 assembly. In part 3, I'll discuss these hBASIC libraries and work with the xLIB functions as provided by the Doors CS shell. With that special hybrid function, four special keycodes are used to represent up-left, up-right, down-left, and down-right.

6.4.2 *What about modifier keys?*

As discussed in section 6.2.2, getKey can't directly tell a program if a user pressed a modifier before another key. If you press [2nd] and then [3], or just [3], getKey will return exactly the same keycode for the [3]. Luckily, you can get around this, because [2nd] returns its own keycode. You can track modifier keys in your own program, setting a variable to 1 if [2nd] is pressed and a second variable to 1 if [ALPHA] is pressed. I'll demonstrate a simple program in listing 6.5 that implements this concept.

To read modifier keys, you look for first [2nd] (keycode 21) or [ALPHA] (keycode 31), then the key that you want to check for. You could maintain a variable for whether one of the modifier keys has been pressed. The sample program in the following listing will let you press [2nd][3], [ALPHA][3], or just [3]. The keycode for [3] is 94, so the Repeat loop runs until getKey returns 94.

Listing 6.6 getKey example for modifier keys

```
PROGRAM:MODKEYS
:0→A:0→S
:Repeat K=94
:getKey→K
:If K=21
:1-S→S
:If K=31
:1-A→A
:End
:If A=1 and S=1
:Then
:Disp "[2ND][ALPHA][3]
:Else
:If A=1
:Disp "[ALPHA][3]
:If S=1
:Disp "[2ND][3]
:End
:If A=0 and S=0
:Disp "[3]
```

A is I if the ALPHA modifier is currently set. S is I if the 2nd modifier is set.

Repeatedly look for keys until [3] is pressed

This turns 0 into I and I into 0 (try doing the math for S = 0 and S = I!)

Loop back to the Repeat if [3] hasn't been pressed yet

The code from here on runs only once, when the program ends

This program runs a `Repeat` loop, checking for keys, until the [3] key is pressed. In the meantime, it looks specifically for either the [2nd] or [ALPHA] key. The A variable is set to 1 whenever [ALPHA] is toggled on, and the S variable is set to 1 whenever [2nd] is toggled on. So why, for example, is the expression to set A with `1-A→A` instead of simple `1→A`? The latter assignment would work fine but wouldn't allow [ALPHA] to be toggled on and off. With `1-A→A`, when A = 0, 1 − 0 = 1→A, and when A = 1, 1 − 1 = 0→A. Therefore if you press [ALPHA] twice, it will be as if you didn't press it at all, just like when you press [ALPHA] twice on the calculator's homescreen.

This is a nifty trick, but in all likelihood there will be few places where you'll need to use it, unless you're planning on writing something like a text editor. In the next two chapters, you'll learn even more things that you can do with interactive programs that use `getKey`. For the first time, you'll have a way to manipulate every single pixel on the screen individually, which will allow you to create some complex educational programs and fun games indeed.

6.5   *Summary*

When you've finished reading and understanding this chapter, you should have a basic understanding of using the `getKey` function as a tool for interactive, fast-paced games. More broadly, you should have a passing familiarity with the idea of the event loop, used to concurrently check for input from the user in the form of keypresses while performing other tasks, such as increasing a hunger level and displaying a hunger bar with that value, running timers, or updating the positions of enemies or items. You should understand the basic differences between an asynchronous event and a synchronous event, because these are important concepts in programming at large. An indirectly related pair of terms that you might have picked up is blocking and non-blocking input, another distinction you'll find yourself using in your calculator programs and in your later programming languages, should you continue on to other platforms. But you may well be a bit overwhelmed after going through all this material. If you are, don't worry: take a break, relax, and come back to this tomorrow. It might take you a few reads to own some of the concepts, but once it all makes sense to you, you'll have programming and problem-solving intuition that will serve you well throughout your endeavors.

The next chapter will introduce the graphscreen and will teach you how to manipulate individual pixels in the LCD screen including setting, changing, and reading them. You'll find out how to draw text on the graphscreen. These plus your new event loop knowledge will enable you to create interactive games that also look a lot fancier than anything else you've made.

Pixels and the graphscreen

Say that with your newfound knowledge of `getKey` and event loops, you want to challenge yourself, and you decide to be ambitious. You want to move a mouselike cursor around your calculator's screen. You let your imagination run free, and you envision something like the left side of figure 7.1. Perhaps you keep brainstorming and want to be able to doodle on the calculator's LCD, so that you can draw things like the right side of figure 7.1. But you don't yet know how to manipulate individual pixels. What you do know is how to draw characters on the homescreen, but that won't help you write these programs.

The homescreen is a versatile canvas on which to paint your programs and games, and it's familiar even to most nonprogrammers who use a graphing calculator. But an array of 16 x 8 characters can be quite limiting for programmers, particularly for creating advanced games with good graphics and for rendering mathematical concepts
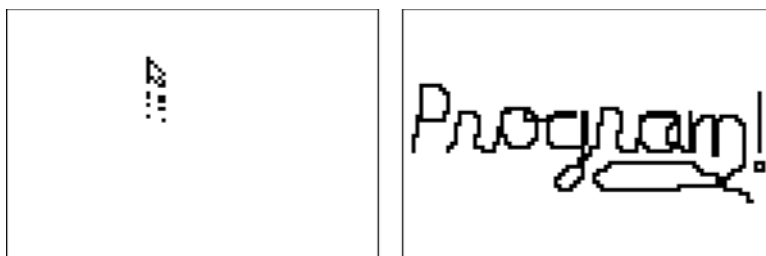
Figure 7.1   Moving a mouse around the graphscreen (left) and using a painting program to doodle (right)

such as annotated graphs, geometrical diagrams, and the like. Say that you want to create a game where you can draw a large, complex maze for the player to navigate or a fast-paced space-themed game. Consider a program in which you want to render 3D objects by drawing a series of lines or a game where you draw 3D corridors for your players to explore. Imagine trying to write a program that labels the minimums and maximums of a graphed function or draws its slopefield.

You now have the experience to think about designing the program structure and flow for some of these types of programs and games, but all of them are impossible to program well on the homescreen. If you tried to represent a graph, a 3D corridor, a puzzle game with expansive puzzles, or any of similar components of thousands of available TI-BASIC programs and games on the homescreen, you'd be disappointed.

 Luckily, you need not lose hope: your trusty TI-83+/84+ calculators have the answers you need in the form of controlling individual pixels in the LCD, the dots that make up any character or image. Although many computer programmers might scoff at having 96 x 64 pixels to work with, you'll soon find that after using the homescreen's 16 x 8 characters, this is more than enough expanse to build complex and feature-rich math and science programs and games that you'll want to play over and over. In this chapter, I'll tell you about the graphscreen, which, contrary to its name, is good for images, not just for working with graphs. I'll then show you one simple thing you can do with it: display text in a smaller font than the homescreen uses. I'll then move on to the trickier task of turning the individual pixels on and off and demonstrate two full example programs that use the technique, the same two programs illustrated in figure 7.1.

I'll begin by explaining what exactly the graphscreen is, how you'll be using it, and what sets it apart from the homescreen manipulation you've been doing so far.

7.1   *Introducing the graphscreen*

As we've previously explored while using input and output commands, the homescreen is 16 columns by 8 rows of characters. The homescreen can be manipulated with commands such as `Output`, `Disp`, `Input`, `Prompt`, and `ClrHome`. Programs that create graphics with letters and symbols can only get you so far, though; at some point

you want to be able to manipulate the individual pixels on the screen. The calculator's LCD is 96 x 64 pixels, small by any modern computing standards but luxurious compared with being constrained to what you can represent with characters. With access to turn individual pixels on and off, you get the ability to draw anything you can imagine. You can combine lines, points, large and small text, graphs, and shapes to render anything your program might need. Unfortunately, this freedom comes with a cost in speed: it's slower to manipulate pixels on the graphscreen than to draw characters on the homescreen.

To understand the difference between the graphscreen and the homescreen, imagine two separate canvases and an easel, as shown at each side of figure 7.2. The two canvases are the graphscreen and the homescreen, respectively, and the easel is the screen of the calculator. You can switch which canvas you're displaying on the easel, and every time you switch, the respective canvases contain the last thing you drew on them unless you've explicitly erased one. But you can't put both canvases on the easel at the same time and be able to overlap them, so you must choose to use one or the other at any given time. From a more technical point of view, the homescreen and the graphscreen are two buffers (areas) in the memory of the calculator, and depending on which is currently being displayed, the content of one or the other buffer is copied to the screen.

The LCD itself is 96 pixels wide and 64 pixels tall, but the OS doesn't let BASIC programs use the last column or the last row of the screen, so you really only have 95 columns and 63 rows with which to work, as shown at the left side of figure 7.2. The top-left corner has coordinates (0,0); the rows increase as you go down the LCD, and the columns increase as you move to the right. As with the homescreen, when you use commands that reference pixel coordinates, you put the row first and the column second, so
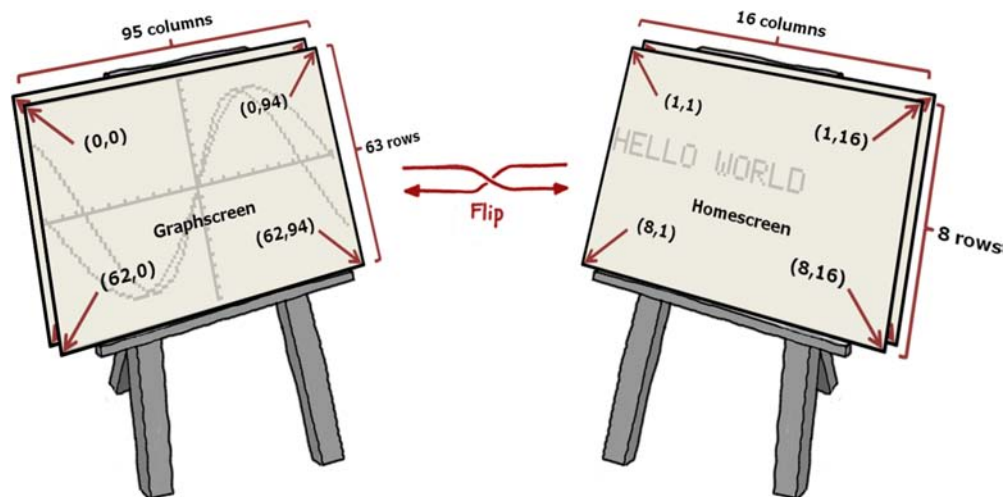
Figure 7.2 Size and coordinates of the graphscreen (left) versus the homescreen (right). You can't show the graphscreen and the homescreen at the same time; one must be tucked behind the other.
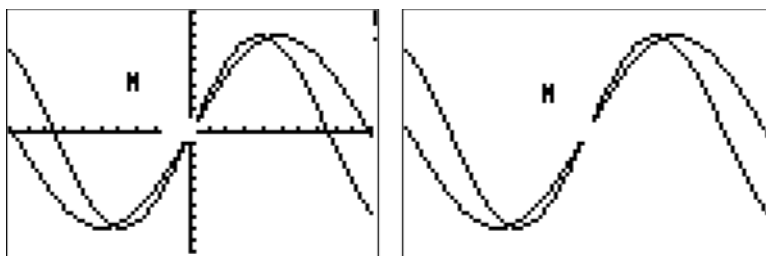
Figure 7.3   An example program, MOVETEXT, running with the graph axes turned on (left) and off (right)

(62,94) is the bottom-right corner, but (94,62) isn't valid, because it would be below the bottom edge of the screen.

Before I teach you your first commands to draw on the graphscreen, I'd like to introduce three commands that deal with preparing the graphscreen for use and cleaning up when you're finished with it. First, there's the `ClrDraw` command, which clears any drawings off the graphscreen just as `ClrHome` clears the homescreen. The `ClrDraw` command can be found in the DRAW tab of the DRAW menu, accessed with [2nd][PRGM]. If there are currently functions entered in the [Y=] menu, `ClrDraw` regraphs the equations after clearing the screen. In addition, if the axes were enabled, they're redrawn.

This brings us to the two other setup/cleanup commands, `AxesOff` and `AxesOn`. When you're editing a program, you can access both commands from the FORMAT menu, in [2nd][ZOOM]. If you're not editing a program, the FORMAT menu lets you turn the axes on and off, but if you're currently creating a program, the `AxesOn` and `AxesOff` options in the menu instead paste the `AxesOn` and `AxesOff` tokens into your program. As you'll see again when we work with graphing functions, the graph axes can be turned on and off, and these are the two commands we'll use. Figure 7.3 shows a program called MOVETEXT running while two functions entered as Y1 and Y2 are active, namely $8\sin(X/\pi)$ and $8\sin(1.3X/\pi)$. In the left screenshot, the axes are enabled; in the right screenshot, `AxesOff` has been used to remove the axes. In most of your programs that use the graphscreen, you'll want to turn off the axes before you begin to draw on the graphscreen with `AxesOff` and then turn them back on with `AxesOn` at the end of your program as a courtesy to the user.

Now you know what the graphscreen is, as well as three commands that will get you started using it effectively. We'll begin your exploration of commands to manipulate the pixels on the graphscreen with the `Text` command, the graphscreen equivalent of `Output`.

7.2   *Drawing text: first steps on the graphscreen*

At the beginning of this chapter, I introduced the graphscreen as a blank canvas in which you can individually manipulate each pixel, rather than being limited to putting

letters and numbers in an 8 x 16 array on the homescreen. One of the simplest ways to ease into using the graphscreen is to consider what it means to draw text on the graphscreen. Think back to the MOVECHAR program from section 6.2.1, which lets you use the arrow keys with `getKey` to move an M around the homescreen. Imagine that you've played with that program for a while and have decided it has limitations. You tell yourself that you want the M to move only a single pixel up, down, left, or right when you press an arrow key. But how do you do that? You know of no way to draw text to the screen with such precision. Luckily, the graphscreen and `Text` provide the solution you need.

I'll show you a program called MOVETEXT, which will let you get your feet wet about the differences between writing text on the homescreen with `Output` and on the graphscreen with `Text`. I'll then give you a more formal introduction to the `Text` command. Let's begin with MOVETEXT.

7.2.1 *Introducing Text: a MOVETEXT program*

In figure 7.3, I showed you a program called MOVETEXT working with the axes on and off. Figure 7.4 shows another view of the MOVE-TEXT program in action, with the functions that were graphed over the program in figure 7.3 deleted. The code for this particular program is shown in listing 7.1; it introduces a new command, `Text`, alongside `ClrDraw`, `AxesOn`, and `AxesOff`. This program is a slight modification of the MOVECHAR program from chapter 2, differing only in those four new commands and in the values used to detect when the M is about to run off the edge of the screen.

Figure 7.4   An annotated view of the MOVETEXT program in action with no graphed functions and no axes. The letter M moves one pixel for each arrow key and can therefore be in any of 92 columns and 58 rows (accounting for its width and height) instead of the 16 columns and 8 rows of the homescreen.

If you test `prgmMOVETEXT`, you'll see it does just what I described: it lets you move a letter M (slightly smaller than a homescreen M) around the screen, but when you press any arrow key, it moves by only a single pixel in that direction. If you have any graph functions active, you may see them behind the M symbol; in chapter 8 I'll teach you how to handle that in your program.

Listing 7.1   MOVETEXT, a modified MOVECHAR on the graphscreen

```
PROGRAM:MOVETEXT
:45→A:28→B
:AxesOff:ClrDraw
:Repeat K=45
:Text(B,A,"M
```

Repeat this loop until the [CLEAR] key is pressed, at which point the loop ends

```
:getKey→K
:If K
:Text(B,A,"[three spaces]
:If K=24 and A≠0
:A-1→A
:If K=26 and A≠91
:A+1→A
:If K=25 and B≠0
:B-1→B
:If K=34 and B≠57
:B+1→B
:End
:AxesOn
```

Text uses a variable-width font: an M is three pixels wide plus one padding column, a space is one pixel wide. Three spaces are needed to draw over the Ms three pixels of width.

Left-right movement and bounds checking. The M is four pixels total, and the columns range from 0 to 94, so the M can start in column 0 through column 9I safely.

Up-down movement. The M is 6 pixels tall, 5 plus I padding row, and the LCD's rows range from 0 to 62. The M can start in rows 0 through 57 safely.

When the Repeat loop ends, be polite and turn the axes back on before terminating.

TIP If the MOVETEXT program throws an ERR:INVALID DIM or ERR:DIM MISMATCH error when you try to run it, check that you haven't accidentally turned on Stat Plots. Quit to the homescreen, press the [Y=] key, and see if Plot1, Plot2, or Plot3 is white text on a black background; if so, Stat Plots is active. To fix this, move the cursor up to whichever plot is active and press [ENTER]. This will turn it back to disabled, shown as black text on a white background. Try rerunning the program.

The first change you'll notice from the MOVECHAR program is that although A and B are still X- and Y-coordinates, we're now initializing them to A = 45 and B = 28. These are roughly the center of the graphscreen, just as A = 8 and B = 4 are roughly the center of the homescreen. Why not A = 47 and B = 31, the middle row between 0 and 62 (62 / 2 = 31) and the middle column between 0 and 94 (94 / 2 = 47)? Because when you draw text on the graphscreen, you tell the calculator the coordinates of the top-left corner where you want to put the text; the calculator will draw what you specify below and to the right of that point.

The character that we want to center on the graphscreen is effectively 6 pixels tall and 3 pixels wide (plus one padding column of width), so we'll say that its top-left corner is 3 pixels above and 2 pixels to the left of its center. Because we said the center of the screen was (31,47), we can place the M character at the center, and its top-left corner will be at (31 − 3,47 − 2), which is (28,45) and which is why we initialized A to 45 and B to 28.

Other changes include switching the usual space used to erase a character to three spaces, because each space is only 1 pixel wide. This provides the three columns of white

Help! There are graphs on my text!
Not to worry; in the next chapter, you'll learn how to turn some or all of the graph functions on or off within your programs to avoid this problem. The `FnOff` and `FnOn` commands will come to your rescue. For now, know that `FnOff` by itself will temporarily disable all Y= functions, just as if you had pressed [Y=] and turned off each of the functions by hand.

pixels to write over the M. A final difference is the coordinates for bounds checking, as noted in the code's annotations. With this initial immersion in a `Text`-based program, we'll move to a more complete description of `Text` and its features.

7.2.2 *The Text command*

Numerical differences aside, the most glaring new feature of `prgmMOVETEXT` is the `Text` command. It can be found under [2nd][PRGM][0], at the bottom of the DRAW menu. It takes at least three arguments: the row or Y-coordinate of the top left of the text to be displayed, the column or X-coordinate of the top left of the text to be displayed, and then either a string or a number. The following `Text` commands are all valid:

```
:Text(1,1,"Hello World
:Text(7,1,1337
:56→A
:Text(13,1,A
:Text(19,1,"ABCDEFGHIJKLMNOPQRSTUVW
:Text(25,1,"abcdefghijklmnopqrstuvwx
:Text(31,1,"()+-*/[]{},.¯:?
```

> **TIP** To access lowercase letters, press [ALPHA] twice; [2nd][ALPHA][ALPHA] locks lowercase alpha mode. If pressing [ALPHA] twice doesn't do anything, you need to enable lowercase letters. Shells such as Doors CS (see appendix C) will unlock lowercase mode for you.

If you turn off the axes and all functions, clear the screen, and run the seven lines just shown, you'll get the output shown in figure 7.5. One of the biggest differences between displaying text on the graphscreen and the homescreen is that although the homescreen uses fixed-width font, the graphscreen uses a smaller variable-width font. As you may have noticed, every character on the homescreen—whether it's a space, a letter, a number, or a symbol—is the same width. On the graphscreen, different characters have different widths. On the positive side, this gives you more flexibility with what you can do with graphscreen text; on the negative side, you need to be more careful if you want items of text to line up with each other and when erasing using spaces.

Almost every graphscreen character is 6 pixels high, where the top row of the character is blank, and has one "extra" column at its right edge that's blank. This blank column is used to separate adjacent letters, as you can see in figure 7.6. Although the capital A character looks like it should be only five rows tall and three columns high, the extra row at the top and the extra column at the right make it six rows tall and four columns wide.
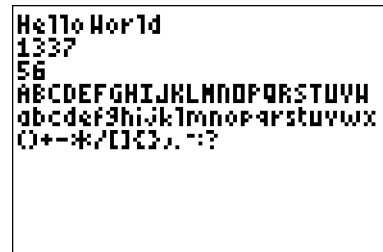
Figure 7.5 Six sample `Text` lines on the graphscreen. Notice that you can fit more text on the screen than with `Output` and that the font is variable width. Characters that don't fit at the right edges disappear.

A lowercase i is two columns wide and six columns tall for the same reason. The main exception is the space character, which is precisely 1 pixel wide.

Text written with the `Text` command doesn't wrap around to the next line, which means when it reaches the right edge of the screen, it stops. Any letters that don't fit onto the screen aren't displayed. Even if a letter would partially fit on the screen, it won't be shown. You could also think of the text as flowing invisibly off the right edge of the screen, if that's easier to imagine. If you want to continue what you were writing onto a second line, you need to use more `Text` commands. In addition, if you try to draw a line of text farther down the screen than Y = 57, or starting at X = 94 or farther right, the command will throw an ERR:DOMAIN error. This means that at least one of the coordinates for the text is invalid. One particularly common mistake that can cause this is reversing the Y and X arguments to the text, for example `Text(80,5,"HI` instead of `Text(5,80,"HI`.

Text drawn with the `Text` command erases whatever was underneath in a 6-pixel-tall swath of the screen. In addition, strings ending in everything except a space erase one extra pixel column to the right of the text because of the one pixel of padding next to every character (see figure 7.6 for a more explicit illustration of this). Although this can be annoying, it's useful for removing a string of text from the screen without having to `ClrDraw` the whole screen. We used this trick in the MOVETEXT program previously: instead of clearing the whole screen every time we wanted to erase the M, we wrote three spaces over it, which erased it to white.
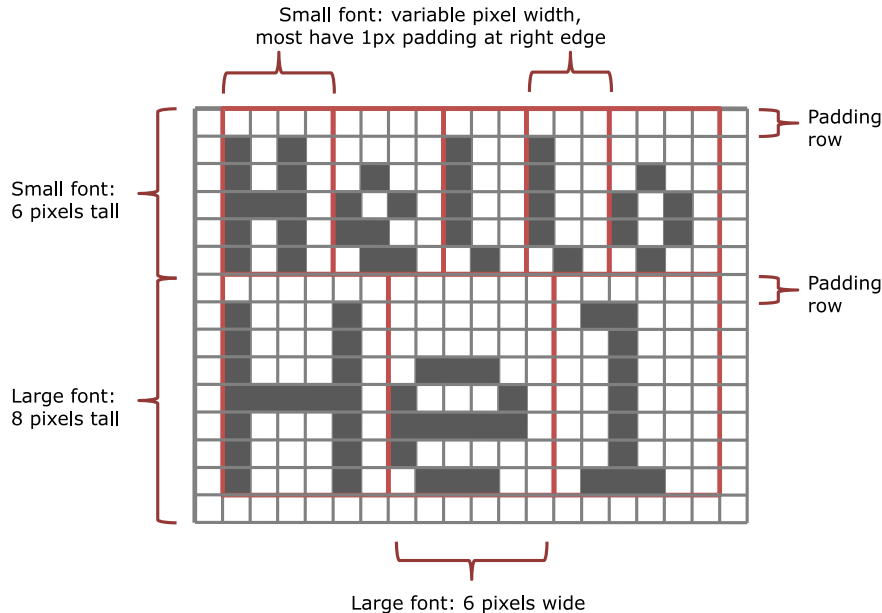
Figure 7.6   Relative size of small-font and large-font characters

At the beginning of this section, I mentioned that the `Text` command takes *at least* three arguments, but thus far we've only seen `Text` used with *exactly* three arguments: the (Y,X) coordinates plus a string or number to display. Multiple arguments are used to concatenate, or join, multiple items such as numbers and strings together. For example, you could use

```
:Text(14,1,"PI IS ROUGHLY ",22/7
```

This will display the string "PI IS ROUGHLY 3.1428571" on the graphscreen. You can add as many arguments to the end, both strings and numbers, as long as they all fit on one line of the graphscreen:

```
:Text(14,1,"PI IS ABOUT ",22/7," OR ",3.1416
```

`Text` has one more useful quirk: the ability to put large homescreen-style text on the graphscreen at arbitrary pixel coordinates. Figure 7.6 shows a comparison of the small and large fonts that the calculator uses. For the large font, you insert a −1 as the first argument to `Text`, before the Y- and X-coordinates of the text. This will make it use the large font, with 8-pixel by 6-pixel characters, rather than the 6-pixel-tall small font, for example:

```
:Text(⁻1,14,1,"Hello World
:Text(⁻1,40,65,"42: ",42
```

You can combine large and small font on the graphscreen for attractive interfaces and games, and in the next chapter, I'll show you how to combine `Text` with lines and shapes.

Now that you've seen the `Text` command on the graphscreen, which lets you turn many pixels on or off at once, I'll show you the fine-grained commands for turning individual pixels on the graphscreen on and off.

7.3 *Playing with pixels*

The TI-83+/TI-84+ LCD is made up of 6,144 pixels, tiny points that can be either white or black. Clever assembly-language programmers have managed to turn the pixels on and off fast enough to make grayscale, but for TI-BASIC purposes, your programs will be setting pixels to either black or white. I'll introduce four commands that will let you directly manipulate pixels: three to change the color of individual pixels and one to determine the current color of a specified pixel. In this section, we'll begin with the four pixel commands, `Pxl-On`, `Pxl-Off`, `Pxl-Change`, and `pxl-Test`. You'll see how to draw a simple mouse cursor with these commands, then how, in two steps, to make it a moveable mouse.

The first step to combining `Pxl-` commands with your existing knowledge is understanding what `Pxl-` commands exist and how to use them.

7.3.1 *Pixel commands*

The four pixel-based drawing commands are all quite straightforward. They all take precisely two arguments, and those two arguments are all (row, column), or if you prefer,
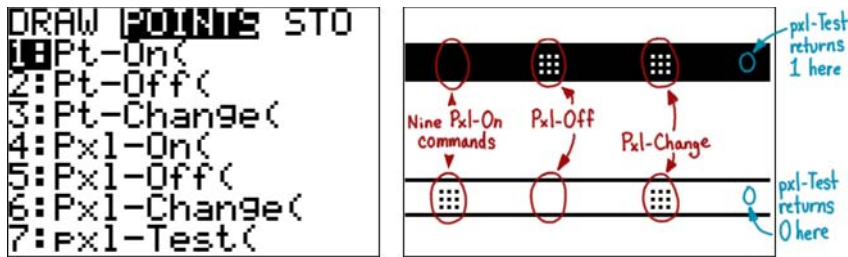
Figure 7.7   The four pixel-based commands we'll be using from the DRAW menu's POINTS tab (left) and their effects on black-and-white backgrounds (right)

(y, x). They're all found in the POINTS tab of the DRAW menu, [2nd][PRGM][▶], demonstrated at the left side of figure 7.7. Options [4], [5], and [6], the Pxl-On, Pxl-Off, and Pxl-Change commands, all modify the state of a pixel, also shown at left in figure 7.7. The pxl-Test command is the odd one out, returning a 1 if the specified coordinates contain a black pixel and a 0 for a white pixel. The right side of figure 7.7 shows the results from using blocks of nine Pxl-On, Pxl-Off, and Pxl-Change commands on black and white backgrounds, as well as the output value of pxl-Test on black and white pixels. As you can see, black pixels are considered on (or 1) and white pixels are off (or 0).

The following snippet of code demonstrates each of the four commands in use:

```
:Pxl-On(1,1
:Pxl-Off(42,80
:Pxl-Change(13,37
:If pxl-Test(42,80
:Then
:Disp "(42,80) BLCK PXL
:Else
:Disp "(42,80) WHT PXL
:End
```

This would turn a pixel near the top-left black, one near the lower-right white, and switch a pixel near the middle of the top edge from black to white or white to black. Because the Pxl-Off command turns the pixel at row 42, column 80 to white (off), which is 0 or false, the If/Then/Else/End condition displays "(42,80) WHT PXL." Recall that the If line is equivalent to the following and that pxl-Test returns 1 for black pixels and 0 for white:

```
:If 1=pxl-Test(42,80
```

With the basics of the commands in hand, let's do something fun with them: draw a mouse-style cursor.
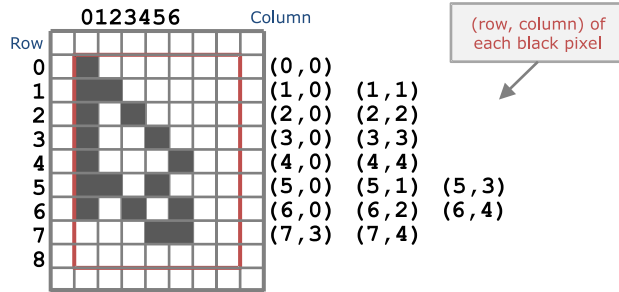
Figure 7.8   A "sprite," or set of pixels, used to create a mouse cursor. The right side of the figure shows the (row, column) coordinates of each black pixel.

7.3.2   *Drawing a cursor*

By the end of the next section, I want you to have a cursor that you can move around your calculator's screen with the arrow keys. A program like MOVETEXT already gives you a good framework to work with, because it lets you move an M around and already handles turning the axes on and off, clearing the screen when appropriate, and working with keys. But the Text commands to draw and erase the M will no longer do the trick. For starters, you need to figure out how you'll draw a mouse cursor. Figure 7.8 shows a simple example sketched in black and white pixels, spanning eight rows and five columns. Because you know that you want to draw this cursor with Pxl-Off/On/ Change commands (because that's the only way you know to manipulate individual pixels), you need to start by figuring out the coordinates of each pixel.

You always label graphscreen pixels with the lowest row and column numbers at the top left, so the tip of the cursor is (0,0); the full list of pixel coordinates for this mouse relative to (0,0) is shown at right in figure 7.8. If you rendered this entirely with Pxl-On commands, it might look something like the left side of the code in the following listing.

Listing 7.2   Two versions of a cursor-drawing subprogram, ZCURSORA/ZCURSORB

```
PROGRAM:ZCURSORA          PROGRAM:ZCURSORB
:Pxl-On(0,0               :Pxl-Change(B,A
:Pxl-On(1,0               :Pxl-Change(B+1,A
:Pxl-On(1,1               :Pxl-Change(B+1,A+1
:Pxl-On(2,0               :Pxl-Change(B+2,A
:Pxl-On(2,2               :Pxl-Change(B+2,A+2
:Pxl-On(3,0               :Pxl-Change(B+3,A
:Pxl-On(3,3               :Pxl-Change(B+3,A+3
:Pxl-On(4,0               :Pxl-Change(B+4,A
:Pxl-On(4,4               :Pxl-Change(B+4,A+4
:Pxl-On(5,0               :Pxl-Change(B+5,A
:Pxl-On(5,1               :Pxl-Change(B+5,A+1
:Pxl-On(5,3               :Pxl-Change(B+5,A+3
:Pxl-On(6,0               :Pxl-Change(B+6,A
:Pxl-On(6,2               :Pxl-Change(B+6,A+2
:Pxl-On(6,4               :Pxl-Change(B+6,A+4
:Pxl-On(7,3               :Pxl-Change(B+7,A+3
:Pxl-On(7,4               :Pxl-Change(B+7,A+4
```

But this only allows you to draw the cursor at the top-left corner of the LCD and gives you no way to erase it again without clearing the screen, which is often something you'll want to avoid if you have many things on the screen and want to erase only one. The program at the right side of listing 7.2, ZCURSORB, solves both of these problems. By using `Pxl-Change` commands, running the program once will draw a mouse, changing all the white pixels to black in the shape of the cursor. If you run the program again, `Pxl-Change` will change all those black pixels back to white, erasing the cursor. In addition, the insertion of the A (X-coordinate or column) and B (Y-coordinate or row) variables, here used as offsets, means that instead of being drawn relative to (0,0), the cursor is now drawn relative to (B,A). If you change A and/or B, the cursor will be drawn at a different location.

Now that you have a small program that can both draw and, if called again, erase a mouse cursor at some coordinates (B,A), you can combine this with fragments from MOVETEXT. In the final subsection of this section, I'll challenge you to put this together yourself and then show you how it's done.

7.3.3 *Exercise: the moveable mouse cursor*

For this exercise, you'll create a program to move a mouselike cursor around the graphscreen, as in figure 7.9. You'll be modifying the MOVETEXT program from listing 7.1 into a new program called CURSOR; if you want to copy and paste the contents of MOVETEXT, recall the lessons with the `Rcl` function from section 2.1.1. For the actual drawing and erasing of the mouse cursor, use the ZCURSORB program as a subprogram; if you need to review subprograms, glance back at the end of chapter 4. Because it's slower to call `prgmZCURSORB` than to use a single `Text` command, try making the mouse move 2 pixels at a time. As a hint, your program will need to have the two `Text` commands in `prgmMOVETEXT` changed, as well as the numbers used for bounds checking and the numbers for updating A and B, but the structure will remain more or less the same. The only structural change should be putting the `getKey` in a tight `Repeat` loop, because running ZCURSORB repeatedly won't repeatedly draw the M over itself with no visual change, as running `Text(B,A,"M` did. Instead, it will repeatedly draw and erase the mouse in place, an unwanted side effect.

Figure 7.9   The results of the CURSOR program you'll write in section 7.3.3, a mouse cursor that moves

Once you have an idea of what your program will look like, or better yet, have written and tested it, read on for my solution.

THE MOVEABLE CURSOR: SOLUTION

My proposed solution for CURSOR is shown in listing 7.3, and it requires that you also have ZCURSORB from listing 7.2 on your calculator.

```
PROGRAM:CURSOR
:44→A:28→B
:AxesOff:ClrDraw
:Repeat K=45
:prgmZCURSORB
:Repeat K
:getKey→K
:End
:prgmZCURSORB
:If K=24 and A≠0
:A-2→A
:If K=26 and A<88
:A+2→A
:If K=25 and B≠0
:B-2→B
:If K=34 and B<54
:B+2→B
:End
:AxesOn
```

The best way to explain a program like this would be to step through it line by line, explaining each chunk.

```
:44→A:28→B
:AxesOff:ClrDraw
```

Initialize A (the X-coordinate) and B (the Y-coordinate) to roughly the center of the screen; then turn off the axes and clear the screen.

Multiple commands on the same line

I introduced putting two commands together on the same line in chapter 6. When you insert a colon between two or more commands, you can put them next to each other without adding a line return (new line) between each command. The only caveat is that you can no longer omit ending parentheses and quotes to save space. The following four commands fit on two lines. This technique is used to save vertical scrolling in long programs.

```
:45→A:28→B
:AxesOff:ClrDraw
```

```
:Repeat K=45
```

Repeat the main loop of this program until the [CLEAR] key is pressed.

```
:prgmZCURSORB
```

Call `prgmZCURSORB` to draw the cursor on the screen at row B, column A.

```
:Repeat K
:getKey→K
:End
```

With the cursor on the screen, run a tight `Repeat` loop, which repeats until K is not 0 (there's an implicit =1 or ≠0 when you have a variable by itself as a conditional). Because `Repeat` loops are guaranteed to run at least once, the first run through the loop with no key pressed will set K to 0, allowing the loop to continue without needing to set K to 0 before the beginning of the loop.

```
:prgmZCURSORB
```

When the `Repeat K` loop ends, a key has been pressed; erase the cursor by running `prgmZCURSORB` again:

```
:If K=24 and A≠0
:A-2→A
:If K=26 and A<88
:A+2→A
:If K=25 and B≠0
:B-2→B
:If K=34 and B<54
:B+2→B
```

Perform bounds checking while updating A and B. The left- and right-arrow keys move the X-coordinate A as far left as 0 or as far right as 88. Technically, because the cursor is 6 pixels wide, it could begin in column 89 and span only as far as column 94. Because the program moves the cursor 2 pixels at a time, it would jump from 88 to 90, at which point the program would throw an ERR:DOMAIN when it tried to change a pixel in column 95, the 96th column. This code also moves the cursor as high as row 0 or as low as row 54. A and B are initialized to even numbers, all updates are in increments of 2, and bounds are checked with even numbers, so all coordinates in this program are always even numbers.

```
:End
:AxesOn
```

End the outer `Repeat` loop, and if it terminates, turn the axes back on (which clears the graphscreen) and reach the end of the program to conclude it.

As with any program you write or that I show you, if there's anything you don't understand, try it out on your calculator, examine and tweak the code, and even add debug statements as taught in chapter 5. If you're still vague on any of the concepts, read on to the next section, where I'll show you a program that lets you doodle on the screen.

7.4   *A painting program*

For the final full program of this chapter, I'll expand the concepts of the CURSOR program to let you doodle on the screen. Instead of moving a mouse cursor around the screen that erases after itself as it goes, you'll be moving a single flashing pixel around the screen with the arrow keys. But pressing [ENTER] in this program is like putting a pencil or pen down on a piece of paper: now the flashing pixel will draw a line behind it as it moves. Press [ENTER] again to lift up the writing implement, and when you've
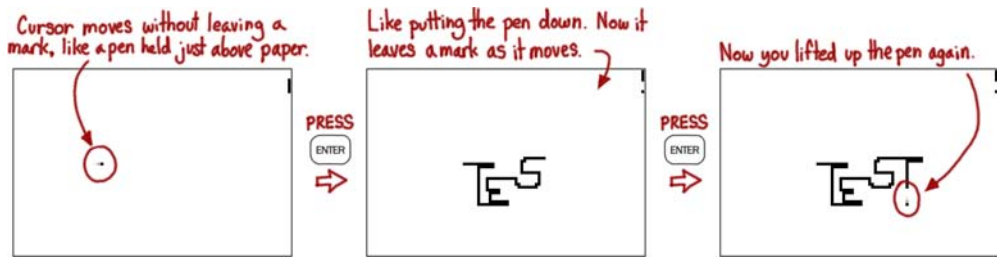
Figure 7.10   A demonstration of `prgmPAINT` in action. Moving the pointer around at far left without drawing and then pressing [ENTER] to hold the pen against the "paper," as in the middle screenshot. Pressing [ENTER] again to lift the pen again, as in the far-right screenshot.

finished, press [CLEAR] to quit. To start, take a look at figure 7.10, which from left to right shows a sample session using the program.

I'll begin by showing you the code for this PAINT program and explain a bit about how it works. At this point, the flow of such programs should be gradually getting clearer to you, so I'll pick out specific features to highlight rather than tracing it line by line.

CODING A PAINTING PROGRAM

The source code for `prgmPAINT` is shown in listing 7.4. It follows the structure of many programs you've examined in the past two chapters that are based on moving something around with `getKey`. Like the CURSOR program, it uses two nested `Repeat` loops, an outer one to handle keypresses until [CLEAR] is pressed and an inner `Repeat` loop to flash a pixel on and off while it waits for any keypress. It does graph setup and variable initialization before the outer loop and cleans up after the end of the outer loop.

Listing 7.4   A painting program, `prgmPAINT`, using `Pxl` commands and `getKey`

```
PROGRAM:PAINT
:AxesOff:ClrDraw
:48→A:32→B
:0→P
:Repeat K=45
:Repeat K
:Pxl-Change(B,A
:getKey→K
:Pxl-Change(B,A
:End
:If K=24 and A>0
:A-1→A
:If K=26 and A<94
:A+1→A
:If K=25 and B>0
:B-1→B
:If K=34 and B<62
:B+1→B
:If K=105
:1-P→P
```

Turn off the axes and clear the graphscreen

Initialize the (B,A) = (row, column) to about the center of the screen

Initialize the P variable, indicating whether the "pen" is touching the "paper," to 0, or false

Flash a pixel to show the currently selected pixel while waiting for a keypress

Toggle the "pen" touching (P = I) or not touching (P = 0) the "paper"

```
:If P
:Pxl-On(B,A
:End
:AxesOn
```

**If the "pen" is touching the
"paper," turn this pixel on**

Like the MOVETEXT and CURSOR programs, this program uses variables A and B to store the X- (column) and Y- (row) coordinates of the pixel currently selected. It also uses variable P to store the current state of the virtual "pen," whether it's touching the paper (P = 1) or not (P = 0). It controls whether a `Pxl-On` command runs right after any movement command, so that when P = 1, the program turns on the pixel each time it moves to a new pixel:

```
:If P
:Pxl-On(B,A
```

The [ENTER] key toggles P between 0 and 1 using the same math trick as `prgmMODKEYS` in listing 6.5 . When the [ENTER] key is pressed, P is set to 1 – P.

```
:If K=105
:1-P→P
```

If P is 0, 1 – P = 1 – 0 = 1, so P switches to 1. If P is 1, 1 – P = 1 – 1 = 0, so P switches to 0.

The final interesting bit of code is the inner loop that displays the swiftly flashing cursor while the program waits for the user to press a key:

```
:Repeat K
:Pxl-Change(B,A
:getKey→K
:Pxl-Change(B,A
:End
```

This code is designed to leave the pixel as it originally was when it ends. As figure 7.7 and the CURSOR program showed, applying `Pxl-Change` twice to the same pixel changes it back to whatever it started as. The first `Pxl-Change` either changes a white pixel to black or a black pixel to white, and the second `Pxl-Change` switches it back. Because there are two `Pxl-Change` commands inside the `Repeat` loop, no matter how many times the loop runs, an even number of `Pxl-Change`s occur, and the pixel at (row,column) = (B,A) will always be the same after the loop ends as before the loop began. While inside the loop, the pixel will flash on and off fast to indicate that that's the currently selected pixel.

As a final note, let me clarify the logic on the `Pxl-On` command that runs when P = 1 is placed at the end of the outer `Repeat` loop. As always, we want our programs to be as small and fast as possible. Therefore, we only want to run the `Pxl-On` command that draws on the virtual paper when either the "pen" is toggled to P = 1 (touching the paper) or the "pen" moves to a new location due to the user pressing the arrow keys. Both of these conditions are handled when the end of the outer `Repeat` loop runs, because the inner loop ends when K ≠ 0, or a key has been pressed. Thus, the end of the outer `Repeat` loop is a logical place for the `Pxl-On` command.

You've now seen programs that let you draw pixels and text on the graphscreen, as well as clear it and turn the axes on and off. You'll soon learn more that you can do

with points, lines, and graphs on the graphscreen, and you should, as in all previous chapters, play with the new skills you've learned in this chapter on your own. Only through practice can you discover the holes in your knowledge and start to appreciate the fun of designing and creating your own programs.

7.5 Summary

This chapter covered your first steps of programming with the graphscreen. It introduced working with individual pixels and how you can use the graphscreen in your own programs and games. You saw a few example programs, including a program to move a mouselike cursor around the screen and a doodling program, both of which combined the event loop lessons of the previous chapter with the precision and control of the graphscreen.

The next chapter will continue your exploration of the graphscreen with a separate coordinate system, one based on points and Cartesian coordinates instead of pixel coordinates. You'll learn to draw graphs, lines, circles, and other shapes and how to combine these with the lessons of chapter 7.

