Programming the
TI-83 Plus/TI-84 Plus

Christopher R. Mitchell
Foreword by Brandon Wilson

MANNING
Chapter 1
**brief contents**

**PART 1  GETTING STARTED WITH PROGRAMMING........................1**

1  ■  Diving into calculator programming  3  
2  ■  Communication: basic input and output  25  
3  ■  Conditionals and Boolean logic  55  
4  ■  Control structures  76  
5  ■  Theory interlude: problem solving and debugging  107

**PART 2  BECOMING A TI-BASIC MASTER .............................133**

6  ■  Advanced input and events  135  
7  ■  Pixels and the graphscreen  167  
8  ■  Graphs, shapes, and points  184  
9  ■  Manipulating numbers and data types  205

**PART 3  ADVANCED CONCEPTS; WHAT’S NEXT........................225**

10  ■  Optimizing TI-BASIC programs  227  
11  ■  Using hybrid TI-BASIC libraries  243  
12  ■  Introducing z80 assembly  260  
13  ■  Now what? Expanding your programming horizons  282
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 streamline your software. But you might be asking yourself why you should bother learning calculator programming 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.

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 operations, 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 students and now have advanced degrees or high-paying jobs in programming and engineering. 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.](image)
**Table 1.1  A side-by-side comparison of a calculator and computer**

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

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.

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

<table>
<thead>
<tr>
<th></th>
<th>Interpreted language</th>
<th>Compiled language</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execution speed</td>
<td>Slower</td>
<td>Faster</td>
</tr>
<tr>
<td>Preprocessing</td>
<td>None needed</td>
<td>Source code compilation</td>
</tr>
<tr>
<td>Syntax error checking</td>
<td>During execution</td>
<td>Before execution</td>
</tr>
<tr>
<td>Executed by</td>
<td>Interpreter program</td>
<td>Computer’s processor</td>
</tr>
<tr>
<td>Examples</td>
<td>TI-BASIC, JavaScript, Java, Python</td>
<td>C, C++, Haskell, Fortran</td>
</tr>
</tbody>
</table>
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
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
:Disp "HELLO, WORLD"
```

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.

**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.

**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 \(^{\wedge}\)[x]\(^2\)[-][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.

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 [\(\uparrow\)] to go to the I/O tab; then press [\(\downarrow\)][\(\downarrow\)][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][\(^{\wedge}\)], [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.
When you’ve finished, press \([2\text{nd}]\)[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 \([2\text{nd}]\)[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 \([2\text{nd}]\)[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 \texttt{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 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
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:

```ti-basic
PROGRAM:QUAD
:Prompt A,B,C
:If 4AC>B²
:Then
:Disp "IMAGINARY ROOTS"
:Else
:Disp (-B+√(B²-4*AC))/(2A)
:Disp (-B-√(B²-4*AC))/(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.
Math programming: a quadratic solver

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"
CHAPTER 1  Diving into calculator programming

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][√]; 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
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
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 non-linear: 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 (→) 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
:0→M
:Repeat G=N
:Prompt G
:If G>N
:Disp "TOO HIGH"
:If G<N
:Disp "TOO LOW"
:N+1→M
:End

Generate the number the player will guess
Repeat the loop from here to the End until G = N
Ask the player to type a guess
Increment the number of guesses used
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.
The first line of the program contains the \texttt{randInt} command, which takes two arguments. In this case, the two arguments are the minimum and maximum possible number that the function should generate.

\begin{verbatim}
:randInt(1,50)→N
\end{verbatim}

As the name suggests, the \texttt{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.

The small arrow indicates that the random number it returns should be stored into variable N, just as \texttt{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 \texttt{randInt} under [MATH]; then use the right- (or left-) arrow key to get to the PRB submenu, and choose \texttt{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 \texttt{(→)}, used to update the contents of variables, is on the lower left of the keypad above the [ON] key, labeled the [STO>] key.

\begin{verbatim}
:0→M
\end{verbatim}

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.

\textbf{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.

\begin{verbatim}
:Repeat G=N
 :Prompt G
 :If G>N
 :Disp "TOO HIGH"
 :If G<N
 :Disp "TOO LOW"
 :M+1→M
 :End
\end{verbatim}

The majority of the program is between the \texttt{Repeat} and \texttt{End} commands, which together enclose what is called a \textit{repeat loop}. In a repeat loop, execution will repeatedly restart at the \texttt{Repeat} command every time the \texttt{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 \texttt{End}. \texttt{End} triggers the interpreter to reexamine the \texttt{Repeat} condition to decide whether to go back and start executing at the \texttt{Repeat} again, prompting for another guess, or continue directly to the code under \texttt{End}, finishing the looping process. If the condition is true, that is, $G = N$, then the code after \texttt{End} will be executed. If the condition is false, or $G$ isn’t equal to $N$ (written as $G \neq N$), then the user has still not guessed the correct number, and execution will return to the \texttt{Repeat}, \texttt{Prompt} section of the code.

\begin{verbatim}
:Disp "CORRECT AFTER:
:Disp M
:Disp "GUESSES"
\end{verbatim}

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 \texttt{Disp} used with strings so far, \texttt{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 \texttt{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.

\textbf{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 \texttt{Disp}, \texttt{End}, \texttt{Prompt}, and \texttt{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 \texttt{prgmGUESS} by quitting to the homescreen, finding GUESS in the Program menu, and pressing \texttt{[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.
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
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!
Programming the TI-83 Plus/TI-84 Plus
Christopher R. Mitchell

The TI-83 Plus and TI-84 Plus are more than just powerful graphing calculators—they are the perfect place to start learning to program. The TI-BASIC language is built in, so you have everything you need to create your own math and science programs, utilities—even games. Programming the TI-83 Plus/TI-84 Plus teaches universal programming concepts and makes it easy for students, teachers, and professionals to write programs for the world’s most popular graphing calculators. This friendly tutorial guides you concept-by-concept, immediately immersing you in your first programs. It introduces TI-BASIC and z80 assembly, teaches you tricks to slim down and speed up your programs, and gives you a solid conceptual base to explore other programming languages.

What’s Inside
• Works with all models of the TI-83, TI-83+, and TI-84+
• Learn to think like a programmer
• Learn concepts you can apply to any language
• Advanced concepts such as hybrid BASIC and ASM

This book is written for beginners—no programming background is assumed.

Christopher Mitchell is a PhD candidate and a recognized leader in the TI-83+/TI-84+ programming community. He hosts discussions and collaboration on calculator programs and projects at his website, Cemetech.

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit manning.com/ProgrammingtheTI-83Plus/TI-84Plus

“All there is to know about TI-BASIC, assembly language, and everything in between.”
—From the Foreword by Brandon Wilson, Advanced Call Center (ACT)

“Makes advanced mathematics and programming techniques accessible to everyone.”
—Ryan Boyd, researcher
North Dakota State University

“Provides a way to look at your calculator that you never thought of before.”
—Jon Walker, 12th-grade student
Steele High School

“Your one-stop guide to TI-BASIC programming.”
—Peter Beck, math teacher
Carmel High School

“A complete TI-BASIC tutorial and a good overture to more languages!”
—Louis Becquey, student
Joseph-Fourier University