SAMPLE CHAPTER

Programming the TI-83 Plus/TI-84 Plus

Christopher R. Mitchell
Foreword by Brandon Wilson

MANNING

Programming the TI-83 Plus/TI-84 Plus

by Christopher R. Mitchell

Chapter 6

brief contents

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.

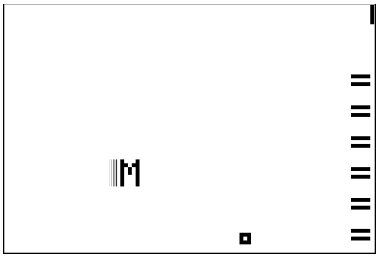

Figure 6.1 You imagine making a Mouse and Cheese game.

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.

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 keypresses, 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,

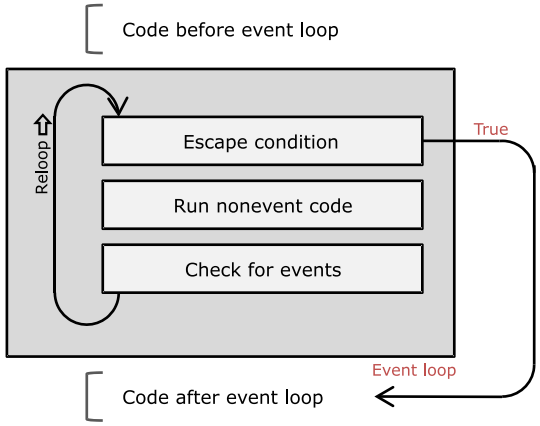

Figure 6.2 The structure of an event loop

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
:ClrHome
:Repeat KEY=[CLEAR]
:X+1→X
:Output(1,1,X
:Check for keypresses
:Store any keys pressed into KEY
:End
```

The variable used for counting

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 homescreen. 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
:ClrHome
:Repeat K=45
: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<16
:A+1→A
:If K=25 and B>1
:B-1→B
:If K=34 and B<8
:B+1→B
:End
```

Initialize X and Y coordinates (A is X, B is Y)

Repeatedly check for keys and move the M until [CLEAR] is pressed

Recall that this is shorthand for If K≠0

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].

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

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.

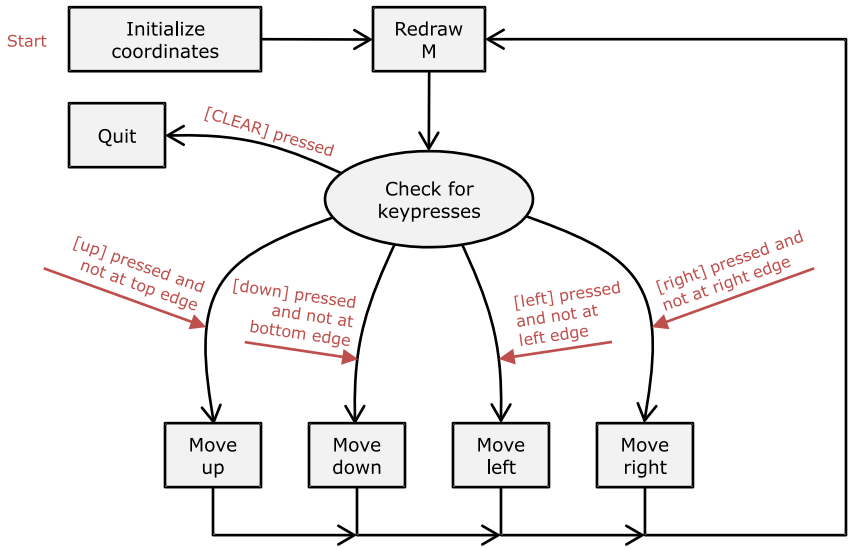

Figure 6.3 Flowchart of functionality for the four-direction M-moving program, MOVECHAR

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.

