CHRISTOPHER LAMPTON

How to Create Computer Games

A COMPUTER AWARENESS FIRST BOOK

HOW TO CREATE COMPUTER GAMES

CHRISTOPHER LAMPTON

HOW TO CREATE COMPUTER GAMES

A Computer-Awareness
First Book

Franklin Watts
New York | London | Toronto
Sydney | 1986

Library of Congress Cataloging in Publication Data

Lampton, Christopher.
How to create computer games.

(A Computer-awareness first book)
Includes index.
Summary: Provides instructions for creating computer games using BASIC.
1. BASIC (Computer program language)—Juvenile literature. 2. Computer games—Juvenile literature. [1. Computer games. 2. BASIC (Computer program language) 3. Programming languages (Computers) 4. Programming (Computers)] I. Title. II. Series.
QA76.73.B3L3535 1986 794.8'2 85-26635
ISBN 0-531-10120-7

Printed in the United States of America
5 4 3 2 1

CONTENTS

HOW TO CREATE COMPUTER GAMES

CHAPTER ONE

THE GHOST IN THE MACHINE

Everybody likes to play with simulations.

What are simulations? Usually, they are scaled-down models of things that are too expensive or too dangerous or too rare to play with for real. A dollhouse is a simulation of a real house, for instance. A model airplane is a simulation of a real airplane. Toy soldiers are simulated soldiers that can fight simulated battles. And so on.

Simulations aren't just for kids. Airplane pilots train on flight simulators before they fly real planes. Investors test their skills with stock market games before risking real cash.

It is even possible to simulate things that have never happened. Toy rocket ships simulate battles in space that may not take place for hundreds of years, if ever; toy animals simulate creatures that exist only in someone's imagination.

A computer is a simulation machine. When we program a computer, we are teaching it to simulate some process that may or may not exist in the real world. (This book, for instance, was typed on a computer running a program called a word processor, which simulates a typewriter but includes some features that a real typewriter could never offer.)

Anyone who has played games on a computer has some idea of the power of computer simulation. Arcade games, for instance, simulate events that have never taken place on the planet Earth—and maybe nowhere else in the universe! A computer can show us what it is like to be a space pilot defending a planet from vicious invaders—or a mythological knight jousting with sword in hand on the back of a giant bird!

This is why a computer is such a marvelous game machine. Any game or sport ever played in the real world (and many games and sports that could *never* be played in the real world) can be played on a computer. Of course, some games lend themselves better to simulation than others. For instance, while it is possible to simulate a swimming meet by depicting the pool and the swimmers on the computer's video display, the computer can never duplicate the tingling thrill of plunging into cool water or the exhilaration of physical exercise. Something is definitely lost in the translation.

On the other hand, games and sports such as auto racing and skydiving are best played on a computer—at least for those of us who would rather not risk our lives to have fun.

A computer is a number-crunching machine, a high-class calculator that adds and subtracts numbers at high speeds. How can it simulate games that have nothing to do with numbers?

The ability of the computer to play games is an illusion, but it is a wonderful illusion. When a computer simulates a game—or anything else, for that matter—it converts the elements of that game into numbers, and performs special operations on those numbers.

When you play games on a computer, though, you don't see these numbers, because they are inside the computer. A clever computer programmer found a way to make those numbers look like something else: a deck of playing cards, or a game board, or a race track, or the surface of an alien planet.

In this book, we will show you how to create simple computerized simulations of familiar games. We will assume that you

already know a few things about computer programming. For instance, you should know how to write computer programs using the programming language called BASIC. If you don't know anything about BASIC, or if you don't even know what a programming language is, then this book will probably be too tough for you. We suggest that you read at least one introductory book on BASIC programming before you continue.

The program examples in this book are intended to run on any computer equipped to understand BASIC—that is, a computer that is running a BASIC interpreter, the program that translates BASIC instructions into an electronic form that the computer understands. Of course, this means that we've left out features such as graphics (pictures) that you might expect to find in a game program, because the way in which graphics are programmed varies widely from one computer to another. However, we'll make suggestions as to how you can add these features, if you wish.

The key to simulating games on a computer is choosing the proper *data structure*. What is a data structure? It is the way in which information, or data, is organized within the internal memory storage of the computer. The BASIC programming language does not offer us as many possible data structures as some more sophisticated languages, like Pascal, do. Nonetheless, with a little creative imagination we can make fine use of the data structures that BASIC does offer. By arranging information properly inside the computer, we can simulate anything from a deck of cards to a checkerboard, to the surface of an entire planet. In this book we'll stick with cards and board games, but by the time you are finished, you will probably have a good notion of how just about anything can be simulated using BASIC data structures.

As you know from your previous programming experience, BASIC stores data in *variables*. A variable is a symbolic label (usually one or two letters of the alphabet or a letter and a number) that identifies a place inside the computer where a piece of

information is stored. There are several types of variables, but we will concentrate on *numeric* variables—variables that store numbers. We will use these numbers to represent all sorts of things—playing cards, positions on a game board, even, well, numbers.