Graphs, shapes, and points

8

This chapter covers

- Drawing lines, shapes, and points
- Creating and annotating graphs from programs
- Techniques for making graphical programs and games

For the first six chapters, you worked with the homescreen, an 8-row, 16-column matrix of characters. In chapter 7, you learned about manipulating the individual pixels of the graphscreen, turning them on and off and checking if certain pixels were on. If you wanted to draw a line, a polygon, or a circle, render a graph, or save a full-screen picture, you'd be stuck with many, many pixel commands. Figure 8.1 shows some of the things you might want to draw in programs.

Once again, TI-BASIC comes to your rescue. There's a second coordinate system on the graphscreen, the one used whenever you plot a graph, a system overlaid on the pixel coordinates you learned in chapter 7. In this chapter, you'll learn all about this coordinate system, called Cartesian coordinates, and the commands TI-BASIC provides to draw lines, circles, points, and graphs and even store and recall whole screen images. Although there's some gentle use of trigonometry and geometry concepts, don't worry if you haven't seen them before: I'll explain anything that your math classes may not have yet introduced.
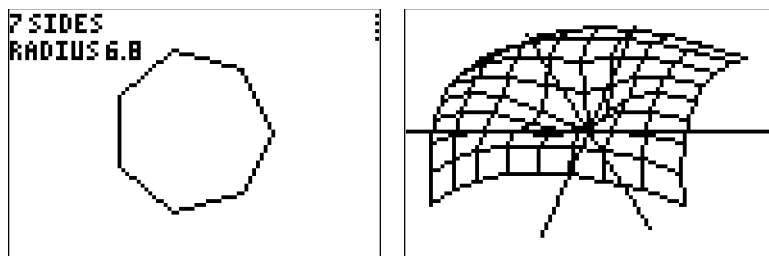
Figure 8.1   Two examples of what you can do with this chapter's lessons. The left screenshot is from the program POLYGON, which section 8.4 will show. The right screenshot is from a TI-BASIC 3D graphing program called Graph3D v4.0.

We'll start with a comparison of the pixel and point coordinate systems, then examine how and why you can manipulate what part of the Cartesian coordinate plane is shown onscreen at any given time. Because your graphing calculator was originally designed for graphing, I'll explain how your programs can generate and annotate graphs, and then I'll move on to drawing with shapes, lines, and points. Finally, we'll explore the picture commands that let you store and recall whole screenfuls of pixels.

We'll begin by discussing the Cartesian coordinate system and seeing some of the similarities and differences between it and the pixel coordinates from the previous chapter.

8.1   *Another coordinate system: points versus pixels*

If you display anything on the screen, you need to be able to turn the individual pixels in the LCD on and off. To do that, you need a way to address or distinguish the pixels so that you can properly specify which ones you want to turn on or off. In the previous chapter, that addressing method was the pixel coordinate system where the top-left pixel in the LCD is (0,0), and the bottom-right one is (62,94). That system always defines the coordinates of each pixel and doesn't change. But on a device used for math where you might want to draw graphs, this pixel coordinate system isn't quite good enough on its own.

The key concept of this chapter is that the graphscreen has a second coordinate system, one that can map different coordinates to the same pixels. Imagine a two-dimensional plane (flat sheet) that extends infinitely up, down, left, and right, as illustrated in figure 8.2.

When you display part of that plane on your calculator's LCD, it's as if you set the calculator down on top of the plane and looked through the LCD at the underlying plane, including whatever graph might be sketched on that plane. Your calculator would be a literal window through which you could see a graph. You could slide your calculator around on the plane to see different parts of the graph through the LCD "window." You could lift it away from the plane to see more of the plane (thus zooming out) or move it closer to zoom in.

And lo and behold, as you might expect because I'm making this analogy, the Cartesian or point coordinate system in your TI-83+/84+ is just like this. As shown in
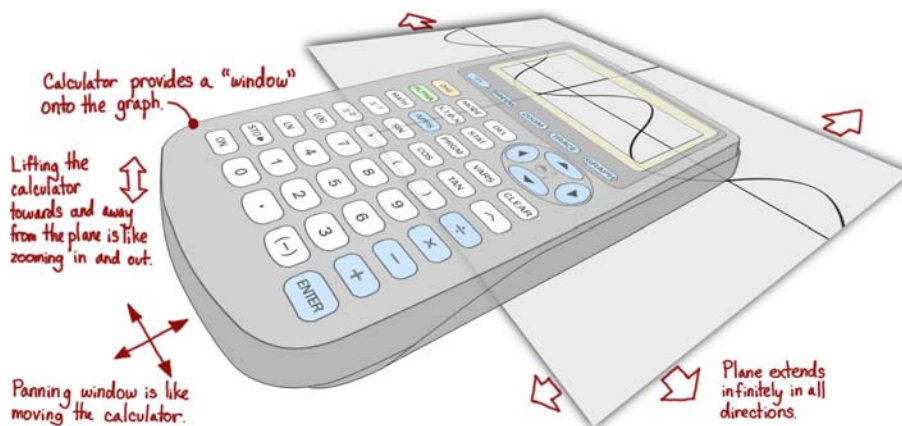
Figure 8.2   The analogy presented in the text about the 2D coordinate plane, which extends infinitely in every direction. Your calculator's screen shows a small portion of that plane at any time, including any axes and graph that may be drawn on the plane; panning and zooming the window is like moving the calculator to see different parts of the plane.

figure 8.3, you specify which part of the Cartesian coordinate system is mapped to your calculator's display with several window variables. The left-to-right axis is the X axis; as you move farther to the right, the X-coordinate increases. The bottom-to-top axis is the Y axis; the Y-coordinate increases as you move up. In the pixel coordinate system, column 0 always refers to the leftmost column of pixels, whereas in the point coordinate system, X = Xmin is the leftmost column, X = Xmax is the rightmost, and Y = Ymin and Y = Ymax are the bottom and top rows.

Table 8.1 shows more about the six variables that govern the size of the point coordinate window. When you're not in a program, you can adjust these in the WINDOW menu, accessed with the [WINDOW] key. If you're writing a program, you can find them in [VARS][1] (Window Variables). They're like the numeric variables A, M, Z, and their 24 siblings (numeric variables can be A–Z and θ, a total of 27 variables) in that you can store to them or read their values back.
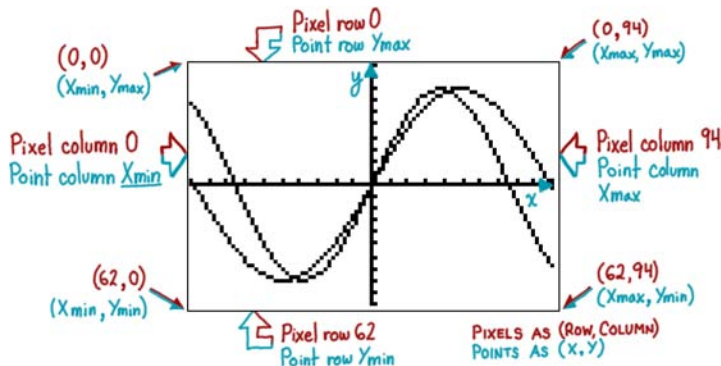
Figure 8.3   Comparing the pixel and point coordinate systems. The pixel coordinates exist only on the screen, whereas the point coordinates map a certain area of the Cartesian plane onto the LCD, but the plane still extends infinitely offscreen in every direction.

In addition, modifying some of the window variables automatically updates others to keep the set of six consistent with each other. If you change ΔX or ΔY, the calculator changes `Xmax` or `Ymax` to `Xmin+`ΔX or `Ymin+`ΔY, respectively. Conversely, if you change `Xmax`, `Xmin`, `Ymax`, or `Ymin`, then ΔX and/or ΔY are automatically adjusted. You may notice `Xscl` and `Yscl` in the Window Variables menu as well, but TI-BASIC programs don't often use these.

Table 8.1   The six useful variables that govern the size of the coordinate window. You can see how four of these six (`Xmin`, `Xmax`, `Ymin`, `Ymax`) relate to their pixel cousins in figure 8.2.

Variable	Function
`Xmin`	The left edge of the screen, or the lowest onscreen X value
`Xmax`	The right edge of the screen, or the highest onscreen X value
ΔX	The difference between horizontally adjacent pixels, equal to `(Xmax-Xmin)/94`
`Ymin`	The bottom edge of the screen, or the lowest onscreen Y value
`Ymax`	The top edge of the screen, or the highest onscreen Y value
ΔY	The difference between vertically adjacent pixels, equal to `(Ymax-Ymin)/62`

8.1.1   *Pixel-point coordinate system conversion*

In some programs, you'll want to combine pixel-based commands such as `Text` with point-based commands such as `Line`, which you'll learn about later in this chapter. For such programs, you may want to convert between the two coordinate systems or adjust the point coordinate window to match the pixel coordinates.

To translate a pixel coordinate (row, column) = (B,A) to point coordinates (X,Y), you translate the row (B) and column (A) separately, as shown in figure 8.4. The resulting point will have X value `Xmin+A*`ΔX, because `Xmin` is the leftmost column and each pixel column is ΔX from the previous one. The Y value, by similar reasoning, will have Y value `Ymax-B*`ΔY, so pixel (B,A) is point (`Xmin+A*`ΔX, `Ymax-B*`ΔY).

You can also go the other way around, translating a point (X,Y) to a pixel (B,A); this is also shown in figure 8.4. The row variable B will have pixel coordinate `(Ymax-Y)/`ΔY, and the column variable A will have pixel coordinate `(X-Xmin)/`ΔX. Inaccuracies in the way your calculator does math may cause those new A and B coordinates to not be exactly integers, so the `round` command, found in the NUM tab of the [MATH] menu, should be used. Point (X,Y) maps to pixel (`round((Ymax-Y)/`ΔY`),round((X-Xmin)/`ΔX`))`.

You can adjust the point coordinate window to almost match the pixel coordinates, making translation largely unnecessary. For the X-coordinate, you can set `Xmin` to 0 and ΔX to 1, making point coordinate X perfectly match the pixel coordinate columns. The Y-coordinate is more complicated, because point coordinates increase from bottom to top, whereas pixel rows increase from top to bottom. You can set `Ymin`
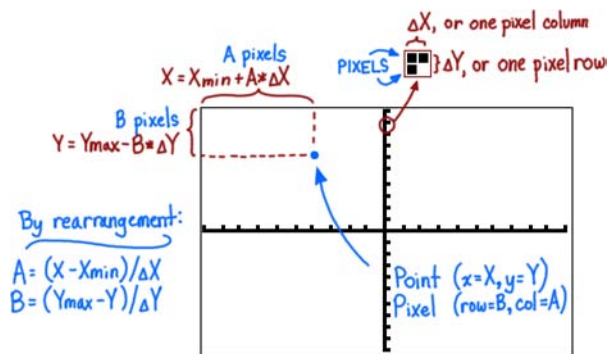
Figure 8.4   **Converting between pixel coordinates and point coordinates for an arbitrary point/pixel on the graphscreen, as described in the text. You can get the point coordinates (x = X, y = Y) for a point given the pixel coordinates (row = B, column = A) for the same point or rearrange the equations for X and Y to get pixel coordinates A and B back from a point coordinate pair (X,Y).**

to -62 and ΔY to 1, which will make Ymax=0. Thus you can negate the row of a pixel to get its point Y-coordinate or negate a point's Y-coordinate to get its pixel Y-coordinate. I'll reiterate the commands to make the pixel and point coordinates match:

```
:0→Xmin:1→ΔX
:⁻62→Ymin:1→ΔY
```

This means that the pixel at, for example, (row, column) = (45,20) would be the same as the point at (X,Y) = (20, -45), because the column and X match and the Y and row are negatives of each other.

> **TIP**   Pixel coordinates are written (row, column), whereas point coordinates are written (X, Y), where X is related to the column and Y to the row. Remember to not mix these up, or the examples herein (as well as your own programs) may be confusing.

Now that you have the basics of the point coordinate system, you'll learn how to make your program generate and manipulate graphs, as befits a program on a graphing calculator.

8.2   *Graphing from programs*

Graphing calculators stand apart from other educational tools like scientific calculators and computers in that they excel at displaying graphs. Considering what an important feature this is, it should come as no great surprise that programs can draw graphs too. You can use your programs to define Y= equations (and equations for the three other graph modes the TI-83+ and TI-84+ support), render graphs, pan and zoom, and annotate and draw on top of graphs. You'll learn about all these capabilities in this section and how to use them in your own programs. I assume that you have a moderate familiarity with drawing graphs on your calculator manually; if not, I recommend you refer to the graphing section in appendix A.

I'll begin with the basics of graphing: defining equations that you want to graph and displaying them. I'll teach you about predefined equations and equations that your program can modify on the fly.

8.2.1 Creating graphs

For all graphs, including those created by a student using a graphing calculator in math and those generated by programs, a two-step process is required. First, the calculator must be given the equation or equations that it should graph. Second, the calculator is instructed in some way to display the graph or graphs, or to at least do some sort of math on the equations entered. For the student, step 1 would be typing in an equation in the [Y=] menu, and step 2 would be pressing the [GRAPH] button or using the tools in the CALC menu under [2nd][CALC]. If any of that sounds unfamiliar, you can review appendix A.

A program can enter equations into the function equation variables, Y_1 through Y_9 as well as Y_0 for rectangular or function mode (Y_0 is the tenth variable, so we list it after Y_9). Like a human user, the program can then instruct the calculator's OS to display the graph or perform calculations on the equations entered.

The simplest way for programs to generate graphs is to store them into a variable, one of the equation variables under [VARS][▶][1]. You must store equations as if they're strings, enclosed in quotes:

```
3X+4→Y₃
"3X+4"→Y₃          ←——— Correct
"3X+4→Y₃
```

Wrong; this will produce an ERR: DATA TYPE error

Also correct; closing quotes and parentheses can be omitted before the store (→) operator

Entering the equations into graph variables isn't enough: you then need to instruct the calculator to draw the graph. For this, you can use the `DispGraph` command under the I/O tab of the PRGM menu, [PRGM][▶][4]. Section 8.5 will tell you more about using the `DispGraph` command with `Disp` to switch between the homescreen and graphscreen.

When you don't want to render an equation that's stored verbatim (as is) inside the program, however, things start to get more complicated.

VARIABLES IN STORED EQUATIONS

One technique you can use to programmatically "modify" stored equations is to include coefficients inside the equation, coefficients stored as variables. Consider the classic linear equation Y = MX + B, where M is the slope of the line and B is the Y-intercept. The statement `"3X+4"→Y₃` would store a line with slope = 3 and Y-intercept B = 4 into the graph equation (function) Y_3, but the slope and the Y-intercept would be constant, and the program wouldn't have a good way to change them. But the program could instead use variables M and B inside the equation and choose values for M and B, as in figure 8.5.

This program, LINCOEF1 (for Linear equation with Coefficients 1), uses just such a technique:

```
PROGRAM:LINCOEF1
:"MX+B→Y3
:3→M:4→B
:DispGraph
:Pause
:0.5→M:⁻1→B
:DispGraph
:Pause
```

Omitting the closing parenthesis, because it comes before a store (→) symbol

Pause works on the graphscreen too!

This is the negative symbol, not the minus operator

The line will be regraphed because the coefficients changed
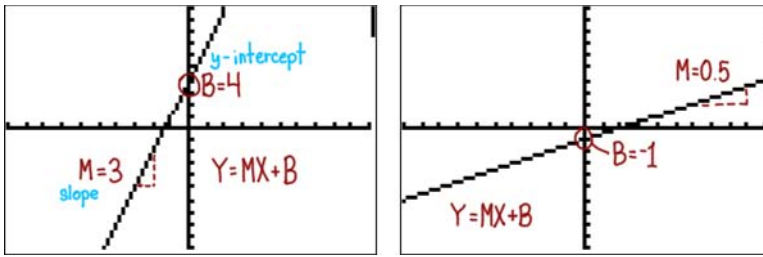
**Figure 8.5   Two linear graphs drawn by `prgmLINCOEF1`, both using the linear
graph equation Y = MX + B. At left, slope M = 3 and Y-intercept B = 4. At right,
M = 0.5 and B = ‾1.**

If you try this on your calculator and you don't see any graphs, try choosing 6:
ZStandard from the [ZOOM] menu and running it again.

This might still not be enough: you might want to convert whole strings containing
custom equations into graphs. In chapter 9, you'll learn to work with strings, includ-
ing concatenating (joining) pieces of strings together and pulling pieces of strings
into other strings. For now, I'll show you how to get a string from a user, turn that into
an equation, and graph it.

GRAPHS FROM STRINGS

Strings are created by putting characters like letters and numbers between pairs of
quote marks. Graph equations are similar, except that the expression between the
quotes needs to be a valid equation for a graph. To convert a string into an equation,
you can use the `String▶Equ` command. It can be found by pressing [2nd][0][4] for
the T section of the Catalog and scrolling up three or four lines. It takes two argu-
ments, the string to convert and the equation into which to store the converted equa-
tion. There's the reverse command, `Equ▶String`, which can also be found in the
Catalog ([2nd][0]) under the E section. Here are the two commands in action:

```
:"1.4sin(X)-2→Str2
:String▶Equ(Str2,Y₁          ◁——— Now YI = I.4sin(X) – 2
:Equ▶String(Y₁,Str8
```

After this code completes, Str8 and Str2 will be identical, and Y_1 will hold the equation
as well. You can quickly test these commands with a new program, CUSTOMEQ, which
prompts the user for a string and then graphs it as an equation.

```
PROGRAM:CUSTOMEQ
:ClrHome
:Disp "ENTER AN EQUATIO","N TO GRAPH
:Input "Y=",Str1
:String▶Equ(Str1,Y₁
:DispGraph
:Pause
```

Why not just directly use `Prompt` or `Input` with Y_1 and its kin? The TI-OS doesn't accept
graph equation variables as arguments to `Prompt` and `Input` and will produce an
ERR:DATA TYPE error if you try.

Once you've defined some functions to graph, there's much more you can do in your programs. You can zoom and pan around the graph and add annotations. You can change extra items shown with the graph, including the axes and something called the grid. I'll even show you formulaic "prolog" and "epilog" code that will ensure your graphscreen programs don't frustrate users who want to render graphs after using your programs and games.

8.2.2 *Manipulating graphs and functions*

Drawing graphs on the graphscreen is just the tip of the iceberg as far as your calculator's graphing features go. Your programs can also manipulate graphs the way you can by hand from your calculator's menus. In section 8.1, you learned how to zoom and pan around graphs by changing the window variables, $Xmin$, $Xmax$, $Ymin$, $Ymax$, ΔX, and ΔY. In the coming sections, you'll learn to draw shapes, lines, and points on the graphscreen, which you can use to annotate graphs or for drawings without any graphed functions. In this section, you'll now learn commands to zoom, modify graph settings and modes, and prolog and epilog code you can apply to your own programs to make them play nice with your users' calculators.

An overview of several useful graphscreen manipulation commands is shown in table 8.2. There are two zoom commands, two commands to control which (if any)

Table 8.2 A small subset of the TI-BASIC commands you can use to manipulate the graphscreen from inside your programs. Explore the [ZOOM] (Zoom), [MODE] (Mode), and [2nd][ZOOM] (Format) menus for more commands.

Command	In menu…	What it does
ZStandard	[ZOOM]	Sets the default window: $Xmin=Ymin=-10$, $Xmax=Ymax=10$.
ZSquare	[ZOOM]	Adjusts $Xmin$ and $Xmax$ or $Ymin$ and $Ymax$ so that $\Delta X=\Delta Y$ (which means squares look square and diagonals are displayed accurately).
FnOff	[2nd][0] or [VAR][▶][4]	Disables all Y= functions (but doesn't delete them). If you specify a number, like FnOff 4, it only disables that Y= equation, for example, Y_4.
FnOn	[2nd][0] (Catalog)	Enables all Y= functions, or with a numeric argument, only enables that Y= equation.
StoreGDB	[2nd][PRGM] [▶][▶][3]	With a number 0–9, saves the current graph functions, the graph settings (axes, window, and grid) to a variable such as $GDB0$, $GDB5$, or $GDB9$. Example: StoreGDB 7.
RecallGDB	[2nd][PRGM] [▶][▶][4]	With a number 0–9, recalls that GDB variable, restoring the graph settings and functions. Example: RecallGDB 3.
GridOn GridOff	[2nd][ZOOM]	Enables or disables a grid of dots on the graphscreen at every (X,Y) location where X and Y are both integers.
Func Par Pol Seq	[MODE]	Changes between Function, Parametric, Polar, and Sequential graphing modes. We work only with Function (Y=) graphing in this chapter, but programs can also graph in the other three modes. Check the resources in appendix C for info.
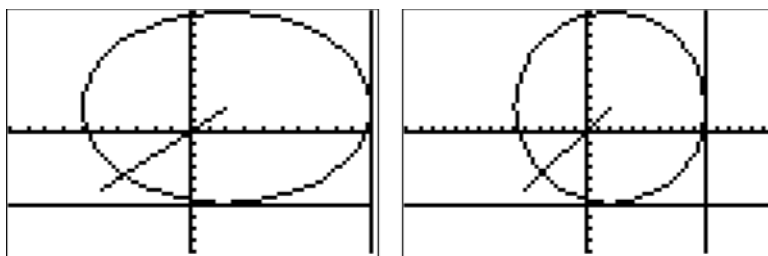
Figure 8.6   The difference between `ZStandard` (left) and `ZStandard` followed by `ZSquare` (right)

Y= equations are graphed, two commands to save and restore graph settings with a data structure called a graph database (GDB), and commands to change graph settings and modes. Besides these, you can find more zoom commands in [ZOOM], other graph formatting options in [2nd][ZOOM], and graph mode commands in [MODE].

Figure 8.6 illustrates the difference between `ZStandard`, which sets `Xmin=Ymin=-10` and `Xmax=Ymax=10`, and `ZSquare`, which increases the `Xmin` and `Xmax` or `Ymin` and `Ymax` of the current window to make ΔX=ΔY. Because the screen is rectangular, `ZStandard` makes the circle at the left side of figure 8.6 appear stretched and the diagonal line appear as not a true diagonal. When the `ZSquare` command is added, the circle looks round and the diagonal looks properly angled. The program that draws these four shapes, DRAWDEMO, is presented in section 8.4. The [ZOOM] menu also contains commands to zoom in and out and set a window that shows trigonometric functions well, among others.

Besides equations in the form Y=, your calculator can graph parametric functions where both X and Y are parameterized, polar functions in the form r=f(θ), and sequential (recursive) functions. You can switch the graphing mode with the `Func`, `Par`, `Pol`, and `Seq` commands, all of which are found under [MODE]. If you want to define equations for each of those graphing modes, the necessary equation names can be found in [VARS][▶], where the Y_0 through Y_9 tokens also reside.

The `StoreGDB`/`RecallGDB` and `FnOn`/`FnOff` commands are particularly useful for helping programs roll back any changes they make to the graphscreen to leave the graphscreen settings of the user's calculator as they were before the program was run.

POLITE GRAPHSCREEN PROGRAMS

Hastily written graphscreen programs may turn off the axes, change the window, disable functions, and generally frustrate the user who may not be savvy enough about their calculator to understand what has happened and how to fix it. To save your programs' users and players that aggravation, I'll show you commands you can put at the beginning and end of your programs to avoid this problem. These prolog and epilog commands will make your life easier by letting your program work with a clean slate and will also restore graph settings and functions when the program is complete for the sake of politeness.

The `StoreGDB` and `RecallGDB` commands respectively save and recall a file that records the current graph mode, any and all equations the user has entered, and the graph settings such as the window and whether the axes are disabled. If you store a GDB (Graph Database), your program can then safely turn off all the equations so that you can use a blank graphscreen unmarred by graphed functions, change the window as your program's needs dictate, and turn off the axes. A simple `RecallGDB` function will then undo all of those changes. If you want to be able to draw on the graphscreen with no graphs and no axes and the standard window settings, your prolog, the first few lines of your program, might look like this:

Set Xmin=
Ymin=-10,
 Xmax=
Ymax=10

```
:StoreGDB 0
:FnOff :AxesOff
:ZStandard
```

Turn off all functions, and turn off the axes

Chosen because it's at the bottom of the GDB list and likely won't conflict with any GDBs that the user manually saved

At the end of the program, you need only have a single line as an epilog to undo all of that:

```
:RecallGDB 0
```

The GDB number used with the `RecallGDB` command must match the number used with `StoreGDB`, because there are 10 GDBs named `GDB1` through `GDB9` plus `GDB0`, just like Y_1 through Y_9 plus Y_0. Keep in mind that any single GDB stores all of the functions Y_1 through Y_9 and Y_0, as well as the current window and zoom settings.

As you've seen, your calculator offers programs many tools to manipulate the graphscreen. We'll conclude this section with a few miscellaneous commands and tricks that you'll be able to use in your own programs.

8.2.3 *Other graph tools and tricks*

There are a few odds and ends that don't fit elsewhere in this chapter but may come in handy in your own program. First, there are a few drawing commands that can annotate or augment graphed functions. Next, there are commands to calculate values and properties of graphed functions. Finally, there's the way to plug a value for X into a Y= equation to get the corresponding Y for that X.

In the remainder of this chapter, I'll be showing you how to use many of the drawing commands in the DRAW menu ([2nd][PRGM]) of your calculator. But there are several useful functions that I won't have space to discuss in detail but are still worth mentioning. The `Shade` command is used to shade the graphscreen, drawing an area of black pixels between two functions, a function and a Y value, or two Y values, for example:

```
:Shade(Y₁,10
:Shade(⁻5,Y3
:Shade(Xmin,Xmax
```

Fills the graphscreen with black

The `Tangent` command takes a Y= function as the first of two arguments and an X value as the second and draws the line tangent to the Y= function at that point, as in `Tangent(Y1,4.5)`. `DrawInv` is used to graph functions that are of the form X = f(Y) instead of the normal Y = f(X). `DrawInv X²` draws the sideways parabola $X = Y^2$.

You can calculate the minimum, maximum, and integral of a function within a given interval, as well as the derivative of a function at a point. All four of these commands, fMin, fMax, fnInt, and nDeriv, are found at the bottom of the MATH tab of the [MATH] menu. fnInt, fMin, and fMax each take as arguments the function to analyze, the name of the independent variable (usually X), and the two bounds for the interval. nDeriv takes the function, the independent variable, and the point at which to calculate the derivative:

```
:fMin(Y3,X,⁻8,6
:fMax(Y5,X,5,8.3
:fnInt(Y1,X,1.5,2.5
:nDeriv(Y9,X,84
```

If your program needs to calculate the Y-value of a function at a given X, it can use one of two code "idioms." First, it can set X to the desired value and can then use Y1 as if it was a number:

```
:3.4→X
:Disp Y1
```

Alternatively, you can put the X value in parentheses after the Y= function name to calculate the same value:

```
:Disp Y1(3.4
```

You now have seen all the basics of graphing from your programs. I'll move on to the many drawing commands your calculator offers to let your programs create fun games, attractive math programs, and much more.

8.3   *Drawing with points*

In chapter 7, I introduced the concept of drawing with pixels. You learned how you could turn pixels on and off, how to flip the color of a pixel, and even how to make a program check if a pixel is black or white. In much the same manner, you can manipulate points in the coordinate plane to turn them on or off. The primary difference from the pixel coordinate system is that a single coordinate pair, such as (0,0) or (-5,6.3) or (992,-496) may refer to different physical pixels depending on how the window is set. If you tried to draw pixels offscreen when I introduced them in chapter 7, you might have discovered that your calculator throws an ERR:DOMAIN error, indicating that you can't use pixel coordinates that are offscreen. Points, however, work differently: no matter where you draw a point, it will "work," even if the point is offscreen. Offscreen points won't be drawn, but an error won't be shown either.

At the end of section 8.1, I mentioned specific window settings that would make pixels and points overlap exactly, save for the fact that the Y-coordinate of each point would be the negative of the row of each pixel. Thus, the pixel at row 30, column 78 would be the point (X,Y)=(78,-30). This is handy if you want to mix points, pixels, and other commands such as Text and the Line and Circle commands from the next section.

The three point commands are Pt-On, Pt-Off, and Pt-Change. All three can be found in the POINTS tab of the DRAW menu with the Pxl commands, accessed with [2nd][PRGM][▶]; table 8.3 lists the three commands and their arguments.

Table 8.3   The syntax and usage of the `Pt-On`, `Pt-Off`, and `Pt-Change` commands

Command	Example	Explanation
`Pt-On(X,Y)` or `Pt-On(X,Y,style)`	`Pt-On(4,4`	Always turns the point at (X,Y) on (to black), even if it was already on
`Pt-Off(X,Y)` or `Pt-Off(X,Y,style)`	`Pt-Off(9.5,0.5`	Always turns the point at (X,Y) off (to white), even if it was off
`Pt-Change(X,Y)`	`Pt-Change(⁻4.2,1`	Switches a black point to white and a white point to black

Pt-On and Pt-Off (but not Pt-Change) can accept an optional third argument, which specifies a point style to use for rendering. If the extra argument is a 2, then Pt-On and Pt-Off draw a 3-by-3-pixel box centered on the given coordinates. If the argument is a 3, the two commands draw a 3-by-3-pixel cross or plus symbol. If the argument is anything else, the commands draw a single point. The left side of figure 8.7 shows the difference between the three point-drawing modes.

8.3.1   *Example: a point-drawing screensaver*

To demonstrate the point commands, I'll show you a simple screensaver-style program; a demo is presented at the right side of figure 8.7. It sets the window to Xmin→Ymin→0, Xmax→47, and Ymax→31. This means that every two pixels up or across the display is an increase of 1 in X or Y or that each pixel is 0.5 in the X and Y directions. This is a square window, which means that going the same number of pixels in the X and Y direction is also the same increase in X or Y. This screensaver randomly draws points on or off at integer X and Y locations, using the three possible point styles, until you press a key.
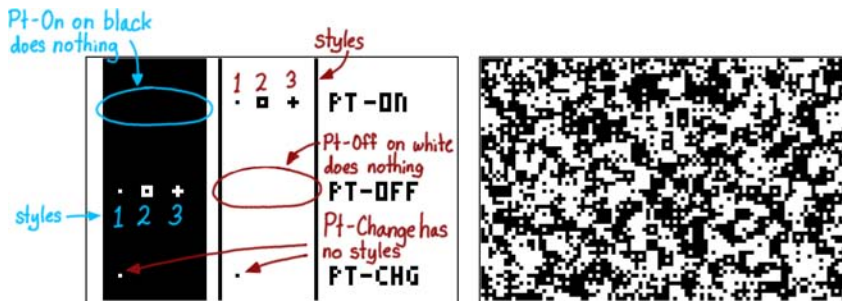
Figure 8.7   The three `Pt-` commands and their optional styles (left) and the output of the PTSAVER screensaver demo (right)

The code for this program is in listing 8.1. After performing the normal prolog functions to set up the graphscreen, `StoreGDB`, and `FnOff`, it turns the axes off and clears the screen. It sets `Xmin`, `Xmax`, `Ymin`, and `Ymax` as specified previously and uses a `While` loop to loop until a key is pressed. The loop chooses X- and Y-coordinates (stored in A and B), a style (S), and then "flips a coin" to decide whether to turn the point with those coordinates on or off with the style S. It cleans up and displays the homescreen when a key is pressed.

Listing 8.1   Program PTSAVER

```
PROGRAM:PTSAVER
:StoreGDB 0
:FnOff :AxesOff
:ClrDraw
:0→Xmin:47→Xmax
:0→Ymin:31→Ymax
:While not(getKey
:randInt(0,47→A
:randInt(0,31→B
:randInt(1,3→S
:If rand>0.5
:Then
:Pt-On(A,B,S
:Else
:Pt-Off(A,B,S
:End
:End
:RecallGDB 0
:Disp
```

Prolog to save the graph equations and state for later restoration and clear the screen and turn off the axes

Set up the window as discussed

Loop until getKey returns non-zero (i.e., a key is pressed)

randInt(M,N) returns a random integer between and including M and N; rand returns a random decimal between 0 and I. Half the random numbers will be above and half below 0.5, so this is like flipping a coin.

Restore the graph settings and display the homescreen

A warning against variable Y

As you begin to create programs that use the point coordinate system, you may be tempted to store arguments in variables X and Y, so that you can call commands like `Pt-On(X,Y)` or `Line(X,Y,X+4,Y)`. You should avoid using the variable Y. In all of the chapters up to here, I've used variables such as (A,B) to represent coordinates, because your calculator's OS has a bug, and I wanted to train you not to use Y as a coordinate. Every time you call the `ClrDraw` command, the Y variable is reset to 0, and whatever you may have stored in Y is lost. Whenever you're writing a program that uses the graphscreen, you shouldn't use the numeric variable Y for anything.

Points are powerful, especially because you can draw them offscreen without worrying about errors and can use various styles for the `Pt-On` and `Pt-Off` commands. The graphscreen has even more in store for you, as you'll see next. From lines to circles to shapes, the next section will teach you more about your calculator's graphscreen drawing commands.

8.4 Lines and shapes

Points, either by themselves or combined with graphs, pixels, and text, are a useful tool, but it's still tedious to try to draw larger images when tools like lines could make the job easier. TI-BASIC offers a number of drawing commands to help you draw such images, of which I'll cover four in this section, shown in figure 8.8. First, I'll tell you about the `Line` command, which can be used to draw or erase lines. I'll talk about two commands that make vertical and horizontal lines fast and easy, and I'll introduce the `Circle` command. Because the commands themselves are fairly self-explanatory, I'll swiftly move on to a small demo program that uses the four commands and a larger example program, POLYGON, that uses the `Line` command to draw polygons.

8.4.1 The drawing commands

The four drawing commands we'll explore in this section all take point coordinates, expressed as X and Y values. Each of the four commands obeys the current graph window. You can draw lines and circles that are partially offscreen and cross an edge, and just as with points, you even draw things that are entirely offscreen without worrying about error messages. First, to draw a line from point (X1,Y1) to point (X2,Y2) use the `Line` command:

```
:Line(X1,Y1,X2,Y2)
```

You could also omit the ending parenthesis, because saving space in your programs is always important. The `Line` command can also be used to erase lines, essentially drawing a line but in white pixels (like `Pt-Off`) instead of black pixels, by adding a fifth argument that's a zero:

```
:Line(X1,Y1,X2,Y2,0)
```

You can make the fifth argument a 1 instead of a 0 to draw a black line, and you can even use a variable instead of 0 or 1, allowing your program to draw either a white or black line depending on the value of the variable.
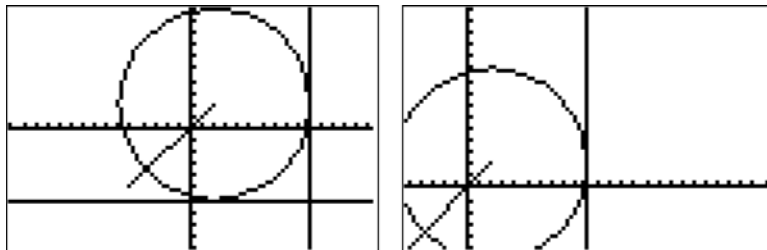
Figure 8.8   The DRAWDEMO program used to make a circle, a diagonal line, a vertical line, and a horizontal line drawn with a centered window (left) and an off-center window (right). Drawing commands will work properly even if the result is partially or entirely offscreen, unlike `Pxl` commands.

The `Horizontal` and `Vertical` commands each take only a single argument and are among the commands that are separated from their argument with a space instead of parentheses. The argument to `Horizontal` is a Y-coordinate, and the argument to `Vertical` is an X-coordinate. The `Horizontal` command draws a line from `Xmin` to `Xmax` at the specified Y value, as long as the Y value is onscreen. The `Vertical` command follows the same logic but draws a line between `Ymin` and `Ymax` at the given X-coordinate:

```
:Horizontal Y1
:Vertical X1
```

The `Circle` command does what you might expect, given an (X,Y) coordinate for the center of the circle and a radius. Of the four commands mentioned in this section, it's noticeably slower, taking about two seconds to complete a circle:

```
:Circle(X,Y,R
```

Clever TI-BASIC programmers have noticed that if you put a list containing the imaginary number *i* (typed with [2nd][.]) as a fourth argument to `Circle`, you can force circles to be drawn about four times faster than normal. This is likely an Easter egg (hidden feature) added by the calculator's OS programmers:

```
:Circle(X,Y,R,{i})
```

To demonstrate all four of these commands together, here's a tiny program that clears the screen and draws a slow circle, a diagonal line, a vertical line, and a horizontal line. It doesn't do anything with the window, the axes, or any functions entered in [Y=], so you can modify the window in [WINDOW] and graph settings in [2nd][ZOOM] (as the two sides of figure 8.8 showed) to see how the drawn circle and lines change. You'll see a more complete program in the next section that carefully handles the axes, functions, and window.

```
PROGRAM:DRAWDEMO
:ClrDraw
:Circle(2,2,8
:Line(2,2,⁻5,⁻5
:Vertical 10
:Horizontal ⁻6
```

With those basic commands under your belt, I'll show you an example program that uses the `Line` command to render polygons on the graphscreen. You'll see defensive programming techniques I've discussed previously, as well as a few notes about using trigonometric commands in your programs.

8.4.2   *Using lines to draw polygons*

This particular example program is both a demonstration of using the `Line` command and an example of how you can draw polygons in your own programs. It prompts the user for how many sides the polygon should have and the distance between each vertex (where the sides meet) and the center, which it refers to as the radius. It makes
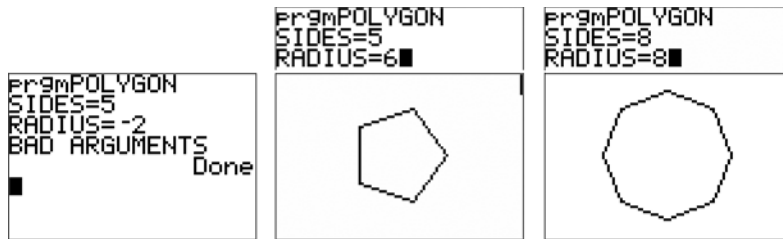
Figure 8.9   Using the `Line` and other graphscreen manipulation commands to draw polygons. The leftmost screenshot demonstrates sanity checking on the inputs; the middle and right screenshots show a pentagon and an octagon, respectively.

sure that the user enters a positive integer for the number of sides and a positive number for the radius, a defensive programming technique (demonstrated at left in figure 8.9). If the checks pass, it draws the resultant polygon and pauses on the graphscreen, as the center and right of figure 8.9 show.

The code for this program is presented in listing 8.2. All of the commands have been presented previously with the exception of `Degree`, but the trigonometric math to calculate the coordinates of each polygon's corners might be new to you, as might be the combination of commands that comes before and after the polygon is drawn. For typing the program in listing 8.2, the `Degree` command is in [MODE]. `sin(` and `cos(` are on their respective keys, and the graph commands such as `AxesOff` and `FnOff` are in [2nd][ZOOM] (the Format menu). `ZStandard` and `ZSquare` are under [ZOOM], and `RecallGDB` and `StoreGDB` are in the STO tab of the DRAW menu ([2nd][PRGM]).

Listing 8.2   The POLYGON program to draw S-sided polygons with the `Line` command

```
PROGRAM:POLYGON
:Input "SIDES=",S
:Input "RADIUS=",R
:If R<0 or S<3 or S≠int(S
:Then
:Disp "BAD ARGUMENTS
:Return
:End
:StoreGDB 0
:FnOff :AxesOff
:ZStandard
:ZSquare
:Degree
:For(θ,0,359,360/S
:Line(Rcos(θ),Rsin(θ),Rcos(θ +(360/S)),Rsin(θ +(360/S
:End:Pause
:RecallGDB 0
```

Defensive programming: don't try to draw the polygon if the user specified fewer than three sides, a noninteger number of sides, or a negative radius

Store the graph settings so that axes and functions can easily be restored

Turn off all Y= functions, turn off the axes, zoom to Xmin=Ymin=-10 and Xmax=Ymax=10

Adjust the window to set ΔX=ΔY, so that squares appear square

Enter Degree (not Radian) mode

Restore the window, axes, and functions from before prgmPOLYGON ran

The `Input` commands and the conditional logic that performs sanity checking should be fairly familiar to you by this point in your TI-BASIC programming journey. The majority

of the new commands are those that set up the necessary graph properties (the pro-log) before drawing the polygon and the one that cleans up afterward (the epilog). As in previous examples, I'll explain the latter half of this program line by line:

```
:StoreGDB 0
```

`StoreGDB 0` will save which functions are enabled, what the current graphscreen window is, and whether the axes are shown or hidden.

```
:FnOff :AxesOff
:ZStandard
:ZSquare
```

Change the graph properties to the way you want them. `AxesOff` and `FnOff` respectively turn off the axes and disable the drawing of any and all functions that may be entered in [Y=]. The pair of zoom commands, `ZStandard` and `ZSquare`, first sets the window to the standard or default size and then adjusts the `Xmin` and `Xmax` values so that $\Delta X = \Delta Y$.

```
:Degree
```

The `Degree` command means that sine, cosine, and tangent will interpret arguments as if they were written in degrees. Conversely, the `Radian` command instructs trig commands to interpret arguments as written in radians. Just as 360 degrees is a full circle, 2π radians are a full circle.

```
:For(θ,0,359,360/S
:Line(Rcos(θ),Rsin(θ),Rcos(θ +(360/S)),Rsin(θ +(360/S
:End:Pause
```

This loop will iterate S number of times, stepping 360/S each time. If S = 3, then it will run with $\theta = 0$, $\theta = 120$, and $\theta = 240$, three equally spaced angles along a circle. For S = 6, a hexagon, it would iterate through $\theta = 0$, $\theta = 60$, $\theta = 120$, $\theta = 150$, $\theta = 180$, $\theta = 240$, and $\theta = 300$. Notice that I made the ending angle 359, not 360, because if it was 360 it would also run with $\theta = 360$, which would redraw the segment it drew when $\theta = 0$.

The arguments to the `Line` command use the fact that if you have an angle θ and a radius R, then $R\cos(\theta)$ is the X-coordinate of the point a distance R from (0,0) in the θ direction, and $R\sin(\theta)$ is the Y-coordinate, as shown in figure 8.10. Therefore, $(R\cos(\theta), R\sin(\theta))$ are the coordinates of the current vertex at each iteration, and $(R\cos(\theta + 360/S), R\sin(\theta + 360/S))$ are the coordinates of the next vertex. Drawing lines between the current and next vertices for every vertex of a polygon as it goes
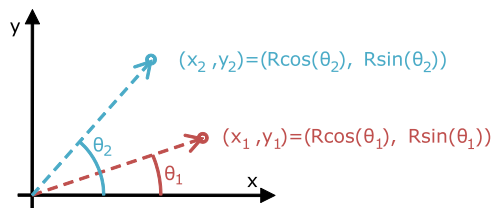
Figure 8.10   Calculating the (x,y) positions of two points from their respective angles θ**. In this polygon-drawing program, each** $\theta_2 = \theta_1 + (360/S)$**, so that** θ_1 **and** θ_2 **are the angles to adjacent vertices.**

through the loop will then draw all the sides of the polygon, producing outputs such as those in figure 8.9.

```
:RecallGDB 0
```

Recall `GDB0`, which will restore the axes, functions, and window to the way they were before `prgmPOLYGON` started running.