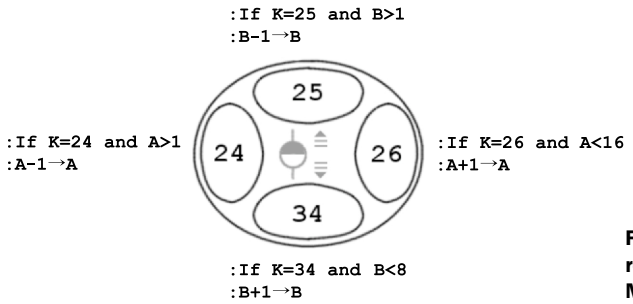

Figure 6.4 Arrow keys and their respective conditional blocks for the MOVECHAR program

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.

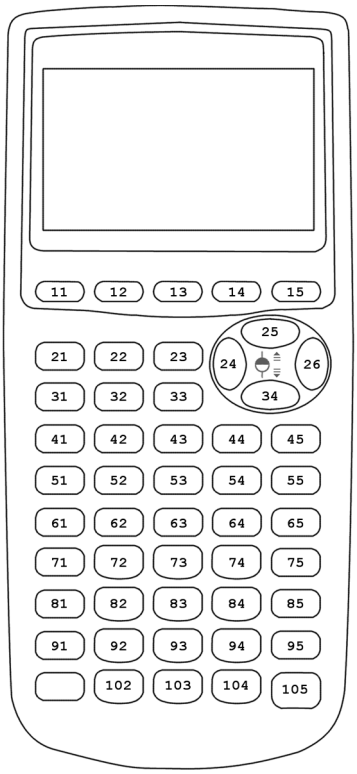

Figure 6.5 `getKey` keycodes for each of the keys on a TI-83+ or TI-84+ keypad. [ON] has no keycode.

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 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

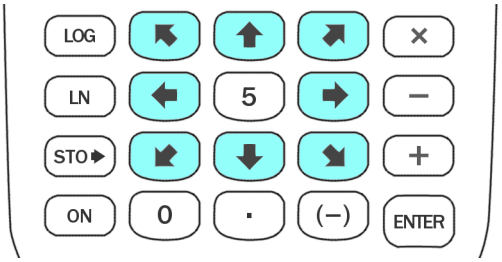

Figure 6.6 Using the number keys for diagonal movement, as shown on the bottom four rows of a TI-83+ calculator's keypad

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
```

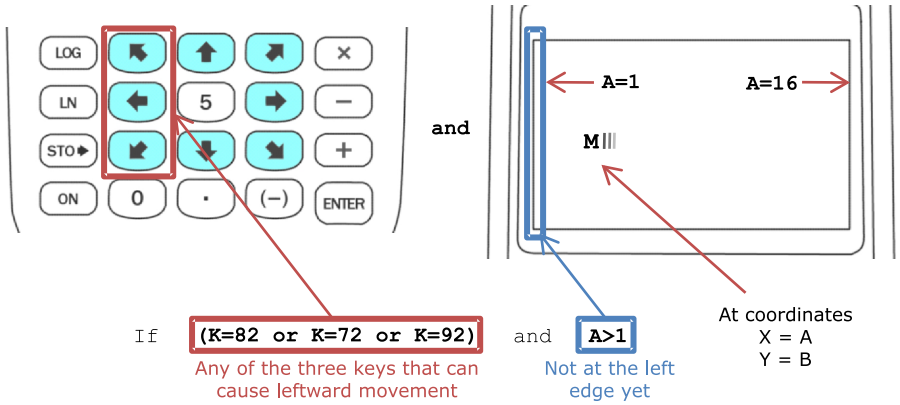

Figure 6.7 Conditional checks for arrow key pressed and M not at the edge of the screen

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
:A-1→A
: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
:B-1→B
:If (K=92 or K=93 or K=94) and B<8
:B+1→B
:End
```

Keys [1], [4], or [7] and not at the left edge

Keys [7], [8], or [9] and not at the top edge

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.

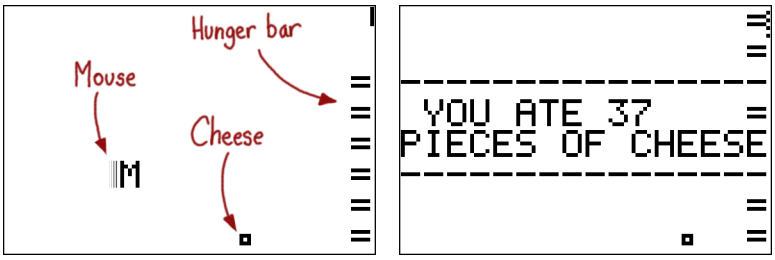

Figure 6.8 Screenshots of the Mouse and Cheese game. The mouse (M) is moved by the player to chase after the pieces of cheese (the square). The player loses if the hunger bar (right edge of the screen) fills up, at which point the game tells the player how many pieces of cheese the mouse ate (right).