We store a number using a numeric variable like this:

```
A = 178
```

This tells the computer to store the number 178 in an internal storage location called "A." An instruction like this is called an *assignment statement,* because it assigns a value to the variable or, rather, to the place in the computer's internal storage represented by the variable. (A very few versions of BASIC require that we place the word LET in front of an assignment statement, like this:

```
LET A = 178
```

If you are using one of these versions of BASIC, you will need to make this change to all assignment statements in this book.)

Once we've assigned a value to a variable, we can retrieve that value just by using the variable's name. For instance, if we write the instruction:

```
PRINT A
```

it will cause the computer to print the value of variable A on the video display. If variable A is still equal to 178, then the computer will print the number 178.

Because the computer treats the variable almost as though it actually were the number we have assigned to it, we often say that we have "set the variable equal to" the number.

Once again, you should know all of this already, because you've written BASIC programs before. However, you may still be surprised at a few of the things we can do with numeric vari-

ables. For instance, there is a special kind of data structure found in most programming languages called the *array*. An array is a list of variables that can be used to hold a list of data items. In BASIC, an array variable looks like an ordinary variable, except that the name of the variable is followed by one or more numbers in parentheses, like this:

```
A(1)
BG(14)
Y6(10,1)
```

All of these are names that can be used for array variables. Before you can use an array variable in a BASIC program, however, you must *dimension* the array using a DIM statement, like this:

```
DIM A(10)
```

This tells the BASIC interpreter that you are establishing an array called A and that this array will have ten elements.

What is an element? Well, when you create an array like the one above you are not just creating a single variable; you are creating ten different variables. Here are their names:

A(1)	A(6)
A(2)	A(7)
A(3)	A(8)
A(4)	A(9)
A(5)	A(10)

(Actually, there is an eleventh variable in this array, called A(0). However, to avoid confusion, we are not going to get into that.)

Each of the ten variables in this array is called an *element* of the array. Each element is a completely separate variable and can be used to store a different item of data. The number in parentheses after the name is called a *subscript*. It tells us which element

of the array we are dealing with. For instance, the element A(6) has a subscript of 6, so we know it is element 6 of the array.

Since the array above is a numeric array—that is, an array made up of numeric variables—we can use each element to store a number. We can assign a value to an element of an array just like we assign a value to a regular variable, like this:

```
A(2) = 45
```

This assigns a value of 45 to the second element of array A. We can print that value on the video display, like this:

```
PRINT A(2)
```

In fact, we can do exactly the same things with an element of an array as we can do with an ordinary variable, including assigning the value of the array element to another variable:

```
B = A(2)
```

or performing arithmetic on the array element:

```
C = A(2) - 15
```

and so forth.

"Wait a minute!" you exclaim. If we can do exactly the same things with an array element as we can with an ordinary variable, what's the point of using an array?

Good question. The answer is that an array can be used to put information in order. If we have a list of numbers and we need to keep that list organized, we can store that list in an array. Furthermore, it is easier to use an array of ten elements to store data than it is to use ten separate variables, with different names. Why? Because we can use a second variable to represent the subscript of the array, like this:

```
A(B)
```

We said before that the subscript identifies which element of the array we are dealing with. But which element is this? Well, that depends on the current value of B.

If the variable B is equal to 2, then we are dealing with element 2 of the array, just as though we had written:

```
A(2)
```

On the other hand, if the variable B is equal to 7, we are dealing with element 7 of the array, as though we had written:

```
A(7)
```

And why is this an advantage? Suppose we wanted to write a program that would allow the user—that is, the person who runs the program—to assign numeric values (numbers) to ten variables. In BASIC, we let the user assign numbers to variables with the INPUT statement, like this:

```
INPUT A
```

As you know, this instruction prints a question mark (?) on the video display and stops the program until the user types a number and presses the key marked RETURN (or ENTER, depending on the computer you are using). It then assigns that number to variable A. We can also put array variables in INPUT statements, like this:

```
INPUT A(7)
```

This lets the user assign a numeric value to element 7 of array A.

If we were writing a program that would let the user assign

values to ten non-array variables, we would need to write ten separate input statements, like this:

```
10 INPUT A
20 INPUT B
30 INPUT C
40 INPUT D
50 INPUT E
60 INPUT F
70 INPUT G
80 INPUT H
90 INPUT I
100 INPUT J
```

You don't need to type and RUN this program, but if you did, it would prompt you to type ten numbers, pressing RETURN (or ENTER) after each. Each number would be assigned to a separate variable, from A through J—though you could change these to any variable names that you might prefer.