Program politeness and the angle mode

In many of the programs you've seen throughout the chapters so far, attempts have been made to make "polite" programs. Such programs try to restore the user's or player's calculator to its state before the program was run, including cleaning the homescreen, fixing the axes, and restoring window settings. The angle mode (whether the calculator is in Radian or Degree mode) should be no exception to this. A given program may need to set `Degree` or `Radian`, but a polite program would remember the original setting and restore it when the program ends. Luckily, a clever trick exists that will reveal the current mode setting. When the calculator is in Radian mode, any angle measure with a degree symbol (that is, 3°) is converted to its Radian equivalent (in this case, 0.052…). In Degree mode, it will remain 3. If 3° = 3, then the calculator is currently in Degree mode; if not, it's in Radian mode. The following code uses this trick to store the current angle mode in the program's prolog and restore it in the epilog:

```
:(3°≠3)→M
:Degree
:...program code...
:If M:Radian
```

8.4.3 *Extras: Text and the polygon*

If you want to make your `prgmPOLYGON` output look like the left side of figure 8.1, all you need to add are two extra lines of code to the program. You can add these two lines between the `Degree` command and the `For` loop in listing 8.2, so that the text will be drawn right before the polygon itself is rendered. You could equally correctly put them between the `End` and `Pause` commands; the only difference is that if the polygon overlapped the text, it would be placed underneath the text. Recall that text and lines both overwrite whatever is already on the screen, so the order in which you render items in your programs dictates how they're layered. Here's the code:

```
Text(0,0,S," SIDES
Text(7,0,"RADIUS ",R
```

Notice that these two lines of `Text` each provide both a number and a string as arguments to display, which `Text` concatenates together, shown in figure 8.1.

Now you know how to draw shapes, how to write text, and how to create points and pixels. What if you've created a great image on the graphscreen and you want to save it? What if you want to repeatedly and quickly show something on the screen in your

program without having to painstakingly redraw it each time? Saving and recalling picture variables is what you need.

8.5   *Working with pictures*

Shape, pixel, text, and graph-drawing commands in TI-BASIC are reasonably fast, but if you have a lot of items to draw at once, the time can add up. Because you always want to make sure your programs are both fast and small, an adage I've mentioned many times before, it's sometimes worthwhile to save the current graphscreen image to memory so you can open it again later without needing to redraw it. The tradeoff is that each such picture takes 767 bytes of memory, but in many cases, the memory is worth the saved drawing time.

In this section, you'll learn about the 10 picture variables your calculator can store, how to store and recall each picture, and how they can be used in your programs and games. First, you'll find out what a picture variable is and how to store and recall such variables.

8.5.1   *What's a picture?*

The old axiom says that a picture is worth a thousand words, but for a calculator programmer, pictures aren't quite as costly. Each picture variable is 767 bytes out of the 24,000 bytes or so of RAM (and 160 KB to 1.5 MB of Archive, depending on calculator model). Your calculator has 10 picture variables, `Pic0`, `Pic1`, `Pic2`, through `Pic9`. Each time you save to a picture, it overwrites the previous contents of that picture. If you try to recall a picture that doesn't exist, you'll get an ERR:UNDEFINED error. As with other commands that create variables, if you don't have enough memory to fit a new picture, you'll may get an ERR:MEMORY error when you store a picture.

The two commands for pictures are `StorePic` and `RecallPic`, and both commands take a single number as an argument, the number of the picture variable to store or recall:

```
:StorePic 1
:RecallPic 6
```

There are several things to note with this pair of commands. First, you can't use a numeric variable with `StorePic` and `RecallPic`, so something like `RecallPic N` is invalid. Unfortunately, if you want to use the contents of some variable like N to determine which picture to store or recall, you'll have to use a long set of conditional statements:

```
:If N=0
:RecallPic 0
:If N=1
:RecallPic 1
:...etc...
```

Interestingly, for both commands' argument, you can either use a number like 3 or 7 or the name of the picture variable, such as `Pic3` or `Pic7`. These variable names can be

found under [VARS][4]. The `RecallPic` command will restore the given picture to the graphscreen and set that to the foreground. This brings us to a pair of commands that can flip between the homescreen and foreground without displaying anything new:

- `Disp`—When run with no arguments, the `Disp` command shows the homescreen.
- `DispGraph`—Found in the I/O tab of the PRGM menu ([PRGM][▶][4]), the `DispGraph` command brings the graphscreen to the foreground.

Recall from figure 7.2 that your calculator's LCD is like an easel, on which you can place either the graphscreen or the homescreen but not both at the same time. You can use `Disp` and `DispGraph` to flip between the two. To return to the motivating example, if you want to recall a picture but continue to show the homescreen for now, run the `Disp` command after `RecallPic`.

Now that you know the mechanics of storing and recalling pictures, I'll discuss how pictures can be useful in your programs.

8.5.2 *Interfaces, optimization, and layering with pictures*

Picture variables can be useful in many types of programs. In math and science programs and in games, any time you want to save a graphscreen image and later restore it, `StorePic` and `RecallPic` will help. If you want to swap among several different graphscreen images, you can store each in a picture and use `ClrDraw` and recall a different picture for each different image you want to display. You can also combine several images. If you don't use `ClrDraw` between recalling pictures, then they're layered together, as demonstrated in figure 8.11; any number of pictures can be combined this way. One real-world example of this is a "3D" first-person shooter game that constructs 3D corridors on the calculator's LCD with pieces stored in different pictures. One picture contains a solid left wall, another a branch to the left, a third a box of health on the ground, and so on. By combining pictures, the game can quickly draw complex environments, rather than painstakingly redrawing the walls and other elements with point/pixel and `Line` commands every time the player moves.

Picture variables are often used for things like splash screens, complex hand-drawn images displayed when a game is started that represent the author of the game or the game itself. Such images can also be part of the program's main menu. When you're designing a program or game, be aware that all of your programs, and all the programs
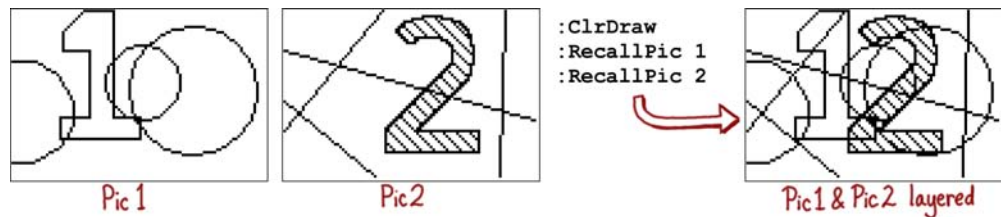
Figure 8.11 Layering two pictures by recalling them without an intervening `ClrDraw` command. Here, `Pic1` and `Pic2` are layered together, but any number of pictures can be combined with the same technique.

that anyone writes, use the same `Pic0` through `Pic9`, and so if your game and another game both use (for example) `Pic3`, one of the two games won't work properly. If possible, it's better to have your program generate any pictures it needs when it's run, save them to `Pic` variables, and use those as necessary. This ensures that even if another program has overwritten the pictures your program needs, your program regenerates the images and fixes them. A TI-BASIC 3D racing game could draw a fancy car dashboard and windshield and save that to a picture, then clear the screen and recall the picture every time the road in front of the car needs to be redrawn.

You now have seen many of the different graphscreen drawing and graphing tools your calculator offers to TI-BASIC programmers, and in the next chapter, you'll learn more about the types of variables and data your calculator can store and use.

8.6   *Summary*

The past two chapters have presented a wide cross-section of commands to draw on the graphscreen, from text and pixel manipulation to graphs, shapes, and lines. In this chapter, you first learned about the point coordinate system and how it provides a window onto an infinite plane on which functions are graphed, the axes are rendered, and drawings are drawn. I presented commands to create and manipulate graphs from programs and to draw from programs. Although I didn't yet show a specific example, you can create attractive and easy-to-use programs by combining pixel-based commands such as `Text` with point-based commands such as `Line`. As with every concept I've shown you, it's vital that you play with the drawing and graphing commands yourself, think of program ideas you want to try, and put them together. You should also try out any DRAW menu commands that I didn't discuss.

In the next chapter you'll learn about the many types of data beyond numbers and pictures that your calculator can store, including matrices, strings, and lists. I'll show you some rudimentary games you can expand on and talk about complex numbers and randomness.

Manipulating numbers and data types

9

This chapter covers

- Working with strings and real and complex numbers
- Making programs use random numbers
- Creating games and programs with matrices and lists

Numbers can take on many roles in your programs. They can represent integers or decimal numbers, they can represent the Boolean values true and false, and they can be used to hold angles or coordinates or health or time or any number of other things. But in TI-BASIC, as in almost any other programming language, numbers aren't expressive enough for everything. Single numbers can't store information about the sequences of characters that form words and sentences. The numeric variables that you've used so far can only clumsily be used for storing sequences of numbers. If you've read appendix A, you already know that TI-BASIC has solutions for these and other problems in the form of strings, lists, and matrices.

In chapter 8, you learned about the picture, GDB, and Y= equation data types; in this chapter, I'll introduce strings, matrices, and lists. I'll show you how each of

these three data types might be used in your own programs and what commands you should know for working with each. The latter half of this chapter will return to numbers, teaching you commands to manipulate real and complex numbers and to generate random numbers. I'll combine the lessons from these three data types to show you how to write the framework of a simple RPG (role-playing game) and challenge you to expand it with features like enemies, pickups, health, and scoring.

A good starting point is the TI-BASIC string, which can store sequences of letters, symbols, and tokens. You'll see how you can create and manipulate strings.

9.1   Using strings

The first data type that you'll learn about in this chapter is the string. You've seen strings scattered throughout the preceding chapters, first with `Input`, `Prompt`, and `Disp` and then with each of the commands that have used letters or words enclosed in quotes. In a few places I introduced the string variables, named `Str1` through `Str9` plus `Str0`, most recently when discussing defining equations to be graphed from within a program. In this section, you'll learn how to define strings, join and cut apart strings, and find substrings within strings. I'll also show you how you can execute TI-BASIC code stored inside strings.

9.1.1   Defining and manipulating strings

A string (in any programming language) is a sequence of characters stored together, such as letters, numbers, and symbols. In TI-BASIC, you can even put commands and tokens, like `Disp`, `sin(`, and `Y₂` into strings as if they were single letters. For example:

> **As always, note the omitted closing quote before the → to save space**
>
> **It also works if you include the closing quote**

```
:"HELLO WORLD→Str1
:"3²+sin(X)→Str4
:"SOMETHING or OTHER→Str0
```

> **The " or " here is from the LOGIC menu, [2nd][MATH][▶]**

You can't put quote marks or the store symbol (→) into a string, because both are used to signal the end of a string. You can put lowercase letters in strings if you use a tool or shell such as Doors CS to enable lowercase, but keep in mind that each lowercase letter takes 2 bytes, whereas each uppercase letter takes only 1 byte. Although "HELLO WORLD" and "hello world" are both 11-character strings, the former is 11 bytes; the latter is 22 bytes. You should use lowercase sparingly in your programs, if at all.

FINDING THE LENGTH OF A STRING

To get the length of a string, you can use the `length` command, which is in the Catalog under [2nd][0][)] (for the L section of the Catalog). It takes either a string variable or a literal string as an argument and returns a number representing the length of that string. Each letter, number, or symbol counts as 1, and each token like `Circle(` or `ClrHome` or `getKey` also counts as 1. Here's a simple program to display the length of a string the user enters:

```
PROGRAM:STRLEN
:Input "STRING:",Str1
:Disp "LENGTH IS:",length(Str1
```

The length of the string that the user can type in for this program (or that your program can store into a string) is limited only by the amount of free RAM in your calculator. If your program tries to store a bigger string than the calculator's RAM can hold, the TI-OS will throw an ERR:MEMORY error.

> **TIP** Rather than saying "Str0 through Str9," the text says "Str1 through Str9 plus Str0." Pic and GDB variables in chapter 8 were the same, for good reason. Although numerically Str0 would come before Str1, on the TI-83+/84+ Str0, GDB0, Pic0, and so on are the tenth of their variable type, appearing at the end of the list after Str9, GDB9, and Pic9.

EXECUTING STRINGS AS CODE

You can even put certain types of TI-BASIC code into a string and execute the contents of the string as if it was its own subprogram. The expr command, also found in the Catalog under [2nd][0][SIN] (the E section), will execute any string containing a math expression and return the result, but it won't work with commands like Disp or RecallPic. The following FAKEHOME program acts like a fake homescreen, displaying the results of math expressions that the user enters until the user types 999:

```
PROGRAM:FAKEHOME
:ClrHome
:Repeat Str1="999"
:Input "",Str1
:Disp expr(Str1
:End
```

Repeatedly get a math expression and evaluate it until the user types 999

This line prompts for a string but displays nothing before waiting

Evaluate the contents of Str1 as if it was math, and display the result

If you're clever, you could slightly modify this as a prank program to make your calculator appear to do math wrong, but I'll leave figuring that out to you, and I caution you to use your expertise with tact and discretion.

JOINING STRINGS

Joining strings, called concatenation, is performed with the plus operator. Putting + between numbers means to add them, putting + between two strings, whether literal strings or Str variables, means to join them. Here are some examples:

```
:Str1+Str4→Str4
:"---"+Str1→Str4
:Disp "H"+"3770"+Str5
```

But you can't concatenate a string with a nonstring, such as a number. The expression "H"+3770 (notice the missing quotes) would yield an ERR:DATA TYPE message.

CUTTING AND SPLITTING STRINGS

You can take out a substring of a string, a piece of that string, using the sub command, once again found in the Catalog under S via [2nd][0][LN]. The sub command takes three arguments: the string from which to take a substring, the offset at which to start

the substring, and the length of the substring. The first character of a string in TI-BASIC is offset 1, the second is 2, and the tenth is 10, in contrast to most other languages, where the first element of a string or list is usually 0 instead of 1. To give you a concrete example, sub("HELLO",5,1) would produce O, and sub("FOLD",2,2) would return OL. If a program tries to get a substring that goes past the end of the string, either because the offset argument is too large or the length is too large, you'll get an INVALID DIM error.

FINDING SUBSTRINGS

A final command, inString, is used to find a string in another string; to keep straight how it works, programmers commonly use the idiom "search for a needle in a haystack." The first argument is the string to search inside (the "haystack"), the second is the string to look for ("the needle"); inString looks for the needle in the haystack and returns the offset of the substring, if it finds it. If it doesn't find a match, it returns 0. There's an optional third argument that specifies the offset at which to start:

```
:inString("HAYSTACK","NEEDLE"
:inString("HAYSTACK","NEEDLE",Offset
```

HAYSTACK and NEEDLE are each either strings or Str variables; offset is a number

If Str1 contains a single letter, the following line of code would produce a number 1 through 26 corresponding to that letter's place in the alphabet:

```
:Disp inString("ABCDEFGHIJKLMNOPQRSTUVWXYZ",Str1
```

I'll conclude this section with an example of sub, but I recommend you play around with combining sub, inString, length, and string concatenate to give yourself a more intuitive understanding of how they fit together.

9.1.2   *String sub example: Xth letter of the alphabet*

You can use the sub command to manipulate strings to figure out the Xth letter of the alphabet, just as I showed you how to find the position X in the alphabet of an arbitrary letter. The results of such a program might look something like the two sides of figure 9.1.

Figure 9.1   Two uses of the sub command to manipulate strings demonstrated in prgmLTRNUM

Not only does this program display the Xth letter of the alphabet, it also appends custom ordinal suffixes (ST, ND, RD, TH) to the number on the first line of the final output, the third line in the screenshots in the figure. If you examine the code in listing 9.1, you'll see that it always uses TH to start, but if the input number X is less than 4, it selects a custom suffix. For X = 1, it plucks the substring of length 2 at offset $2 - 1 = 1$, or "TH." For X = 2, it chooses "ND," and for X = 3, "RD."

Listing 9.1   Program LTRNUM to display the Xth letter of the alphabet

```
PROGRAM:LTRNUM
:ClrHome
:Disp "ENTER A NUMBER
:Repeat X≥1 and X≤26 and X=int(X
:Input "1 TO 26:",X
:End
:ClrHome
:Disp "THE   TH LETTER","OF THE ALPHABET","IS:      ←   Three spaces
:Output(1,5,X                                             between "THE"
:If X<4                                                   and "TH"
:Output(1,7,sub("STNDRD",2X-1,2                      ←   For X = I, 2, and 3,
:Output(3,5,sub("ABCDEFGHIJKLMNOPQRSTUVWXYZ",X,1          displays a special prefix:
:Pause                                                    "IST," "2ND," "3RD"
```

Just as strings are sequences of letters, your calculator has data types that can store one-dimensional and two-dimensional sequences of numbers, called lists and matrices.

9.2   *Lists and matrices*

Now you know that strings let you store sequences of letters together, and numeric variables let you store single numbers. Consider something like a high scores table, where it would be helpful to be able to store a set of numbers together in a single variable. Imagine a game on the homescreen, where each of the character spots on the screen could be something like a tree, a piece of wall, or an enemy, where a two-dimensional set of numbers representing each item would be a great way to store the map. Once again, TI-BASIC comes to the rescue with lists and matrices. I introduce lists and matrices in appendix A, which you may have reviewed before reading chapter 2 if you hadn't previously heard of matrices and lists. If this section is the first you're hearing about them, I strongly encourage you to review the lessons of appendix A, especially how you type lists and matrices. Otherwise, let's start with the usage and manipulation of lists.

Lists, sometimes called vectors or arrays in other programming languages, store between 1 and 999 numbers in a single variable. There are six built-in lists, named L_1 through L_6, but you can also create custom lists named with one to five letters, such as $_LA$ or $_LSCORE$ or $_LMINE8$. These so-called custom lists can be used in your TI-BASIC programs to save settings, save games, and save high scores between runs of your program; the custom names mean that another program is unlikely to overwrite those lists. You can type L_1 through L_6 with [2nd][1] through [2nd][6]; for custom lists, the subscript L

is under [2nd][STAT][▶][▲]. You use the `dim` command to create a list of a given size, resize an existing list, or check a list's size.

```
:5→dim(L₃
:800→dim(ʟBIG                          displays 800
:Disp dim(ʟBIG        ◁─────────┐
```

You can also create a list by specifying its contents, such as {1,2,3}→L₆. Other commands include `Fill(number,list)`, which sets every element of `list` equal to `number`, and `SortA(list)` and `SortD(list)`, used to sort the contents of a list in ascending or descending order. These and other commands to manipulate lists can be found in the OPS tab of the LIST menu, or [2nd][STAT][▶].

Matrices, or two-dimensional arrays, store a grid of between 1 x 1 and 99 x 99 numbers in a single variable. Unfortunately, there are only 10 matrices, named `[A]` through `[J]`; you can find them all under the NAMES tab of the MATRX menu, at [2nd][x⁻¹]. The `dim` command works with matrices as well as lists and can be used to create or resize matrices and find the size of an existing matrix, for example:

```
:{8,16}→dim([G]                  ◁─────────        8 rows, 16 columns, just
:{10,10}→dim([F]   ◁──────  A 10 x 10 square matrix   like the homescreen
```

The `Fill` command can be used on a matrix, as can the many other matrix math commands in the MATH tab of the MATRX menu, [2nd][x⁻¹][▶]. You can take the determinant of a matrix with `det` and flip (or transpose) a matrix with the ᵀ command.

You can get or set the value of the first element of a list with ʟTHIS(1), the second using ʟTHIS(2), and so on. You can get or set the last element (if you're not sure of the list's size) with ʟTHIS(dim(ʟTHIS. Matrices are similar, but you must supply a pair of numbers for the element index, ordered as (row, column). [A](1,1) is the top-left corner of matrix [A], [A](2,1) is the first column of the second row, and [A](1,3) is the third column of the first row. You can use single elements of lists or matrices as if they were numeric variables like B or T, including as values for math and for commands and as storage for the return value of any command or expression.

You can also manipulate whole lists or matrices by adding, subtracting, multiplying, or dividing a single value from an entire list or matrix. You can add or subtract two lists of equal sizes, and you can add, subtract, multiply, and divide matrices of equal sizes. You can use comparison operators to compare single elements of lists or matrices or to compare entire lists or matrices with other lists or matrices of the same dimensions (size).

Although this is a greatly abbreviated overview, lists and matrices are straightforward and powerful, and you'll find them to be especially useful in your games. Between the information in this section, the sample game you'll see at the end of this chapter, and appendix A, you have the essentials of using lists and matrices; I leave it up to your explorations (or your online forum posts; see appendix C) to brainstorm efficient and clever uses for these data types.

9.3   *Working with integers and complex numbers*

Thus far, you've worked extensively with numbers and with variables that can hold numbers, such as `A` through `Z` and θ. Other than the arithmetic operators like addition, subtraction, multiplication, division, and raising to a power, your calculator can manipulate numbers in many useful ways. I'll reintroduce the `int` command along with `iPart`, `fPart`, `round`, and `abs`. I'll then introduce you (or remind you, depending on how extensively you've used your calculator) to complex numbers and the several commands your TI-83+ or TI-84+ offers to work with complex numbers. Let's begin with the most important of the commands TI-BASIC offers for manipulating numbers, especially in dealing with integers and decimal numbers.

The numeric variables you've been working with thus far in this book, `A` through `Z` and θ, are called the Reals, in that they hold real (noncomplex, nonimaginary) numbers. Each contains a single value, a positive or negative number that may have many digits before and after the decimal point. The smallest allowed value is around 10^{-99} (that is, a decimal point, 99 zeroes, and a 1); the largest is 10^{99} (a 1 followed by 99 zeroes). Your calculator can't store that much precision, though, and for very large and very small numbers it stores their approximate value. Your calculator offers numerous commands to work with real numbers, mostly found in the NUM tab of the [MATH] menu. Table 9.1 highlights a few of the most useful commands.

Notice that for positive numbers, `int` and `iPart` work the same way; the difference between the two is for negative numbers. `int` rounds down to the next integer,

Table 9.1   New commands to manipulate the integer and decimal parts of numbers

Command	In menu…	What it does
`int`	[MATH][▶]	Rounds down to the next integer and returns it. `int(6.9)` = `int(6.1)` = 6; `int(⁻6.1)` = `int(⁻6.9)` = -7.
`iPart`	[MATH][▶]	Plucks out the part of the number before the decimal point and returns it. `iPart(6.9)` = 6; `iPart(⁻6.1)` = -6.
`fPart`	[MATH][▶]	Plucks out the part of the number after (and including) the decimal point and returns it; also maintains the sign of the number. `fPart(6.9)` = 0.9; `fPart(⁻6.1)` = -0.1.
`round`	[MATH][▶]	Rounds to the nearest integer. Takes two arguments: the number to round and the number of digits to round to. `round(1.75,0)` is 2; `round(1.75,1)` is 1.8.
`abs`	[MATH][▶]	Returns the absolute value of its argument. `abs(X)` returns X if X is positive or the positive version of X if X is negative.
`Float`	[MODE]	The normal number display mode; shows as many digits after the decimal place (if any) as necessary.
`Fix`	[MODE]	Followed by a number 0–9, such as `Fix 3`. Always displays exactly that many digits after the decimal point, regardless of the number.

`iPart` plucks out the integer part (the bit before the decimal point) of the current number, so if X is a negative decimal (and not an integer) such as -6.1 or -590.04, `iPart(X)-1 = int(X)`. `fPart` is more straightforward, returning the decimal part of the number; the only gotcha is that it also maintains the sign of the number. The `round` and `abs` commands work as any math student might expect. The `Float` and `Fix` commands control the number of digits displayed after the decimal point when a decimal number is displayed.

You've previously seen the `int` command used to check if some number is an integer with the following formulation:

```
:If X=int(X
:Disp "X IS INTEGER
```

You'll find yourself using that in your programs, particularly when you want to check if a number that a user typed for `Input` or `Prompt` is an integer. The `iPart` and `fPart` commands can also be used for "compression," letting you compress two numbers into a single number by storing one before the decimal point and the other after it.

There are a few other useful commands for working with numbers in the MATH and NUM tabs of the [MATH] menu, which I leave you to explore on your own. Let's spend the remainder of this section looking at complex numbers and how you can work with them.

COMPLEX NUMBERS

To understand complex numbers, you must first understand imaginary numbers. You may be familiar with the idea that you can't take the square root of a negative number, which is only partially true. In mathematics, the square root of -1, or $\sqrt{(^-1)}$, is defined as a number called i, the imaginary unit. Because $i = \sqrt{(^-1)}$, $i^2 = -1$. You can use i to help you take the square root of other negative numbers by the rules for simplifying square roots:

$$\sqrt{(A * B)} = \sqrt{(A)} * \sqrt{(B)}$$
$$\sqrt{(^-4)} = \sqrt{(^-1)} * \sqrt{(4)} = i * 2 = 2i$$

Keep in mind that i isn't a variable here; it's part of the number itself, just as a negative sign placed before a number is part of that number. Any imaginary number is written in the form Ni, where N is a positive or negative decimal number. A real number, as distinguished from an imaginary number, is any number that doesn't include i.

You can also have complex numbers, which are numbers that have both real and imaginary parts. A complex number is written a + bi, where a is the real part and b is the imaginary part. One way to think about complex numbers is as a pair of coordinates (a, b) in a plane, just as (x, y) is a point in the Cartesian plane. Figure 9.2 shows this concept:
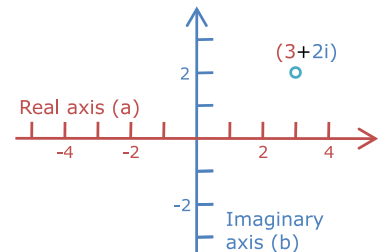
Figure 9.2   Plotting a complex number as a point in a two-dimensional plane. The point 3+2*i* is at 3 on the real axis and 2 on the imaginary axis.

the x-axis is the real axis (corresponding to the value of a in a + b*i*), and the y-axis is the imaginary axis (corresponding to b). The point plotted, $3 + 2i$, is at 3 on the real axis and 2 on the imaginary axis.

You can type the imaginary *i* with [2nd][.]. Complex numbers can be stored inside the usual numeric variables A–Z and θ. Your calculator offers five commands to manipulate complex numbers, found in the CPX tab of the [MATH] menu. They are

- `real`—Returns the real part of a complex number. `real(3+2i)` returns 3.
- `imag`—Returns the imaginary part of a complex number. `imag(3+2i)` is 2.
- `conj`—Calculates the complex conjugate of a complex number. This is like flipping it over the real axis, or negating the imaginary part. `conj(3+2i)` is $3 - 2i$.
- `angle`—Calculates the angle a line would make relative to the real (x) axis if you drew the line from the origin at $0 + 0i$ to the point in question. If you're in Degree mode (from the [MODE]) menu), then `angle(4+4i)` = 45 (degrees). In Radian mode, it's $\pi/4$ or 0.7854….
- `abs`—Returns the magnitude of a complex number; `abs(2+3i)` = $\sqrt{(2^2 + 3^2)}$ = $\sqrt{(13)}$. This is the very same `abs` command used to take the absolute value of a Real.

As a final note, taking the square root of a negative number returns ERR:NONREAL ANS from a program or on the homescreen, if your calculator is in Real mode from the [MODE] menu. If your program switches to a + b*i* mode instead, it will properly calculate the square roots of negative numbers. For politeness, if your program uses the `a+bi` command to switch to complex mode to avoid the NONREAL ANS error, it may wish to switch back to Real mode before it terminates. Be aware that the `real`, `imag`, `conj`, `angle`, and `abs` commands work in Real mode; a + b*i* mode is only needed for taking the square root of negative numbers.

Your math programs may need to take advantage of complex and imaginary numbers, but it's unlikely you'll need them for games. Both games and math programs can make effective use of your TI-83+/84+'s commands for generating random numbers, the final topic we'll cover in this chapter before an example game that summarizes the lessons of the chapter.

9.4 *Revisiting randomness*

In chapter 1, I showed you a simple number-guessing game as your first TI-BASIC game. In chapter 8, you saw a program called PTSAVER that drew points with random positions and styles on the graphscreen. Both programs used the `randInt` command to generate randomness, and the latter also introduced the `rand` command. Both commands generate some sort of random number, a number selected arbitrarily. Random numbers are the same sort of numbers you might get from rolling dice or flipping a coin. In this section, I'll show you how to use `randInt` and `rand`, as in figure 9.3, and mention other commands for generating random numbers. I'll show you what it means to seed the random number generator and why that's useful, and I'll conclude

```
randInt(1,6
                3
                2
                3
                6
                1
                5
■
```

```
rand
    .2523434268
    .1472581069
    .6361803262
    .2958671057
    .7076862165
    .1808546696
■
```
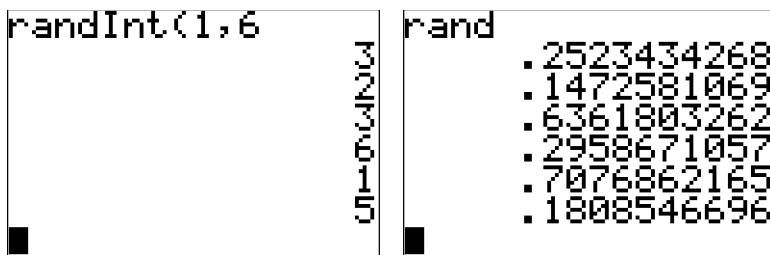
Figure 9.3   Demonstrating the `randInt` and `rand` commands on the homescreen, without a program. `randInt` generates random integers between and including the two specified numbers, whereas `rand` generates random decimal numbers between 0 and 1.

with a coin-flipping program and an exercise for you to create a program that rolls and displays a die.

First, I'll explain the various commands you can use to generate random numbers.

9.4.1   Generating random numbers

Both the `randInt` and `rand` commands generate what are known as uniformly distributed random numbers. Every number in the set of possible numbers is as likely to be selected at a given call to `rand` and `randInt` as any other, as figure 9.3 illustrates. The `rand` command picks a random decimal number between 0 and 1; it takes no arguments. `randInt` generates an integer with a value somewhere between (and including) the values of its two arguments. Both `rand` and `randInt` can be found in the PRB tab of the MATH menu, accessed with [MATH][◀].

```
:randInt(⁻10,15→A
:Disp rand
:rand(50
```

> You can also create a list of N random numbers, by adding (N at the end

As you can see from the third line of these examples, although `randInt` and `rand` are usually used to get a single random number, you can add an extra argument to `rand` to get a list of that many random numbers, generated as if you called `rand` that number of times. `rand(42)` generates a list of 42 random numbers. You'll see in chapter 10 that this trick is also sometimes used as a good way to create a delay.

Your calculator offers two commands that generate nonuniformly distributed random numbers. That is, each of the numbers isn't equally likely to be selected. The first is `randBin`, which generates binomially distributed random numbers; the second is `randNorm`, which generates numbers selected from a Gaussian distribution. You're not likely to need those for your own programs unless you're working with engineering or statistics programs.

SEEDING THE RANDOM NUMBER GENERATOR

In some cases, you may want to be able to generate the same sequence of random numbers more than once. This sounds counterintuitive, because by definition sequences
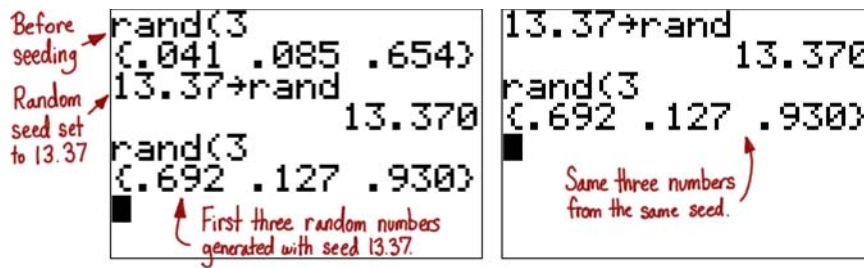
Figure 9.4 Setting the random number generator's seed lets you create the same series of three random numbers twice. This also works with using `rand` and `randInt` to generate single numbers.

of random numbers shouldn't repeat. But your calculator uses something called a pseudorandom number generator, which means that even though it looks as though the numbers are random, they're being calculated by an algorithm. The calculations are based on a number called the random seed, which defines the sequence of numbers that commands such as `rand` will generate. Figure 9.4 demonstrates setting the random seed by storing the new seed to the `rand` command as if `rand` was a variable. Notice that the sequence of numbers repeats the same values after reseeding with the same seed, here 13.37.

Because I mentioned how random numbers are conceptually equivalent to the real-world problems of flipping a coin or rolling a die, I'll conclude this section with a program that shows you coin flipping and a challenge for you to create your own die roller.

9.4.2 *Applying the random number commands*

I'll begin by showing you a simple program to flip a virtual coin, and then I'll challenge you to create your own die-rolling program, complete with a graphical output. The left side of figure 9.5 shows the output of the former program, the right side shows the latter. Let's first dive right into the coin-flipping program, which consists of three lines of code, only one of which forms the core of the program.

```
PROGRAM:FLIPCOIN
:ClrHome
:Disp "COIN FLIP:
:Disp sub("HEADSTAILS",1+5(rand>0.5),5
```

The first two lines of the program clear the screen and display an introductory line; the third line is the meat of the program. It combines a `sub` command, which you learned in the first section of this chapter, with a trick you'll learn more about in the next chapter called an implicit conditional. The `rand` command will return a random decimal between 0 and 1, which means that it's equally likely that its value will be greater than or less than 0.5. You could create a weighted coin by modifying this value, called the threshold, which chooses whether the program displays "HEADS" or "TAILS."
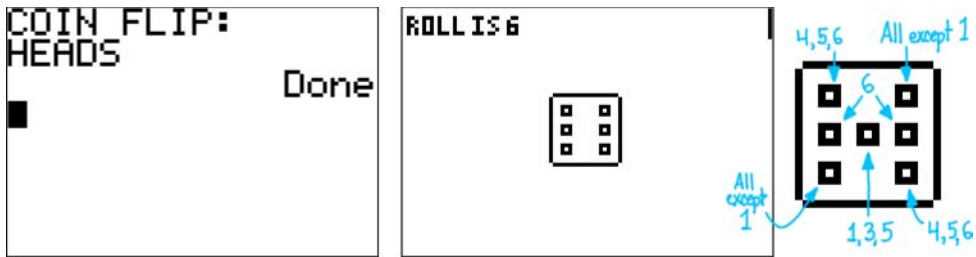
Figure 9.5 The output of the coin-flipping demo (left) and the die-rolling exercise (right). The large die is marked with which die values cause which dots on the die to be drawn for your reference.

When the return value of rand is greater than 0.5, then (rand>0.5) is true, which you'll recall is represented in TI-BASIC by the number 1. Thus $1 + 5(1) = 6$, which is the offset to the "TAILS" substring. If rand returns a value equal to or less than 0.5, then (rand>0.5) is false, which is 0; $1 + 5(0)$ is 1, the offset to the "HEADS" substring. Both "TAILS" and "HEADS" are five characters long, so the third argument is always 5. If this implicit conditional trick, using a conditional in a math expression as if it was a number, doesn't make sense to you yet, don't worry. I'll review it once more in chapter 10.

EXERCISE: ROLL A DIE

As a final exercise for this chapter, I task you with making a program that rolls a six-sided die. As an output, it should display that die on the graphscreen, with dots on the upward-facing face representing what value the die has landed upon. You need not animate the die rolling, though if you're able to do that, I encourage you to try. The output should look something like figure 9.5 when you finish writing the program. Don't forget to use the "polite" prolog and epilog from section 8.2.2 and to add a Pause right *before* the epilog cleans up the screen so that the user can see what the die roll yielded. You can also use Text to write "ROLL IS 3" (the value the die rolled) in the upper-left corner of the screen.

DIE-ROLLING SOLUTION

The solution to this exercise is shown in listing 9.2. Nothing in this program should be particularly new. The first line selects a random integer between 1 and 6, the value that the die will display. The program then does a standard prolog to save and set up graphscreen settings. It draws a square to form the edges of the die and draws the dots as per the right side of figure 9.5. Finally, it pauses, cleans up, and ends after switching to the homescreen.

Listing 9.2 ROLLDIE, a dice-rolling program

```
:randInt(1,6→X          ◁————————   Choose the value
:StoreGDB 0                          the die will display
:FnOff
:AxesOff:ClrDraw        Standard
:0→Xmin:1→ΔX            prolog and
:⁻62→Ymin:1→ΔY          window setup
```

```
:Text(1,1,"ROLL IS ",X
:Line(39,⁻22,55,⁻22
:Line(38,⁻23,38,⁻39
:Line(56,⁻39,56,⁻23
:Line(55,⁻40,39,⁻40
:If X≥4:Then
:Pt-On(42,⁻26,2
:Pt-On(52,⁻36,2
:End
:If X=6:Then
:Pt-On(42,⁻31,2
:Pt-On(52,⁻31,2
:End
:If X≠1
:Then
:Pt-On(42,⁻36,2
:Pt-On(52,⁻26,2
:End
:If X=1 or X=3 or X=5
:Pt-On(47,⁻31,2
:Pause
:RecallGDB 0
:Disp
```

Draw the outside of the die

All the Pt-On commands use style 2, which draws a box

As an additional challenge, try making this program ask the user for a number between 1 and 5 and roll and display that many dice. For this extra step, you should make a separate program ZDIE that takes arguments in variables A and B (the position of one corner of this die on the graphscreen) and X (the value of the die). You could also try working with fancier dice that have more than six faces. For an extra challenge, you can try animating the roll of each die. From this program, you could make a simple game quite easily.

The last section of this chapter will present a different game, a single-screen roleplaying game (RPG) in which you move a player around a map and have to find a coin while avoiding an enemy.

9.5   *Fun with data types: a single-screen RPG*

You've reached the end of the built-in TI-BASIC commands you've been learning for the past nine chapters. In the remaining chapters, you'll be learning about optimizing your programs, using hybrid BASIC, a little bit of z80 assembly, and you'll conclude with where you can go from there. In the chapters up to this point, you've built up knowledge of using the homescreen, working with conditionals and program flow, creating graphics on the homescreen and graphscreen, using getKey and event loops for interactive programs, and most recently, manipulating data types. As a capstone on all that learning, let's look at a game that applies many of the lessons you've learned.

The game I'll show you in this section, called MATRXRPG, challenges players to get to a spot in the map while avoiding an enemy that's actively hunting them. The game has a map, which includes the walls, the blank spaces, and the goal that players need to safely reach. Figure 9.6 shows this game in action. The pi symbol (π) is the player,
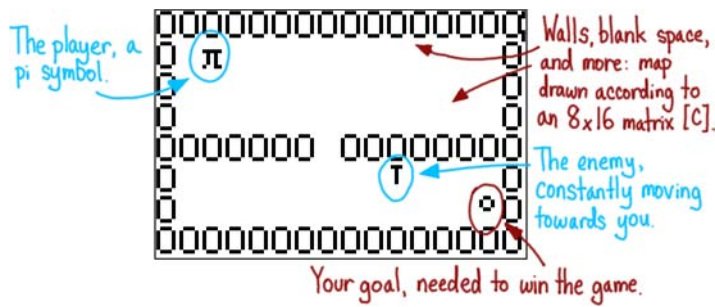
Figure 9.6   The matrix-based homescreen RPG. A matrix contains the map, including the walls, blank spaces, and the goal. The player at (B,A) and the enemy at (N,M) are drawn when needed; the player must move to get to the goal while avoiding the enemy. Touching the enemy hurts the player, and if the player's health drops to zero, the game ends.

the transpose ^{T} is the enemy, and the walls are Os. The goal is the degree symbol at the lower right.

To build this game, you'll work with many of the skills you've learned so far. As the name of the game suggests, a matrix will be used to hold the map. An event loop will be used to get keypresses from the player; the non-key code inside the event loop will handle moving the enemy toward the player. I'll use Menu and Lbl/Goto to create a main menu that can jump to the Help section, to the Quit section, or to the game itself. The sub string command will make the map-drawing code fast and small. Before we get into the specifics of how this game works, let's look at the flowchart in figure 9.7. This flowchart, the description of the game, and the interface design in figure 9.6 form our plan for the MATRXRPG program.

As you can see from the flowchart, the game will start by creating the matrix that will hold the map, [C]. Because the map is the same size as the homescreen, the matrix will be 8 rows by 16 columns, making each element of the matrix correspond to one character on the homescreen. The main menu can lead to a Quit function, which currently only displays credits, a Help section, which does what it says on the tin, and Play, which leads to the main gameplay. In the gameplay area, it first sets up variables and draws the map, then enters the main loop. This loop draws the player and enemy, runs an event loop, and then ends with erasing the player and performing relatively standard arrow key–handling code. The main loop for this game should continue until any of three things becomes true:

- The player's health drops to zero, or $H \leq 0$.
- The player presses clear, or $K = 45$.
- The player reaches the goal.

We can use a Repeat loop for this outer loop to avoid having to initialize K. Inside the event loop, we'll need to use getKey to search for keys, but we also need to move the enemy toward the player. To make the game slightly easier, the enemy will only
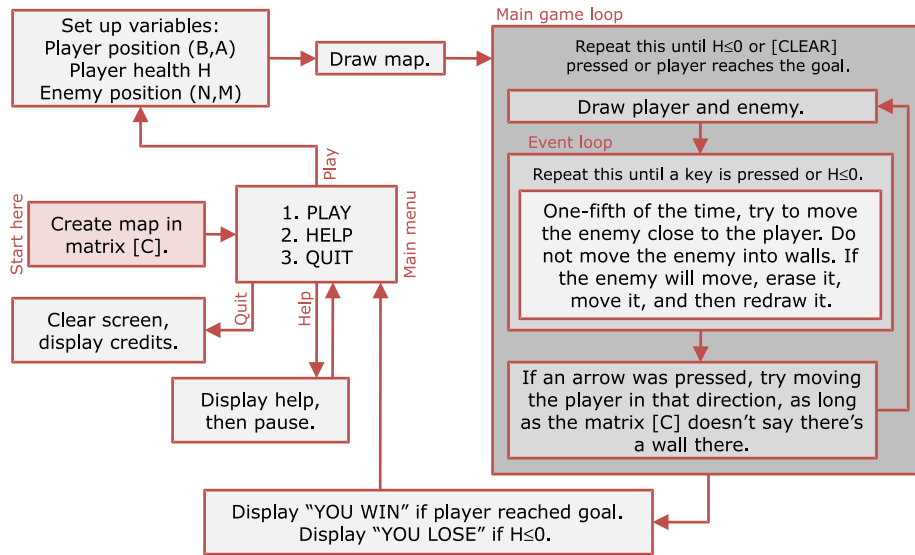
Figure 9.7   A flowchart of the MATRXRPG game. The left side is the main menu, Help, and Quit. The top is pregame setup; the right side is the main game loop. Inside the main loop is an event loop, as you learned in chapter 6.

move in roughly one of every five event loop iterations; we can use `rand` to achieve this. For both moving the player and moving the enemy, we need to perform bounds-checking to ensure neither character goes offscreen or into a wall. Luckily, because the map's walls form a border around the entire screen, we only need to make sure that the characters aren't going into walls. Because we have the entire map stored in a matrix, we can check the object at the position to which the player or enemy is about to move and see if it's a wall, a blank space, or something else.