The program will be structured with two nested `Repeat` loops; the following pseudocode 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.

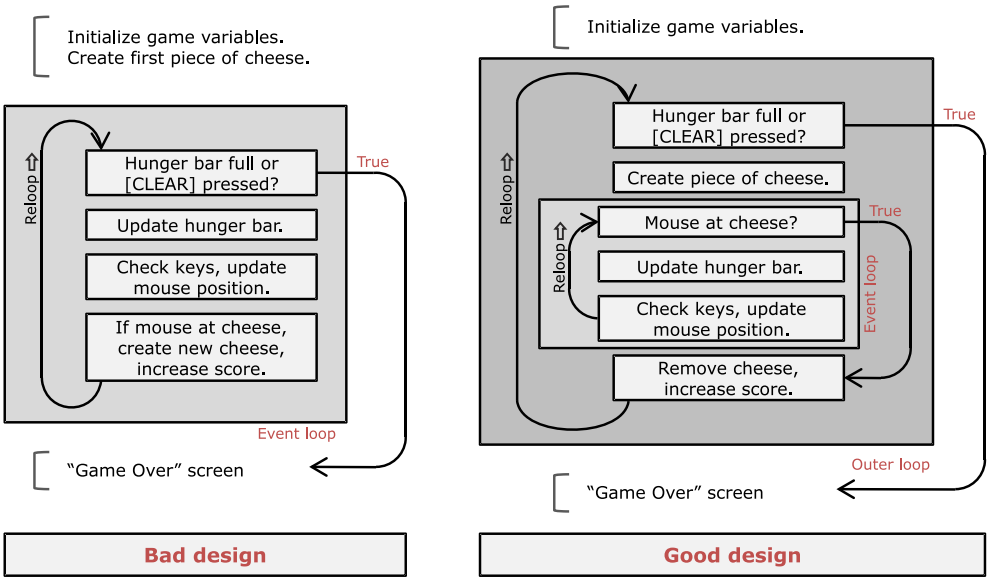

Figure 6.9 Putting the full game loop including cheese management inside the event loop (left) and properly reducing the amount of work in the inner event loop (right)

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
```

Add another notch to hunger bar if hunger has increased enough

Only erase mouse if it might move

Repeat until hunger bar is full, [CLEAR] is pressed, or mouse reaches cheese

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

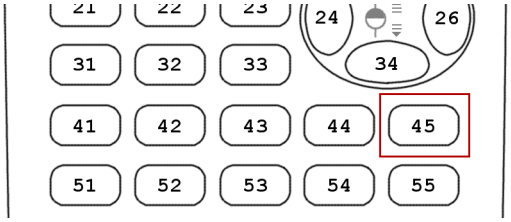

Figure 6.10 An excerpt of the keycode chart showing that the [CLEAR] key is code 45

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 `or`ed 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

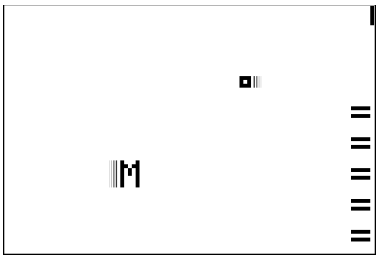

Figure 6.12 By allowing the cheese to move randomly around the screen, the game starts to get more challenging.

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, 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 nonblocking 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 1 if the ALPHA modifier is currently set. S is 1 if the 2nd modifier is set.

Repeatedly look for keys until [3] is pressed

This turns 0 into 1 and 1 into 0 (try doing the math for S = 0 and S = 1!)

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 nonblocking 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.

PROGRAMMING/GRAPHING CALCULATOR

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.

"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

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

ISBN-13: 978-1617290770
ISBN-10: 1617290777
52999
9 781617 290770

MANNING US $29.99 / Can $31.99