This program would be much simpler if we used array variables, rather than ordinary variables. We would dimension the array in the first line of the program, like this:

```
10 DIM L(10)
```

We would then set up a FOR-NEXT loop that would repeat ten times:

```
20 FOR X=1 TO 10
```

In the middle of the loop, we would place an INPUT statement:

```
30 INPUT L(X)
```

Then we would end the loop:

```
40 NEXT X
```

The entire program would look like this:

```
10 DIM L(10)
20 FOR X=1 TO 10
30 INPUT L(X)
40 NEXT X
```

Do you see how this program works? The FOR-NEXT loop will be executed ten times. Each time, the variable X will be equal to a different number, from 1 to 10. Therefore, the variable L(X) in line 30 will be equal to a different element of array L each time the loop executes. The first time, it will be L(1), because X will be equal to 1. The second time, it will be L(2), and so on. Each time, the user will be prompted, with a question mark, to type a number, and that number will be assigned to an element of the array. (If you don't already know how a FOR-NEXT loop works, you might find this example confusing. Once again, you should read an introductory book on BASIC if you find these program examples too difficult.)

In short, this program does exactly what the earlier program did, but it does it with four instructions instead of ten. And if we suddenly decided that we wanted to input 100 numbers instead of 10, we would need to make only one small change to the program, like this:

```
10 DIM L(100)
20 FOR X=1 TO 100
30 INPUT L(X)
40 NEXT X
```

Now the program will input 100 numbers. To change the earlier version so that it would input 100 numbers, we would need to add

ninety new instructions and ninety new variable names. See how much easier the job becomes when we use an array?

Why would you want to put a list of numbers in an array? There are almost as many reasons as there are computer programs. Maybe you are writing a program to keep track of the batting averages of your favorite players. Or maybe your program contains the grade averages of all the students in a class, in order of their names in the class roll book.

The arrays that we've been showing you are called *one-dimensional arrays.* This is a fancy way of saying that each element of the array has one subscript after it. Most versions of BASIC, however, allow us to create arrays that have more than one subscript after each element, like this:

```
DK(2,9)
BG(1,9,13)
Z(89,3,1,2)
```

An array that has two subscripts after each element is called a *two-dimensional array.* An array that has three subscripts after each element is called a *three-dimensional array,* and so forth. Some versions of BASIC have a limit on the number of dimensions you can have in an array, but most allow at least three-dimensional arrays. A more important limit is the amount of internal memory storage your computer has. This is usually measured in kilobytes (Ks), as in 64K or 256K memory. When an array has a lot of subscripts, it eats up a lot of memory. Your computer will tell you if you try to use more memory than it has. (Don't worry: you won't lose this memory permanently. It will return as soon as the program and its arrays are removed from the computer's memory.)

We create multidimensional arrays just as we create one-dimensional arrays, with a DIM statement. The difference is that we have to specify the extra subscripts, like this:

```
DIM AB(14,7)
```

This DIM statement creates an array called AB, with two dimensions. There are fourteen possible subscripts in the first dimension and seven possible subscripts in the second. (Actually, there are fifteen and eight possible subscripts in each dimension, but we are still ignoring the 0 subscript.)

Here are a few possible elements in this array:

```
AB(1,5)
AB(14,1)
AB(9,2)
```

How many elements are there in this array? Well, at a guess, you might think that there would be twenty-one elements, because 21 is the sum of 14 and 7. But if you guessed this, you would be wrong. Actually, there are ninety-eight elements in this array, because 98 is 14 *times* 7! And that is why multidimensional arrays eat computer memory—they have a lot of elements in them!

We use one-dimensional arrays to hold lists of data items, such as numbers. Why would we use multidimensional arrays? Once again, there are lots of reasons. In this book, we will show you several ways in which both one- and two-dimensional arrays can be used in game programs. Once you have finished reading this book, you will probably have a better understanding of how these data structures can be used in other kinds of programs.

CHAPTER TWO

A FISHING EXPEDITION

Card games lend themselves especially well to computer simulation.

Why? Because card games have very precise, fixed rules, and because cards are usually numbered. With a little imagination, we can come up with simple data structures to describe all of the parts of a card game. For instance, a deck of cards can become a numeric array, with each suit . . .

But that's getting ahead of the game, so to speak. The best way to learn to write a computer simulation of a card game is to write one. This chapter will show you how to set up the data structure for the game. The next chapter will show you how to play the game with (and possibly beat) the computer.

What game should we choose? Well, it should be a game with simple rules, and one that can be played by two players—so that the user of the program can pit himself or herself directly against the computer, with the computer playing one hand of cards and the user playing the other. It should also be a game that is familiar to most people, or that is so simple it can be explained easily.

There are several card games that match this description. One of these is Go Fish.

You've probably played this game before, though perhaps not recently. In case you haven't played it, or have forgotten the rules, we'll describe it here briefly.