A matrix can only store numbers, so how are we going to use the matrix to draw a map that's clearly made of characters? The easy solution would be to come up with a system where each element of the matrix would store a number and have each of those numbers represent one possible character (or item). Here's the system we'll use, a rudimentary version of a technique called tilemapping:

- 1 = A blank space (a space character)
- 2 = A wall (O)
- 3 = A health pickup (+)
- 4 = The goal (a degree symbol)

You can use the `sub` command to turn a number X into one of these characters like this:

```
:sub(" O+°",X,1)
```

With these basic lessons, take a look at the following listing, the code for this game.

Listing 9.3   The framework of a simple homescreen RPG, MATRXRPG

```
PROGRAM:MATRXRPG
:{8,16→dim([C]                       Create matrix [C] for the map,
                                     with 8 rows and 16 columns
:Fill(1,[C]                                              Fill the whole matrix
:For(X,1,16                                              with 1s, or blank space
:2→[C](1,X
:2→[C](5,X          Add walls at the
:2→[C](8,X          top, bottom, and
                    middle of the map               Replace the center of
:End                                                the middle wall with
:1→[C](5,8                                          a single blank space
:For(X,1,8
:2→[C](X,1          Add walls at the
:2→[C](X,16         left and right
                    sides of the map                Add the final goal to
:End                                                the bottom-right
:4→[C](7,15                                         corner of the map
:Lbl MM
:Menu("MATRIX RPG","PLAY",A,"HELP",H,"QUIT",Q
:Lbl Q:ClrHome
:Disp "MATRIX RPG 1.0","CHAPTER 9
:Return                                             Initialize player's health
:Lbl H                                              to 100 and the player's
:Pause "HELP GOES HERE                              position to (2,2)
                              The main gameplay
:Goto MM                      starts here
:Lbl A
:100→H:2→A:2→B
:14→M:7→N                                           Initialize the
:ClrHome                                            enemy's position
:For(D,1,8              Clear the screen and draw   to (7,14) (row 7,
:For(C,1,16            the map. Use the sub         column 14)
:Output(D,C,sub(" O+°",[C](D,C),1   command to turn the
:End:End              numbers 1–4 into letters.
:Repeat K=45 or H≤0 or 4=[C](B,A                    The outer game loop,
:Output(B,A,"Π                                      which continues until
:Output(N,M,"T        Draw the player               [CLEAR] is pressed, or
:Repeat K or H≤0      and the enemy                 the player dies or wins
:If rand<0.2          Repeat the inner event loop
:Then                 until a key is pressed or the
:Output(N,M,"[one space]  player dies               If we're moving the
:If rand>0.5                                        enemy, erase it first
:Then                 Randomly try to move
:If A<M and 2≠[C](N,M-1   the enemy either left or
:M-1→M                right toward the player,
:If A>M and 2≠[C](N,M+1   as long as it won't be
:M+1→M                moving into a wall
:Else
:If B<N and 2≠[C](N-1,M                             Alternatively, try moving
:N-1→N                                              the enemy up and down
:If B>N and 2≠[C](N+1,M                             toward the player (but
:N+1→N                                              not into a wall)
:End                  Now that the
:Output(N,M,"T        enemy moved,
:End                  redraw it
:If N=B and M=A
:H-30→H               If the enemy has reached the player,
                      removed health from the player
```

About 1/5 of the time, rand will be less than 0.2, so try moving the enemy

```
:getKey→K:End
:Output(B,A,"[one space]
:If K=24 and 2≠[C](B,A-1
:A-1→A
:If K=26 and 2≠[C](B,A+1
:A+1→A
:If K=25 and 2≠[C](B-1,A
:B-1→B
:If K=34 and 2≠[C](B+1,A
:B+1→B
:End
:If H≤0
:Pause "====YOU LOSE!===
:If 4=[C](B,A
:Pause "====YOU WIN!====
:Goto MM
```

End the inner event loop

Adjust player's position based on arrow keys. Instead of bounds-checking, check that the spot the player will be moving to isn't a wall (2) spot.

If players finish with zero or negative health, they died

If the game ends with players on the degree symbol, they won

Rather than go through this entire program line by line, I'll pick out a few sections in particular that you should notice:

```
:For(D,1,8
:For(C,1,16
:Output(D,C,sub(" O+°",[C](D,C),1
:End:End
```

This double `For` loop draws the map on the screen. It loops the row (D) from 1 to 8, and for each D, it loops column C from 1 to 16. The homescreen is 8 rows tall and 16 column wide, and matrix [C] is the same size, so the map can be drawn by figuring out what character the number in each element of [C] represents and drawing that character at the same coordinates on the homescreen.

```
:Repeat K=45 or H≤0 or 4=[C](B,A
```

Repeat the main outer loop until [CLEAR] is pressed, the player's health hits zero, or the player reaches the goal (represented by a 4 in the matrix). The comparison `[C](B,A)=4` is written reversed so that the closing parenthesis can be omitted, a simple optimization.

```
:Repeat K or H≤0
```

If the program is currently inside the inner loop, and the outer loop needs to end because the player died, the inner loop has to end too. The inner loop has to end either when a key is pressed or when the player's health hits zero.

```
:If rand<0.2
:Then
:Output(N,M,"[one space]
:...code...
:Output(N,M,"T
:End
```

The body of this `If/Then/End` will only run if `rand` returns a value below 0.2. Because `rand` selects a random number between 0 and 1, about one-fifth of the time it will pick

a value below 0.2 (1 / 5 = 0.2), so during roughly one out of every five inner loop iterations the game will try to move the enemy. The Output statements ensure that the enemy gets erased before it moves, because the program doesn't remember where it used to be, which would be needed to erase the enemy after moving it.

```
:If rand>0.5
:Then
:...code...
:Else
:...code...
:End
```

Instead of moving both horizontally and vertically toward the player, flip a coin to determine whether to try moving closer in the horizontal or vertical direction.

```
:If K=24 and 2≠[C](B,A-1
:A-1→A
```

If the player presses an arrow key, only let the player move if the matrix [C] says that the new location doesn't contain a wall piece. This formulation is used three more times for the three other arrow keys, as well as for the four conditional statements that try to move the enemy.

As always, you should examine and understand this code, and you should also try out the program. If there are pieces you don't understand, try changing them and seeing how the program's behavior changes. If you understand it well, you can try modifying and expanding this game to make it more expansive and more complex. I encourage you to make a full, fun game out of this and release it to the public!

MATRIX RPG: YOUR CHALLENGE

The MATRXRPG program presents a simple framework for an RPG-like game (or, perhaps, an arcade game). But in its current shape, it's far from complete. Here are a few ideas of items you could add to it or a similar game to make it more fun, more complex, and more challenging, and I'm sure you can think of many of your own ideas.

- What if you or the enemy moves over something like a pickup that stays there? It would get erased! Change how the Output command is used to erase the player/enemy when it moves.
- Make the map larger than the homescreen, perhaps multiple homescreens wide and tall. Figure out how to select which piece of the map the player is in at any given point, and figure out how to redraw the map when the player moves between sections. Hint: maintain both the onscreen position of the player and the offset of the top-left corner of the current screen from the top-left corner of the matrix.
- Try adding pickups, more items to be collected, and more obstacles.
- Add doors and puzzle minigames that must be solved to get through those doors.
- Give the player a weapon with which to fight the enemy.

This is certainly the longest program I've shown you so far, but it's on the shorter end of the spectrum for a complete TI-BASIC game. As you grow to write longer and more complex programs, remember to plan thoroughly and keep notes to avoid becoming confused and frustrated. As you try to expand this game or create your own programs and games, be methodical and careful, but above all, have fun!

I'll wrap up this chapter with a summary of what you've learned and give you a chance to take a deep breath before we begin the third and final part of the book.

9.6   Summary

As you read through this chapter, you learned about many of the numeric features and data types that are specific to your TI graphing calculator. Almost every programming language has For loops of some sort, conditional statements, and subprograms, concepts that you learned in previous chapters. Many have graphics commands to draw pixels or shapes. Most can also store one-dimensional and two-dimensional arrays of numbers or letters (lists and matrices) and have ways to store and manipulate strings of letters. Your TI-83+/TI-84+'s TI-BASIC language is unique in that it can perform math on full matrices and lists, understand real, complex, and imaginary numbers, and generate different types of random numbers with just the software with which it originally shipped. For such mathematically aligned features, most other programming languages require extra software or libraries to be installed.

But even with TI-BASIC these data types and numeric features aren't easy, and to understand them, you must try writing your own programs. In particular, to understand how lists can be vital to your programs and games, and to a lesser extent matrices, you must try using them in your games and see how they can help you. Try creating a custom-named list that holds information about a player's progress, such as health, levels, and current weapons. Experiment with using a list to hold a high scores table, and figure out the loop you'd need to use to insert a new high score at the proper place in the list. Lists and matrices have direct applications in math and science programs, so if those are more your cup of tea, I encourage you to give such programs a try with your newfound knowledge as well.

In the final chapters, I'll be discussing optimizing pure TI-BASIC, hybrid BASIC, and where you can progress from here with your programming hobby or career, so I once more encourage you to experiment, try, fail, and eventually succeed. Don't hesitate to post questions at the forums listed in appendix C, because you may well meet a concept that you can't wrap your head around without someone offering you a new perspective. With that in mind, let's move on to a recap and further lessons on optimizing your programs to be fast and small.

Advanced concepts; what's next

By the time you reach this section, you'll have created many TI-BASIC programs, and you may feel you've learned practically everything there is to know about graphing calculator programming. Not so fast! There's more in store for you!

Alternatively, you may have picked up this book with some preexisting TI-BASIC knowledge, so this part is where you'll really start encountering those moments of inspiration. Part 3 shows you cutting-edge optimization tips to speed up and slim down your programs, explains how to use hybrid BASIC libraries to bring a whole new bag of tricks to your programs, and introduces the z80 assembly programming language. It concludes with a look forward at how you can go further with general programming, calculator programming, and hardware development with the skill set you built throughout this book.

Chapter 10 introduces techniques to optimize TI-BASIC programs, including removing extraneous characters, simplifying logic, compressing stored data, and using the `Ans` variable. The extra features of hybrid BASIC are the first features you'll explore that aren't built into your calculator and are introduced in chapter 11. You'll see how the hybrid BASIC libraries can help you draw faster, fancier graphics, find and manipulate programs and files, and more. The language used to create those hybrid BASIC libraries, z80 assembly, is the topic of chapter 12. I'll show you enough about the underpinnings of assembly and a high-level view of the instructions and program-flow constructs

in z80 assembly to whet your appetite. The final chapter takes a look at where you can go from here as a graphing calculator programmer and how you might use your TI-BASIC knowledge as a stepping-stone into computer, mobile, and web programming. It also introduces hardware development as a field you might like to pursue.

10

Optimizing TI-BASIC programs

This chapter covers

- Making programs smaller and faster
- Using implicit conditionals, `Ans`, and other tricks
- Compressing numbers and lists of strings

Being a programmer means knowing not only how to write programs but how to write them *well*. In all of the chapters so far, I've emphasized the importance of creating programs that are not only correct but are also fast and small. Your TI-83+ or TI-84+ calculator has either a 6 MHz or 15 MHz processor, 24 KB of RAM, and somewhere between 160 KB and 1.5 MB of Archive (ROM). You can archive programs to free up RAM. When you run a program, it must fit entirely in RAM, which means that the largest TI-BASIC program you can run is smaller than 24 KB. It's important to be clever in order to fit complex interfaces on the relatively tiny screen of your calculator, but being sufficiently skilled to write small, fast code is at least as valuable. If your programs are faster, they can do more; if they're smaller, you can cram in more features. You've been learning skills up to this point to write code that's correct and efficient; in this chapter, you'll learn more tips to fit your programs within your calculator's RAM and make them fast enough to impress your programs' users.

227

In the chapters so far, you've seen optimization tips woven throughout the programming lessons. I showed you how you could omit ending quotes and closing parentheses that fall at the end of a line or right before a store ($\rightarrow$) operator. You also saw the idea of putting two commands on the same line separated by a colon, which doesn't save space but does reduce the amount of scrolling you need to use to get around your program.

In this chapter, you'll learn general optimization methods that you can apply to any TI-BASIC program. The first topic discussed is implicit conditionals, a way to compress conditional statements together. We'll look at the `Ans` variable, which I mentioned once before, and how it can save using other variables. You'll see how to compress multiple numbers and strings together, similar to how the FLIPCOIN program at the end of chapter 9 worked. I'll conclude with assorted tips that don't fit well into other categories.

You'll first learn about implicit conditionals. This trick takes variable updates controlled by conditional statements and combines them with their comparisons to save space and accelerate programs.

10.1   *Implicit conditionals*

The conditional constructions that you learned about in chapter 3 can also be called explicit conditionals. Here, the "explicit" means that there's a clearly defined comparison and one or more commands that run based on the value of the comparison. The conditional `If` statements that you've often seen are explicit conditionals. Implicit conditionals are usually used when you have an `If` statement that controls a single variable update, such as adding or subtracting a number or variable from a variable. In this section, I'll show you how to convert an explicit conditional statement to an implicit conditional, present two small examples, and show you a new version of the MOVECHAR program from chapter 6.

Let's begin with how to convert explicit conditionals into implicit conditionals.

10.1.1   *Converting explicit conditionals to implicit conditionals*

Implicit conditionals can easily be written from scratch once you understand the concept, but to get started, it's easier to convert an explicit conditional. Implicit conditionals take advantage of the fact that a Boolean false in TI-BASIC is just the number 0 and that true is the number 1. With an implicit conditional, you combine an `If` statement and an assignment statement that together form an explicit conditional. This conversion can be completed in two steps:

1 Identify the condition, the variable that's conditionally updated, and by how much the variable is updated when the original condition is true.

2 Write the implicit conditional so that the variable is updated to itself if the condition is false or to itself plus/minus/times/divided by the desired update value if it's true.

These two steps are applied slightly differently whether you're converting addition or subtraction or are instead working with multiplication or division.

	Example 1: Addition	Example 2: Subtraction
Explicit Conditional	`:If K=105` `:A+2→A`	`:If C=11 or C=12` `:B-10→B`
Implicit Conditional	`:A+2(K=105)→A`	`:B-10(C=11 or C=12)→B`
When K=9 and C=9	`:A+2(0)→A` `False`	`:B-10(`False` or `False`)→B` `(same as) :B-10(0)→B`
When K=105 and C=11	`:A+2(1)→A` `True`	`:B-10(`True` or `False`)→B` `(same as) :B-10(1)→B`

Figure 10.1 Turning explicit conditionals (first row) that control addition (left) and subtraction (right) into implicit conditionals, and how they work when the comparison is false (third row) or true (fourth row)

CONVERTING CONDITIONAL ADDITION AND SUBTRACTION

For conditional addition and subtraction, you want your new code to add or subtract the value of the desired change multiplied by the conditional, as shown in figure 10.1. When the conditional is false, this is like adding or subtracting 0 times the update value to/from the variable or adding or subtracting 0.

CONVERTING CONDITIONAL MULTIPLICATION AND DIVISION

For multiplication and division, you need a slightly more complex formulation, because you need to multiply or divide by 1 if the condition is false or divide by the desired update value if the condition is true. Figure 10.2 shows two more examples, the one at left demonstrating multiplication and the one at right showing division.

To determine the implicit conditional's coefficient on the target variable, subtract 1 from the desired multiplicand or divisor, use that as the coefficient for the true-or-false comparison, and add 1. Figure 10.2 shows how this works. If you want to multiply A by 3 when the comparison is true, you also want to multiply by 1 if the comparison is false. Similarly, you want to divide M by 5 when T > 5 and divide it by 1 otherwise, because division by 1 leaves M as it was.

	Example 3: Multiplication	Example 4: Division
Explicit Conditional	`:If K=105` `:3A→A`	`:If T>5` `:M/5→M`
Implicit Conditional	`:A(1+2(K=105))→A`	`:M/(1+4(T>5))→M`
When K=21 and T=4	`:A(1+2(`False`))→A` `(same as) :A→A`	`:M/(1+4(`False`))→M` `(same as) :M→M`
When K=105 and T=9	`:A(1+2(`True`))→A` `(same as) :3A→A`	`:M/(1+4(`True`))→M` `(same as) :M/5→M`

Figure 10.2 A demonstration of turning explicit conditionals for multiplication (left) and division (right) into implicit conditionals. Note that in order to preserve the original variable when the condition is false, the implicit conditional must either multiply or divide by 1.
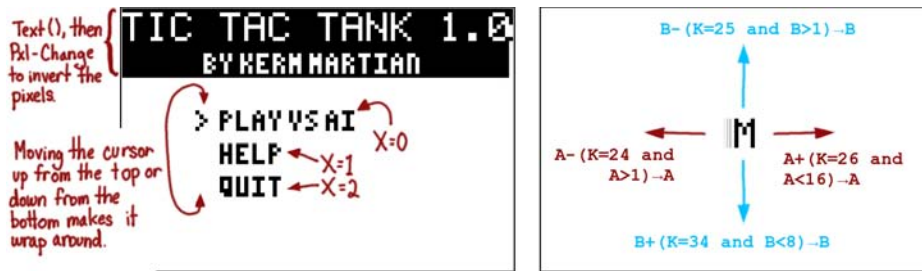
Figure 10.3   A menu with a cursor that can wrap around the top and bottom of the menu (left), and the 4-directional ICMVCHAR program using implicit conditionals (right), both described in the text

EXPLICIT TO IMPLICIT EXAMPLES: ABSOLUTE VALUE AND A MENU

Assume your calculator had no `abs` command, and you wanted to take the absolute value of variable V and store in back into V in a single line. The two-line code might be something like this:

```
:If V<0
:⁻V→V
```

If you wanted to change this to an addition/subtraction update, it might look as follows:

```
:If V<0
:V-2V→V
```

V – V = 0, and V – 2V = -V, so this update is identical to ⁻V→V. You now have a comparison (V<0) and an update (V-2V→V), so you can build an implicit conditional:

```
:V-(V<0)2V→V
```

Another example might be a menu that displays three items on the screen, such as Play, Help, and Quit, using the `Text` command. The left side of figure 10.3 shows such a menu.

A variable X, holding 0, 1, or 2, could be used to indicate which option was currently selected, and an arrow such as > or -> could be drawn next to the selected option with `Text` based on X. Assuming that you already have code to update X when [▲] (key code 25) or [▼] (code 34) was pressed, you also need to deal with what happens when X is moved above the first item (X = -1) or below the third item (X = 3). One popular option is to make the pointer wrap around: if the user presses [▼] at the last option, the arrow should reappear at the first option, and vice versa. Here it is with explicit conditionals:

```
:If X=⁻1
:2→X
:If X=3
:0→X
```

Once again, converting these straight stores to updates:

```
:If X=⁻1
:X+3→X          ⟵⎯⎯| X must be -I, so -I + 3 = 2
```

```
:If X=3
:X-3→X
```
        ⊲────┘ **By the same logic, X must be 3, so 3 – 3 = 0**

The conversion to implicit conditionals follows the same strategy you've seen thus far:

```
:X+(X=⁻1)3→X
:X-(X=3)3→X
```

As a final optimization, you can even combine these, because the second update doesn't depend on the result of the first:

```
:X+(X=⁻1)3-(X=3)3→X
```

Now that you've seen the mechanism behind creating implicit conditionals out of explicit conditionals, plus a few example fragments, let's move on to a full program that uses implicit conditionals to move an M around the homescreen.

10.1.2 *Implicit conditionals for four-directional movement*

In chapter 6, you worked with the Mouse and Cheese game, in which you learned how to move an M around the homescreen. The mouse was at column A, row B on the homescreen, and its position was updated by a keypress in K. The code for updating A and B based on K looked something like this:

```
:If K=24 and A>1
:A-1→A
:If K=26 and A<16
:A+1→A
:If K=25 and B>1
:B-1→B
:If K=34 and B<8
:B+1→B
```

As you can see, each position update is dependent on two checks. The proper key must have been pressed, and the bounds check must be true to avoid the mouse going out of bounds off the edge of the screen. Because true and false are just numbers, something like (K = 24 and A > 1) is equal to 1 if true or 0 if false. With the first update, decreasing column A, you want to leave A as it is if K ≠ 24 or A = 1 or subtract 1 if the anded comparisons are both true. Therefore, you can just subtract the pair of comparisons from A and store that back into A. When K = 24 and A > 1, then the anded condition statement is equal to 1, and A will decrease. If not, the line is equivalent to A-0→A, and A doesn't change. You can follow the same logic for the other three conditional/update statements, as also illustrated on the right in figure 10.2:

```
:A-(K=24 and A>1→A
:A+(K=26 and A<16→A
:B-(K=25 and B>1→B
:B+(K=34 and B<8→B
```

You can compress this even further by recognizing that even though there are two lines that work with A, it's not necessary for the first update to A to occur in order for the second to run, because your program will never be moving A both left and right in

the same iteration of the game loop. Therefore, you can combine the two A updates and the two B updates:

```
:A+(K=26 and A<16)-(K=24 and A>1→A
:B+(K=34 and B<8)-(K=25 and B>1→B
```

Let's put this into a program to demonstrate the implicit conditional in action, a version of the MOVECHAR program from chapter 6 called ICMVCHAR. The following listing shows the source code for this program, complete with two compound implicit conditional lines that update the row and column of the M.

> **Listing 10.1   An implicit conditional program to move a character, program ICMVCHAR**

```
PROGRAM:ICMVCHAR
:8→A:4→B
:ClrHome
:Repeat K=45
:Output(B,A,"M
:getKey→K
:If K
:Output(B,A,"[one space]       ←—— Adjust column
:A+(K=26 and A<16)-(K=24 and A>1→A   ←—— Adjust column
:B+(K=34 and B<8)-(K=25 and B>1→B    ←—— Adjust row
:End
```

As always, you should try this out on your calculator to understand how it works. As an extra challenge, you could try converting one of the eight-direction character movement programs from chapter 6 to use implicit conditionals, and see how small you could make it.

Implicit conditionals are merely one space-saving/time-saving trick in our bag of optimizations. The next technique I'll show you is the `Ans` variable and how it can be used.

10.2   *Exploiting Ans*

The `Ans` variable is perhaps the most versatile variable your calculator offers to TI-BASIC programmers. Unlike other variables, it can be any type: a real or complex number, a string, a list, or a matrix. In this section, I'll show you three cases where `Ans` can be used to save variables and shorten your programs. You'll first see the simple usage of `Ans` to save intermediate variables. I'll then demonstrate how `Ans` can be used to simplify `If/Then/Else/End` conditionals and finally how `Ans` is used as the argument and return value to and from subprograms.

First, we'll look at using `Ans` to simplify mathematical expressions and conditionals.

10.2.1   *Ans to save variables and conditionals*

In many programs, you'll need to calculate the value of an expression and use that value in the next line of your program. You could use a variable, but if you're not going to use that value again, it's a waste of a variable and a waste of the bytes that a storage operation like →T takes in the source code of your program. Any time you

have a math expression or an assignment (store) expression, the value of the `Ans` variable is changed to the result. All of these expressions modify `Ans`:

```
:3
:4(X+5→N
:"HELLO"
:Ans+" WORLD"→Str3
```

⟵ **Now Ans is a number, and Ans = 3**

Now Ans is a number, and N and Ans contain the same value

Now Ans and Str3 are the string "HELLO WORLD"

Now Ans is the string "HELLO"

Notice that it's never necessary to explicitly store to `Ans`, and if you try to do so, you'll get an ERR:SYNTAX:and storing Ans error. You can use `Ans` to save variables and simplify expressions, as I'll show you in three steps.

SIMPLIFYING MATH EXPRESSIONS WITH ANS

Consider some fictitious equation involving the trigonometric functions `sin` and `cos`:

```
:sin(3cos(θ))+4sin(3cos(θ))sin(3cos(θ))+5→X
```

This is a fairly long expression. You could use a temporary variable to help yourself:

```
:sin(3cos(θ→T
:T+4T²+5→X
```

Because sin(3cos(θ))sin(3cos(θ)) = T ∗ T = T²

With `Ans`, you can shorten this even more while removing the use of the T variable:

```
:sin(3cos(θ
:Ans+4Ans²+5→X
```

This final pair of lines is 13 bytes together; the original single line was 24 bytes. The intermediate solution using temporary variable T was 15 bytes and wasted T.

You can also use the fact that a number, string, list, or matrix expression by itself on a line, even if it's not explicitly stored into anything, will update the contents and type of `Ans`. Let's now examine how `If/Then/Else/End` expressions can be shortened with this trick.

SIMPLIFYING CONDITIONALS WITH ANS

In many programs, the value of a comparison or conditional statement will dictate which of two similar commands is executed. Consider the following two examples, both of which use `If/Then/Else/End` constructs:

```
:If B²-4AC<0              :If A=15 and B=7
:Then                     :Then
:Text(4,18,"IMAGINARY ROOTS    :3→W
:Else                     :Else
:Text(4,18,"REAL ROOTS     :0→W
:End                      :End
```

The code on the left determines whether the roots of a quadratic equation $AX^2 + BX + C = 0$ are real or imaginary. The code on the right might be one way to check if the user won in the mini-RPG from chapter 9.

The key insight to simplifying these conditionals is that you can start by assuming that the false case will occur, modify `Ans` based on that assumption, and then only modify it again if the true case is executed. In the left-hand code, assume that the

"REAL ROOTS" case will occur, so put `"REAL ROOTS"` on a line by itself to store that to `Ans`. Then, only overwrite `Ans` with `"IMAGINARY ROOTS"` if it turns out that $B^2 - 4AC$ is indeed negative. `Ans` can then be used in a single `Text` command, as shown here at left:

```
:"REAL ROOTS                        :0
:If B²-4AC<0                         :If A=15 and B=7
:"IMAGINARY ROOTS                    :3
:Text(4,18,Ans                       :Ans→W
```

By the same token, for the code at right, start by setting `Ans` to 0, on the assumption that the conditional won't be true. If it is, `Ans` can be overwritten with 3, and whatever `Ans` contains (be it 0 or 3) is stored to W. Why not use an implicit conditional for the explicit conditional on the right side? You could indeed, and it would be up to you the programmer to decide whether to use the explicit form shown previously or this implicit form:

```
:3(A=15 and B=7→W
```

`Ans` is a powerful optimization for simplifying mathematical, string, list, and matrix manipulation and for shortening conditional constructs. Another common use is as the argument and/or return value to and from subprograms.

10.2.2 *Ans with subprograms*

As introduced in chapter 4, subprograms are a great way to avoid having to repeatedly type out sections of code. By putting the code in question into a separate program and calling that program, you can execute the code inside the subprogram. In some cases you may want to give your subprograms an argument, a number, string, or other data type that dictates what the subprogram does. In some cases, you'll want your subprograms to provide a value back to the main program. `Ans` can be used effectively for both these cases.

Your subprograms may be responsible for displaying some graphical element, as in chapter 4, where a subprogram was used to draw a border around the homescreen. A subprogram may be used to repeatedly apply some formula to input values to get some output value or values. You could use a variable such as Q or L_1 or `Str8` as input to the subprogram and another variable for output, but as with any program or piece of code, this is wasteful if you can exploit `Ans` instead. A good solution is to use `Ans` as the argument or arguments to the subprogram and reuse `Ans` as the return value or output value of the program.

Consider a simple subprogram that calculates the length of the hypotenuse of a triangle, given the lengths of the other two sides. As you saw in chapter 5, when we discussed Pythagorean Triplets, the Pythagorean Theorem ($A^2 + B^2 = C^2$) can be used to calculate the length of the hypotenuse of a right triangle. Indeed, a subprogram might take A and B as arguments, apply the Pythagorean Theorem to calculate C, and return C:

```
PROGRAM:ZHYPTNS1
:sqrt(A²+B²→C
```

Using `Ans` as the arguments and return value would save the use of variables A, B, and C. But wait! This program takes two arguments, and there's only one `Ans`. What will we do?

We can solve this conundrum by using `Ans` as a two-element list, the first element of which is A and the second of which is B. Remember, `Ans` can be just about any data type, such as a number, a list, a matrix, or a string. When the code that calculates the length of the hypotenuse executes, it will update `Ans` to be a number, the return value:

```
PROGRAM:ZHYPTNS2
:sqrt(Ans(1)²+Ans(2)²
```

As you can see, just as you can select elements from lists L_1 or $_LTHIS$, you can select elements from `Ans` as `Ans(1)` and `Ans(2)` when it's a list. If `Ans` contains the lengths of two triangle legs, this code calculates the length of the third, and because it doesn't specifically store it anywhere, only `Ans` is updated and in this case changed to a number first as well. A simple "driver" program to use this subprogram might look like the following listing.

> **Listing 10.2 Using an `Ans`-based subprogram to calculate a triangle's hypotenuse**

```
PROGRAM:HYPOTNUS
:Prompt A,B
:{A,B                    ⟵——— "Store" a two-element list to Ans as the
:prgmZHYPTNS2                   argument to the subprogram, just like {A,B→LI
:Disp "HYPN LENGTH:",Ans ⟵——— Display the return value, also Ans
```

This is a small subprogram, but this technique could be applied to much more complex subprograms as needed.

`Ans` is useful for reducing the numbers of variables that you use and shaving off a few bytes here and there from your code. In the next section, we'll look at larger-scale techniques for compressing data and code.

10.3 Compressing numbers and choices

A recurring theme throughout the preceding chapters has been using the limited memory and processing speed of your graphing calculator efficiently. On any platform, it's important to optimize your code for size and speed, but it's also important to shrink the external data that your program uses or creates; a graphing calculator is no different. In this section, I'll show you a trick to fit several numbers into a single numeric variable (or list or matrix element). You'll then learn two ways to shrink your code, one for selecting different strings based on conditionals and a second for shrinking a long series of `and` and `or` operations.

Let's first look at fitting multiple integers into a single number.

10.3.1 Compressing numbers

From scores and positions to health and coefficients, numeric variables serve a multitude of duties in your TI-BASIC programs. You've already seen lists and matrices in

appendix A and chapter 9 and how they can be used to store many numbers together. But every number, whether it's in a variable, a list, or a matrix, takes up memory. Numeric variables are 18 bytes each, whereas lists and matrices are 9 bytes per element, regardless of what sorts of numbers are stored in each element. If a list contains {0, 0, 1}, it's the same size as a list holding {0.45246, 1.553, -4569.31}. When you need to store a lot of simple numbers at once, you can compress multiple numbers into single variables or list elements.

Because your calculator can store up to 14 digits for each number, plus an exponent and a sign, you can compress several integers into a single variable or list or matrix element. The key to the technique is that if you multiply one integer by a power of 10 (such as 100 or 10000), you effectively insert zeroes at the end of the number. If you then add that to another integer, the digits of the two numbers will appear concatenated, but you're still working with a single number. The left side of figure 10.4 demonstrates multiplying 42 by 100 to insert two zeros at the end, then adding 93, so that both 42 and 93 are compressed into the single number 4293. If you wanted to fit more integers into that number, you could multiply by 100 again, making it 429300, and add another one- or two-digit integer. `prgmCOMPRESS` in listing 10.2 shows packing seven such integers into the number N.

The right side of figure 10.4 shows how you can extract those numbers out again. You first divide by 100, so that the number you want to remove is on the right side of the decimal point, then use `fPart` to extract it, and multiply by 100 to return the original number. As with compression, you can repeat this process to extract as many integers as you have compressed into the longer number. `prgmDECMPRSS` in listing 10.3 takes the number N generated by `prgmCOMPRESS` and extracts the seven integers

You need not deal with only two-digit integers. You could easily fit in three four-digit numbers instead, but you'd need to multiply and divide by 10000 instead of 100, because you'll need to shift left and right by four digits at a time rather than two. This technique is most easily applied to integers, but many of the variables in your programs will be integers anyway.
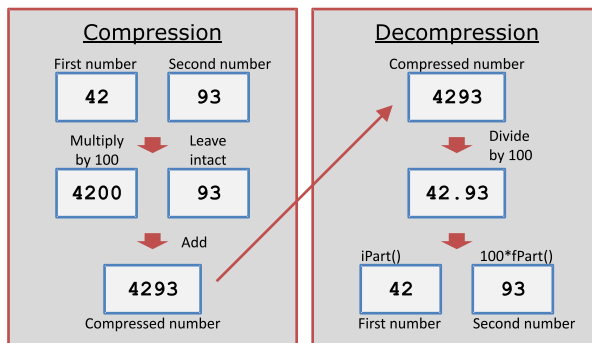
Figure 10.4   **Compressing two integers into a single number and then decompressing them. You can safely fit 14 digits into a Real, list element, or matrix element, so you could fit 2 seven-digit integers, 7 two-digit integers, or even 14 single-digit integers.**

Listing 10.3   A pair of programs to pack/unpack seven integers into a single number

```
PROGRAM:COMPRESS                    PROGRAM:DECMPRSS
:0→N                                :For(X,1,7
:For(A,1,7                          :100fPart(N/100
:Disp "ONE OR TWO-DIGIT             :Disp Ans
:Input " INT:",X                    :iPart(N/100→N
:100N+X→N                           :End
:End
```

This particular technique is good for data that will be compressed infrequently, stored for a while, and decompressed infrequently. It's generally too slow for real-time use within your programs and games. But the remaining two topics in this section should be used as often as possible to slim down and speed up your programs. First, I'll show you how to combine string options.

10.3.2 *Compressing string options*

In many programs, you'll want to display one of several strings based on some number. In chapter 9, you saw a program that displayed "HEAD" or "TAILS" depending on whether `rand` returned a number above or below 0.5. Another example might be an RPG that could display a class such as "WIZARD," "ROGUE," or "FIGHTER." Consider a program that puts one of these three strings in `Ans` depending on whether X is 1, 2, or 3. This program might look like the code at left; if you use the optimizations from section 10.2, you'll get the code at right:

```
:If X=1                 :"WIZARD
:"WIZARD                :If X=2
:If X=2                 :"ROGUE
:"ROGUE                 :If X=3
:If X=3                 :"FIGHTER
:"FIGHTER
```

But you can use the `sub` command to algorithmically select one of these three strings. You need to perform three steps to use this trick:

1 Find the length of the longest string among the choices. Put spaces at the end of the shorter string choices to make all the choices equal in length. For example, `"WIZARD "`, `"ROGUE "`, and `"FIGHTER "`, are all seven characters long.

2 Concatenate the strings into a single string: `"WIZARD ROGUE FIGHTER"`

3 Use the length of each choice as the length argument for `sub`, and design the offset argument to select the correct substring based on the number used to choose which string should be returned.

The third step is the most challenging. For our example, we want seven characters beginning at character 1 to get "WIZARD" if X = 1, seven characters at character 8 if X = 2, and seven characters at character 15 if X = 3. The final command looks like this:

```
:sub("WIZARD ROGUE  FIGHTER",7X-6,7
```

The 7X-6 takes care of properly mapping X = {1,2,3} to the desired {1,8,15}. I got this expression from knowing that each substring we'll want to grab is seven characters long, so 7X will return a number (an offset into the string) that's 7 characters further as X is increased by 1. But 7X by itself would yield {7, 14, 21} for X = {1, 2, 3}. We want {1, 8, 15}, so we can subtract 6 each time, hence 7X − 6. Note that this general technique only works well if the spaces at the end aren't a problem for your program or game; if they are, you should stick with the explicit conditional code.

Here's the "HEADS"/"TAILS" solution from chapter 9, demonstrating the same technique:

```
:sub("HEADSTAILS",1+5(rand>0.5),5
```

Here, each option is 5 characters long, and the conditional (rand>0.5) is either 0 or 1, so 1+5(rand>0.5) maps {0,1} to {1,6}, as needed to display "HEADS" or "TAILS." Here are some extra examples from real programs:

```
:sub("0123456789",1+X,1
:sub("SUNMONTUEWEDTHUFRISAT",3X-2,3
:sub("FALSETRUE ",5(X>Y)+1,5
:sub("EASYMED HARD",4D-3,4
```

For a final compression tip, let's look at how you can shorten long conditionals that include a series of comparisons ored or anded together.

10.3.3 *Compressing or and and*

As your programs get more complex, you'll find conditional structures that you want to execute when one of a range of comparisons is true or that require a set of comparisons to all be true. Your training thus far from chapter 3 forward indicates that you'll need several comparisons separated either with or or and. But this can get quite cumbersome. Consider the following code, which will display "NUMBER KEY" if you press any of the number keys, [0] through [9]:

```
:Repeat K
:getKey→K
:End
:If K=102 or K=92 or K=93 or K=94 or K=82 or K=83 or K=84
➡  or K=72 or K=73 or K=74
:Disp "NUMBER KEY
```

Keycodes for [0] through [9] as in chapter 6

This code waits until a key is pressed; once a key is pressed, the loop ends, and if the key was a number key, the Disp line executes. It would be far preferable to be able to compress this line. Fortunately, min and max from the LIST MATH menu, [2nd][STAT][◄], can come to the rescue.

You can compress or and and whenever you have a single variable (or number) that you're comparing to several numbers (or variables), where the comparisons are joined by and or or operators. To replace and, use the min command; for or, use max. The steps require starting with the convoluted conditional as shown previously and breaking it into pieces:

1. Identify the common number or variable in all of the equality comparisons. In the previous example, this is variable K. If there's no such commonality, you can't use this.

2. Construct a list of all the things you're comparing to that variable (or number). For the number key example, this would be {102, 92, 93, 94, 82, 83, 84, 72, 73, 74}.

3. Complete the new conditional with `If max(` or `If min(` followed by the item from step 1, an equals symbol, and the list from step 2, for example:

```
:If max(K={102,92,93,94,82,83,84,72,73,74
```

For another example, consider making sure that variables A, B, and C are all equal to 2. The long form of this conditional might be

```
:If A=2 and B=2 and C=2
```

To simplify this, you can follow the same three steps. As per step 1, the common item in all of the equality comparisons is the number 2. Step 2 instructs you to make a list of all the things being compared to 2, specifically {A,B,C}. As step 3 directs, you put it together, using `min` because the original used `and`:

```
:If min(2={A,B,C
```

HOW DOES IT WORK?

When you compare a number or variable to a list, you create a new list of the same length. This new list contains the Boolean values (0 or 1) indicating what the results would be if you individually compared each list element to the outside value. Every true comparison will result in a 1 in this new list; false comparisons will yield a 0. When you `and` several comparisons together, you want to make sure that all of the comparisons are true, which would mean the new temporary list generated from the new comparison you constructed would have to contain all 1s. The minimum (and maximum) value in this list would be 1. But if any of the comparisons was false, there would be a 0 in the new list, and the minimum of the list would be 0. Thus, `If min(…)` would yield 0, because it implies `If min(…)≠0`.

If at least one of the comparisons is true, there will be one or more 1s in the new list. For `or`, you use the `max` operator, because any 1s in the new list will make `If max(…)` also true. As you master TI-BASIC, you may even discover ways to expand this trick to cover more ways to manipulate multiple comparisons as lists. In the interest of space, rather than go deeper into these compression techniques, we'll conclude with a medley of optimization tips and tricks.

10.4 *Space-saving tips and tricks*

In the preceding chapters, I mentioned tips to speed up and slim down your programs; these tips are important enough to bear repeating. In this section, I'll reiterate those tips as well as present a few other techniques that will add up to more efficient and user-friendly programs.

In the preceding chapters, I repeated two particular "best practices" that makes your programs smaller and easier to maintain.

10.4.1 Shortening your programs

Of these two tips, the first will help you make your completed programs fit into fewer bytes, the second leaves your program the same size but will make it easier for you to scroll up and down your program and reach errors. The first tip is to remove closing quotes and closing parentheses that are immediately followed by a store operator ([STO], or →) or a new line ([ENTER]). You can also remove closing square brackets and curly braces ( ] and }). Each removed symbol will save 1 byte in the program, and you can even remove several closing symbols at the same time. The following two lines are equivalent:

```
:Text(4,5,"HELLO")
:Text(4,5,"HELLO
```

Closing parenthesis removed first, which means closing quote can be removed too

You can also rearrange lines to help you remove more closing punctuation, such as switching the items on either side of an equals or not-equal sign. Here are another two equivalent lines:

```
:If L₁(3)=3 or L₁(3)=4
:If L₁(3)=3 or 4=L₁(3
```

By reversing 4 and $L_1(3)$ that are being compared, the closing parenthesis can be removed. A final note: you can't remove closing punctuation before a comma, such as the commas that separate the arguments to a command.

The other tip is that you can put two or more commands on the same line by separating them with the colon (:) character, under [ALPHA][.]. A colon is similar to a new line in TI-BASIC; the only difference is that you can't remove closing parentheses and quotes before a colon. Combining multiple lines using colons doesn't make your program faster or smaller (though the technique also doesn't make your program slower or larger). Instead, by reducing the number of lines in your program, it reduces how far you need to scroll in order to reach an error or to insert more code. You can also scroll more quickly by entering Alpha-Lock mode before pressing the [▲] and [▼] arrows.

You've seen both of these tips before, but there are many more methods of removing bytes from programs while making them faster and more efficient that programmers have developed over the years. Let's take a brief look at some of the outstanding ones.

DELETING VARIABLES AND SETTING VARIABLES EQUAL TO ZERO

In many of your programs, you may want to set a variable equal to 0, such as 0→V. This takes 3 bytes (4 if you include the colon or [ENTER] after the assignment). TI-BASIC has a command called DelVar which can be used for the same purpose, and which takes only 2 bytes in your program. DelVar is used to delete one of the number variables, namely A–Z and θ, which not only saves 21 bytes of your calculator's RAM if you

don't use it again but also makes the TI-OS act as if that variable exists and is equal to 0 if you use it again. `DelVar` is also unique in that you can put any command directly after it without needing a new line or a colon. These two pieces of code both set A and B to 0 and display "HELLO" on the screen.

```
:0→A:0→B              :DelVar ADelVar BDisp "HELLO
:Disp "HELLO
```

REMOVING =0 AND ≠0

You've seen this particular tip once or twice before, but I only discussed it in passing. If you're comparing a variable or an expression to 0, using either an = or ≠ symbol, you can remove the `=0` or `≠0` thanks to the fact that Boolean true and false values are equivalent to the numbers 1 (or any nonzero number) and 0 respectively in TI-BASIC. If you want to ensure that a variable or expression is non-zero, you could use `If A` instead of `If A≠0`. Because any non-zero number is true, `If A` will execute its conditionally controlled statement or block whenever A is not 0. If you want to instead make sure that a variable or expression is equal to 0, you might try `If A=0`. To save space, you can instead use `If not(A`, because the `not` operator converts any non-zero number to 0 and 0 to 1. When A is 0, `not(A)` is 1 (or true), just as A=0 is true when A is 0. When A is not 0, `not(A)` is 0 (or false).