Go Fish, like most card games, is played with a deck of fifty-two cards. We'll assume you know what a deck of cards looks like. For our purposes, the most important thing about a deck is that each card has a suit and a denomination. The suit for each card is either hearts, diamonds, clubs, or spades. The denomination is either ace, two through ten, jack, queen, or king.

The game of Go Fish is played by two to six players. In this book, we will restrict ourselves to two players.

When the game begins, one of the two players is chosen to be the dealer. (In our computerized version, the computer will always deal.) The dealer shuffles the deck and hands seven cards to each player. These cards become the players' *hands*.

The players then alternate turns, usually beginning with the player who did not deal the cards. The player whose turn it is asks the other player for all of the cards of a certain denomination that he or she holds. For instance, one player may say to the other, "Give me all of your nines." If the other has cards of this denomination, he or she must give all of these cards to the player who requests them. Before a player can ask for cards of a certain denomination, however, that player must have at least one card of that denomination in hand. For instance, the player who asks for "nines" must have at least one nine in hand already.

If the other player does *not* have any cards of that denomination in hand, he or she says "Go fish!" The player asking for the cards must then draw a card from the top of the deck and add it to the cards in his or her hand.

If the player gets the denomination of card asked for—either from the other player or from the deck—that player gets an extra turn and can continue asking for cards until he or she fails to receive the requested denomination.

When a player has four cards of the same denomination in hand, that player is said to have a *book* and must discard those cards—that is, put them aside. The game continues until one player runs out of cards. The winner is the player who has discarded the most books. If both players create the same number of books, the game is a tie.

How do we turn a game like Go Fish into a computer program? We start by looking at the essential parts of the game—that is, the actual physical structures with which the game is played—and deciding what sort of data structures can be used to represent these physical structures.

What are the essential parts of Go Fish?

The most obvious is the deck of fifty-two cards. In fact, this is an essential part of almost every card game. We will need a data structure to represent a deck of cards.

The second element is the players' hands—that is, the cards that each player holds. We can use a second data structure to represent the cards or we can simply make the hands part of the same data structure that represents the deck. More about this in a minute.

Finally, there are books—the sets of four cards that the players set aside periodically throughout the game. Once again, we can use a separate data structure for the books, or we can use the same data structure that represents the deck.

Next, we must look at the actions that take place during the game—shuffling and dealing cards, finding and discarding books, moving cards back and forth between hands, drawing cards off the top of the deck, etc.—and invent program instructions that will be the equivalent to these actions.

Remember, a computer works with numbers. We must find a way to represent all the parts of this game as numbers. And when we perform actions like moving cards from one hand to another or rearranging the order of cards, we must perform these actions with numbers.

This isn't quite as hard as it sounds. Honest!

In our game of Go Fish, we will represent the deck of fifty-two cards as a two-dimensional array called DK (deck). This array will be created at the beginning of the game, like this:

```
20 DIM DK(4,13)
```

(Don't type this line or any of the rest of the program yet. At the end of Chapter Three, we'll give you the complete program. You can type it, and play with it, then.)

This line tells the computer that the first subscript can be any of four possible numbers and that the second subscript can be any of thirteen possible numbers. Do these numbers sound familiar? We are going to use the first subscript to represent the four possible suits:

1 = hearts
2 = clubs
3 = diamonds
4 = spades

And we will use the second subscript to represent the thirteen possible denominations of each suit:

1 = ace
2–10 = two through ten
11 = jack
12 = queen
13 = king

In this way, we can represent any card in the deck as an element of array DK. For instance, if we wrote:

```
DK(2,13)
```

this would represent the king of clubs, because the 2 in the first position represents clubs and the 13 in the second position represents the king.

We could also write:

```
DK(SU,DE)
```

with SU representing any possible suit from 1 to 4 and DE representing any possible denomination from 1 to 13. By changing the values of these two variables, DK(SU,DE) can represent any possible card in the deck.

Why do we want to represent the deck as a two-dimensional array? Because at any time in the game, any of the fifty-two cards will be in one of four places: in the deck, in the human player's hand, in the computer's hand, or in a discarded book. We can represent each of these as a number, like this:

0 = in the deck
1 = in the player's hand
2 = in the computer's hand
3 = in a discarded book

In this way, we can keep track of each card, by setting the appropriate element of array DK equal to one of these numbers. For instance, if we want to put the 8 of diamonds in the player's hand, we would write:

```
DK(3,8) = 1
```

Remember, the 3 represents the suit of diamonds and the 8 represents the denomination 8. By setting this element equal to 1, we are making note of the fact that this card is in the player's hand (a position that is represented by the number 1). We don't actually

have to "move" the card anywhere; rather, we use array DK to keep track of it, in a way that the computer can understand.

This data structure, the array DK, is the key to our entire game. Everything we do in the actual program will be based on the fact that DK represents a deck of cards, that the first subscript of DK is the suit, that the second subscript of DK is the denomination, and that the value of an element represents the whereabouts of that card.

Now let's begin writing the program. Once again, you don't need to type the program until we show you the complete listing at the end of this chapter, but you should follow along as we build the program line by line and routine by routine. (A *routine* is a set of program instructions that accomplish a single task or related group of tasks. A *subroutine* is a special kind of routine that is called, or activated, by the GOSUB instruction.)

We've already written line 20, which DIMs the array DK. Line 10 should clear the video display of your computer, but the instruction for this will vary from one version of BASIC to another. For instance, if you are using a Radio Shack, IBM, or Timex-Sinclair computer, you should use the instruction:

```
10 CLS
```

If you are using an Apple computer, you should use the instruction:

```
10 HOME
```

If you are using a Commodore computer, use the instruction:

```
10 PRINT CHR$(147)
```

and if you are using an Atari computer, use the instruction:

```
10 PRINT CHR$(125)
```

If you are not using one of the above computers and don't know what instruction will clear the video display, just leave this line out. It isn't really necessary; it just makes things look neater. The rest of the program will be pretty much the same no matter which computer you are using.

Next, we need to create a variable that will tell us how many cards are currently in the deck, so that we will know when we run out of cards. In Go Fish, we should *never* run out of cards in the deck; by the time we reach the bottom of the deck one of the players has to run out of cards, ending the game. Nonetheless, we will check for this event in case something goes wrong with our program. We create the variable like this:

```
30 ND=52
```

The variable name ND stands for "number in deck." We set it equal to 52 because the game begins with fifty-two cards in the deck.

Next, we must shuffle the deck and deal the cards. Shuffling the deck is easy. Since we will be dealing out cards from the deck at random (as you will see), it is not necessary to shuffle the deck into random order. It is only necessary to set all elements of array DK to 0, indicating that all cards are in the deck, as explained above. Although BASIC will automatically set the elements of a newly created array equal to 0, we should nonetheless *deliberately* set every element equal to 0, especially if we plan to play more than one round of the game, in which case this routine will need to be repeated at the beginning of each round. The easiest way to set the entire array equal to 0 is with a pair of FOR-NEXT loops, each inside the other, like this:

```
50 FOR SU=1 TO 4 : REM STEP THROUGH EVERY SUIT . . .
60 FOR DE=1 TO 13 : REM  . . . AND EVERY DENOMINATION
70 DK(SU,DE) = 0 : REM SET IT EQUAL TO 0
80 NEXT DE
90 NEXT SU
```

The first loop ("FOR SU=1 TO 4") will execute four times. The second loop ("FOR DE=1 TO 13") is *inside* the first loop, so it will execute thirteen times every time the first loop executes, for a grand total of fifty-two executions, once for every card in the deck. The first loop steps the variable SU through all four values that a suit can take—that is, 1 through 4. The second loop, inside the first, steps the variable DE through all 13 values that a denomination can take—that is, 1 through 13. Between them, they step DK(SU,DE) through every possible element in the array. Line 70 sets each of these elements to 0.

Now the action starts. We must deal seven cards to each player from this deck. Dealing cards is such an important part of the program that we will put part of this operation in a subroutine, because it will be performed several times in the game. A subroutine, as you should already know, is a routine that is called with the GOSUB statement and ends with a RETURN statement. We will write a subroutine that will deal one random card from the deck of 52. That subroutine will look like this:

```
2000 IF ND>0 THEN GOTO 2020 : REM IF CARDS IN DECK, DEAL'EM!
2010 PRINT "THE DECK IS EMPTY!" : END : REM IF NOT, END GAME
2020 SU=INT(RND(0)*4+1) : DE=INT(RND(0)*13+1) : REM RANDOM CARD
2030 IF DK(SU,DE)<>0 THEN 2020 : REM ALREADY DEALT?
2040 DK(SU,DE)=CP : ND=ND-1 : REM IF NOT, CHECK IT OFF
2050 RETURN
```

This is a very important subroutine, so we will describe in some detail what it does.

Line 2000 checks to make sure that there are still cards in the deck. In Go Fish, the deck shouldn't run out of cards, so the instruction GOTO 2020 should always be executed. If not, something is very wrong and line 2010 will stop the program with a

message reading "THE DECK IS EMPTY!" If this happens, check your program to make sure it is correctly typed.

Line 2020 uses the RND function to choose a random suit, which it stores in variable SU, and a random denomination, which it stores in variable DE. How does the RND (random) function work? Unfortunately, it works differently in the various versions of BASIC, so you may have to make some changes here.

In most versions of BASIC, RND(0) will be equal to a random fraction that is larger than 0 and smaller than 1. This may be difficult to understand, but you can simply think of the RND function as a way of coming up with an unpredictable number between 0 and 1, but not including 1. The first time we use RND(0) in the above subroutine, we write:

```
SU=INT(RND(0)*4+1)
```

The RND(0) function produces a random fraction between 0 and 1. Multiplying it by 4 changes this fraction into a random number between 0 and 4, but not including 4. Adding 1 to this changes it into a random number between 1 and 5, but not including 5. The INT (integer) function, with which we have surrounded all of these other operations, chops off any fractional value of the result, so that what we end up with will be either the number 1, the number 2, the number 3, or the number 4, chosen at random.