HIDING THE RUN INDICATOR

The run indicator, the scrolling group of pixels that appears at the top-right of the LCD when a TI-BASIC program is busy, as shown at the left in figure 10.5, can be helpful for users to know when your program is working hard. But it can also be distracting in some cases or make your program look less professional. Luckily, the large-font feature of the `Text` command can be used to draw over the run indicator, as at the right in figure 10.5, but this command must be run repeatedly in your inner event loop:

```
:Text(¯1,0,90,"[one space]
```

USING UPPERCASE LETTERS

If you use a shell that lets you put lowercase letters in your programs, the programs can look more refined to users. But be aware that lowercase letters take 2 bytes, uppercase letters take 1 byte. This means that a string written in all lowercase letters will take
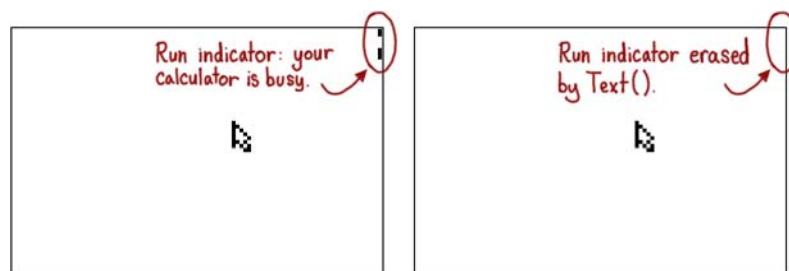
Figure 10.5   The run indicator (left) and using the `Text` command to remove it (right)

twice as much memory as a string written entirely in uppercase letters. Lowercase can be used effectively and is often worth the space, but be sure you're using it judiciously. In addition, every token such as `Disp`, `Degree`, or `and` takes up either 1 or 2 bytes, no matter how long it is. You can use these tokens in your strings to save space.

FINAL OPTIMIZATION THOUGHTS

There are many more optimization tricks that you'll see in others' code or that you may discover on your own. At the risk of sounding like a broken record, experiment with your code and with others' code, don't be afraid to break programs, and learn to see what makes programs fast and small and what just makes your code confusing. Shorter programs are often also faster, but this isn't always the case, so choose for yourself whether speed or size is more important to you for each of the programs you write.

10.5   *Summary*

Throughout the preceding chapters, you've read repeatedly about the importance of making programs for any platform fast and small and how writing good calculator programs in particular requires careful attention to such details. In this chapter, we worked through simple and through advanced tricks to shorten and accelerate your programs, from removing extraneous punctuation to compressing comparisons into lists. It's vital to practice these techniques in your own programs to fully internalize them, even more than for the programming commands you've learned. In addition, unlike many of the lessons you've learned to date, the optimization tricks in this chapter are customized for the TI-83+/TI-84+, although the important lesson of thinking creatively when trying to improve your code is a universal constant in programming.

This chapter concludes your introduction to pure TI-BASIC. The next chapter will begin an all-too-brief look at how TI-BASIC can be expanded with extra tools, what other languages you can write in, and what else besides software development you can do with your calculator.

Using hybrid TI-BASIC libraries

11

This chapter covers

- Understanding the xLIB, Celtic III, DCSB, PicArc, and Omnicalc libraries
- Using sprites and tilemaps in hybrid TI-BASIC, demonstrated with a full game
- Manipulating programs and data files via hybrid libraries

The TI-BASIC language provided by your calculator has been the focus of the chapters thus far. We've been working with only the capabilities that your calculator included when you first bought it, and you've seen many features you can use in your programs, from simple to complex. You've been able to do math, write games, get input and display output, and work with graphics. But if you've tried to get particularly advanced with your graphics as in figure 11.1, you may have found that it's impossible to make an extremely graphical game also be very fast. If you've investigated working with subprograms, you may have been frustrated to find you have no way of moving files between RAM and Archive. The limitation of having only 10 picture variables to use might have led you to scale back ambitious program designs. Around 2004, TI-BASIC coders just like you decided to do something about those limitations, and hybrid BASIC was born.
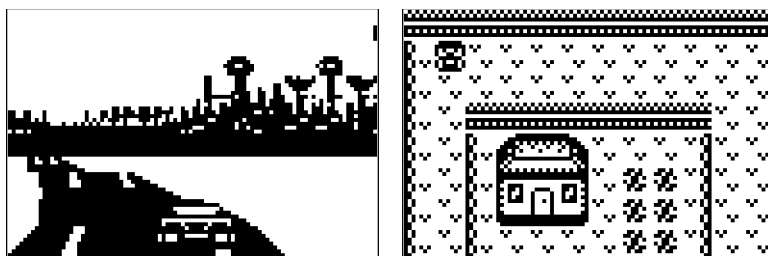
Figure 11.1   Examples of graphics created with hybrid BASIC libraries by coders Kevin Ouellet (left) and Patrick Prendergast

Hybrid BASIC is a term describing a TI-BASIC program that makes use of hybrid libraries. Hybrid libraries are written in z80 assembly, a more complex but more powerful programming language that I'll show you in chapter 12. These libraries offer many functions to TI-BASIC programs that the programs wouldn't otherwise be able to access, including quickly drawing small pictures to the graphscreen, copying archived programs to RAM, modifying the contents of programs, getting a list of all the programs on the user's calculator, accessing up to 255 picture variables, and much more.

In this chapter, you'll learn some of the particularly attractive functions of the hybrid BASIC libraries xLIB and Celtic III, both of which are included inside the Doors CS shell. I'll first explain getting Doors CS on your calculator and where to read full documentation about the functions it offers TI-BASIC programmers. We'll then look at two hybrid BASIC techniques you can use for fast, complex graphics in your programs and games, followed by a section on finding and running programs from other programs. The chapter will conclude with an abbreviated introduction to a few of the other handy features of hybrid libraries.

Let's begin with a more in-depth look at what exactly hybrid TI-BASIC is, what it means for you as a programmer, and what it means for your users.

11.1   *Introducing hybrid TI-BASIC*

The simplest explanation of hybrid BASIC is TI-BASIC expanded with functions from third-party libraries (that is, those not written by Texas Instruments). Each library is a collection of related functions that a TI-BASIC program can use but is not itself a program that can be run. The five major libraries are xLIB, Celtic III, the DCSB Libs ("Libraries"), PicArc, and Omnicalc, listed in table 11.1. All five have been rolled into a single entity, the shell called Doors CS 7, incidentally written by me. You're more than welcome to download each of these individually, but some of the original versions are known to have bugs and incompatibilities. xLIB and Celtic III don't work properly under the latest TI-84+ OS versions, the so-called MathPrint versions 2.53MP and 2.55MP.

For this reason, I'll walk you through using Doors CS for your hybrid BASIC needs. I'll show you how to get Doors CS and make sure the hybrid BASIC library tools are enabled, and then I'll show you what you need to add to the beginning of your programs so that

they can ensure the libraries they need are present on the user's calculator. For your reference as you go through this chapter, the two most important documentation links are these:

- http://dcs.cemetech.net/?title=Hybrid_Libs
- http://dcs.cemetech.net/?title=DCSB_Libs

11.1.1 Downloading the hybrid libraries

Four of the five libraries in table 11.1 were originally released as separate applications, each of which takes up memory on your calculator. To save space and add even more functions for TI-BASIC programmers to use, the Doors CS shell, version 7.0 and higher, includes all five libraries.

Table 11.1   The hybrid libraries you'll need from Doors CS, including each library's main purpose. Each library is a group of related functions, and all of these libraries are built into the single Doors CS application.

Library	Original author	Functions
xLIB	Patrick Prendergast ("tr1p1ea")	Graphics functions for games: extra pictures, tilemaps, sprites, scroll the screen. Also special text-output modes and the ability to run archived programs.
Celtic III	Rodger Weisman ("lambian")	Primarily manipulating data, strings, numbers, and programs.
DCSB Libs	Christopher Mitchell ("Kerm Martian")	Primarily GUI (graphical user interface) functions, including mouse functions, windows, and forms.
PicArc	Rodger Weisman ("lambian")	Like xLIB, graphics functions for programs, including "databases" of pictures, sprites, screen scrolling, and advanced text drawing.
Omnicalc	Michael Vincent ("Michael_V")	*(Partial support in Doors CS)* Sprites and executing assembly code.

You need to send DoorsCS7.8xk to your graphing calculator. You can download Doors CS from http://dcs.cemetech.net; at the time of writing, Doors CS 7 is the latest version, although further versions may be developed in the future. Appendix A describes how to send files to your TI graphing calculator, either with a USB SilverLink if you have a TI-83+ or TI-83+SE, or with a mini-USB cable for a TI-84+ or TI-84+SE.

Unfortunately, if you can't send files from your computer to your calculator, or you can't get Doors CS or other hybrid library apps from a friend's calculator, you won't be able to write (or run) hybrid BASIC programs. Doors CS and all of hybrid libraries are assembled applications, which means that their content would be meaningless viewed in the TI-BASIC editor. A TI-BASIC program, on the other hand, could be copied out line by line. Therefore, there's no way to type out Doors CS by hand on your calculator.

Once you have Doors CS, run it once from the [APPS] menu, and hybrid programs will work. You can run programs from inside Doors CS; the shell also installs a HomeRun

hook that lets Doors CS intercept programs when they're run from the homescreen and provide the hybrid libraries to them. If you have any difficulties, the Doors CS manual zipped with the .8xk file can help. The Doors CS SDK linked in appendix C, section C.2, also contains in-depth documentation of the hybrid functions, as do the links at the top of section 11.1.

11.1.2   Calling hybrid functions

Two pieces are required to use hybrid BASIC library functions in your programs. One is that whenever you want to run a hybrid command, you need to call it. The five hybrid libraries listed in table 11.1 all reuse TI-BASIC commands, so you use the `sum` command to call DCSB Lib functions or the `real` command to call xLIB functions. In the following sections, you'll see the arguments for each command.

The second piece is to make sure that the hybrid libraries that provide the functions your program needs are on the user's calculator. If you distribute your programs over the internet, you can add a readme.txt file that explains how to use your program, that it needs a shell for hybrid libraries, and how to contact you with questions. Because many programs are shared between calculators, your programs need to check if all required hybrid libraries are available and refuse to run if not. I use the following code at the beginning of my programs:

```
:If 1337≠det([[42
:Then
:Disp "NEED DOORS CS","DCS.CEMETECH.NET
:Return
:End
```

You could instead have this code instruct the user to read your readme or to contact you for help. The first line of this section of code is one of the hybrid commands. The 1 x 1 matrix `[[42]]` is passed to the `det` (determinant) command, which returns 42 if no libraries are present or 1337 if Doors CS intercepts the function, indicating that all five of the libraries in table 11.1 are available.

Now that you have Doors CS on your calculator for its hybrid libraries and know how to make sure your programs' users know how to get it too, let's look at how to draw small pictures quickly with hybrid libraries.

11.2   Working with hybrid sprites

In game design, a *sprite* is a small picture that you frequently draw (and perhaps erase) at different locations on the screen. If you were writing an RPG game, then your character and enemy characters would be sprites. If you worked with our mouse program from chapter 7, the cursor was a sprite as well. On a TI-83+ or TI-84+, sprites are always multiples of 8 pixels wide and any number of pixels tall. The most common size is an 8-pixel by 8-pixel sprite. In this section, I'll teach you the two ways you can encode and draw sprites with hybrid BASIC, focusing on the method where you encode sprites in your program in hexadecimal format. I'll then show you how to update the
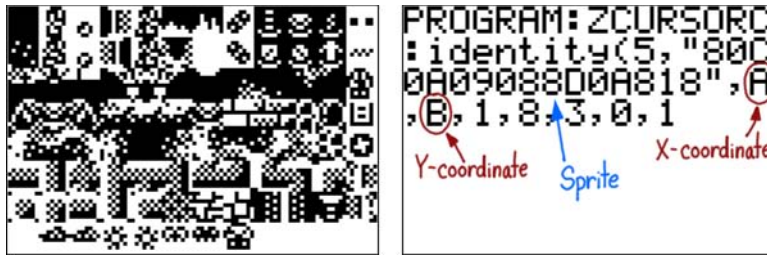
Figure 11.2   An example of a spritesheet (by Brad Sparks) made of many 8-pixel by 8-pixel sprites (left) used in one method of drawing sprites with hybrid BASIC. Sample source code (right) that defines a sprite as hexadecimal code (see figures 11.3 and 11.4) and draws the sprite.

CURSOR program from chapter 7 to be smaller and run faster using the hybrid BASIC sprite commands.

Let's begin with the two ways you can define and draw sprites in hybrid BASIC.

11.2.1 Defining and drawing sprites

Sprites in hybrid TI-BASIC can be drawn two possible ways. You can either copy a sprite from a predefined image called a spritesheet onto the screen, or you can directly draw a single sprite's worth of data onto the screen.

As shown at left in figure 11.2, you can define a picture variable containing all of the sprites you want to use. This picture is called a spritesheet, and you specify one of the sprites in the sheet to be drawn at a given place on the screen. The xLIB function known as `DrawSprite` is used for this method; `real` is under the CPX tab of the [MATH] menu:

```
:real(1,Sprite_X,Sprite_Y,Sprite_Width,Sprite_Height,Pic_Num,
➥ Pic_X,Pic_Y,Method,Flip,Update_LCD)
```

All xLIB functions are accessed through the overridden `real` command; `DrawSprite` is command number 1 in the xLIB library. The other arguments are:

- `Sprite_X`—The pixel column for the upper-left corner of the sprite, 0 to 95.
- `Sprite_Y`—The pixel row for the upper-left corner of the sprite, 0 to 63.
- `Sprite_Width`—The width (in 8-pixel groups) of the sprite. An 8-pixel-wide sprite would have width 1 here, and a 24-pixel-wide sprite would have width 3.
- `Sprite_Height`—The height (in rows) of the sprite. An 8-pixel-tall sprite has height 8.
- `Pic_Num`—The number of the `Pic` variable used as the spritesheet containing the desired sprite.
- `Pic_X`—The X-position of the desired sprite within the spritesheet. Also in 8-pixel groups, so `Pic_X=4` means that the sprite starts at column 32 in the spritesheet.
- `Pic_Y`—The Y-position of the sprite in the spritesheet. Doesn't need to be aligned.

- `Method`—How to draw the sprite, a value 0–4. 0 overwrites the background, and 3 is `XOR`, the method to use to draw and erase a sprite with the same command.
- `Flip`—Normally 0, or 1 if you want to horizontally flip the sprite.
- `Update_LCD`—1 if you want to update the LCD immediately. You can draw several sprites by setting this to 0 and then make them all appear at once by setting this to 1 instead of 0 for the last sprite.

The other method, shown at the right side of figure 11.2, is to define the sprite in hexadecimal form. The sprite is then stored as a string within your code, and you don't need to use a `Pic`. In the remainder of this section, I'll focus on the latter method. This method is from the PicArc library, so it uses the `identity` command, from the MATH tab of the MATRIX menu, accessed with [2nd][x⁻¹]:

```
:identity(5,"SPRITE HEX",Sprite_X,Sprite_Y,Sprite_Width,
➥ Sprite_Height,Method,Flip,Update_LCD
```

As you can see, this command and its arguments are similar to the `real(1` command that uses a spritesheet, differing in that the sprite is directly defined within the command. It also omits the `Pic_Num`, `Pic_X`, and `Pic_Y` arguments, because those are now meaningless. But you need to learn how to represent sprites in hexadecimal format.

11.2.2 Sprites as hexadecimal

Hexadecimal (or "hex") is a way to represent numbers, just as decimal is a way to represent numbers. You may also have heard of binary and octal, two other number formats. Hex is good for representing numbers in computers. If you consider the decimal number system, there are 10 digits, 0 to 9. Each successive number place as you move from right to left is worth 10 times as much as the previous place. For example, in the number 98, the 8 is worth 8, but the 9 is worth 90. The 3 in 345 is worth 300.

Hexadecimal numbers have 16 digits. The digits 0–9 mean 0–9, just as in decimal, but A means 10, B is 11, C is 12, D is 13, E is 14, and F is 15. In hex, each successive place as you move from right to left is worth 16 times more than the previous place. The hex number 0x10 (0x is a prefix indicating "this is a hex number") is equal to 16 in decimal, because it's $1 * 16 + 0$. 0x123 would be $1 * 16 * 16 + 2 * 16 + 3 = 256 + 32 + 3 = 291$. Hex will become important to you for numbers in assembly in chapter 12, as well as for different types of computer programming, but here we'll use hex to represent sprites.

If you looked carefully at the arguments to the `identity(5` and `real(1` functions in section 11.2.1, you may have noticed that both take widths in multiples of 8. If you pass the number 1 as the width of a sprite, it means it's 8 pixels wide; a width of 2 is 16 pixels, 3 is 24 pixels, and so on. This is because sprites are made up of 8-pixel-wide, 1-pixel-tall chunks, as shown in figure 11.3. Each pixel can be one of two states, off (white or 0) or on (black or 1). Conveniently, bytes in computers and calculators are also 8 bits, each of which can be a 0 or a 1, so sprites are made up of bytes. Each 8-bit byte can be broken up into two 4-bit nibbles, also shown in figure 11.3. Remember,
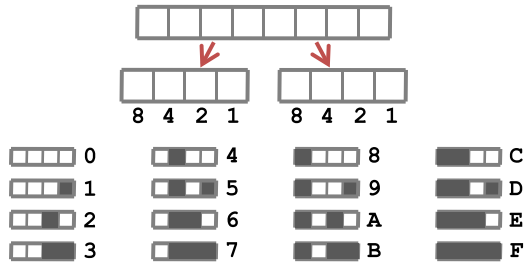
Figure 11.3 Each 8-pixel chunk of the sprite is divided into two 4-pixel chunks called nibbles. Each nibble is converted into a hex digit, 0–F. Summing the value of the spot for each black pixel in the nibble yields the equivalent value of that nibble; the spots are worth 1, 2, 4, and 8, from right to left.

we're representing each pixel with 1 bit, and just as a bit can be either a 0 or a 1, each pixel can be off (0) or on (1). Because each byte is 8 bits and each nibble is 4 bits, an 8 x 8 sprite is 8 bytes, or 16 nibbles.

You now know that each horizontal group of 4 pixels is a nibble and that sprites must be multiples of 8 pixels wide; to that, I'll add that each nibble is represented by one hexadecimal digit, 0 through F. There are 16 hex digits 0 through F, and there are also 16 possible ways to arrange 4 pixels that are either on or off. The bottom of figure 11.3 shows each of these 16 possible arrangements, along with the hex digit used for that particular arrangement. In each group of 4 pixels, the rightmost is worth 1, the next 2, the third 4, and the leftmost 8. If all 4 pixels are black, then the hex digit for that nibble is 8 + 4 + 2 + 1 = 15, which is an F in hex. If the pattern is white-black-white-black, the pixels worth 4 and 1 are on (black), so the nibble is worth 5 in decimal and hex.

As a further example of converting a sprite to hex, look at figure 11.4. The mouse cursor sprite shown is 8 pixels wide and 8 pixels tall, so because a byte is 8 bits, this sprite is 1 byte wide and 8 high. Recall also that each byte is 2 nibbles, so this sprite will be 16 nibbles (or 16 hex digits). You convert each group of 4 pixels to a hex digit as the text and figure 11.3 demonstrated, treating each black pixel as a 1 and each white pixel as a 0, and converting each 4-bit nibble to its hexadecimal form.

This particular sprite gets converted to the 16-nibble hex string "80C0A09088D0A818", which can be used with `identity(5` to draw or erase a mouse cursor sprite on the screen. Let's conclude this section with an improvement to `prgmCURSOR` from chapter 7 that demonstrates the speed and space improvements of this technique in action.
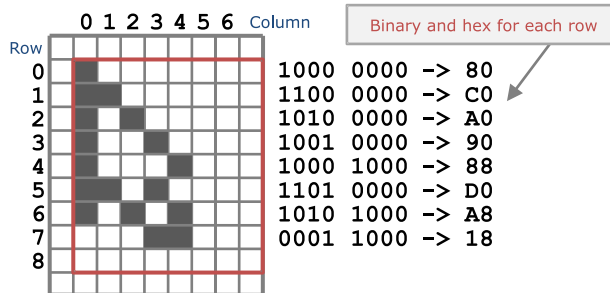
Figure 11.4 Turning the mouse cursor into binary and then hexadecimal. The final hexadecimal string encodes the sprite left to right, top to bottom as "80C0A09088D0A818". Notice the nibble encoding matching figure 11.2, where each black pixel is worth 1 and each white pixel 0.

11.2.3 *The hybrid BASIC mouse: CURSORH*

We can take the CURSOR program from chapter 7 and convert it to use hybrid BASIC commands to draw the cursor instead of a series of `Pxl-Change` commands. Listing 11.1 shows the completed code for this program. In the original CURSOR program, a sub-program called ZCURSORB containing many pixel commands was called to draw or erase the cursor. Because we can now draw or erase the cursor with a single command, I've removed the subprogram, instead putting the `identity(5` command directly into `prgmCURSORH`. Just as `Pxl-Change`ing a sprite over itself causes it to be erased, `XOR`ing a sprite over itself erases it, hence the `identity(5` call in CURSORH.

Like CURSOR in chapter 7, `prgmCURSORH` in the following listing provides a mouse cursor that you can move around the screen with the arrow keys. [CLEAR] ends the program.

Listing 11.1   The hybrid cursor program, CURSORH

```
PROGRAM:CURSORH
:If 1337≠det([[42
:Then
:Disp "NEED DOORS CS","DCS.CEMETECH.NET        Make sure required
:Return                                        libraries are present
:End
:44→A:28→B
:StoreGDB 0
:AxesOff:ClrDraw
:Repeat K=45                                   Draw the cursor with
:identity(5,"80C0A09088D0A818",A,B,1,8,3,0,1   XOR logic (Method = 3)
:Repeat K                                      at X = A, Y = B
:getKey→K
:End                                           XORing a sprite over itself
:identity(5,"80C0A09088D0A818",A,B,1,8,3,0,1   erases it, so do that here
:A-2(K=24 and A≠0)+2(K=26 and A<88→A
:B-2(K=25 and B≠0)+2(K=34 and B<54→B           Explicit conditionals converted
:End                                           to implicit conditionals
:RecallGDB 0
```

As a final improvement, the four explicit conditional statements in `prgmCURSOR` have been converted to a pair of implicit conditionals. This final CURSORH program is a mere 18 lines of code and very fast, compared to the slower CURSOR and ZCURSORB, which together were 35 lines. If `prgmCURSORH` is called from another program or game, you could save five more lines by removing the check for libraries at the beginning of this program.

Sprites are great for quickly drawing and erasing small icons or pictures of objects or characters that need to move around the screen, but what about static full-screen collages of sprites, such as might be used to draw the map and house in figure 11.1?

11.3 *Tilemapping and scrolling*

In many games, you draw a background and then move characters, ships, or other sprites around on top of that background. Such games might include space shooters,

where you must survive waves of enemy ships, RPGs where you explore a large world, or even puzzle games or platformers. In each case, it would be helpful to have a fast way to build that background out of a collection of sprites (collectively called a spritesheet). Given a set of sprites and some data that define which sprite to put at each location on the screen, the routine could build the complete background and draw it to the screen.

Such a routine is a tilemapper, and you've already seen a rudimentary tilemapper in the RPG game in chapter 9. A matrix was used to define a number corresponding to each character spot on the homescreen, and a sub command turned those numbers into characters. A 1 was a blank spot, a 2 was a wall, and so on. We'll review that tilemapper and how it can be made to smoothly scroll with hybrid BASIC. You'll then see the hybrid BASIC tilemapping command and how it uses a similar matrix method to your pure TI-BASIC tilemapper. We'll begin with a review of the single-screen tilemapper from chapter 9 and how it can be expanded to draw a map larger than a single 16 x 8–character array.

11.3.1 *Expanded TI-BASIC tilemapping with scrolling*

The simple game we worked with in the final section of chapter 9 used the contents of an 8-row, 16-column matrix to draw a map on the homescreen. The code to draw the tilemap looked something like the following:

```
:ClrHome
:For(D,1,8
:For(C,1,16
:Output(D,C,sub(" O+°",[C](D,C),1
:End:End
```

Loop over 8 rows of the homescreen

Loop over 16 columns

Display one of four possible characters, depending on matrix element [C] (D,C)

I briefly mentioned after presenting the program that you could expand it even further by allowing larger, scrollable maps. You could use one of two techniques to do this:

- You could scroll the map by one row or column horizontally or vertically if the player walks over one of the four edges of the screen, as shown in figure 11.5. With this method, the map could be any size, from 8 rows by 16 columns up to the maximum size of a matrix, 99 by 99 elements.
- You could scroll the map by an entire screen (16 columns and 8 rows) each time the player crosses an edge. This means that the map must be a multiple of 16 columns and 8 rows.

We'll pursue the first technique. I'll show you a tilemapper that will move two homescreen rows or columns each time the player moves off one of the edges. It will use hybrid BASIC to smoothly scroll the screen when the player reaches an edge.

Let's first look at what you'd need to do to make this sort of smooth-scrolling tilemapper, and then I'll show you the program that generated the screenshots in figure 11.5.
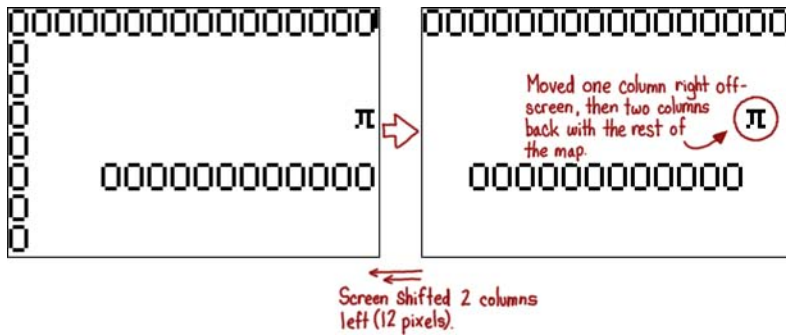
Figure 11.5   A smooth-scrolling homescreen-style tilemapper, showing reaching an edge and scrolling the map by two columns

WRITING A HOMESCREEN TILEMAPPER WITH COMPLEX SCROLLING

In chapter 9, we dealt with a simple tilemapping engine. Its job was to take the numbers in a matrix, each element of which corresponded to one character on the homescreen, and draw those characters onto the homescreen. If we expand such an engine to handle a map larger than the homescreen, then to use the hybrid BASIC smooth-scroll function, we need to change a number of characteristics of the engine:

- The map matrix [C] is now bigger than 8 rows by 16 columns but is still a multiple of two columns and rows, because our design will scroll the map two rows/columns at a time.
- It will need to track the X and Y offsets of the top-left of the screen into the map matrix. We'll use variables C and D respectively for this.
- It will use hybrid BASIC to smoothly scroll the screen. This means that the graphscreen rather than the homescreen must be used.
- The `Text(⁻1` command is used instead of `Output` to draw large text on the graphscreen.
- More nested loops are needed, because the map now needs to be drawn more than once. The outermost loop redraws the entire screenful of the map when the map is scrolled, the next inner one draws and erases the player's character as it moves around the screen, and the innermost loop tightly waits for a keypress.

We can continue to use (A,B) for the (X,Y) position of the player on the screen. For a rudimentary map, we'll put a border around the edges and a partial horizontal wall in the middle. Let's now look at the code for this program to see how it can be put together.

FINAL TILEMAPPER CODE

The code for `prgmBTILE1`, the hybrid BASIC tilemapper, is presented in listing 11.2. It uses the standard five-line construct to make sure that the required hybrid libraries are present. Because it operates on the graphscreen, it begins and ends with code to save and restore graphscreen settings. It sets up a matrix [C] as a 12-row by 20-column map, then runs the three main nested loops. The outermost draws the map, the next

inner one moves the player around the map, and the innermost waits for keypresses. This program has to use the graphscreen for the hybrid screen-scrolling function to work, so the `Text(‾1` lines draws the map and the player. All characters are drawn at multiples of 6 in the X direction and 8 in the Y direction, because homescreen characters are each 6 pixels wide and 8 pixels tall.

Listing 11.2   A scrolling homescreen-style hybrid BASIC tilemapper

```
:If 1337≠det([[42
:Then
:Disp "NEED DOORS CS","DCS.CEMETECH.NET
:Return
:End
:StoreGDB 0                     Set up the
:FnOff :AxesOff                 graphscreen
:ClrDraw
:{12,20→dim([C]
:Fill(1,[C]
:For(X,1,12
:2→[C](X,1
:2→[C](X,20                     Define a l2-row,
:End                            20-column map and
:For(X,2,19                     fill in rudimentary
:2→[C](1,X                      contents
:2→[C](12,X
:If X>4 and X<17
:2→[C](6,X                      Set (C,D), map offsets, to 0;
:End                            (A,B) is the player location
:DelVar CDelVar D2→A:2→B        onscreen
:Repeat K=45                                    Repeat outer
:For(M,1,16                      Draw the map.    loop until
:For(N,1,8                       Need to offset by D  [CLEAR] is
:Text(‾1,8N-8,6M-6,sub(" O+o",[C](N+D,M+C),1  rows and C columns  pressed
:End:End                         into the matrix.
:C→E:D→F                                        Erase the
:Repeat C≠E or D≠F or K=45       Repeat inner loop   player,
:Text(‾1,8B-8,6A-6,"Π           until map needs to  because it
:Repeat K                       Draw the  be redrawn or  will probably
:getKey→K:End       Repeatedly loop until  player on  [CLEAR] is pressed  move
:Text(‾1,8B-8,6A-6,"[one space]  a key is pressed  the screen
:A-(K=24 and 2≠[C](D+B,C+A-1))+(K=26 and 2≠[C](D+B,C+A+1→A   Implicit conditionals
:B-(K=25 and 2≠[C](D+B-1,C+A))+(K=34 and 2≠[C](D+B+1,C+A→B   to move the player
:If B=0:Then
:2→B:For(X,1,16                 If player walked off top edge,
:real(4,1,1,1                   scroll two rows, with real(4...;
:End:D-2→D                      then fix player coordinate
:End                            and update map offset D
:If B=9:Then
:7→B:For(X,1,16
:real(4,0,1,1                   Scroll the graphscreen
:End:D+2→D                      I6 pixels (two rows) up
:End                            if the player walked off
:If A=0:Then                    the bottom edge
```

```
:2→A:For(X,1,12
:real(4,3,1,1
:End:C-2→C
:End
:If A=17:Then
:15→A:For(X,1,12
:real(4,2,1,1
:End:C+2→C
:End
:End:End
:RecallGDB 0
:Disp
```

Scroll the graphscreen l2 pixels (two columns) right if the player walked off the left edge

Scroll the graphscreen l2 pixels (two columns) left if the player walked off the right edge

End the inner and outer loops

End by displaying the homescreen

The main new command in this code is the xLIB command `real(4`, which is used to smoothly scroll the graphscreen. It requires the following arguments:

```
:real(4,Direction,Number_of_Pixels,Update_LCD
```

`Number_of_Pixels` is how many pixels to scroll the display in the given direction. `Update_LCD` is identical in function to the argument of the same name to the sprite functions, dictating whether the image on the LCD should be updated when the scrolling is complete. `Direction` is a number between 0 and 7: 0 = up, 1 = down, 2 = left, 3 = right, 4 = up-left, 5 = up-right, 6 = down-left, and 7 = down-right.

One weakness of this program is that that's an unnecessarily long delay when the map is scrolled. This lag is due to the program redrawing the bulk of the map that's already onscreen from being scrolled. A faster game that would be less frustrating for the player might keep track of which way the screen had been scrolled and redraw only that part of the map.

As a final note, you'll see how this can be easily converted to the second type of BASIC homescreen-style tilemapper, which scrolls the map by one screen at a time.

CREATING THE SCREEN-BY-SCREEN SCROLLING TILEMAPPER

A few simple changes will turn this into a tilemapper that scrolls an entire screen at a time. Other than changing the setup of [C] to define a map matrix that's a multiple of 16 columns wide and a multiple of 8 rows tall, the only parts that must be changed are the four conditionals that scroll the map. Each should scroll the entire screen away, either 96 pixels horizontally or 64 pixels vertically. Each should set the player to be on the opposite edge, to move to row 1 if it walked off the bottom of the map. The left and right edge cases will have to add or subtract 16 from C; the top and bottom edge cases will have to add or subtract 8 from D.

Hybrid BASIC also provides a graphscreen function to let you make even more complex tilemaps constructed from sprites.

11.3.2 *Hybrid tilemapping*

Imagine that you want to create a city-simulator game for your calculator. You design types of buildings, zoning, roads, environment sprites, and even railroads. But once you go to create the routine that renders the player's city, you have a decision to make.
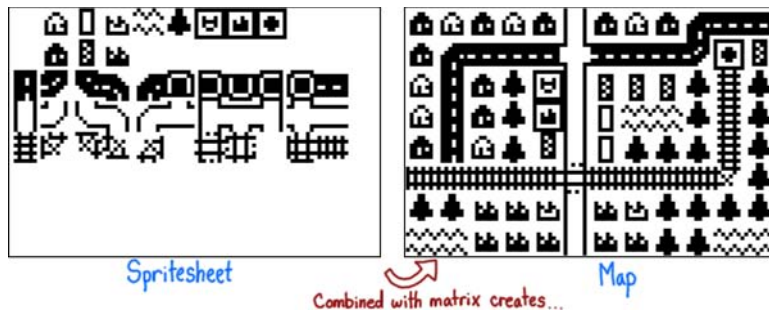
**Figure 11.6   A spritesheet (left) for a city-building game; used with the
`real(2, ...)` tilemapping command**

Presumably, you'll decide to store the city in a matrix. You'll need to decide what numbers put into the matrix define which sprite, and you'll have to decide how to draw the actual map. Figure 11.6 shows an example of the sort of spritesheet you might be using at left, along with a city that might be created from the spritesheet, shown at right.

The solution for drawing the map is the `real(2` command, the xLIB hybrid BASIC command sometimes called `DrawTileMap`. This lets you pass a matrix containing values, matrix offsets, and a tilemap and will draw the tilemap onto the screen like the right side of figure 11.6. The arguments are as follows:

```
:real(2,Matrix_num,Col_offset,Row_offset,Width,Height,SStartX,
➥ SEndX,SStartY,SEndY,Pic#,Logic,TileSize,Update_LCD)
```

The `Matrix_num` argument indicates which matrix holds the map data being accessed; `Matrix_num=0` is matrix [A], 1 is [B], and up to 9 for [J]. `Col_offset` and `Row_offset` specify the offset of the current screen's data in the given matrix. `Width` and `Height` are the dimensions of the tilemap, in pixels. `Pic#` indicates which picture contains the spritesheet, such as 1 for `Pic1`, 5 for `Pic5`, and 0 for `Pic0`. The `Logic` arguments are the same as for sprites, including 0 for overwrite and 3 for XOR. `Update_LCD` is also the same as for sprites, refreshing the contents of the screen when this argument is 1. `Spr_Flip` defines whether or not to flip each sprite horizontally, and `TileSize` defines whether this routine works with 8 x 8 sprites (when `TileSize` = 8) or 16 x 16 sprites (when `TileSize` = 16).

The remaining four arguments specify where the resulting sprites will be drawn on the graphscreen and are in numbers of sprites (not in numbers of pixels). `SStartX` and `SStartY` define the top left of the area to be drawn; if you specify `TileSize` = 8, then `SStartX` = 3 and `SStartY` = 1 would mean the area drawn would be offset 8 pixels down and 24 pixels across. By the same token, `SEndX` and `SEndY` define the bottom right of the area to be drawn, also in sprite coordinates.

This concludes the introduction to the graphical tools available in the hybrid BASIC libraries. If you want to learn more about the many graphics features of hybrid BASIC, the resources listed in section 11.1.1 and appendix C have additional information and a

full listing of the functions provided by the five major libraries. The remainder of this chapter will explore nongraphical hybrid BASIC features, starting with tools to manipulate programs.

11.4   *Finding and executing programs*

Several of the hybrid BASIC libraries, particularly Celtic III and xLIB (both built into Doors CS), offer functions to find, modify, and execute programs. Celtic III functions can read and write programs in RAM and ROM, generate lists of programs meeting certain criteria, and lock, archive, and delete programs and AppVars. xLIB can temporarily copy programs from Archive to RAM and delete the temporary copies after they're used as subprograms.

This section will first cover listing and finding files and then will show you how to copy and execute programs from Archive. Let's begin with finding programs, AppVars, and groups.

11.4.1   *Finding files*

You can find files either by name or by contents with hybrid BASIC. The commands for both types of searches are in the Celtic III portion of the libraries, so they're accessed via the det command. The first function we'll examine finds all programs on your calculator.

This function det(9) returns a list of all programs on your calculator as a string. The names are separated by spaces, so by repeatedly applying inString and sub to the result with space as the search character, you can get each program's name in a separate string.

```
:det(9
```

You can also search for programs, AppVars, or groups by name using the det(32 command. The arguments to det(32 specify the beginning of the name of the desired item, plus the type of file:

```
:det(32,"NAME",Type)
```

The string "NAME" must be at least one character long and must match the beginning of the name of each file returned. This function will also return a space-delimited list of names or an error such as the code ".7" if no such programs are found. The Type argument must be 0 to look for programs, 1 to find AppVars, and 2 for groups.

You can search for programs by contents with an optional second argument to det(9:

```
:det(9,"STARTS_WITH
```

The string "STARTS_WITH" must be the beginning of each of the programs for which you're searching, so it can be text or commands. You can use this feature to make level files for your games or save files for programs. If you make the beginning of each level or save a special string, you can use det(9 to return a space-separated string containing the names of all the programs holding levels or saves.

Celtic III also exposes commands to examine and extract groups, to read, write, insert, and erase lines from programs and files, and much more. The resources in appendix C, especially http://dcs.cemetech.net/?title=Hybrid_Libs, list them all.

11.4.2 Running subprograms from Archive

As your programs and games get more elaborate and complex, they'll also become larger and larger. You'll need more lists, pictures, and matrices, but you'll also need many more subprograms. For sufficiently large projects, the total size of all of your programs, subprograms, and level and data files may grow to overwhelm the meager 24 KB of RAM that your calculator offers. One solution might be to start archiving programs and files, and then selectively unarchive subprograms as needed, run them, and rearchive them.

To save wear and tear on your calculator's Flash memory, which can fail if written more than about 100,000 times, xLIB offers a function to copy out an archived program to a temporary program without unarchiving it and then can delete the temporary program after it's executed. The `real(10` command provides three separate subfunctions:

```
:"PROGNAME":real(10,0,Temp_Number
:real(10,1,Temp_Number
:real(10,2
```

Copy prgmPROGRAM to a temporary RAM program

Delete one temporary program

Delete all temporary programs

xLIB offers up to 16 different temporary programs, named `prgmXTEMP000` through `prgmXTEMP015`. Your programs can copy an archived program to RAM by putting the name of the program as a string in `Ans` and calling `real(10,0` with the number of the target temporary program. If there's room in RAM, and the specified XTEMP0XX program doesn't already exist, it will be created. You can then run each of these temporary programs as you would any other subprogram, as, for example,

```
:prgmXTEMP004
```

When your program no longer needs one of the temporary programs, it can delete temporary program XX using the command `real(10,1,XX)` or delete every temporary program XTEMP000 through XTEMP015 with `real(10,2)`.

Many other hybrid BASIC functions exist in the five main libraries; we'll now review some of the most useful ones.

11.5 Other hybrid tools

In the preceding sections, you saw tools to draw sprites and tilemaps on the graphscreen, to scroll the screen, and to find and execute programs using hybrid libraries. The hybrid libraries contain more functions too numerous to cover fairly, so this section lists other useful functions that you may wish to explore in depth on your own.

We'll discuss functions that manipulate files and data in your calculator's memory, as well as tools to read information about the calculator itself. I'll then present functions for advanced I/O and for working with complex GUIs.

11.5.1 *Manipulating files and data*

You can work with files and programs using Celtic III functions. These are of particular use for games that need to read or write a lot of data, including levels, stats, and sprites, and to save files. They would also be helpful for office-type programs, such as text editors.

- Read, modify, insert, or erase lines in programs with `det(5)`, `det(6)`, `det(7)`, and `det(8)`.
- Read binary data from programs with `det(14)`. Insert data with `det(15)`, or delete data with `det(16)`.
- When reading and writing binary data, you may want to convert raw binary to and from ASCII-encoded hexadecimal. `det(17)` converts ASCII-encoded hex to binary, and `det(18)` converts binary read from a file back to ASCII-encoded hex. This is particularly useful if you store many sprites as compressed lines in files.

Several functions return statistics or information about files and the current calculator or can modify the state of the calculator:

- You can use `det(4)` to find out information about the calculator. `det(4,0)` returns the number of bytes of free RAM, `det(4,1)` returns the bytes of free Archive, and `det(4,2)` returns this calculator's OS version.
- `real(5)` can check or change the contrast of the calculator's LCD. `real(5,1,0)` returns the current contrast, 0 through 39 where 0 = very light and 39 = very dark. `real(5,0,X)` sets the current contrast to X, also 0 to 39.
- The type of the current calculator can be determined with `real(11)`. This will return 0 for a TI-83+, 1 for a TI-83+SE, 2 for a TI-84+, and 3 for a TI-84+SE.

One particularly useful function for working with numbers and strings is `real(1,X)`, which converts the number X into a string. To perform the reverse function, use `expr("10.2")`, which turns a string back into a number, if it indeed represents a number.

A final category of useful hybrid functions facilitates input and output.

11.5.2 *Hybrid TI-BASIC I/O and GUIs*

Hybrid BASIC can be used for more expressive input and for GUIs. Here are the functions for input:

- xLIB provides a `getKey` that can handle all keypresses as well as diagonal movement, `real(8)`. This outputs its own set of keycodes, different from `getKey`'s keycodes. The documentation for `real(8)` shows the full set of keycodes.
- You can use a very fast mouse cursor in your programs with `sum(6,X,Y)`, where X and Y are the starting location of the mouse. This will return a list containing {X,Y,Click}, where X and Y are the final coordinates of the mouse, and Click is 1 for a left click ([2nd] or [TRACE] or [ENTER]) or 2 for a right click ([ALPHA] or [GRAPH]).

Complex GUIs are also possible with the DCSB Libs. `sum(9)` and `sum(7)` are used to create a GUI containing layered windows, buttons, text input elements, and more. `sum(12)` will then call a special mouse routine that can interact with all of those elements and notify the program about text entered and form elements and buttons clicked.