You'll remember that the four suits in a deck of cards are represented in this program by the numbers 1 through 4. This instruction sets the variable SU equal to one of the four suits. Since the number is chosen at random, there is an equal chance of SU being equal to *any* of the four suits—just as we have an equal chance of drawing any of the four suits when we deal a card from a shuffled deck.

This instruction will work in most versions of BASIC, but the number in parentheses following "RND" may have to be changed. If you are using Applesoft BASIC, for instance, you must change "RND(0)" to "RND(1)." The instruction will work as writ-

ten in Radio Shack BASIC, Atari BASIC, and Commodore BASIC. Consult your computer's manual for details, if necessary.

The next instruction in line 2020 is:

```
DE=INT(RND(0)*13+1)
```

This works pretty much like the last instruction, setting DE equal to a random number in the range 1 to 13. Since the denomination of a card can be any number from 1 (ace) to 13 (king), this sets DE equal to a random denomination of card. We have now produced both a random suit and a random denomination, as though we had drawn a random card from a shuffled deck.

However, we must make sure that this card has not already been drawn from the deck. Line 2030 reads:

```
IF DK(SU,DE)<>0 THEN 2020
```

DK(SU,DE) is the element of array DK that represents the card we have just "drawn from the deck"—that is, the card with suit SU and denomination DE. This line checks to see if this element is not equal to 0, which would mean that the card is no longer in the deck. If so, this line sends the computer back to the previous line, to choose another random card. The computer will continue to choose random cards until it finds one that has not already been dealt.

The first instruction in line 2040 reads:

```
DK(SU,DE)=CP
```

The variable CP stands for "current player." Before calling this subroutine with a GOSUB instruction, we must set CP equal to either 1 or 2, depending on whether we are dealing a card to the human player or to the computer. If we are dealing a card to the player, we will set CP equal to 1. If we are dealing a card to the computer, we will set CP equal to 2. This instruction simply

records the whereabouts of card DK(SU,DE), so that we will know whether it is in the player's hand or the computer's.

The next instruction reads:

```
ND=ND-1
```

The variable ND, you will recall, is equal to the number of cards in the deck. Since we have just removed a card from the deck, we now subtract 1 from variable ND.

Finally, line 2050 ends the subroutine. At this point, remember, SU will be equal to the suit of our random card, DE will be equal to the denomination, and DK(SU,DE) will be equal to 1 or 2, depending on who got the card.

Now that we have a subroutine for dealing a random card, we must call this subroutine with a GOSUB. As we said above, the first use for this subroutine will be to deal seven cards each to the human player and the computer. Therefore, we will need to call the card-dealing subroutine fourteen (7 cards × 2 players) times. The best way to do that is with a FOR-NEXT loop, like this:

```
100 FOR I=1 TO 7 : REM REPEAT 7 TIMES
110 CP=1 : GOSUB 2000 : DK(SU,DE)=1 : REM DEAL A RANDOM
CARD TO PLAYER 1
120 CP=2 : GOSUB 2000 : DK(SU,DE)=2 : REM DEAL A RANDOM
CARD TO PLAYER 2
130 NEXT I
```

The loop that starts on line 100 repeats seven times. Each time it calls the card-dealing subroutine twice, once with CP equal to 1 and once with CP equal to 2, dealing one card to the human player and one card to the computer, for a total of seven cards in each hand.

All of this dealing and shuffling takes time, and the player may wonder what the computer is up to. To calm the player's

fears that the computer might have abandoned him or her, we'll print a message on the video display, like this:

```
40 PRINT "DEALING"
```

Notice that this message goes way back in line 40, so it will be printed *before* the dealing starts.

After the dealing, we must take care of some important business. We must establish the number of cards in the player's hand and the number of cards in the computer's hand, because these numbers will keep changing throughout the game and we must keep track of them. We will use the variable PN ("player's number") to store the number of cards in the player's hand and the variable CN ("computer's number") to store the number of cards in the computer's hand. At the beginning of the game, both hands have seven cards in them, so we will set both variables equal to 7, like this:

```
150 PN=7 : CN=7
```

We also need to keep track of the number of books that each player has discarded, since this number will determine who wins the game. We will use the variable PB ("player's books") to store the number of books the player has discarded and the variable CB ("computer's books") to store the number of books the computer has discarded. At the beginning of the game, neither will have discarded any books, so we will set both variables equal to 0, like this:

```
160 PB=0 : CB=0
```

All of the basic elements of the game are now in place. Sharpen your Go Fish skills—it's time to play!

CHAPTER THREE

PLAYING THE GAME

We have now completed *all* of the preparation required for a game. The cards have been dealt and the variables have been initialized. It is time for the main loop to begin.

Since the computer dealt the cards, we will let the human player make the first move. In a fancy computerized card game simulation, we would draw pictures of the cards on the video display, to show the player what cards he or she is holding. However, as we said earlier, we are going to write this program without any pictures, since different computers draw pictures using different BASIC instructions. If you want to add some instructions to the program that will draw pictures of the seven cards we have just dealt, go right ahead. In fact, this would be a terrific way of exploring the graphics capabilities of your computer. Of course, you will have to consult the user's manual for your computer to find out what sort of graphics instructions are available, if you don't know already.

Instead of drawing a picture here, we will print out a list of the cards that the human player is holding. We will also print out other important information: the number of cards that the player

is holding and the number of books that the player has discarded.

Printing out the number of books that have been discarded is easy. This number, you will remember, is contained in variable PB ("player's books"), so we can print it out like this:

```
170 PRINT "YOU HAVE DISCARDED";PB;"BOOKS"
```

Notice that we don't leave any blank spaces around the number of books. This is because most versions of BASIC will automatically place blank spaces before and after a number. If you are using a version of BASIC that does *not* do this, such as Applesoft or Atari BASIC, you will need to add some spaces inside the quotes, until this sentence looks right on your video display.

Printing out the number of cards that the player is holding is also easy. This number is contained in the variable PN ("player's number"), and can be printed out like this:

```
180 PRINT "YOU HAVE";PN;"CARDS"
```

Finally, we must list the cards themselves:

```
190 PRINT "THEY ARE:    ";
```

But wait! Naming the cards is trickier than numbering them. First, we will need a special notation for the cards—that is, a way of naming them on the video display. Because there is a limited amount of space on the video display for printing the card names, this notation should be short. We will use only one or two characters to represent the denomination:

A = ace	8 = eight
2 = two	9 = nine
3 = three	10 = ten
4 = four	J = jack

5 = five
6 = six
7 = seven
Q= queen
K= king

We will use two letters to represent the suit:

Ht = hearts
Cl = clubs
Di = diamonds
Sp = spades

We will put a hyphen ("-") between the denomination and the suit. The jack of clubs, to give one example, would be represented like this:

J-Cl

The five of spades would be represented like this:

5-Sp

The ace of hearts would be represented like this:

A-Ht

and so forth.

Now we need a subroutine that will take a suit number from 1 to 4 and a denomination number from 1 to 13 and print the card name on the video display using this notation. Here is such a subroutine:

```
3000 IF DE>1 AND DE<10 THEN PRINT CHR$(DE+ASC("0")); :
GOTO 3060: REM NUMBER CARDS
3010 IF DE=10 THEN PRINT "10"; : GOTO 3060 : REM TEN
3020 IF DE=1 THEN PRINT "A"; : GOTO 3060 : REM ACE
```

```
3030 IF DE=11 THEN PRINT "J"; : GOTO 3060 : REM JACK
3040 IF DE=12 THEN PRINT "Q"; : GOTO 3060 : REM QUEEN
3050 IF DE=13 THEN PRINT "K"; : GOTO 3060 : REM KING
3060 PRINT "-"; : REM PRINT A HYPHEN IN THE MIDDLE
3070 IF SU=1 THEN PRINT "Ht"; : RETURN : REM HEARTS
3080 IF SU=2 THEN PRINT "Cl"; : RETURN : REM CLUBS
3090 IF SU=3 THEN PRINT "Di"; : RETURN : REM DIAMONDS
3100 IF SU=4 THEN PRINT "Sp"; : RETURN : REM SPADES
```

This is a pretty simple subroutine and doesn't need as detailed an explanation as the last one did. Before you call the subroutine, you must set DE equal to the denomination of the card you wish to name and SU equal to the suit of the card you wish to name. The first six lines (3000–3050) check the value of DE and print out the first part of the name. If the value is between 2 and 9 just the number is printed. Otherwise, it prints "A," "10," "J," "Q," or "K," whichever is appropriate. (You might wonder why "10" is singled out here, since it could be printed as any of the other numbers are. We'll explain why in a moment.)

Line 3060 prints the hyphen. Lines 3070 through 3100 check the value of SU and print out the correct suit name, then RETURN to the main program.

There is one problem with line 3000, as it is now written. We mentioned earlier that most versions of BASIC print extra blank spaces before and after a number. If this happens in line 3000, it will mess up the card name, by adding spaces to it. If you are using one of the versions of BASIC that does not add these extra spaces, such as Applesoft or Atari BASIC, you can use this subroutine as is. However, in the final version of this program we will make a change to this line to remove the extra spaces, like this:

```
3000 IF DE>1 AND DE<10 THEN PRINT CHR$(DE+ASC("0")); :
GOTO 3060
```

(If you are using Timex-Sinclair BASIC, you will need to change the letters "ASC" to "CODE.") You don't need to understand

how this line works. Just take our word for it that it does. Unfortunately, it won't work properly if the card is "10," which is why we have deliberately singled out "10" for special treatment on line 3010.