11.6 *Summary*

You've now seen the extra power your programs can gain from using hybrid TI-BASIC, or TI-BASIC augmented with special community-created libraries. From fast graphics drawing to diagonal key input, from executing archived programs to changing the LCD's contrast, hybrid libraries can make your programs much more advanced. But you're still working within the limitations of the TI-BASIC language.

If you want to go further and write programs that are even more powerful, fast, and complex, you can learn z80 assembly, the language in which the hybrid libraries are written. The following chapter will give you a first taste of z80 assembly, as well as resources where you can learn more.

Introducing z80 assembly

12

This chapter covers

- The basics of numbers and commands in z80 assembly
- Your first z80 assembly programs
- All the tools to get started with assembly programming

TI-BASIC can be used to create programs and games running the gamut from extremely simple to very complex. After spending some time with TI-BASIC and running into some of its limitations, you might have wanted to reach for more power and more features, which the hybrid TI-BASIC libraries introduced in chapter 11 provided. Perhaps you've used the hybrid libraries for a while, and have wondered why those libraries can do so much. Why are they so fast, and how come they can access so many features that your TI-BASIC programs can't use? The answer is that they're written in z80 assembly, a more-powerful, less-restrictive language that can take full control of your calculator. You too can learn how to write programs in assembly!

In chapter 1, you learned the difference between an interpreted language like BASIC and a compiled language. Interpreted languages are processed as they're

run by an interpreter, which translates each command into something that the device's CPU can execute. For TI-BASIC, the TI-OS's interpreter translates each command into z80 assembly, which then runs on your calculator's processor. The interpreter can detect if you've made an error in your program and alert you accordingly. On the flip side of the coin, it limits what you can do with the hardware and doesn't let you arbitrarily access memory and features of the calculator. The interpreter slows your programs way down, because it must do a great deal of work to run each command in your TI-BASIC programs.

Assembly (ASM) runs directly on the processor. It can run much, much faster and can manipulate the hardware to do almost anything, including working with the screen, the memory, the keyboard, and even the link port and USB port on your calculator. The downside is having to be more careful to check for mistakes, because now coding errors will at best make your program misbehave and at worst crash your calculator.

In this chapter you'll learn the basics of z80 assembly programming. I'll introduce the tools used to write assembly code and the similarities and differences between assembly and BASIC. We'll look at the classic Hello World program as written in z80 assembly and then walk through important concepts such as binary and hexadecimal, manipulating bits and bytes, and performing math in assembly. You'll see how to control program flow and work with input and output in z80 assembly. I'll leave you with a few final thoughts on ASM and how you can pursue it further.

Let's begin with the tools you'll need to work with z80 assembly, hereafter also known as ASM, and how ASM compares to TI-BASIC.

12.1 What is assembly?

TI-BASIC is one programming language; C, Java, PHP, JavaScript, Python, C#, C++, and Lua are also programming languages. z80 assembly is yet another programming language, one that runs on your calculator. Assembly language is a low-level language; unlike languages such as C or C++, where a compiler translates your code into simpler instructions that will be executed on the processor, you directly write instructions for the CPU. Each different CPU architecture (x86, ARM, and z80, to name a few) has a different assembly language, corresponding to the instructions available for that CPU.

We'll be working with the z80 CPU. First created by the Zilog Corporation in 1976, it's an 8-bit processor, which means it deals mostly with 8-bit integers. It can handle 16-bit addresses, which means it can handle at most 64 KB of RAM. Figure 12.1 shows a simple block diagram of the z80 processor connected to memory, the LCD, the keypad, and your calculator's serial or I/O link port; this is a more detailed version of figure 1.1 from chapter 1. The CPU is responsible for executing commands, which may perform math and read in or write out to memory and hardware, and also maintains a set of internal variables called registers.

In this section, I'll talk about the differences between TI-BASIC and z80 assembly, particularly as they pertain to programming directly for the hardware in figure 12.1.
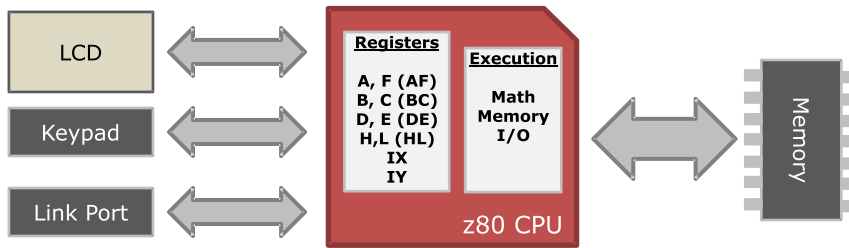
Figure 12.1   How the z80 CPU in your calculator talks to the memory and hardware. The CPU performs computation and talks to RAM and hardware devices; the registers are built-in variables.

I'll also introduce the tools you'll need to write z80 assembly on your computer. Let's begin with a comparison of ASM and TI-BASIC.

12.1.1   z80 assembly versus TI-BASIC

You learned in chapter 1 that TI-BASIC is an interpreted language, in which the calculator reads and translates each line of code into instructions the underlying hardware can understand. You gain peace of mind, because the interpreter will catch mistakes in what you or the user types and display error messages. On the downside, the language is slow, because it needs to spend time examining and translating each line before executing it.

z80 assembly is an assembled language, which means you're directly writing the instructions that will run on the processor. The only thing the assembler does is translate human-readable instructions (like `add`, `ld`, `call`, and `push`) into numbers that the processor understands. Each of these instructions is much simpler than the TI-BASIC commands you've been using. Because it runs directly on the CPU, z80 assembly is fast. The minuses are that it's more complex to understand than TI-BASIC and if you make a coding error, your calculator might crash. Luckily, because the language has been in use for over 30 years, and calculator programmers have been using it for more than 15 years, there are many tools to make your life as an ASM programmer easier, several of which I'll discuss in this section.

To get an idea of some of the similarities and differences between z80 assembly and TI-BASIC, glance at table 12.1. As you can see, you're working with much lower-level concepts, such as registers and memory instead of variables, lists, matrices, and pictures. Some of the TI-OS functions are still available to assist your programs, and if you choose to make your programs run under a shell like Doors CS or Ion, you have an additional set of functions available.

With z80 assembly, you can't (easily) access any of the variables and data types to which you're accustomed in TI-BASIC. You use the registers, a set of 8-bit or 16-bit integers, to hold temporary values. Where you might use several of the 27 numeric variables A–Z and θ to hold values used throughout your TI-BASIC programs, you'll now store those values directly at memory locations. Instead of referring to data locations

Table 12.1   A comparison of features and concepts between TI-BASIC and z80 assembly

In TI-BASIC	In z80 assembly
27 real *variables* that store floating-point numbers.	Eight *registers* that store 8-bit integers (0 to 255 or -128 to 127) or three registers that store 16-bit integers (0 to 65535 or -32768 to 32767).
Subprograms that can be called.	*Functions* within the same program that can be called.
Access TI-OS features via *commands*.	Access TI-OS features via *bcalls*.
Store *external data* in matrices and strings.	Store permanent data in *memory chunks* of the program itself. Temporary data can be placed in special areas of the calculator's memory.
Separate *homescreen* and *graphscreen*.	Single *gbuf* stores all LCD contents.
Programs are *interpreted*; no compilation is necessary.	Programs must be *assembled* from a human-readable format down to machine code.
Comments in code are discouraged; they slow down and bloat programs.	Extensive comments are encouraged: they're removed during assembly and don't affect the final assembled program.

by name and letting TI-BASIC figure out where exactly that data resides, you'll now work with numeric memory addresses (although you can name them to make your life easier). Most z80 ASM programs you'll write will use positive integers, holding values between either 0 and 255 or 0 and 65535. You can also use signed integers, ranging from -128 to 127 or -32768 to 32767.

Another significant difference is that most z80 ASM programs are written on your computer, then assembled and sent to your calculator via a link cable and linking software. This makes programming easier, because you can use a text editor on your computer's big screen to work with your code. You can also use lots of comments, notes to yourself that describe how nearby code works and what it does but that isn't included in your assembled programs. You'll need tools to help you write, assemble, transfer, and run assembly programs, which we'll now discuss.

12.1.2   z80 assembly programming tools

You need a few tools in your utility belt in order to program z80 assembly in a way that will be fun and minimally frustrating. You need an editor to help you create, edit, and organize your code. You need an assembler and linker to turn your code into a format your calculator can understand. You need link software to send programs to your calculator. You can protect your calculator from crashes and lost data if you use an emulator instead of your physical calculator to test your programs.

I recommend the following toolset:

- *Editor*—Notepad++, which can keep many files open in different tabs, is an excellent text editor. Programmer's Notepad is another alternative, or if you're using Linux, you should try to learn how to use vim. (http://notepad-plus-plus.org/)

- *Assembler/linker*—The Doors CS SDK (http://cemete.ch/DL470) can be used to create ASM programs that run under Doors CS, but it also can create any ASM programs, including those that require no shell at all (called nostub programs). I recommend it because it includes an assembler called Brass and a linker called BinPac8x, and it works on 32- and 64-bit OSs. It works on Windows, Mac OS, and Linux, requiring only Python (and, if you're on Mac OS or Linux, Mono).
- *Linking software*—TI-Connect or TILP. See appendices A and C for more information.
- *Emulators*—jsTIfied, Wabbitemu, or PindurTI. See appendix C.

There are other assemblers and linkers, but I chose the Doors CS SDK despite my bias because it packages everything together neatly to help beginners start programming quickly.

Once you have linking software and/or an emulator installed, unpack your assembler and linker. If you're using the DCS SDK, you can unzip it in a location of your choice. To create a program, pick a name between one and eight letters, such as *project*, and create project.asm in the /source/ folder of the SDK. Be careful not to use the Windows Notepad program, because this will give your file the name project.asm.txt (which is wrong). To help yourself quickly build your project from its ASM source into a .8xp program you can load onto your calculator, make a batch file in the main SDK directory called project.bat (again, where *project* is the name of this program):

```
compile project
pause
```

The `pause` will let you see any errors that the assembler found, so that you can correct them.

If you have any difficulties with setting up or using these tools, or by the time you read this you can't find the tools mentioned, try asking for assistance on one of the forums mentioned in appendix C. Assuming that you were successful in setting up your environment and can create projects, let's look at your first z80 ASM program.

12.2   "Hello, World"

A Hello World program was your first TI-BASIC program, and it will be your first z80 assembly program as well. The source code for this program is shown in listing 12.1; remember that this needs to be assembled before it can be tested on your calculator. This program clears the screen, moves the cursor to the top-left corner, and prints "Hello, World!" It then moves the cursor to the second line and ends.

> **Listing 12.1   Hello World in z80 assembly, your first ASM program**

```
.nolist
#include "ti83plus.inc"
#include "dcs7.inc"
.list
.org progstart
    .db $BB,$6D
```

Every z80 ASM program starts with something like this

```
Start:
    bcall(_ClrLCDFull)
    ld a,0
    ld (curcol),a
    ld (currow),a
    ld hl,HelloWorldMessage
    bcall(_puts)
    bcall(_newline)
    ret
HelloWorldMessage:
    .db "Hello, World!",0
.end
END
```

a is the accumulator, the most frequently used 8-bit register

This is a 2-byte pointer to the string "Hello, World!"; the actual string can't fit into the hl register

bcalls are like commands in TI-BASIC; _puts, is like Disp

Every z80 ASM program should end with these two lines so that the assembler knows where the end of the source code is

As we did with your first few TI-BASIC programs, let's walk through this code line by line.

```
.nolist
#include "ti83plus.inc"
#include "dcs7.inc"
.list
```

The ti83plus.inc and dcs7.inc files (in the /tasm/ folder of the DCS SDK) define a lot of constants for you and the assembler to use. ti83plus.inc tells the assembler where in the calculator's OS the _puts, _ClrLCDFull, and _newline bcalls can be found. dcs7.inc defines things such as the address at which programs start in memory, progstart, so that you don't have to type its hexadecimal equivalent $9D93 instead.

> **Labels, addresses, and hexadecimal**
> Every assembly program starts from an address in memory, generally 0x9D93. This is usually written as $9D93 for assembly programs; $ indicates to the assembler that the following number is hexadecimal. Every instruction in your program takes between 1 and 4 bytes, and data that you store in the program (including strings such as "Hello, World!") also take up bytes. Each line of the program is at some memory address, which the assembler determines by counting bytes from the beginning of the program and adding $9D93. When you add a label to your program, such as Start: or HelloWorldMessage: in listing 12.1, you're assigning the memory address for that line a name that you can then use to refer to it. To the assembler, though, you're still referencing 2-byte addresses consisting of 4 hexadecimal digits.

When the assembler turns your program into a .8xp file, it also creates a second file called a listing, which shows your source code side by side with the hexadecimal equivalents of each line of the program, as well as the address of each line (see the "Labels" sidebar). The .nolist/.list tells the assembler that the files included after .nolist and before .list should not be shown in the listing file, because they aren't specific to the program and would just clutter the listing file.

```
.org progstart
    .db $BB,$6D
```

.org, for "origin," tells the assembler to start counting addresses at progstart, defined in dcs7.inc as address $9D93. The first command that's actually part of the program is the .db $BB,$6D. This tells the assembler to put the two literal bytes $BB and $6D into the program as the first 2 bytes. These special bytes signal the TI-OS that this is an assembly program instead of a BASIC program, necessary because ASM programs show up in the [PRGM] menu just like TI-BASIC programs.

```
Start:
```

The label Start: isn't used in this program, in the sense that nothing jumps to it, but in other programs, it would be necessary if you wanted to restart the program. Labels are named locations in the program, so defining labels doesn't raise program size. The program could jump to this label, just like a Goto/Lbl in TI-BASIC.

> **Formatting code**
> All labels must be aligned on the left margin, and all commands must be tabbed or spaced into from the margin. I recommend a single tab. Comments are placed after semicolons; the assembler ignores everything from the semicolon to the end of the line.

```
bcall(_ClrLCDFull)
```

bcall() tells the calculator to execute a ROM call, which means to go into the TI-OS and run some function that the OS provides. In this case, the command is _ClrLCDFull, which clears the screen.

```
ld a,0
```

The accumulator, register a, is the most frequently used register in z80 assembly. Used somewhat like variables in TI-BASIC, you can store to and read from the accumulator. The ld command is short for "load" and works from right to left. ld a,0 means to load 0 into the accumulator; in TI-BASIC, the line might look like 0→a.

```
ld (curcol),a
ld (currow),a
```

These two lines use the zero stored in a to set the cursor row and cursor column, used to define where text is printed. These work similar to the row and column arguments to Output, except that you must set curcol and currow before calling command to display text. In z80 ASM, the homescreen columns range from 0 to 15 and the rows from 0 to 7; in TI-BASIC the columns and rows start at 1.

The z80 processor can't directly load a number into a numeric memory address, so you can't write ld (curcol),0. Instead, you must first load the number to be stored into a register, then store the contents of that register into memory. curcol and currow are the addresses in memory where the value of the cursor column and row are stored; the parentheses are used for an operation called indirection. They tell the

processor that you want to access the byte stored at that location, not the actual number of the address. Whenever you want to store directly to locations in memory or read back from memory, you need to use indirection.

```
ld hl,HelloWorldMessage
```

`HelloWorldMessage` is the address in the program of the first byte of the string "Hello, World!" Because you don't use parentheses, this refers to the address of the byte rather than the contents (which would be the single character H). Whenever a program deals with a string, you refer to the string by the address of its first character; you can then move forward in memory from that point to find the remainder of the string. This line of code loads that address (once again, a 2-byte number represented by four hex digits) into the 2-byte register `hl`. The accumulator `a` holds 1 byte; `hl` holds 2, and because addresses in memory are 2 bytes, you can't put an address in `a`.

```
bcall(_puts)
bcall(_newline)
```

Remember that `bcall()`s are like commands; here, you call the `_puts` command and then the `_newline` command. In TI-BASIC, you put the arguments to commands in parentheses after the name of the command, but in z80 ASM, you first put the arguments in registers and then `bcall` the command. `_puts` takes one argument, the string to be displayed, in `hl`. It displays that string in homescreen font at `currow` and `curcol`, which is why you set both to 0. `_newline` takes no arguments, so you call it directly after `_puts`.

```
ret
```

Short for "return," the `ret` command is similar to its TI-BASIC counterpart. If you're in the main body of code for a program, it makes the program quit; if it's inside a function called with `call`, it returns to the point from where the function was called.

```
HelloWorldMessage:
    .db "Hello, World!",0
```

This is an example of a string embedded in a program. `.db` (data bytes) embeds the characters in the string into the final program, one character per byte. The last byte of the string, a 0, is called a zero-terminator and marks the end of the string. When the string is displayed, `_puts` keeps displaying characters until it reaches a 0.

```
.end
END
```

These two lines are always required at the end of your program. They're used to tell the assembler when it reaches the end of the source file but don't become part of the final program.

12.2.1 Running Hello World

If you save the source code as hiworld.asm in the /source/ folder of your DCS SDK directory, you can assemble it as described in the previous section. If you make no errors, hiworld.8xp will appear in the /exec/ directory, which you can send to your calculator or emulator. If you have Doors CS on your calculator, you can run the HIWORLD program from your calculator's [PRGM] menu; if you have any other shell, enter the shell and then run HIWORLD. If you have no shell, go to the Catalog under [2nd][0], select Asm(, and paste prgmHIWORLD from the [PRGM] menu. Hit [ENTER] to run the Asm(prgmHIWORLD command.

> **I wrote my own program and my calculator froze!**
>
> z80 assembly gives you complete control over your calculator. There's no longer a safety system that will pop up error messages when you make a mistake; instead, errors will be a bit more dramatic. At best, programs will unexpectedly quit to the homescreen. If the error is a bit worse, your calculator will freeze. If you make particularly egregious errors, you can even get your calculator to get stuck displaying random gibberish.
>
> In most cases the fix is easy. Just pull out one of the four AAA batteries, reinsert it, and you should be met with a RAM clear. If you still have no luck, pull the battery again, hold down the [CLEAR] button, and with [CLEAR] still held, reinsert the battery. Tap [ON] and release [CLEAR], and the calculator should be fixed. In the worst cases, you might need to take out all five batteries (including the button cell under its cover) for a few hours or even resend the OS.
>
> To put your mind at ease, there are few known instances of destroying ("bricking") a TI-83+/84+ with a bad assembly program, and almost every such case involved an intentionally malicious program. With that said, be careful, and understand that by writing or even using z80 ASM programs you're indeed risking the well-being of your calculator.

Now that you've seen a simple z80 assembly program, let's step back and go through the skills you'll need to write ASM programs. First, we'll discuss how numbers in assembly work.

12.3 Bases and registers

We discussed the concept of hexadecimal once before, as it relates to encoding sprites, in chapter 11. With z80 assembly, you'll need to frequently work with binary, hexadecimal, and decimal, three different bases used to represent numbers. A good assembly programmer must be able to intuitively understand binary and hexadecimal, so this section will introduce the two bases and how they relate to decimal. We'll then look at how you can work with the individual bits within bytes, as well as why you'd want to, and how to do math with registers.

Let's begin with the relationship between the three bases you'll be using most frequently: hexadecimal, binary, and decimal.

12.3.1 *Working with binary, hex, and registers*

In TI-BASIC, you're almost always dealing with decimal numbers, the familiar numbers you use in everyday life. Such numbers are made up of the digits 0–9 and may include portions before and after the decimal point. The farther left in a number a digit is, the more it's worth; the farther right, the less it's worth. Hexadecimal and binary are two more ways of representing the exact same numbers but in a way that's closer to how numbers are stored in a calculator's (or computer's) memory. We'll be working with integers, numbers that have no decimal point or fractional parts.

I'll first show you how to convert between binary and decimal and to read binary numbers and then move on to hexadecimal. We'll also look in more detail at registers, the variables inside your calculator's processor where numbers are stored during calculations.

BINARY AND CONVERSION

Binary numbers are made up of only the digits 0 and 1; decimal numbers consist of the digits 0–9. In decimal, each digit as you move from right to left is worth 10 times as much as the preceding digit. The 3 in 123 is worth $3 * 1$, whereas the 2 is worth $2 * 10$ and the 1 is worth $1 * 100$. Each digit in a binary number is worth twice as much as the previous digit, as shown in figure 12.2.

To convert a decimal number into a binary number, find the largest power of 2 less than or equal to the decimal number, and set that digit in your binary number to 1. Subtract the value of that place from the decimal, and continue setting 0 bits (digits) in your binary number to 1 and subtracting from the decimal until the decimal number is equal to 0. As you can see in figure 12.2, 3 bits must be set to 1 in order to make the binary representation of the number 42. 32 is the highest power of 2 less than or equal to 42, leaving $42 - 32 = 10$. Next, 8 is the highest power of 2 less than or equal to 10, leaving $10 - 8 = 2$. Finally, 2 is the highest power of 2 less than or equal to 2, leaving 0, so the process is complete. Note that just as putting zeroes at the left end of a decimal number doesn't change its value, putting zeroes at the left end of a binary number also makes no difference.

Converting back to decimal is easier; all you have to do is sum the place values of all the "1" bits in the binary number. Technically, you can multiply each bit by its place value and sum all the products together, which might look like this:

$$0 * 1 + 1 * 2 + 0 * 4 + 1 * 8 + 0 * 16 + 1 * 32 = 0 + 2 + 0 + 8 + 0 + 32 = 42$$
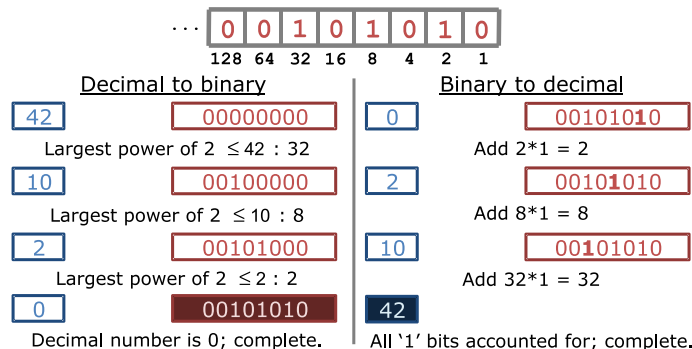
Figure 12.2 Converting the decimal number 42 to binary 00101010 and back to decimal

But you can also simply add up the bit values of the places with "1" bits, namely 2 + 8 + 32 = 42.

In figure 12.2, the binary numbers are all written with 8 bits (digits). In general, computers, calculators, and all electronic devices store numbers in series of bytes, each 8 bits long. The largest value that can be stored in a byte is 11111111b, or 255 (the b suffix indicates binary); the smallest is 00000000b, or 0. To store bigger numbers, you put several bytes together. Two bytes are 16 bits and can store up to 1111111111111111b = 65535.

> ### Endian-ness
> Your calculator's z80 CPU can work with 1- and 2-byte numbers and is little-endian. This means that when you have a 2-byte number in the calculator's memory, the first byte is the less significant byte (LSB, with places worth 128 through 1), and the second is the more significant byte (MSB, with places worth 32768 through 256). Big-endian machines store the MSB before the LSB.

BINARY AND HEXADECIMAL

Converting between binary and hexadecimal is similar to converting between binary and decimal. Hex numbers consist of hex digits, which can be 0–9 and A–F. As mentioned in chapter 11, 0 to 9 are 0 to 9, A is 10, B is 11, and up to F is 15. Just as each binary digit is worth twice as much as the previous digit, and each decimal digit is worth 10 times the one to its right, each hex digit is worth 16 times as much as the preceding digit.

For example, the hexadecimal number $A5 (also written as 0 x A5 or 0A5h) is $A * 16 + 5 * 1 = 10 * 16 + 5 * 1 = 160 + 5 = 165$. By the same token, $3F is $3 * 16 + 15 * 1 = 48 + 15 = 63$. Finally, $105B is $1 * 16 \wedge 3 + 0 * 16 \wedge 2 + 5 * 16 \wedge 1 + 11 * 16 \wedge 0 = 4096 + 0 + 80 + 11 = 4187$. Hex numbers are easy to convert to and from binary and decimal:

- To convert a *hex* number to *decimal*, you can follow the process in the preceding paragraph, multiplying each digit by its place value.
- To turn a *decimal* number into *hex*, repeatedly divide it by 16. Each remainder will be one digit of the hexadecimal number; stop when the decimal number is less than 16. Say you want to convert 500 to hex. First divide by 16, yielding 31 remainder 4: the rightmost digit is thus 4. Divide 31 by 16 again, yielding 1 remainder 15, so the next digit is 15, or F. Finally, 1 is less than 16, so it's the final digit: $1F4.
- Converting *hex* into *binary* is easy, because each hex digit is equivalent to exactly four binary bits. Convert each hex digit 0–F to binary as if it was a single number by itself, with its binary bits worth 8, 4, 2, and 1 left to right. For example $195B becomes 0001 1001 0101 1011 = 0001100101011011b.
- *Binary* into *hex* follows the same procedure in reverse. The binary number to be converted must be a multiple of 4 bits long and is converted 4 bits at a time.

Each set of 4 bits should be treated as a separate 4-bit binary number and converted to a hex digit. For example, 01101100 would be 0110 1100, or $6C in hex (because 1100 is 12 is C).

All of these different types of numbers would be more useful to your z80 programming education if you knew where you could store them, which we'll now discuss.

z80 REGISTERS

In TI-BASIC, you're accustomed to storing most of the numbers needed for programs and games in variables, each named by a letter A–Z or θ. These variables can hold positive or negative integers and floating-point (decimal) numbers covering a huge range of values. In z80 ASM, you store values temporarily in registers, small variables inside the processor itself, as shown in figure 12.1. Values that are used and modified more occasionally in your program should be stored at named locations in memory areas called SafeRAM, which we'll discuss shortly.

Registers are all either 8 bits or 16 bits and hold integers. They're usually used to hold positive integers, 0–255 ($00 to $FF) for 8-bit registers and 0–65535 ($0000 to $FFFF) for 16-bit registers. Because the processor doesn't care what you're storing in registers, they can also be used to hold flags: each bit of an 8-bit or 16-bit register is one flag holding 0 or 1, so you can fit 8 or 16 flags per register. The TI-OS uses flags for things like whether text is drawn normally or inverted and whether or not the Polar graph mode is set.

> ### Negative numbers in z80 assembly
> You can use registers to hold signed (positive or negative) instead of unsigned (positive only) numbers. If you make the leftmost bit in an 8-bit or 16-bit number indicate that the number is negative, you can instead make 8-bit registers hold -128 to 127 and 16-bit registers hold -32768 to 32767, but the difference is purely in how you make your programs read the registers. The processor has no way of knowing how you've decided to interpret the bits in each register. To convert a negative number to a positive or a positive number to negative, you invert all the bits to change 1s to 0s and 0s to 1s with the `cpl` instruction and then add 1. You could also use the `neg` instruction, or you could subtract the number from 0.

Table 12.2 lists most of the registers you'll be using in your z80 ASM programs. Note that although you can often use all the registers for general-purpose math, almost all have special meanings. The most important registers are a and hl. If you want to load a byte into a fixed memory address, for instance, it has to be in a.

Although there are 8-bit registers like b and c and 16-bit registers like bc, you can't use the three independently. If you modify b, you're also modifying the highest 8 bits of bc, and if you modify c, you're also changing the lowest 8 bits of bc. You can use this to your advantage, as long as you don't accidentally think you have more (unique) registers available to use than you actually do. For example, to check if bc holds

Table 12.2   8-bit and 16-bit registers in z80 assembly. Note that if you modify a 16-bit register like `bc`, you also modify registers `b` and `c`, and vice versa.

8-bit register	16-bit register	Description
a f	af	The accumulator, `a`, is the most frequently used 8-bit register for 8-bit math. `f` is the flags register and can't be directly modified. `af` is generally not used to hold a 16-bit value but can be used to save and restore `a` and/or `f` using the stack.
b c	bc	`b` is used as a loop counter, especially with the `djnz` instruction. `b` and `c` are also general-purpose 8-bit registers. `bc` is a general-purpose 16-bit register, also often used for counting.
d e	de	`d` and `e` are general-purpose 8-bit registers. `de` is used as the destination address for copies with `ldir` as well as general use.
h l	hl	`hl` is the 16-bit equivalent of the accumulator, used for most 16-bit math operations. `h` and `l` can be used separately as 8-bit registers.
	ix iy	`ix` and `iy` are index registers, used to access various bytes of memory near a given point. `iy` should not be modified, because the TI-OS uses it for its flags. `ix` may be modified.

$0000, it's convenient to check if `b` contains $00 and `c` contains $00. The same rules apply to `d`/`e`/`de`, `h`/`l`/`hl`, and `a`/`f`/`af`.

There are several more registers that you won't be using in most cases, including `sp` (the stack pointer), `pc` (the program counter), `i` (the interrupt register), and `r` (the refresh register). I've mentioned that values you're not currently using can be put in special memory addresses, but you can also use the stack to save them.

12.3.2 *The stack: saving registers*

The stack is a type of data structure (and an area of memory) used for saving and restoring registers. It's also used to track which functions have called which, so that when a function ends, it can properly return to the code that called it. The stack has several unique characteristics:

- It grows from higher memory addresses down into lower addresses. Most storage in memory is from low to high addresses.
- The stack has two operations: `push` (onto the stack) and `pop` (off the stack).
- You can only push and pop 16-bit registers: `af`, `bc`, `de`, `hl`, `ix`, or `iy`.
- When a function returns, it should always have performed the same number of pushes and pops, leaving the stack level where it was when the function started.
- You should never put more than about 200 items (400 bytes) on the stack.

Figure 12.3 demonstrates the z80's stack, with three `pushes` and one `pop` performed. Each `push` pushes the contents of that register onto the "top" of the stack, and `pop` always pops off (and removes) the "top" element of the stack.
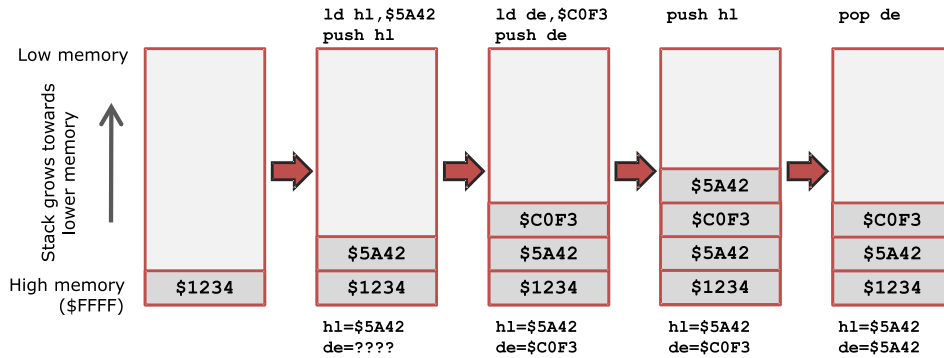
Figure 12.3   The z80 stack, going through three pushes and one pop. Each push expands the stack lower into memory, while pops contract the stack into higher memory. All pushes and pops are 16-bit values and must be performed on 16-bit registers, namely `af`, `bc`, `de`, `hl`, `ix`, or `iy`.

Notice that the stack only stores the numbers that you `push` onto it but doesn't remember which register that number came from. Therefore, if you `push hl` and then `pop de`, the value pushed onto the stack from `hl` will end up in `de`. The stack is often used to hold the value of intermediate calculations while additional calculations are carried out, after which the intermediate values can be restored to registers and used in some final computation. When you `pop` to a register, keep in mind that its previous value is wiped out, unsaved.

If you want to save the contents of a register after a function ends, or without burying it deep in the stack, you can save it in memory.

12.3.3   Integers in memory: long-term storage

If you need to store integers for longer than they can survive on the stack or in a register, given that you have few registers to juggle and that the stack must be restored before a function can exit, you can use memory. The TI-OS and shells like Doors CS define several areas called SafeRAM that can be used to hold variables while your program is running but that are wiped out when your program ends.

STORING VALUES IN MEMORY FOR THE DURATION OF YOUR PROGRAM

The largest of these SafeRAM areas is `AppBackupScreen`, also called `SafeRAM1`, and is 768 bytes long. It can hold one saved LCD image, because a monochrome 96 x 64-pixel image is 768 bytes, or up to 768 bytes of variables. The following code placed in the header of a game called Invalid Tangram defines a variety of spots in which to place 1- and 2-byte variables:

```
ShotsOnscreen      .equ    AppBackupScreen      ;1 byte
EnemiesOnscreen    .equ    ShotsOnScreen+1      ;1 byte
CurrentLevel       .equ    EnemiesOnScreen+1    ;1 byte
TotalScore         .equ    CurrentLevel+1       ;2 bytes
ShipX              .equ    TotalScore+2         ;1 byte
ShipY              .equ    CraftX+1             ;1 byte
```

Comments, everything from a semicolon (;) to the end of the line, are ignored by the assembler

For l-byte memory variables, the next variable is l byte higher in memory

For a 2-byte variable (TotalScore), the next variable (ShipX) is 2 bytes higher in memory

Just as adding labels to your code associates a name with the memory location where that code will be, the `.equ` statements in this code name memory addresses inside `AppBackupScreen`. When you assemble the program, those names will no longer exist and will become the numeric addresses that they represent.

You can use these variable names in your code like this:

```
ld a,4
ld (ShipX),a
ld hl,9001
ld (TotalScore),hl
;...more code...
ld hl,(TotalScore)
inc hl
ld (TotalScore),hl
```

Load decimal 4 in accumulator a, then load a into (Ship)

Load 2-byte decimal 9001 into (TotalScore) via hl

Load, increment, and store TotalScore via hl

As first introduced in the Hello World program, the `ld` ("load") instruction works from right to left, meaning that the first line of this code loads 4 into `a`, not `a` into 4. You must use the accumulator `a` to load an 8-bit value from a register into memory; if the value is in another register, you must load that register into the accumulator `a` first. You usually use `hl` to load and store 16-bit values to and from memory, although you can also use `de` or `ix`.

If you want variables to persist between sequential runs of your program, you need to use a different technique.

STORING VALUES IN MEMORY BETWEEN PROGRAM RUNS

You can store 8-bit and 16-bit values, and even arrays and strings, inside your program's own memory. This means that when you're writing the code, you could have something like this:

```
SomeStaticString:
    .db "This contains text",0
StorageByte:
    .db 0
StorageWord:
    .dw 0
```

Every string ends with a 0 to mark the end (zero-terminated string)

A word means 2 bytes/16 bits

You can use these named memory locations just like the ones for SafeRAM locations, but the values will be within your program itself. When your program ends, shells like Doors CS, Ion, or MirageOS will store the updated program with an operation called writeback, so that when it's run once more, `StorageByte` and `StorageWord` will contain the same contents that they did after it ran the first time. It's important to note that the TI-OS's `Asm(` command won't perform writeback, which means that this technique won't work. The special program headers listed on http://dcs.cemetech.net/?title=ASM_Header will force your programs to only run with, for example, Doors CS to avoid this problem.

Now you know the basics of representing numbers and passing them around in your programs. Let's discuss the basics of register math.

12.4   z80 math with registers

Math was relatively easy in TI-BASIC: you used the same sorts of equations you used on the homescreen for math. You could use operators including +, -, *, and /, you could group with parentheses, and you could even use exponents, square roots, and functions. z80 assembly is much lower level, so performing math is more challenging. As you know, you'll usually be working with integers, although if you choose to pursue assembly, you'll find you can also use some of the TI-OS's tools for floating-point math. You can only perform one operation at a time, and multiplication and division instructions don't exist.

   Let's first look at basic register math and then explore a few ASM-specific bit tricks.

12.4.1   Register math and flags

Integer math is performed using instructions and registers. Although you can perform similar math on both 8-bit bytes in 8-bit registers and 16-bit words in 16-bit registers, the instructions for each aren't exactly the same. In this section, you'll see addition and subtraction as well as the z80's flags; next, I'll talk about multiplication and division.

ADDITION AND SUBTRACTION

One of the simplest operations you can perform is adding or subtracting 1, also called incrementing and decrementing. You can `inc` or `dec` any 8-bit or 16-bit register like this:

```
inc hl
dec b
dec ix
inc a
```

If you increment from the maximum value ($FF or $FFFF) or decrement from the minimum ($00 or $0000), the registers will loop around to the minimum or maximum, respectively.

   You can also add or subtract in assembly. Addition or subtraction for 8-bit registers always must use `a` as the destination register and one of the operand registers, for example:

```
add a,b      ⟵  Add a + b and store the result in a        Subtract a - c and
sub c                                                  ⟵   store the result in c
add a,5                                    ◁⎤
sub 42       ⟵  ...and subtract them too      You can also add
                                              "immediates," or numbers...
```

Notice that for the 8-bit `sub` command, the `a` is implied. If the value you want to add to or subtract from isn't in the accumulator, you must use `ld` command(s) to put it there. 16-bit addition is similar, but instead of `a`, it's `hl` and `ix` that must be the destination and one of the operands.

```
add hl,de         ◁⎯⎯  hl + de → hl
add ix,bc                            ◁⎤  ix + bc → ix
```

Also, you can't add an immediate (number) to `hl` or `ix`; you must first load it to a register, for example:

```
ld de,5004
add hl,de
ld bc,-999
add hl,bc
```

Set de to 5004, and add to hl (and store back into hl)

Set bc to -999, or 65536 − 999 = 64537. Numbers wrap around!

◁── **Subtract 999 from hl and store back into hl**

You can add a negative number to `hl` to subtract from it, but what if you want to actually subtract? You can't just `sub` from `hl`; you have to use a different instruction called subtract with carry, or `sbc`. An example is `sbc hl,de`. This subtracts `de` from `hl`, as you might expect, but if the carry flag is set, it will subtract one more. You can set the carry flag with `scf`, clear (reset) the carry flag with `or a`, and change (flip) the carry flag with `ccf`. Therefore, this code would properly subtract `de` from `hl`:

```
or a
sbc hl,de
```

◁── **You could substitute scf (set carry flag) followed by ccf (change carry flag). There's no "clear carry flag" instruction.**

But wait, carry flag? Setting and resetting flags? This isn't something we've discussed.

FLAGS: THE F REGISTER

Good point! We haven't gone over flags before; let's remedy that. The flags are stored in 6 of the 8 bits of register `f`. You can't directly modify the `f` register, but most mathematical operations affect 1 or more bits within `f`, each of which has a special meaning:

- *Sign (S)*—Stores the rightmost bit of the result of the last math operation. Because negative numbers are represented as having this bit set as 1, flag S set (equal to 1) means that the result of the last math operation could be taken as negative.
- *Zero (Z)*—Set (1) when the last math operation yielded 0. Reset (0) otherwise.
- *Carry (C)*—Set (1) if addition caused a register to overflow from large to small numbers or if subtraction caused a register to underflow from small to large numbers.
- *Parity/Overflow (P/V)*—For some instructions, counts the number of 1 bits in the accumulator `a`; an even number of 1 bits sets this flag, and an odd number resets it. For other instructions, it tracks *signed* overflow and is set if a math operation made the highest bit of the destination register change from 0 to 1 or 1 to 0.

These flags are often used to affect the flow of an assembly program and may also be useful with math and bitmath.

You have seen addition, subtraction, and flags; speaking of bitmath, let's look at what you can do with manipulation of the individual bits in registers and numbers.

12.4.2 *Masking and using bits*

The low-level characteristics of ASM are a double-edged sword, giving the programmer a great deal of power with significant additional complexity compared with TI-BASIC. Nowhere is this exemplified as well as with bitmath. It's entirely absent from TI-BASIC, and

if used correctly, it can make many operations in ASM effortless and fast. If ignored, the operations it's meant to simplify can be laborious. What can bitmath do?

- Fast multiplication and division by powers of 2
- Turn single or groups of bits on or off
- Flip single or groups of bits between on and off
- Shift or rotate whole bytes and words

Let's briefly discuss the instructions for manipulating bits and then see one specific application, performing multiplication and division.

BIT MANIPULATION OPERATIONS

An abridged listing of the important instructions for bit manipulation is provided in table 12.3, along with relevant examples.

Table 12.3 Major groups of instructions for manipulating bits, with examples

Type	Instructions	Description

Bitwise Boolean operations	`and b`
	`or 5`
	`xor h`
		Perform the bitwise `and`/`or`/`xor` operation between the accumulator `a` and a register or number, and put the result in `a`. Bitwise operations individually `and`/`or`/`xor` the corresponding bits in the two operands.

Changing bits in registers	`set 4,a` `res 0,d`	Set or reset the Nth bit of an 8-bit register. The rightmost bit is bit 0, and the leftmost is bit 7.

Changing bits in memory	`set 7,(ix+5)`
	`res 5,(iy-9)`
	`set 1,(hl)`
		Set or reset the Nth bit of a byte in memory. The address can be in `hl` or an offset from the index registers `ix` and `iy`.

Shifting registers	`srl b`
	`sla a`
	`sra e`
		`srl` and `sla` mean "shift right logical" and "shift left arithmetic." Move each bit one place left or right, and put in a 0 at the end. `sra`, or "shift right arithmetic," performs sign extension, which makes it work like signed division.

Rotating registers	`rr h / rrc h` `rl c / rlc c`	Rotate left/right without or with carry. `rlc` and `rrc` directly rotate bytes as groups of 8 bits. `rr` and `rl` rotate as groups of 9 bits, with the carry flag forming the 9th bit.

Bit manipulation can be used for a wide variety of operations, including masking (setting a group of bits to 0 or 1), packing (putting multiple numbers into the same byte/word), and working with flags. If you get into assembly programming, you'll have a chance to explore the many uses of these instructions, but one in particular stands out.

MULTIPLICATION AND DIVISION

Recall from our discussion of binary that each place in a binary number is worth twice as much as the place to its right and half as much as the place to its left. If you move all the bits in a number one place to the left, you're making each worth twice as much: you've doubled the number. If you move all the bits one place to the right instead,

each is now worth half as much, and you've halved the number. Therefore, you can use `srl`, `sla`, and `sra` to perform multiplication and division by powers of 2.

Working with numbers and data is great but isn't particularly useful without program flow features like loops, jumps, and functions. Let's take a high-level look at program flow control in z80 assembly.

12.5   Functions and control flow

No real-world programming language would be complete without jumps and loops, commands that you can use to avoid reinventing the wheel in every program, and functions (subprograms) to hold code you use frequently. Assembly offers all of these features, and in this final section you'll see how each can be used. We'll begin with functions and commands.

12.5.1   Using bcalls and ASM functions

When I discussed math and numbers in assembly, you saw instructions such as `add`, `set`, and `ld`. These instructions correspond directly to operations your calculator's CPU performs. Often, you'll want to repeatedly run a more complex operation in your code; in TI-BASIC, you might have used a command or made a subprogram for this. You'll see the ASM equivalents to both commands and subprograms in this section.

A BCALL REFRESHER

In TI-BASIC, you learned to use commands, things like `Disp`, `Output`, `ClrDraw`, `randInt`, and `Menu`. Each of these commands performed some moderately complex function, certainly something that the processor couldn't do with a single simple math operation like a load or an add. These commands are composed of large chunks of code inside the TI-OS.

Your calculator's OS exposes many similar commands that can be used from within your assembly programs. Called ROM calls or bcalls, each reads arguments from registers and/or memory, performs some operation, and may return values back to registers and/or memory. The following code from the Hello World program runs one bcall that takes a pointer to a string in the `hl` register and displays it at the cursor coordinates defined in memory locations `currow` and `curcol`, and it runs a second bcall that takes no arguments but moves the cursor down one row.

```
ld hl,HelloWorldMessage
bcall(_puts)
bcall(_newline)
```

You can find a full listing of the available bcalls in a PDF called "TI-83 Plus System Routines" released by Texas Instruments.

MAKING AND USING FUNCTIONS

Bcalls are a complex sort of function, similar to the subprograms you created in TI-BASIC. In TI-BASIC, you probably used subprograms sparingly; in assembly, you should use functions as liberally as possible. You can call functions from other functions with wild abandon; the main caveat is that when a function finally ends, it should

almost always leave the stack as it originally found it, by making sure it pushed and popped the same number of times. Every function begins with a left-aligned label that names the function and can include arbitrary instructions, loops, jumps, calls, and bcalls. To call a function, you use the `call` instruction; the `ret` instruction ends a function and makes it return to whatever function called it.

You're still stuck with code that runs exactly the same way every time. For code that can make choices, you used comparisons and conditions in TI-BASIC.

12.5.2 Conditionals and jumps

In assembly, you still use comparisons and conditionals to control the flow of your programs. There aren't even built-in loops; you create your loops with conditionals, labels, and jumps. You can conditionally do several things in z80 assembly: conditional jumps, conditional calls, conditional bcalls, and conditional returns. Every conditional instruction runs—or not—based on one of the six flags in the `f` register. You must either use a math operation or a comparison operation to set or reset flags in `f`. Comparisons must be performed between the accumulator `a` and some 8-bit register or number using the `cp` instruction; a comparison is similar to a subtraction, except that a isn't changed. For example, `cp 5` when a is 5 will set the z (zero) flag; `cp 7` when a is 5 will set the carry flag, because 5–7 causes an underflow.

Jumps in z80 assembly are performed with the `jr` or `jp` instruction, both of which operate like `Goto` in TI-BASIC. Both instructions will jump to a named label; `jr` takes up fewer bytes in your program but can only jump a short distance forward and backward. `jp` takes up an extra byte but can jump anywhere in memory. This code will compare a to 42 and then jump to `MeaningOfLife` if a holds 42:

```
cp 42
jr z,MeaningOfLife
```

Just as you saw `For`, `While`, and `Repeat` loops written in terms of comparisons and jumps in chapter 4, you compose loops out of comparisons and jumps in z80 assembly.

12.5.3 Loops in z80 assembly

Assembly loops are composed by figuring out the initialization needed, what should happen in each iteration of the loop, and what condition should make the loop end. It's up to you as a programmer to decide whether to have the comparison at the beginning of the loop, like a `While` loop, or at the end, like a `Repeat` loop. The left side of figure 12.4 shows a `Repeat`-style loop in z80 assembly.

One final instruction is great for loops that run a specific number of times, similar in function to a `For` loop in TI-BASIC. The `djnz` instruction, short for "Decrement and Jump If Not Zero," takes a label as an argument. Each time it runs, it decrements the `b` register and jumps to the specified label if b is not zero. You set b to the number of times you want the loop to run before the loop, as shown at right in figure 12.4.

Program flow in z80 assembly can range from simple combinations of calls and jumps to complex stack manipulation, but once you learn to think like an assembly
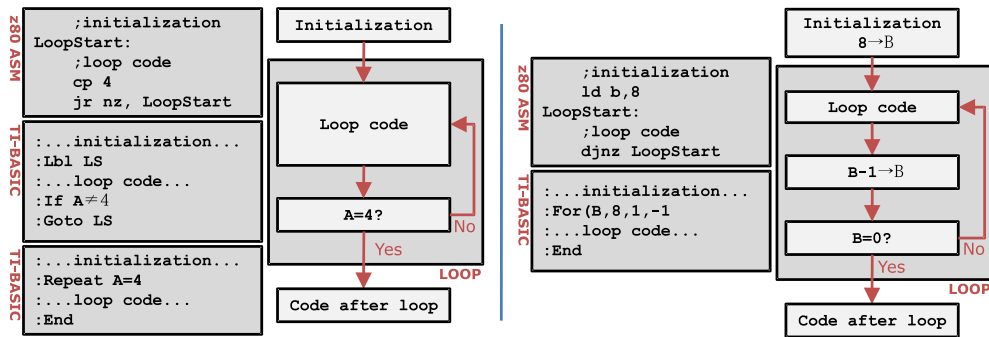
Figure 12.4   Conditional `Repeat/While`-style loops in z80 ASM (left) and one type of `For`-style loop (right)

programmer, you'll likely discover that program flow is at least as intuitive as in TI-BASIC and perhaps easier.

12.6   *Summary*

z80 assembly is a topic to which you'd need an entire book to do justice, or at least a few chapters to build the fluency to create programs like those in figure 12.5. In this chapter you saw a brief overview of numbers, registers, math, and program flow in assembly, as well as tools and resources for ASM development. By now you should have enough insight into the language to pursue more extensive tutorials and references
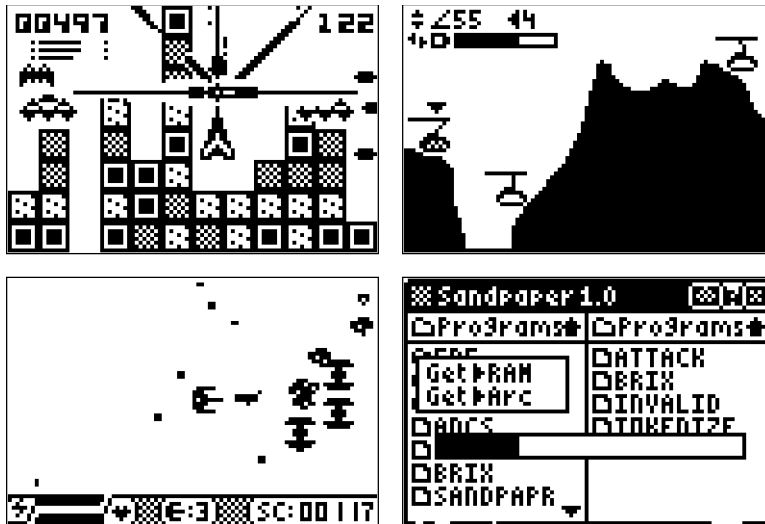
Figure 12.5   Four examples of assembly games and programs in action. Clockwise from top right: "Invalid Tangram" (a puzzle/space shooter), "Obliterate" (a scorched-earth game), and "Sandpaper" (an FTP client/server) by the author; "Phantom Star" by Joe Pemberton.

and to start writing your own programs. As with TI-BASIC and any other language, reading others' code is one of the best ways to deepen your understanding of the structure and flow of assembly code. If you want to learn more about assembly, visit the forums listed in appendix A for additional resources.

In the next and final chapter, we'll explore where you might want to go from here. We'll examine the calculator projects you could attempt in BASIC and ASM and the hardware projects you might try. I'll introduce new programming languages that you might want to take for a spin and platforms that you might enjoy working with.

13

Now what?
Expanding your
programming horizons

This chapter covers

- Going further with calculator programming
- Exploring programming for computers, mobile devices, and the internet
- Working with hardware development and modification

TI-BASIC can be used to create programs and games running the gamut from extremely simple to surprisingly complex on your calculator. You learned input and output, program flow and logic, and event loops, graphics, and the use of strings, matrices, and lists. You proceeded to advanced topics like optimization, programming hybrid TI-BASIC, and a toe dip into the pool of z80 assembly. From here, you have limitless programming possibilities open to you, from increasingly complex calculator programs and participation in the larger calculator programming and development community to computer programming and electronics hardware development.

This capstone chapter will teach you about the many places you can go from here with programming and engineering. We'll begin with a look at how you can continue the calculator programming journey, from pursuing more TI-BASIC experience and learning z80 assembly to publishing your programs, sharing your knowledge, and

working with other calculators besides the TI-83+/84+. I'll discuss computer and web programming, enumerate some of the popular languages currently used for each, and explain how the programming and problem-solving skills you've learned will carry over to new languages and platforms. You'll see how you might explore hardware development, both with your calculator and with popular microcontroller development kits like the Arduino.

Because you've spent the last 12 chapters learning to master the programming capabilities of your TI-83+/84+ graphing calculator, let's begin with how you can guide yourself toward further calculator programming knowledge and achievements.

13.1 *Taking your calculator programming further*

The primary topic of the past 12 chapters has been TI-BASIC programming, starting with the simplest lessons on input/output and leading up to complex, optimized, interactive programs and games. You may well consider yourself quite experienced with TI-BASIC, particularly if you've been working independently to write increasingly complex programs and games of your own with the new lessons of each chapter as you read along. We also spent a chapter on some of the highlights of hybrid TI-BASIC, and I told you where you could find additional reference materials on the many functions that hybrid BASIC offers to your programs. Chapter 12 gave you a solid framework on which to build an education in z80 assembly, including the importance of hex, binary, decimal, bits, and flags, and let you window-shop through the types of commands and program flow used with ASM.

We'll start with how you can take your TI-83+/84+ programming further and then look at other graphing calculator platforms you might choose to investigate.

13.1.1 *Continuing with TI-83+/SE and TI-84+/SE programming*

I recommend that you continue to aggressively pursue at least one of the programming languages that we've worked through in the preceding chapters. You can continue to write TI-BASIC programs, discover additional features and tricks you can use, and expand to more fun and more useful projects. You can play with hybrid TI-BASIC libraries and read up on their features. You can explore the extensive power of z80 assembly, which comes at the cost of more complexity compared with writing TI-BASIC.

If you work by yourself, you can certainly find a wealth of good tutorials and resources to help you along, including those listed in appendix C. If you work with pure or hybrid TI-BASIC, you have a number of references available to you. If you choose to pursue z80 assembly, the current leading tutorial is "Learn TI-83 Plus Assembly in 28 Days v2.0," available at www.ticalc.org/archives/files/fileinfo/268/26877.html. Written in 2003 and revised in 2004, this comprehensive tutorial covers everything from the simplest assembly instructions to manipulating programs, creating graphics, and working with the keyboard. Unfortunately, it has several weaknesses, including recommending an outdated toolchain for assembling (I recommend the DCS SDK, as mentioned in chapter 12) and few sample programs to assemble and test.

For writing TI-BASIC, hybrid BASIC, and z80 assembly, most programmers find it more rewarding to participate in the several vibrant TI graphing calculator programming community forums. Appendix C lists the top forums, each of which is free to join and has a staff and membership of friendly, dedicated users. You can ask for help with your BASIC and ASM programs, show off your latest projects, and discuss the finer points of programming. Several of the forums also have a large subset of members who have expanded into web and computer programming and hardware development after their initial involvement with calculators and would be happy to help you get into those fields.

On these forums, you can search for threads that cover subjects you want to learn more about. If you can't find what you're looking for, choose a subforum that closely matches what you want to discuss, such as TI-BASIC or Your Projects, and start a thread of your own. Most forums follow common "netiquette," dictating being respectful, helpful, and writing with proper spelling and grammar, but be aware that each forum has its own userbase and set of specific rules.

PUBLISHING YOUR WORK

One of the great things about graphing calculator programming is that programs spread from calculator to calculator, student to student, and school to school. In addition, there are large archives of calculator programs online. The most complete archives are at www.ticalc.org/pub; the end of appendix C lists other archives that you might want to consider for uploading your programs and finding others' source code to learn from. Issues you might want to consider are keeping your programs and games you publish original, to avoid infringing on companies' copyrights or trademarks, including a readme.txt file that explains how to use your program or game, and whether you want to release your projects under your real name or a pseudonym. I, for example, have used the pseudonym "Kerm Martian" for many years to avoid publishing my real name online.

13.1.2 *Programming other graphing calculators*

The TI-83+/SE and TI-84+/SE are great calculators to use for math and for programming. They're easy to use, and everything you need to get started as a TI-BASIC programmer is built into the calculator. Once you master TI-83+/84+ coding, you may decide to look at what other alternatives are out there.

The TI-89 is an older TI calculator but one of the most revered for advanced mathematics and engineering, because it can do symbolic algebra, trigonometry, and calculus. It has since been replaced by the TI-Nspire CX CAS calculator. The two calculators that directly follow from the TI-83+/84+ in terms of target audience and features are the Casio Prizm and the TI-Nspire CX, both compared to the TI-84+SE in table 13.1. The Casio Prizm is the less-expensive option, with a slightly weaker processor but a bigger screen. The TI-Nspire CX is more expensive but has a faster processor and more memory.

Table 13.1 **Other graphing calculators you might explore, the Casio Prizm and the TI-Nspire CX, as compared to the TI-84+ Silver Edition**

	TI-84+ SE	Casio Prizm	TI-Nspire CX
Processor and speed	6 MHz Zilog z80	58–102 MHz Renesas SH3/4	132 MHz ARM
Screen	96 x 64 monochrome	396 x 224 full color	320 x 240 full color
RAM	24 KB user/32 KB total	61 KB user/2 MB total	16 MB user/64 MB total
Archive/Flash	163 KB user/512 KB total	16 MB user/32 MB total	20 MB user/100 MB total
Communication	9.6 Kbps serial/mini USB	Serial/mini USB	Serial/mini USB/ dock connector
Programming Languages	TI-BASIC, z80 ASM	Casio BASIC, C, C++	Lua

From a programmer's perspective, the TI-Nspire CX is more locked down than its Casio counterpart. The Nspire CX offers a BASIC language that can't perform the input and output of TI-83+/84+ BASIC, though it supports the more powerful Lua language. You can't write C, C++, or assembly for the calculator without "jailbreaking" it, and anyone who wants to use your programs must go through the same jailbreaking process. The Casio Prizm offers a built-in BASIC language, plus you can write C, C++, or even SH3/4 assembly for the calculator using community-developed tools. As of this writing, I recommend the Prizm over the Nspire for programmers, but as the two lines of calculators develop, the balance may well change. Be sure to research both and make your own decision if you choose to explore a new model of calculator for math or for programming.

Calculators certainly provide far from the only programming opportunity out there; computers, the internet, and even mobile devices offer many more ways to stretch your coding expertise.

13.2 *Expanding your programming horizons*

Hearken back to chapter 1, where you first learned the characteristics of TI-BASIC. In introducing the fact that it's an interpreted language, I presented the distinctions

between interpreted and compiled languages and gave a few examples of each. Languages like C, C++, Python, Java, and JavaScript were mentioned, languages that are currently popular for computer programming. The programming and problem-solving lessons that you've learned throughout these chapters can be applied to almost any programming language you might try to learn, and I encourage you to see if computer or web programming might be right for you. Calculator programming is fun for the challenges it provides, especially making powerful, fun, and useful programs with little memory and processing power and a tiny screen, whereas computer/web programming is rewarding in seeing what you can do with more capabilities and a wider audience.

A summary of programming languages you may want to consider is shown in table 13.2. It's far from exhaustive; there are hundreds or thousands of languages currently in use. If you're particularly interested in one platform, such as computers, the internet, or mobile devices like smartphones, then your options will be limited by the platform.

You'll find that as you learn more programming languages, each new language is easier to learn, because at a certain point you're just learning new syntax to apply to your existing knowledge about building the structure and flow of programs. Which

Table 13.2   Popular programming languages with their target platforms and significance

Language	Platform	Description
C	Computers	One of the most widely used of all modern programming languages. Similar to TI-BASIC in program flow but with functions used extensively to run repeated sections of code. Used for programs and games on almost every operating system.
C++	Computers	An object-oriented extension of C. Code and data are often contained in instances of objects, containers that can hold member functions and data fields. Used for programs and games.
C# ("C Sharp")	Computers (mainly for Windows)	A relative newcomer, a just-in-time (JIT) compiled language, similar to Java. Easy to create graphical games and GUI programs, but mostly used for Windows programs.
Java	Computers and mobile devices	Java is object-oriented and is used for a huge range of games and programs. It can run on computers, smartphones, and even in web pages as Java applets.
JavaScript	Web programming (client side)	Javascript is used to write dynamic web pages. It runs inside the user's browser when a page is loaded.
PHP	Web programming (server side)	PHP is usually used to create web pages, generating HTML (Hypertext Markup Language, which is *not* a programming language) to construct web pages. It's often used with a database program that organizes data, such as MySQL.
Python	Computer and web programming	An interpreted language used to write command-line and graphical programs and games. Well suited for beginners, with lots of libraries for many types of programs.

language you choose to start with depends on your personal preferences and your programming goals, such as whether you want to eventually get a programming job, write programs and games for yourself, or build websites. If you want to get a general feel for computer programming without making too much of a commitment to a language, I recommend Python. The syntax is somewhat familiar for a TI-BASIC programmer, and all you need is a Python interpreter and an editor like Notepad++ to try writing programs. It runs on all major operating systems. Programs are interpreted and are usually distributed as source code, so it's easy to learn from others' projects.

Another popular beginner choice is Java, because it not only runs on computers but can also be executed on many mobiles devices including tablets and phones and can be embedded in web pages. If you prefer to work at a lower level, and the power and control of something like z80 assembly appeals to you, then C or C++ might be more your speed. If your interest lies more in creating interactive web pages or web applications, you'll want to learn PHP and Javascript, as well as how to design the look and feel of web pages with the descriptive "languages" HTML and CSS.

For many programmers, it's not enough to make devices do your bidding through software. Hardware development is a fun and increasingly accessible field to explore if you have an interest in electrical engineering or are just curious about what makes gadgets tick.

13.3   Working with hardware

Hardware development describes the broad field of building and programming circuits, microcontrollers, or embedded devices. Embedded processors and microcontrollers are generally slower and lower powered than computers or mobile devices, and they have the processor, memory, and other circuitry combined into a single integrated circuit. Hardware development often involves building hardware that connects to one of these microcontrollers or embedded systems and then writing software to control your hardware.

TI-83+/84+ graphing calculators can be used for hardware development, because they have a serial port at the bottom that you can control with z80 assembly. Among the many projects that have used this port are sound and music programs, networking systems, and attempts to enhance the calculator with more memory. The TI-84+/Silver Edition's USB port has been used to connect to USB flash drives and to Bluetooth and Wi-Fi modules. More ambitious developers have opened the calculator and added backlights, touchpads, and extra ports and have overclocked the processor.

Let's begin with a few hardware projects that could start you on hardware development with your calculator, along with a few important warnings, disclaimers, and caveats.

13.3.1   Calculator hardware and modifications

Recall from chapter 1 that your calculator is essentially a full computer. It has a processor, long- and short-term memory, and input and output devices. It also has the

ability to connect to the outside world using its serial I/O port and on some calcula-tors a mini-USB port. If you're careful and respectful of your calculator, you can safely use it to get started with simple electronics development, communicating with exter-nal hardware or even modifying your calculator itself.

Be aware that if you do pursue hardware development, you can *damage* or *destroy* your hardware. With programming, the worst you can do is generally crash your calcu-lator and be forced to reset it or reload the operating system. When you start connect-ing hardware or opening the calculator, you risk permanently and irreparably breaking the device. If you're not willing to potentially break your calculator if you're not careful, you should probably steer clear of using it for hardware development and instead take a look at some of the microcontroller development options presented in the next section. You can moderately safely test audio output and networking with your calculator's link port, but if you start attaching more complex hardware like LEDs, circuits, and especially batteries, you risk damaging the device. If you open the case, you might accidentally damage the mainboard or the delicate cables that con-nect the screen to the mainboard.

If you want to test sound output, all you need is a 2.5mm to 3.5mm stereo adapter, available from some audio or electronics stores. Programs like Piano83, mobileTunes 3, and QuadPlayer play sound through the link port by oscillating it quickly between 0 and 5 volts, which you can hear through headphones attached to an adapter. Networking is straightforward with the CALCnet protocol, which requires no extra hardware other than the link cables that come with the calculator. It can also be used to connect your calculator to the internet and to other calculators across the internet. With a few tran-sistors, resistors, and batteries, you can make your calculator turn LEDs on and off. The right side of figure 13.1 shows testing a circuit that flashes LEDs when link activity occurs on a CALCnet network.

Hardware modifications for TI-83+ calculators, which are easier to modify than TI-84+ calculators, have included overclocking, backlights, case modifications, PS/2 and 3.5mm audio ports, and touchpads. The left and middle of figure 13.1 show an example of a calculator modified by the author with a PS/2 port, a repainted case, a backlight, and a touchpad. If you want to attempt modifications of your own, be aware that the TI-83+ has more space inside to fit additional electronics and a simpler LCD

Figure 13.1   Calculator hardware projects in action. The "Ultimate Calculator 2," left and center, with several hardware modifications; experimenting with calculator networking using CALCnet, at right.

mainboard cable, which is somewhat repairable if you have a steady hand and good soldering skills. You'll also need a Torx-6 screwdriver to remove six of the seven screws that secure the case.

Why bother working with calculators and hardware? It's a great way to understand more about what makes your calculator or any piece of computing technology tick, and as you get into more complex projects, you'll have to delve further into memory, registers, busses, synchronization, and the many aspects of building hardware devices. If you want to get more serious about developing complex devices that don't rely on a computer for control, you should consider working with microcontrollers or embedded hardware.

13.3.2 *The wonderful world of microcontrollers*

A range of easy-to-use kits have been created to introduce beginners to microcontrollers and embedded development. Figure 13.2 shows three of the most popular current systems, the Arduino, BeagleBone, and Raspberry Pi.

The Arduino is a simple platform, built around an 8-bit Atmega microcontroller and shown at left in figure 13.2. It runs at 16 MHz, contains 32 KB of ROM and 2 KB of RAM, and has a host of digital and analog inputs and outputs to interface with hardware. You can connect LEDs, LCDs, sensors, and extra boards called shields that add functionality.

The BeagleBone is a much more powerful device, with a 720 MHz ARM processor, similar to the CPU in many smartphones. Pictured at center in figure 13.2, it has 256 MB of RAM and uses a 4 GB SD card as a hard drive. It has USB ports, an Ethernet port, and more than 60 I/O pins to connect to hardware, and it runs a full version of Linux. But it's about three times as expensive as the Arduino and is overkill for many simple projects.

The Raspberry Pi is somewhere in between the two. It's closer to the price of the BeagleBone than the Arduino but runs a 700 MHz ARM processor with 256 MB of RAM. The final version is the size of a credit card and roughly resembles the prototype at right in figure 13.2. It has USB, Ethernet, and HDMI ports and also has I/O pins to connect hardware.

Figure 13.2   Three microcontroller/embedded development platforms. From left, the Arduino Uno, the BeagleBone, and an early prototype of the Raspberry Pi.

WHY WORK WITH EMBEDDED HARDWARE?

With an embedded system like any of the three discussed here, you're responsible for constructing peripheral hardware and writing the software to control and interface with that hardware. Although you have a basic set of software and functions to help you, you must make most of the hardware and software design decisions that will dictate how the complete system works. Indeed, with a bit of work, you can make any of the three function like a real computer (or calculator), complete with input devices and a screen. If you want to test the basics of microcontroller development, I recommend starting with the Arduino. It's much simpler than the BeagleBone or Raspberry Pi, and the C-like language that you use to write your programs, or "sketches," is straightforward.

You've now seen how you can go further with calculator software and hardware development and with the wider fields of programming and embedded hardware; I'll leave you with a few final thoughts.

13.4   *Final thoughts*

We've taken quite a journey together, from the rudimentary basics of graphing calculator programming up through the complexity of highly optimized programs, using assembly-powered hybrid libraries, and even experimenting with ASM itself. In this chapter, we looked at new horizons you can explore from here, including continuing to program calculators, working with calculators and hardware, and learning with embedded devices. I hope that it has been as fun and rewarding for you to follow and learn from as it has been for me to write, and I hope I've given you some of my enthusiasm for programming. The skills you need to be an effective calculator programmer can help you become a great programmer of other languages, a better problem solver, and perhaps even a great engineer.

I wish you fun and success in continuing to pursue programming calculators and perhaps programming computers and other devices and working with hardware. I encourage you to publish programs you create in the community and to seek out fellow programmers and developers at the forums listed in appendix C to present your work and ask for or offer assistance as necessary; perhaps I'll see you around Cemetech. Good luck!

appendix A
Review: using
your calculator

Your graphing calculator is a powerful piece of computing hardware, able to perform many of the functions of a full-sized computer, as discussed in chapter 1. As you embark on learning to program, in particular to program on a calculator, you'll find yourself drawing on many of the skills that you developed as a calculator user. You'll need to be able to navigate around the calculator's menus and be familiar with storing data in variables. You should have a basic familiarity with how the calculator does math, including typing in equations for the calculator to solve, its rules for grouping and order of operations, and the strengths and limitations of the device.

From the moment you unwrap a new TI-83+ or TI-84+ graphing calculator, you can use it to make math and science easier and more understandable even without programming. As befits a calculator, it can do arithmetic, trigonometry, and algebra. It can be used to solve equations and systems of equations. To fulfill the other half of its name, "graphing," your calculator can draw rectangular, polar, parametric, and recursive graphs and even perform numerical calculus on the graphscreen, including differentiation and integration. You can examine multiple lines graphed at once, calculate areas and points of intersection, and generate tables of graphed points. You can perform statistical analysis on collections of numbers and points. You can use a range of drawing tools to annotate graphs and to draw on the graphscreen.

In this appendix, I'll review common calculator skills every beginner programmer should know, so you'll have the necessary basic skill set when you begin to read chapter 2. If you feel that you already are proficient with the lessons of this appendix, you're more than welcome to jump straight to chapter 2 and begin to learn about the output and input commands used in programs. If you run across a concept that you don't understand, you can flip back to this appendix as a reference section.

I'll begin by discussing what your calculator is and can do and, equally important, what it can't do. I'll continue with an overview of navigating through your calculator's menus and present the function of each of the menus you'll need as a programmer, to familiarize you with where to find each of the calculator's features. I'll proceed to how to do math on the calculator and how to use variables that store a single number and then expand to variables that can store one-dimensional and two-dimensional sets of numbers as well as sequences of letters. I'll conclude with a review of using your graphing calculator to graph equations and the basics of the important skill of moving programs and files between your calculator and your computer.

As you start out on your calculator-programming career, you'll find yourself constantly drawing on your preprogramming calculator skills. One of the most fundamental is navigating the calculator's menus to find commands and features, and thus this is the first skill that I'll review.

A.1   Navigation and menus

As you use your calculator, you'll find that not everything can be accessed from the homescreen, the main screen your calculator goes to when you turn it on, the screen where you do math. Many functions and tokens are hidden inside menus, which you access by pressing certain keys on the calculator. One such menu you may have encountered before is the Y= menu, where you enter equations that you can then graph by pressing the [GRAPH] key. There are numerous menus, exemplified by the six shown in figure A.1.

Almost every menu in your calculator has either the name of the menu or a set of tabs in the first row of the screen, followed by items to be selected. In figure A.1, the Catalog menu has no tabs, just a menu title (CATALOG) and a list of tokens. You can use the arrow keys to scroll up and down the items and press [ENTER] to select one of the entries. The Math menu, by contrast, has four tabs: MATH, NUM (Numerical), CPX (Complex), and PRB (Probability). You can use the left- and right-arrow keys to switch between the tabs; the currently selected tab is drawn in white text on a black background.

The items in a menu may be commands or symbols that when selected will be pasted into the homescreen, the program editor, the Y= menu, or wherever you were before you entered that particular menu. They may also be nested menus with their own options. Whenever you see a menu item that ends in an ellipsis (…, three dots), it means that that option leads to another menu. In figure A.2, which shows the Link menu accessed with [2nd][X,T,θ,n], all seven of the onscreen options shown lead to other menus, because they each end with an ellipsis. In addition, you know that there are more items to be found if you scroll down, because the last item on the screen shows a down-arrow symbol instead of a colon between its item number and its name. Similarly, there are more menu items above, because the first item has an up-arrow symbol between its number and name.

To select an item in a menu, you can use the up and down arrows to move the white-on-black highlight to that item and press [ENTER], or you can directly type the number
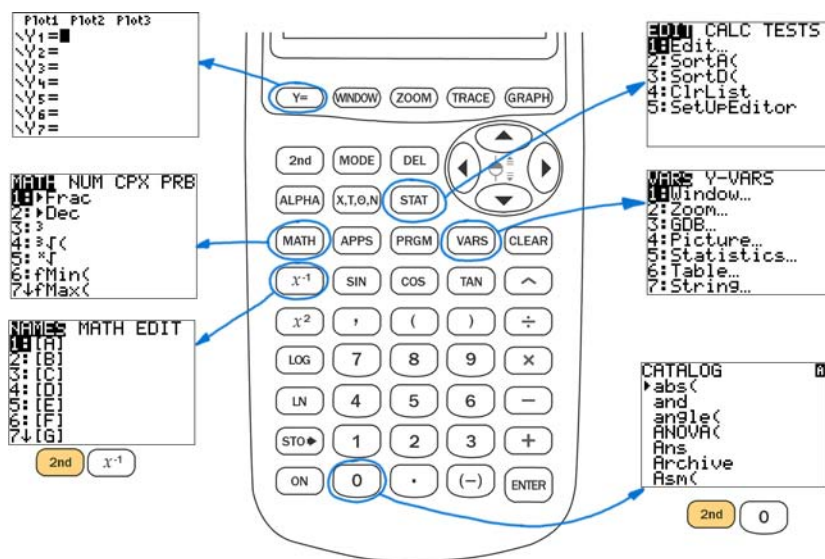
Figure A.1   Six examples of the many menus of the TI-83+/TI-84+ graphing calculator. Clockwise from upper right, the Statistics menu, containing three tabs for working with statistics; the Variables menu; the Catalog menu with a list of all the typeable commands on your calculator; the Matrix menu with the names of the matrix variables and a matrix editor; the Math menu; and the Y= menu for entering functions to graph. Some of these are accessed by pressing a single key ([VARS] or [MATH], for example); others are accessed with a combination of keys ([2nd][0] to access Catalog, marked in yellow (TI-83+) or blue (TI-84+) above the [0] key on your real calculator's keypad).

or letter at the left margin of the screen next to the item. As mentioned, selecting most items that don't lead to additional menus will paste the item shown into the editor you were using most recently. If you enter a menu from the Y= equation editor, and you select a token like `int` from the NUM tab of the Math menu, then `int(` will be pasted into whichever equation you're currently editing. If you select the same option after entering the Math menu from the program editor, then `int(` will instead be pasted at the current cursor location inside that program. If you're not in any editor, the command or symbol will end up on the homescreen. If you wish to exit
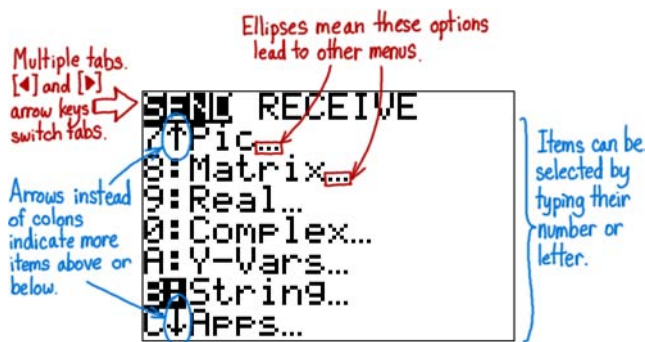
Figure A.2   The anatomy of any menu on your TI-83+/84+ (in this case the Link menu). The left- and right-arrow keys change tabs, the up- and down-arrow keys scroll the menu, and pressing [ENTER] selects the highlighted option. The number keys and the letter key combinations can be used to select items as well.

a menu without selecting an option, you can press [2nd][MODE] (Quit), or in some cases [CLEAR].

As a reference, table A.1 summarizes the TI-83+/TI-84+'s menus, including the key sequence to access each menu, the tabs within that menu, and for what the different items in that particular menu can be used.

Table A.1   The major menus of the TI-83+/TI-84+ calculator series and the purpose of each menu. Some menus have submenus, which are mentioned in the descriptions in the right column.

Menu	Access key(s)	Function
Y=	[Y=]	Enter functions to graph.
Stat Plots	[2nd][Y=]	Modify which, if any, statistical plots are displayed on the graph screen.
Window	[WINDOW]	Change the limits displayed on the graph. In rectangular mode, these are the X and Y values of the edges of the screen.
TblSet	[2nd][WINDOW]	Modify the settings for the table feature in [2nd][GRAPH].
Zoom	[ZOOM]	Zoom in and out, or reset the zoom to one of several presets.
Format	[2nd][ZOOM]	Change how graphs look, including turning the axes and grid on and off.
Calc	[2nd][TRACE]	Features that can be calculated from graphed functions, such as minima and maxima and intersections.
Mode	[MODE]	Modify settings, such as the graph mode, number of digits of precision in calculations, Real/Complex mode, Degree/Radian mode, and more. On the TI-84+/SE, also used for setting the time and date.
Link	[2nd][X,T,θ,n]	Select variables including programs to send to another calculator, or go into Receive mode to accept variables from another calculator.
Stat	[STAT]	Commands to calculate statistics from lists of numbers.
List	[2nd][STAT]	Shows all the lists on your calculator in the NAMES tab and commands for manipulating lists in the OPS and MATH tabs.
Math	[MATH]	The four tabs of the Math menu, MATH, NUM, CPX, and PRB, contain commands for working with real and complex numbers and for generating random numbers.
Test	[2nd][MATH]	Contains two tabs: TEST with the equality-testing symbols (=, ≠, >, <, ≥, ≤) and LOGIC with the Boolean operators (and, or, xor, not).
Apps	[APPS]	A list of all applications (not programs) on your calculator. Select any of them to run that app.
Angle	[2nd][APPS]	Angle entry and conversion commands and symbols.
Prgm	[PRGM]	If accessed from the program editor, contains three tabs (CTL, I/O, and EXEC) containing programming commands. From anywhere else, lists programs to run or edit and allows you to create new programs.
Draw	[2nd][PRGM]	Three tabs, DRAW, POINTS, and STO, for drawing shapes, points, and pixels and for storing and recalling pictures.

Table A.1 The major menus of the TI-83+/TI-84+ calculator series and the purpose of each menu. Some menus have submenus, which are mentioned in the descriptions in the right column. *(continued)*

Menu	Access key(s)	Function
Vars	[VARS]	Access most of the string, picture, GDB, and statistics variables in the VARS tab and the graph equation variables in the Y-VARS tab.
Distr	[2nd][VARS]	Calculate and draw probabilistic distributions from the DISTR and DRAW tabs.
Matrix	[2nd][x^{-1}]	Access the names of the 10 matrices in NAMES, commands to manipulate matrices under MATH, and a matrix editor in EDIT.
Mem	[2nd][+]	View, archive/unarchive, and delete any program, file, or variable on your calculator. Check the used and available RAM and archive.
Catalog	[2nd][0]	A nearly complete alphabetic list of the tokens in other menus.

You may have used some of the menus previously; we explore many more starting in chapter 2. Of particular interest for basic math is the Mode menu, found by pressing the [MODE] key.

A.1.1 Changing modes

Your calculator has a number of modes that govern how it performs math, handles complex numbers, and displays graphs. Almost all such modes are switched from the Mode menu, accessed by pressing the [MODE] key in the upper-left corner of the keypad. Figure A.3 shows the Mode menu on a TI-83+/SE (left) and TI-84+/SE (right); notice that the TI-84+ series also has a space to set the date and time for its internal clock.

In general, these modes need only be changed if you need to use engineering or scientific notation, you wish to change whether you calculate trig functions from angles in radians or degrees, you need to enter a different kind of graph equation, or you have to work with complex numbers.

Modifying these modes is not particularly useful if you don't know how to use your calculator for general math calculations, so let's continue with a review of how to perform math on your calculator and use different types of data storage.
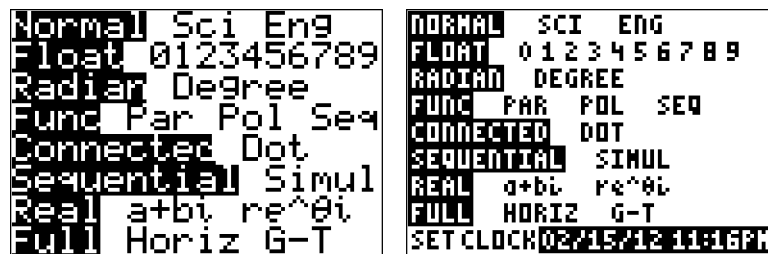
Figure A.3 The Mode menu for the TI-83+ (left) and TI-84+ (right). The options control the display of numbers, calculation of angles and trig functions, how graphs are drawn, how complex numbers are handled, and on the TI-84+ and TI-84+ SE, the current time and date.

A.2   Simple math, variables, and data types

Your graphing calculator excels at many tasks, particularly, as its name might lead you to believe, calculating and graphing. In this section, I'll provide a brief overview on using your calculator for arithmetic, for simple numbers as well as for one-dimensional lists of numbers and two-dimensional arrays of numbers. I'll also review storing numbers for later use and a related data storage task, saving and loading sequences of characters, called strings.

A.2.1   Math, Ans, and numeric variables

Every calculation performed on your calculator involves at least numbers and the homescreen. The homescreen, the blank area of your calculator with a blinking cursor that first appears when you turn the device on, is used for the majority of calculations for which you might need your graphing calculator. As you perform calculations, the equations that you enter accumulate along the left edge of the screen, and the results from numerical calculations appear along the right edge. If you fill up the screen, it will scroll to make room for new equations.

EQUATIONS AND KEYS

Other than entering equations and viewing the results of calculations, several useful key combinations help you use the homescreen:

- [CLEAR] wipes the homescreen and returns the cursor to the top-left corner. When you're typing in a mathematical expression, one tap on [CLEAR] blanks only the current line, and a second [CLEAR] clears the screen.
- [2nd][ENTER], or Entry, recalls the last equation that you typed in. You can press [2nd][ENTER] up to 10 times to move backward through the history of equations that you typed; only at most 10 entries are saved.
- If the result of a calculation is a long list or a large matrix, you can use the arrow keys to scroll around the list or matrix. Once you start entering the next equation you can no longer scroll around the previous answer.

Expressions that involve simple math, such as $2 + 2$, or $3 * (5 - 2)$, or $6.13 * \pi$, are all entered just as you might write them on paper or see written in a math book. Your calculator supports implicit multiplication, which means that $3 (5 - 2)$ is interpreted as $3 * (5 - 2)$ and that 2π is equal to $6.283185...$. Your graphing calculator follows the "PEMDAS" order of operations rules, meaning that in order, it evaluates parenthetical expressions, exponents, multiplication, division, addition, and subtraction. Take a look at the left side of figure A.4 for a gotcha when using things like division with a complex denominator, because the division operator takes precedence over (is evaluated before) the addition operator.

The right side of figure A.4 shows the proper use of implicit multiplication and the difference between subtraction and negation. The subtraction key, [−], produces the larger horizontal bar symbol; the negative key, [(-)], produces the smaller, superscripted minus
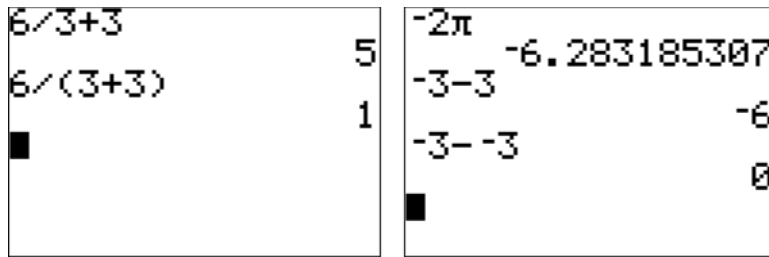
Figure A.4 **The importance of remembering operator precedence (left) and the proper use of implicit multiplication and the negative symbol (right). At the left side, PEMDAS precedence means that 6/3 is evaluated first, so 6 / 3 + 3 = 2 + 3 = 5. Using parentheses forces the result to be 6 / 6 = 1. On the right side, the negative symbol, the [(-)] key, is used to denote a negative number, and the subtraction key, [–], is used for subtraction.**

sign. If you confuse which is which, either your calculator will produce a syntax error or you'll get an incorrect result.

VARIABLES AND ANS

As you perform calculations, you may need to reuse the results of your previous calculations. You could look upward and type the numbers in again or write them down on a piece of paper and retype them, but those are both rather inefficient solutions, not to mention that you could make an error writing and typing the numbers. To save you from such problems, your calculator has a set of 28 variables that it can use to store numbers. Each variable is like a named box that you can put a number into; when you want to use that number, you can just refer to the name of the box. The variables are A through Z, θ (theta), and Ans. Each can hold a single number, and if you store another number into it, the first one is lost. You can store the result of a calculation by adding the [STO>] key's symbol, →, followed by the variable name to the end of an equation. The left side of figure A.5 demonstrates storing to the C variable and the M variable and then using those two variables in an equation.
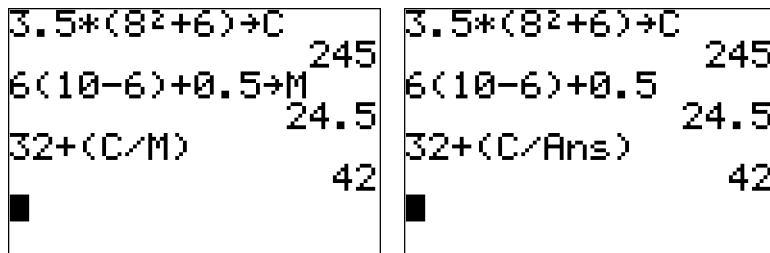
Figure A.5 **Using variables. The left side shows storing the results of two different calculations in C and M and then using those two variables. The quotient C/M is 10, so 32 + (C / M) = 42. On the right side, the value of C is stored as before, but the Ans variable is used instead of M, which in a program is a faster, space-saving alternative.**

`Ans` is a special variable: the result of every calculation performed is automatically stored into `Ans`, even if it's also stored into another named variable. Conversely, you can't explicitly add →`Ans` to the end of an equation, because this will produce an error. The right side of figure A.5 shows how you can use `Ans` instead of the M variable in the equations shown at the left side of figure A.5. You'll find that you can use variables and `Ans` in exactly the same way in your programs and that each line in your program that performs a calculation will set the `Ans` variable just as calculations on the homescreen store to `Ans`.

Just as variables can store single numbers, there are other data structures called lists and matrices, which also can be stored in their own special variables.

A.2.2   *Working with lists and matrices*

In TI-BASIC, sometimes a variable that can hold one number isn't enough. Consider tracking your earnings each month of the year for one year. For that, you'd need a sequence of twelve numbers. You could store them in variables A through L, but it would be easier to have a single structure or variable of some sort to store them. Such a variable is called a list, and it can hold between 1 and 999 numbers as a single unit. The names of six lists are written directly on the calculator's keypad, L_1 through L_6, typed by pressing [2nd][1] through [2nd][6]. You can also create your own lists with names between one and five characters long, which must be prefaced with the small L character at the bottom of [2nd][STAT][▶]. Lists are typed between curly braces, with commas separating elements:

```
{1,2,8}       ⟵── A three-element list          A one-element
{⁻9.3}                                          list                    A six-element
{⁻5,⁻3,⁻1,1,3,5}                                                        list
```

The rules for math on lists are:

- Adding, subtracting, multiplying, and dividing lists perform the given operation on respective elements of two lists, for example, {1,2} + {5,6} = {6,8}.
- You can perform math on two lists only when they're the same length.
- You can perform the same operations between a list and a single number, for example, 3 ∗ {1,2,3} = {3,6,9} and {10,15,20} / 5 = {2,3,4}.
- You access individual elements of a list with the name of the list followed by a number in parentheses. $L_1(2)$ is the second element of L_1, ∟MINE(799) is the 799th element of ∟MINE, and $L_5(M)$ is the Mth element of list L_5.

When you store a list to a list variable, that list variable holds the list just as with numeric variables discussed in section A.2.1. Because lists can be of any size between 1 and 999 elements, there's a command that's used to get and set the length of lists, `dim`. If you type the following two commands,

```
{5,8,9,100}→L1
dim(L1)
```

your calculator will display 4. If you store a number to the dim of a list, you'll manually change its size to that number. If the list grows as a result, any new elements will be filled with zeros; if the list shrinks, elements will be deleted from the end. There are many additional commands for working with lists in the three tabs of the List menu in [2nd][STATS].

A list holds a sequence of numbers; a matrix (sometimes called an array) stores a two-dimensional set of numbers. The matrix variables are named [A] through [J], and you're limited to only those 10 matrices. The matrix names can be accessed from the NAMES tab of the [2nd][x⁻¹] Matrix menu; typing the three characters "[", "A", "]" is not the [A] matrix (see the "Tokens versus text" sidebar in chapter 2). The MATH tab of the [2nd][x⁻¹] menu contains functions for working with matrices, and the EDIT tab lets you directly edit matrices.

If you'd like to type out a matrix from the homescreen instead of using the matrix editor, you'd type each successive row as a pair of square brackets enclosing elements separated by commas and wrap the whole thing in a pair of square braces. A few examples with their properly drawn equivalents are shown in figure A.6. Matrices can contain up to 99 by 99 elements, and individual elements of a matrix may be extracted or stored to with the syntax [matrix](row,column), for example:

```
54→[B](4,2)     ⟵   Stores to row 4, column 2 of matrix B        This will return 7,
[B](4,2)/8                                                        since 54 / 8 = 7
```

Like lists, matrices work with the `dim` command, except for matrices the dimension is a two-element list, {rows,columns}. As with lists, you can use the `dim` command to check or change the size of a matrix. Adding and subtracting matrices is performed element-wise and thus must be performed with two matrices of the same size, but matrix multiplication and division are performed on full matrices.

A graphing calculator must excel at calculations, and your trusty TI-83+/SE or TI-84+/SE graphing calculator certainly does. But it must also be a pro at graphing, and I'll now move on to rendering graphs.

```
[[2,3][8,5]]        [[2,3,4][8,5,9]]        [[1,2,3,4][-1,-2,-3,-4][5,6,7,8]
                                             [0,0,0,0][9,8,7,6][0.1,0.2,0.3,0.4]]

  ⎡ 2  3 ⎤            ⎡ 2  3  4 ⎤                     ⎡  1    2    3    4  ⎤
  ⎣ 8  5 ⎦            ⎣ 8  5  9 ⎦                     ⎢ -1   -2   -3   -4  ⎥
                                                      ⎢  5    6    7    8  ⎥
[[9][99.9]]         [[0,1,0,3,5]]                     ⎢  0    0    0    0  ⎥
                                                      ⎢  9    8    7    6  ⎥
  ⎡  9   ⎤            [ 0  1  0  3  5 ]                ⎣ 0.1  0.2  0.3  0.4 ⎦
  ⎣ 99.9 ⎦
```
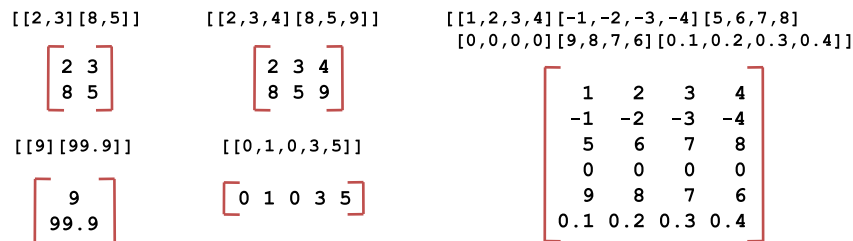
Figure A.6   The TI-83+/84+ representation of several matrices, along with their properly drawn forms. Notice how row and column matrices are represented and that the elements of rows are grouped together.

A.3   *Graphing and the graphscreen*

Your calculator can take various kinds of equations, plug in values for the independent variable(s), and display the results as one or more graphed lines. In this section, I'll remind you of the skills for entering such equations, viewing the graph, and modifying how the resulting graph appears.

Before you can render a graph, you must first enter the equation(s) that define that graph. Your calculator can render graphs in four different modes:

- In *Function* (Func) mode, you can enter up to 10 equations in the form $Y = f(X)$, that is, equations that use X as the independent variable and produce values for Y.
- *Parametric* (Par) mode lets you specify up to six pairs of equations for X and Y, parameterized by the variable T.
- For *Polar* (Pol) mode, you enter up to six equations for the dependent variable r (radius) in terms of the angle θ (theta).
- *Sequential* (Seq) mode calculates iterative functions, where each value of the series depends on previous values.

Function mode is by far the most commonly used mode, so it's on that mode that I'll focus. If your calculator isn't already in Function mode, you can press [MODE] to choose the current graph mode.

To enter one or more equations to graph, you first press the [Y=] key at the top of the keypad. You'll see several rows of Y= functions. If any of them already has an equation entered, moving the cursor to that line and pressing [CLEAR] will remove it. When you enter an equation for one of the Y= functions, the equals sign will turn to white on black to indicate that that function is enabled, as shown at the left side of figure A.7. If at some point you want to disable a function without deleting it, move the cursor over its equals sign and press [ENTER] to toggle the function on and off. Once you're satisfied with the equations you've entered, press the [GRAPH] key and wait while the graph is drawn. The right side of figure A.7 shows the graph for the equations in the same figure, with the calculator's default graph window.
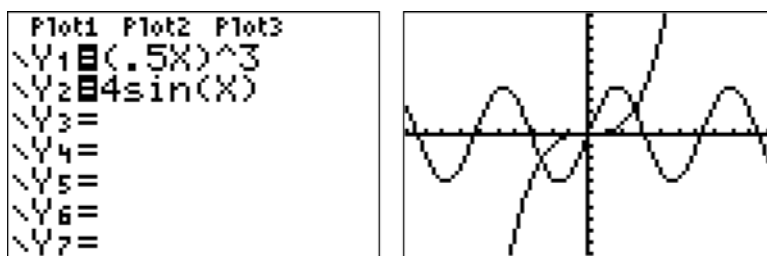
Figure A.7   Entering equations in the Y= equation editor and viewing the respective graph. Note the highlighted equal signs that indicate enabled functions. If you change the line type at the far left of each equation, you can make that function be drawn in a bolder line or shaded above or below.

```
WINDOW              ZOOM MEMORY
 Xmin=-10           1:ZBox
 Xmax=10            2:Zoom In
 Xscl=1             3:Zoom Out
 Ymin=-10           4:ZDecimal
 Ymax=10            5:ZSquare
 Yscl=1             6:ZStandard
 Xres=1             7↓ZTrig
```
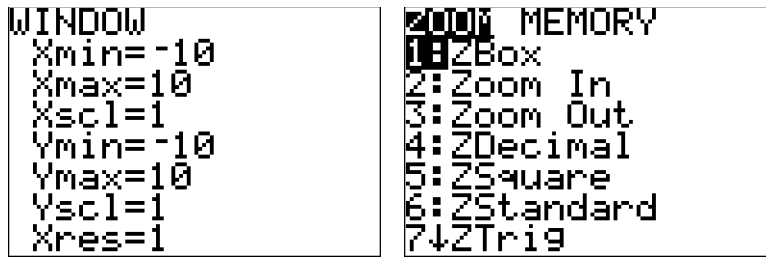
Figure A.8   Modifying the window used for graphing from the Window menu (left), and zooming in, out, and to one of several presets from the Zoom window (right).

A.3.1   Zooming and modifying the window

If you'd like to change the window, the X and Y values of the edges of the screen, you can use the Window menu or the Zoom menu. In the Window menu, you can manually set the Xmin, Xmax, Ymin, and Ymax values of the screen edges, as well as the graphing step and the spacing between the ticks on the axes. You can zoom in and out or reset to the default zoom from the Zoom menu. The Window and Zoom menus are accessed respectively with the [WINDOW] and [ZOOM] keys directly below the screen; figure A.8 shows these menus.

One final set of skills that is important for any calculator user, especially a programmer, is the ability to transfer files and programs between your calculator and computer.

A.4   Uploading and downloading programs and files

As a programmer and a calculator user, you'll find that saving your own programs and using others' programs are valuable skills to have. If you want to share your programs with other people across the internet or even just back them up in case your calculator crashes, you need a way to transfer them to your computer. Likewise, if you want to try other programmers' works or examine the source code of other TI-BASIC programs to learn from them (which I strongly encourage), you need a way to get programs from your computer or the internet onto your calculator.

Like any good computer, your calculator has a way to communicate with the outside world. If it is a TI-83, TI-83+, or TI-83+ Silver Edition, the bottom edge of the case will have a hole slightly smaller than a headphone jack that you can plug a cable into. If you have a TI-84+ or TI-84+ Silver Edition, the top edge of your calculator will have that jack as well as a standard mini-USB socket. Figure A.9 shows the respective connection ports of the four different main calculator models I address in this section.

To transfer files between your calculator and a computer, you need two things: a cable and software. You need a cable to connect your calculator and your computer; two types of cables are in wide use. For any of the calculators mentioned in the previous paragraph, you can use a SilverLink cable. It has a 2.5mm stereo plug at one end (a smaller version of a headphone plug) and a full-sized USB plug at the other end. As

Serial I/O port

Mini-USB port

Figure A.9   The serial (I/O) and/or mini-USB ports at the edges of, from left to right, the TI-83+, the TI-83+ Silver Edition, the TI-84+, and the TI-84+ Silver Edition

the name implies, it is silver, with a silver box halfway along its length. With the introduction of the TI-84+ and TI-84+ Silver Edition, graphing calculators entered the modern world of USB; both of the TI-84+ calculator models have a mini-USB port at the top-right edge of the calculator. For those models, you can use any standard mini-USB cable to connect your calculator to your computer.

The second thing you need is a piece of software on your computer that lets it talk to your calculator. Unfortunately, regardless of which cable you use, your computer can't automatically talk to your calculator. Unlike Casio's Prizm graphing calculator, TI's calculators can't appear as a folder, what you might see if you plugged a USB flash drive into your computer. On Windows and Mac OS, you can use Texas Instruments' own TI-Connect software. Alternatively, you can use TiLP II, a piece of software written by the calculator programming community, which works on Linux as well as on Windows and Mac OS. You can get TI-Connect from in the Downloads section of http://education.ti.com/, or if you'd prefer TiLP, from www.ticalc.org/pub/win/link/. TI-Connect is the official TI software. If you have problems with it, you can contact Texas Instruments for help, but it is known among calculator programmers as occasionally having bugs and problems, especially with the direct USB (mini-USB) cable.

A.4.1   *Installing and using linking software*

Both TI-Connect and TiLP have a fairly simple installation process; both contain documentation that explains how to install them. Both require installing special drivers that let your computer understand how to communicate with the cable and/or calculator, so make sure that you install the software before you plug your calculator cable into your computer.

Both TI-Connect and TiLP have a simple two-pane interface to send files between your calculator and your computer, as shown in figure A.10. With TI-Connect, you use the TI DeviceExplorer program. If it finds your calculator, it will display the full contents of the device, including programs, pictures, strings, and settings. You can either drag items out of this window into folders on your computer or drag files into the window to transfer them to your calculator. If you are instead using TiLP, start TiLP and select the appropriate cable from the menu that appears; the program will connect to your calculator. In the two-pane window that appears, you will need to press the DirList button to get a list of all the files on your calculator, after which you can drag
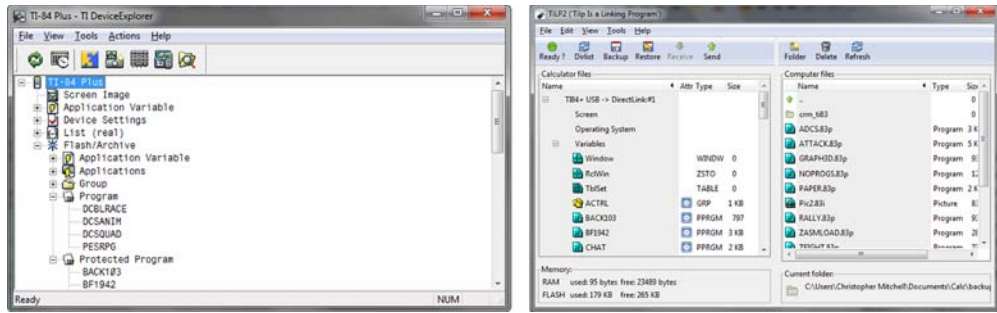
Figure A.10 TI-Connect's TI DeviceExplorer software (left) and the open-source TiLP software (right). Both are used to transfer files between a computer and a graphing calculator.

programs from right to left to transfer them to your calculator or left to right to transfer them to your computer. If you have TI-Connect, you can also quickly send a file to your calculator by right-clicking it in its folder and choosing Send to TI Device.

Calculator programs and games are generally distributed either as .8xg (group) files, which contain multiple programs and can also hold pictures, strings, and lists, or individual programs and other files with .8x* extensions (such as .8xp, .8xl, .8xs, .8xi, and others). Most programs include a file called a readme that explains which files to transfer to your calculator and how to use the particular program or game.

If you have difficulty installing or using TI-Connect or TiLP, I strongly encourage you to visit this book's forum or my website's forum (see appendix C) and post a topic describing your difficulties.

You now know how to save your programs to your computer, how to restore them to your calculator, and how to find and load other programmers' projects onto your calculator for testing and examination. You can certainly learn a lot from others' code, but you can also learn a lot about becoming a better and more efficient programmer from troubleshooting your own code, a vital skill for any programmer.

A.5 *Summary*

In this appendix, I reviewed basic calculator skills for the TI-83+/SE and TI-84+/SE lines of graphing calculators. I discussed navigating menus, doing math, using the graphing features, and transferring files and programs between your calculator and a computer. With these skills under your belt, you have most of the background that you need to be an effective calculator programmer and to use your calculator to learn the general concepts of programming. Your remaining training will come from using your calculator in everyday life, perhaps occasionally exploring new features and/or while learning to program beginning in chapter 2.

appendix B
TI-BASIC
command reference

TI-BASIC comprises many commands for interacting with users, creating graphics, manipulating program flows, and working with math concepts. Throughout the first 11 chapters, many commands are discussed; this appendix presents all of these commands, plus extras, as an easy-to-use tabular reference. Each section references the chapters where the concepts in that section are first introduced, so that you can look back for more detail.

This appendix is organized to mirror the progression of the chapters. Input and output are first introduced, followed by the control-flow commands. The third section focuses on the many TI-BASIC commands to manipulate graphics and functions on the graphscreen. Math and data manipulation commands are presented fourth, and the appendix concludes with the hybrid BASIC commands.

Each command is presented with a short description and a sample usage scenario. For some commands, the arguments are shown separately where I feel additional clarity is necessary. Unless otherwise specified, arguments shown in square brackets (`[` and `]`) are optional and may be omitted. To prevent confusion, optimizations such as omitting closing quotes and parentheses are not performed in the Usage column. Many commands can be found in their own menus. Most can be found in the Catalog, [2nd][0]. Variables such as strings and pictures are in the [VARS] menu.

B.1   *Input and output*

Most of the commands in table B.1 work only on the homescreen; `getKey`, `Input`, and `Pause` also work on the graphscreen. Chapters 2 and 6 cover this content.

Table B.1   Input and output commands

Command	Description	Usage
ClrHome	Clears the homescreen and resets the cursor for Disp to the top row.	`:ClrHome`

Disp	Displays a number, string, matrix, list, or line of text. If you separate several items with commas, each will go on a line of its own. Strings are left-aligned on the LCD; all other items are right-aligned.	`:Disp 32`
		`:Disp "HELLO, SARA"`
		`:Disp 5,L1,[A],"CODE"`

getKey	Checks if any keys are pressed in a nonblocking manner (always returns immediately). Returns 0 for no key or one of the key codes listed in chapter 6 if a key is pressed. Can detect only one key at a time.	`:Repeat K=45`
		`:getKey→K`
		`:End`

Input	Displays a string and waits for the user to type in a value to store in a string, variable, matrix, or list. If used on the graphscreen (with no arguments), will allow the user to move a cursor, and sets X and Y.	`:Input "VALUE:",X`
		`:Input "HOW MANY?",N`
		`:Input M`
		`:Input`

Output	Displays a number, string, numeric variable, matrix, or list at a specified row and column on the homescreen. Row can be 1 to 8; column can be 1 to 16.
	Output(row,column,item_to_display)
		`:Output(4,7,"ROSE")`
		`:Output(8,15,42)`
		`:Output(7,2,Str5)`

Pause	Stops execution and waits for the user to press [ENTER]. Optionally, can be given a string, number, or variable to display during the pause. If the item is a long string, list, or matrix, the user can scroll left, right, up, and down.	`:Pause`
		`:Pause 3`
		`:Pause "PRESS ENTER"`
		`:Pause [C]`
		`:Pause LALIST`

Prompt	Displays the name of a variable, and asks the user to enter a value for that variable. Works with numeric variables, strings, lists, and matrices.	`:Prompt X`
		`:Prompt Str8`
		`:Prompt [C]`
		`:Prompt L4`

B.2   Conditionals and control flow

The conditional and control-flow commands in table B.2 are covered in chapters 3 and 4. The Boolean logic commands in table B.3 are discussed in detail in chapter 3.

Table B.2   Conditional and control-flow commands

Command	Description	Usage

Else	Marks the end of the true block of code and the beginning of the false block of code for an If/Then/Else/End conditional.	`:If X=2:Then`
		`:Disp "X IS 2"`
		`:Else`
		`:Disp "X IS NOT 2"`
		`:End`

End	Marks the end of a Repeat, While, or For loop or the end of the false block for an If/Then/Else/End conditional.	`:For(R,-10,10,2)`
		`:Disp R`
		`:End`

Table B.2   Conditional and control-flow commands *(continued)*

Command	Description	Usage

For	Loops a fixed number of times, modifying a variable called the loop variable on each iteration. `For` loops end with an `End` command and can contain zero or more commands inside the body of the loop. `For(variable,start,end[, step])`	`:For(X,1,10)`
		`:Disp X`
		`:End`
		`:For(M,3.0,0.5,⁻0.1)`
		`:N+M→N`
		`:End`

Goto	Jumps forward or backward to a named label in the same program and continues executing there. Throws an error if the label is not found.
	`Goto label`
		`:Goto NX`
		`:Lbl 1`
		`:...code...`
		`:Goto AA`
		`:Lbl NX`
		`:...code...`

If	Evaluates a comparison or conditional statement and determines if it is true or false. `If` can be used to run or not run a single line of code or with `Then`/`End` or `Else`/`Then`/`End`, multiple lines of code.	`:If X<0`
		`:Disp "NEGATIVE"`
		`:If N=42`
		`:Then`
		`:X+1→X`
		`:3N→N`
		`:End`

Lbl	Marks a spot in the program to which a `Goto` can jump. Execution flows straight through this command if there is code before it.	`:5→M`
		`:Lbl A`
		`:M^4→M`
		`:If M<999999`
		`:Goto A`

Menu	Draws a menu of one to seven items with a title. Lets the user use the arrows, number keys, and [ENTER] key to select an item. Each item in the menu, when selected, makes the program jump to a named `Lbl`.
	`Menu("TITLE","OPTION 1",label_1`
	`[,"OPTION 2",label_2,...]`
		`:Menu("MY GAME 1.0",`
		`  "PLAY",P,"HELP",`
		`  H,"QUIT",Q1)`

prgm	When followed by the name of a program, runs that program as a subprogram from the current program.	`:Disp "RUNNING SUBPRGM"`
		`:prgmSUBPRGM`
		`:Disp "RAN SUBPRGM"`

Repeat	A loop that continues to execute until the specified condition becomes true ("repeat until…"). The condition is checked at the end of each iteration, so the loop always runs at least once.
	`Repeat condition`
		`:Repeat K=105`
		`:getKey→K`
		`:End`
		`:Disp "ENTER PRESSED"`

Return	Exits from the current program to the program that called it. If it was run from the homescreen or shell, returns there instead.	`:Return`
Stop	Exits from this program and every program that called this program, back to the homescreen. If the program was run by MirageOS, returns to the homescreen; if from Doors CS, returns to Doors CS.	`:Stop`

Table B.2   Conditional and control-flow commands *(continued)*

Command	Description	Usage

`Then`	Separates an `If` command with a condition from a block of two or more statements run only if the `If` condition is true. The block must end with `End`.	`:If X>3 and X<6`
		`:Then`
		`:ClrHome`
		`:Disp "X IS 4 OR 5?"`
		`:End`

`While`	Begins a loop that loops only while the given condition is true. The condition is checked at the beginning of each loop, unlike `Repeat`, so it might be run zero times.
	`While condition`
		`:1→X`
		`:While X<999`
		`:2X→X`
		`:End`

Table B.3   Boolean logical operators

Command	Description	Usage
`and`	Returns the logical `and` of the operands on either side. Returns true (1) if both operands are nonzero. `Boolean and Boolean`	`:If 2<3 and 6>5` `:Disp "SANITY RETAINED!"`
`not`	Returns the logical `not` of the argument: true (1) if the argument is false (0), false (0) otherwise. `not(boolean_value)`	`:If not(X>0)` `:Disp "X IS NEG/ZERO"`

`or`	Returns the logical `or` of the operands on either side. Returns true (1) if either operand is nonzero, false (0) otherwise.
	`Boolean or Boolean`
		`:If X<10 or X≥100`
		`:Disp "X DOES NOT HAVE",`
		➥ `"TWO DIGITS"`

`xor`	Returns the logical `xor` of the two operands. This is true (1) if exactly one operand is false and false (0) if both are true or both are false.	`:If X=0 xor Y=0`
		`:Disp "X OR Y MUST BE",`
		➥ `"0, BUT NOT BOTH"`

B.3   *Working with graphics*

The TI-BASIC commands to manipulate graphs and graphics on the graphscreen in table B.4 are introduced in chapters 7 and 8. Pixel-based (`Pxl-`) commands and `Text` are presented in chapter 7, and graphs and the point coordinate system and commands are discussed in chapter 8.

Table B.4   Graphics commands for manipulating and drawing on the graphscreen

Command	Description	Usage
`AxesOn` `AxesOff`	Turns the graphscreen axes on or off.	`:AxesOn` `:AxesOff`

`Circle`	Draws a circle with center (X,Y) and radius R. Special feature: extra argument {i} makes it fast.
	`Circle(X,Y,R[,{i}])`
		`:Circle(0,0,8)`
		`:Circle(5,-2.4,5.5)`
		`:Circle(0,0,8,{i})`

Table B.4   Graphics commands for manipulating and drawing on the graphscreen *(continued)*

Command	Description	Usage
ClrDraw	Clears the graphscreen and possibly redraws the axes, grid, and any enabled equations.	`:ClrDraw`
DispGraph	Switches to displaying the graphscreen.	`:DispGraph`
fMin fMax	Finds the minimum or maximum of a function within given bounds. `fMin(equation,var,low,high)`	`:fMax(3sin(X),X,0,1)` `:fMin(4cos(T),T,5,5.1)`
fnInt	Calculates the integral of (area under) a function between given bounds. `fnInt(equation,var,low,high)`	`:fnInt(sin(U),U,0,π)`
FnOff FnOn	Turns one or all graph functions on or off. With a numeric argument, toggles that equation.	`:FnOff` `:FnOn 2`
Func	Sets the graph mode to Function mode.	`:Func`
GridOff GridOn	Turns the graphscreen grid on or off.	`:GridOff` `:GridOn`
Horizontal	Draws a horizontal line at the specified Y coordinate.	`:Horizontal 6.5`
Line	Draws a line from (X1,Y1) to (X2,Y2). Either coordinate can be offscreen. A fifth argument of 0 erases the line instead of drawing it. `Line(X1,Y1,X2,Y2[,0])`	`:Line(5,5,4,6)` `:Line(0,0,4,-2.6,0)`
nDeriv	Takes the derivative (slope) of a function at a point. `nDeriv(equation,var,X)`	`:nDeriv(X²,X,5)`
Par	Sets the graph mode to Parametric mode.	`:Par`
Pol	Sets the graph mode to Polar mode.	`:Pol`
Pt-On Pt-Off	Turns a point (X,Y) on (black) or off (white). Point may be offscreen.	`:Pt-On(6.04,90)` `:Pt-Off(3.11,87)`
Pt-Change	Turn a point (X,Y) from white to black or vice versa.	`:Pt-Change(B,C)`
Pxl-On Pxl-Off	Turns a pixel at (row, column) on or off. Must be onscreen.	`:Pxl-On(31,47)` `:Pxl-Off(62,94)`
Pxl-Change	Switches a pixel (row, column) between on and off.	`:Pxl-Change(13,37)`
Pxl-Test	Returns 1 if a pixel (row, column) is black, 0 if white.	`:If Pxl-Test(4,5)` `:Disp "PXL IS ON"`
StoreGDB RecallGDB	Stores or recalls a graph database of current functions and graph settings (axes, grid, window, zoom, etc).	`:StoreGDB 5` `:RecallGDB 0`
StorePic RecallPic	Stores or recalls a picture of the graphscreen.	`:StorePic 9` `:RecallPic 1`

Table B.4 **Graphics commands for manipulating and drawing on the graphscreen** *(continued)*

Command	Description	Usage
`Seq`	Sets the graph mode to Sequential.	`:Seq`
`Shade`	Shades the graphscreen between two functions. Optionally specifies min and max X. `Shade(low_f,high_f[,min_x,max_x])`	`:Shade(⁻10,10)` `:Shade(X,X2,1,5)`
`String▶Equ` `Equ▶String`	Converts a string to a Y equation, or back.	`:String▶Equ(Str0,Y1)` `:Equ▶String(Y4,Str5)`
`Tangent`	Draws the line tangent to a curve at a given X. `Tangent(equation,X)`	`:Tangent(Y1,5.5` `:Tangent(X2+4X+1,0.5)`
`Text`	Draws a string (or number) at (row,column). Inserts -1 as the first argument to draw using large font. `Text([-1,]row,column,string[,...])`	`:Text(6,6,"42=",42)` `:Text(⁻1,5,4,"HELLO")`
`Vertical`	Draws a vertical line at the given X-coordinate.	`:Vertical -5.693`
`ZStandard`	Sets the standard window, with Xmin=Ymin=-10 and Xmax=Ymax=10.	`:ZStandard`
`ZSquare`	Increases the Xmin and Xmax values or the Ymin and Ymax values to make each pixel represent a square (not a rectangle).	`:ZSquare`

B.4   Number and data type commands

Chapter 9 introduces commands to work with real and complex numbers, strings, lists, and matrices. Those commands are summarized in this section, grouped by the type of data they manipulate.

B.4.1   Numbers

Commands to manipulate and generate numbers are presented in table B.5.

Table B.5 **Number generation and manipulation commands**

Command	Description	Usage
`abs`	Returns the absolute value of the argument: itself if it is positive, or its positive version if it is negative.	`:If abs(5)=abs(⁻5)` `:Disp "ABS WORKS"`
`angle`	Returns the angle of a complex number in the complex plane, equal to $\tan^{-1}(\text{imag}(X)/\text{real}(X))$.	`:angle(3i+4)→θ`
`conj`	Returns the complex conjugate of a complex number, found by negating the complex part.	`:If conj(1+2i)=1-2i` `:Disp "CONJ WORKS"`
`Degree`	Sets the angle mode to Degree.	`:Degree`
`DelVar`	Deletes a numeric variable, which can also be used to set a numeric variable to zero.	`:DelVar A` `:DelVar M`

Table B.5   Number generation and manipulation commands

Command	Description	Usage
Fix	Fixes the number of decimal places displayed, regardless of the precision of the number.	`:Fix 0` `:Fix 6`
Float	Sets the number display format to floating-point: the number of decimal places displayed changes.	`:Float`
fPart	Returns the fractional part of a real number, the part after the decimal point.	`:fPart(52.409)`
imag	Returns the imaginary part of a complex number.	`:imag(8.3i-9)`
int	Rounds down to the nearest integer.	`:int(-95.2)`
iPart	Returns the integer part of a number. Works like `int` for positive numbers, differently for negative numbers.	`:iPart(5.3)`
Radian	Sets the angle mode to Radian.	`:Radian`

rand	Returns a random decimal between 0 and 1. If given an argument, returns a list of that many numbers. Also used to seed the random number generator.	`:If rand>0.5`
		`:Disp "50-50 CHANCE"`
		`:6.45→rand`
		`:rand(99)`

randBin	Returns one or a list of binomially distributed numbers with binomial parameters n and p. `randBin(n,p,[count])`	`:randBin(4,0.4)` `:randBin(8,0.25,5)`
randInt	Returns one or a list of random integers between and including two bounds, low and high. `randInt(low,high[,count])`	`:randInt(5,10)` `:randInt(-10,10,40)`
randNorm	Returns one or a list of normal or Gaussian-distributed random numbers. `randNorm(mu,sigma[,count])`	`:randNorm(0.5,1)` `:randNorm(0,1.5,6)`
real	Returns the real part of a complex number.	`:real(-2+4.1i)`

round	Rounds up or down. Specifies the number of decimal places, 0 to round to the nearest integer.
	`round(value,decimal_places)`
		`:round(5.5,0)`
		`:3.14159→X`
		`:round(X,3)`

B.4.2   Strings

The few commands for working with strings are in table B.6.

Table B.6   Manipulating string variables and data

Command	Description	Usage
expr	Evaluates a string as if it was a math equation, and returns the resulting value.	`:expr("3X+2")→B`
inString	Searches for a substring in a string. Returns the index of the substring if found, or zero otherwise. `inString(haystack,needle[,start])`	`:inString("HELLO","LO")` `:inString("CATC","C",2)`

Table B.6 Manipulating string variables and data *(continued)*

Command	Description	Usage
`length`	Returns the length of a string, in tokens.	`:Disp length("HELLO")` `:length("NODROFF")→D`
`sub`	Returns a substring of a larger string. `sub(string,start,length)`	`:sub("THRESH",3,3)`

B.4.3 Lists and matrices

Some commands are unique to lists or matrices, but other commands can be used with both data types. Both classes of commands are presented in table B.7.

Table B.7 Some helpful commands for manipulating lists and matrices

Command	Description	Usage

`dim`	Gets or sets the size of a list or matrix. Always takes the name of the list or matrix as an argument. If you store to it, changes size. If you use it as a value, returns size. Lists have numeric sizes 1 to 999; matrices have sizes that are two-element lists formatted as {rows, columns}. Matrices can be 1 x 1 up to 99 x 99.	`:Disp dim(L₅)`
		`:45→dim(ᴸAREA)`
		`:dim(ᴸGROW)+1→dim(ᴸGROW)`
		`:{5,3}→dim([E])`
		`:dim([A])→ᴸMSIZE`

`Fill`	Sets every element of a list or matrix to a given real or complex value.	`:Fill(0,ᴸAREA` `:Fill(4i-6.2,[G]`
`min` `max`	Finds the smallest or largest element of a list. Also used for optimization tricks with strings of `or` or `and` logic. See chapter 10 for more information.	`:If min(LVALS)>0` `:max({4,1,9,0})→X`
`seq`	Creates a list by plugging values to an equation. Specifies the equation, variable to plug into, and start and end values. Optionally, specifies a step, as with `For`. `seq(equation,var,start,end[,step])`	`:seq(4X+3,X,0,8,2)` `:seq(9-B,B,⁻7,7)`
`SortA` `SortD`	Sorts a list in ascending or descending order, and returns the list. Does not modify the original list, so you must use the store (→) operator.	`:SortA(ᴸ1)→ᴸ1` `:SortD(LHS)→ᴸHS`

`sum`	Returns the sum of all the elements in a list. If given optional start and end arguments, calculates the sum of a portion of the list. If you specify just a start index, sums to the end of the list.
	`sum(list[,start_index[,end_index]])`
		`:sum(ᴸ3)→B`
		`:Disp sum({A,B,C})`
		`:sum(ᴸ5,5,10)→D`

`rref`	Reduces a matrix to Reduced Row-Echelon form. This can be used to solve a system of simultaneous equations, by putting the coefficients in a matrix and calling `rref` on the matrix.	`:[[5,1,2][1,5,3]]` `:Pause rref(Ans)`

B.5   *Hybrid BASIC commands*

The four commands in table B.8 are used to run hybrid library functions. A number is supplied as the first argument to each to choose which of the library's functions will actually be run. `det` can be used with a single argument to check which libraries are present.

Table B.8   Accessing commands from the major hybrid BASIC libraries

Command	Description	Usage
det	If used with a 1 x 1 matrix [[42]], checks for hybrid libraries. If det([[42]]) is 1337, all libraries are available. If 0, all except the DCSB Libs are available. If 42, no libraries are available. Also used to call Celtic III functions, mostly used for data, file, and program manipulation.	`:If 1337=det([[42]])` `:Then` `:Disp "NEED HYBRID LIBS"` ➡ `,"DCS.CEMETECH.NET"` `:Return` `:End` `:det(20,"C9C9")`
identity	Runs a function from the PicArc library of hybrid BASIC commands.	`:identity(10,6,2,2,` ➡ `"INVERTED TEXT")`
real	Executes a function from xLIB, mostly graphics, keyboard, and program commands.	`:real(12,8,5,5,40,40,1)`
sum	Runs a DCSB Libs function, which works with graphical user interface (GUI) elements, among miscellaneous other features.	`:sum(6,47,31)`

appendix C
Resource list

Over the past two decades, Texas Instruments and a dedicated, vibrant community of third-party hackers and programmers have independently built a large variety of their own tools, tutorials, reference materials, and perhaps most important, programs and games. From Texas Instruments, you can download OS updates and software to connect your calculator to your computer, but the company offers no programming assistance. The author's website, Cemetech ("KEH-meh-tek"), provides program downloads, tutorials, a forum for programming help and project discussions, an online TI-BASIC editor and TI-83+ emulator, and news about recent calculator developments. The third major resource is www.ticalc.org, which has the definitive archive of programs and games for your graphing calculator. This appendix provides links to these three websites and a few other smaller sites with handy resources, and it lists specific highlights within the various websites that you might find useful, including tools, program downloads, and tutorials.

First, I'll enumerate links to some of the most useful resources if you get stuck with learning to program or working on a project: discussion forums to talk to experts.

C.1   Programming and project help and discussions

Although you might work hard to learn the contents of this book and rigorously apply the debugging and planning lessons in chapter 5, you may find yourself with a problem you don't know how to fix. You might also discover that you're stuck on a particular command or concept that doesn't make sense to you, no matter how many times you stare at it used in programs. When that happens, your best recourse is to reach out to experts. One of the best things about the graphing calculator programming community is that its members are, on the whole, friendly and willing to help out new users. My own website, Cemetech, has for over a decade been helping thousands of users with calculator programming, computer and web

programming, and DIY hardware and electrical engineering. As far as discussions, a few areas of the site may be particularly helpful:

- Discussion subforum specifically for this book, *Programming the TI-83 Plus/TI-84 Plus* (www.cemetech.net/forum/f/70 or http://cemete.ch/f70)
- Discussion for TI-BASIC topics (www.cemetech.net/forum/f/19 or http://cemete.ch/f19)
- General list of discussion forums (www.cemetech.net/forum)

There are other discussion forums online as well, with varying degrees of experience and numbers of users. In addition, there are assorted sites that specialize in programming help in different languages. Some of these websites:

- *TI-Planet*—French projects and programming help (http://tiplanet.org)
- *TI-Freakware*—TI-89/83+/84+ help and tutorials (http://tifreakware.net)
- *Omnimaga*—TI-Nspire/83+/84+ programming help (www.omnimaga.org)
- *MaxCoderz*—Inactive reference for z80 ASM (www.maxcoderz.org)
- *United-TI*—Inactive BASIC and ASM reference, now absorbed by Cemetech (www.unitedti.org)

The flagship calculator community downloads site, www.ticalc.org, unfortunately does not have general discussion boards.

As you begin to program, you may find that you need more resources than just your calculator. These might include a virtual calculator to test programs or to work on projects when your calculator is far from you.

C.2   *Tools and emulators*

Over the two decades that enterprising programmers and hackers have been trying to make their graphing calculators do more, the community has created a number of polished tools, from emulators to IDEs (integrated development environments) to linking software.

An *emulator* is a piece of software that lets you run a virtual device on another device; figure C.1 shows screenshots of several emulators. These emulators let you run a virtual calculator on your computer (or, for jsTIfied, on anything with a web browser):

- *jsTIfied*—An online emulator written in JavaScript that emulates the TI-83+ and TI-84+ calculators (www.cemetech.net/projects/jstified)
- *Wabbitemu*—An emulator that runs on your computer; also emulates the TI-83+ and TI-84+ calculators (http://wabbit.codeplex.com)
- *PindurTI*—A more accurate emulator and has four linked calculators, but you need to memorize all the keys (http://sgate.emt.bme.hu/patai/pindurti)

To use an emulator you need a ROM image, a copy of your calculator's OS. Legally, you must get the ROM image from your own calculator. The best tool for the job is called ROM8x (www.ticalc.org/pub/dos/rom8x.zip).
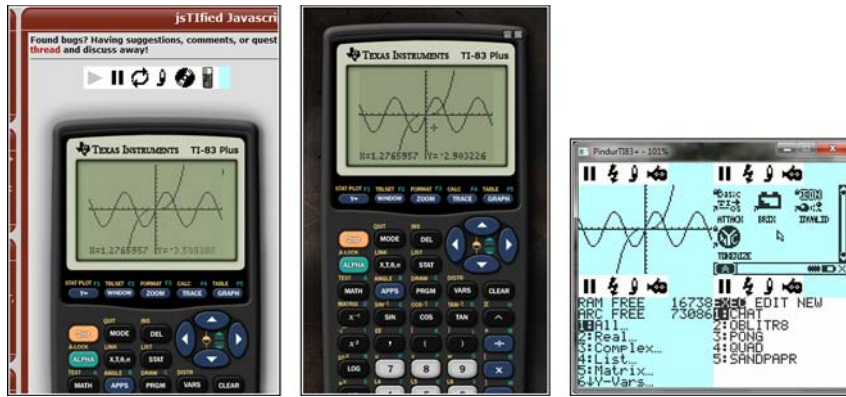
Figure C.1 TI-83+/84+ emulators, from left to right, jsTIfied, Wabbitemu, and PindurTI

Just as an emulator lets you use a virtual calculator, an IDE lets you edit TI-BASIC (or z80 ASM) programs on your computer. For TI-BASIC, two options exist: SourceCoder and TokenIDE. For z80 ASM, the Doors CS SDK and WabbitCode are both good options.

- *SourceCoder 2.5*—An online TI-BASIC IDE, connected to the jsTIfied emulator; also can turn .8xp into text source code and vice versa and can convert lists, pictures, matrices, and other files (http://sc.cemetech.net)
- *TokenIDE*—An offline TI-BASIC IDE; includes a sprite editor (http://cemete .ch/DL515)
- *Doors CS 7 SDK*—A complete z80 ASM assembly toolkit for TI-83+/84+ calculators; works well with the Notepad++ text editor (http://cemete.ch/DL470)
- *WabbitCode*—Part of the Wabbitemu project (http://wabbit.codeplex.com)

If you use an emulator and/or an IDE, you'll want to transfer programs and files between your computer and calculator. As a programmer, it's a good idea to back up your projects from your calculator, and as a calculator user, you'll want to try programs you find online. Two major options for linking software exist:

- *TI-Connect*—Texas Instruments' official software, used to transfer files and take screenshots (http://education.ti.com/calculators/downloads)
- *TiLP*—Community-made linking software that works on Linux, Windows, and Mac OS (http://lpg.ticalc.org/prj_tilp/)

These are specific excellent tools, but many other tools exist. In addition, there are tens of thousands of calculator games and programs available for free download, plans and videos of hardware projects for calculator projects, and tutorials for specific skills. The next section presents these resources.
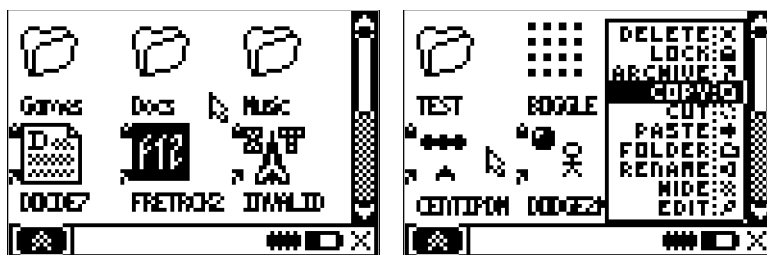
Figure C.2   Two sample screenshots of the Doors CS 7 desktop, showing programs and folders

C.3   *Downloads and tutorials*

In the heyday of the budding TI graphing calculator community, quite a few program archives were maintained on different websites. As the community matured, the archives gradually merged or fell into obsolescence; at present, two main websites maintain archives, ticalc.org (www.ticalc.org) and Cemetech (www.cemetech.net). Ticalc.org, the flagship community website, has about 40,000 programs for graphing calculators; www.ticalc.org/pub/83plus has math and science programs, games, graphics and sound programs, and more. The Cemetech archives are smaller and more specialized and can be found at www.cemetech.net/programs.

Many programs need to be executed with a special program called a shell rather than directly from the homescreen, because shells add extra functions for programs to use. Although a shell called MirageOS was at one time the popular choice, it's now outdated and doesn't work properly on many modern calculators. The recommended shell du jour is Doors CS (currently Doors CS 7, figure C.2), which can be found at http://dcs.cemetech.net. Its interface uses a mouse, just like most computer operating systems, and it can run almost every kind of TI-83+/84+ program.

Besides the TI-Connect software, Texas Instruments' official website (http://education.ti.com) also has OS updates and other downloads and documentation. It has no tutorials; for those you should visit Cemetech, www.ticalc.org, and TI-Freakware's collection of TI-BASIC, hybrid BASIC, and z80 ASM tutorials (http://tifreakware.net/tutorials/).

index

317