Now that we have a subroutine that prints out the name of the card, we can call this subroutine to print out the cards in the human player's hand. We determine which cards are in the player's hand by looking through the entire deck, and checking to see which elements of array DK are equal to 0, like this:

```
200 FOR DE=1 TO 13 : REM CHECK ALL 13 DENOMINATIONS
210 FOR SU=1 TO 4 : REM CHECK ALL 4 SUITS
220 IF DK(SU,DE)=1 THEN GOSUB 3000 : PRINT " "; : REM IF THE
PLAYER IS HOLDING THE CARD, PRINT NAME AND 2 BLANK
SPACES
230 NEXT SU
240 NEXT DE
250 PRINT
```

The two FOR-NEXT loops run through all the denominations and suits. Every time it finds one that is in the player's hand—that is, if DK(SU,DE) is equal to 1—it calls the subroutine at line 3000 to print the name of that card on the video display. It then prints two blank spaces, to separate the names. The PRINT statement on line 250 simply prints a carriage return at the end of the list of cards.

Before we proceed with the game, we should check to be sure that there are no books of cards in the player's hand. Although the game has not really begun, it is possible that the player has been dealt four cards of the same denomination. Not very likely, but possible. Of course, we will also need to check again for a book later, after the player has drawn cards from the computer's hand or the deck. And we will need to check the computer's hand for books too, when it is the computer's turn to play. Therefore, we should create a subroutine that will check for books, since we will be doing it often. Here is such a subroutine:

```
4000 FOR DE=1 TO 13 : REM CHECK ALL 13 DENOMINATIONS
4010 NC=0 : REM SET NUMBER OF CARDS IN DENOMINATION TO
0
4020 FOR SU=1 TO 4 : REM CHECK ALL 4 SUITS
4030 IF DK(SU,DE)=CP THEN NC=NC+1 : REM COUNT NUMBER
OF CARDS OF THIS DENOMINATION IN CURRENT PLAYER'S
HAND
4040 IF NC=4 THEN 4090 : REM IF IT'S A BOOK OF FOUR, SKIP
AHEAD
4050 NEXT SU
4060 NEXT DE
4070 DE=0 : REM NO BOOK WAS FOUND
4080 RETURN
4090 FOR SU=1 TO 4 : REM ELSE, IF FOUND, THEN DISCARD IT
4100 DK(SU,DE)=3 : REM 3 = DISCARDED BOOK
4110 NEXT SU
4120 RETURN
```

Before calling this routine, variable CP must be set equal to 1 if you wish to search the player's hand and 2 if you wish to search the computer's hand.

Once again, we use a pair of nested FOR-NEXT loops to proceed through the entire deck—that is, one FOR-NEXT loop inside a second FOR-NEXT loop, as in our shuffling routine. The FOR-NEXT loop that begins in line 4000 steps variable DE through all thirteen denominations.

The second FOR-NEXT loop, which begins in line 4020, steps variable SU through all four suits. This loop executes four times as the first loop executes once. Variable NC counts the "number of cards" of each denomination that the current player (CP) is holding. This variable is set to 0 each time the first loop executes, right before the second loop executes. Line 4030 counts the number of cards of the current denomination in CP's hand. Line 4040 checks to see if the count reaches 4, meaning that there is a full book of that denomination in CP's hand. If such a book is found, the

search is terminated and the computer jumps ahead to line 4090. (There can never be more than one book in hand at a time, so there is no need to search further.)

If no book is found, DE is set equal to 0 and the subroutine RETURNs to the main program. If a book is found, the loop beginning on line 4090 sets all cards of that denomination—DK(SU,DE)—equal to 3, indicating that they have been discarded from the hand. The variable DE now contains the number of the denomination for which we found a book.

We first call this subroutine immediately after we print out the contents of the human player's hand, like this:

```
260 CP=1 : GOSUB 4000 : REM LOOK FOR BOOKS IN PLAYER'S HAND
```

We then check to see if a book was found:

```
270 IF DE=0 THEN 310 : REM IF NO BOOK, SKIP AHEAD
```

If the book was found—that is, if DE is not equal to 0—we announce the fact:

```
280 PRINT "YOU HAVE A BOOK OF"; : GOSUB 5000
290 PRINT "YOU DISCARD THIS BOOK"
```

What is this mysterious subroutine on line 5000? It will print out the name of the denomination that we have discarded, based on the value of variable DE. Here is that subroutine:

```
5000 IF DE=1 THEN PRINT "ACES" : RETURN
5010 IF DE=2 THEN PRINT "TWOS" : RETURN
5020 IF DE=3 THEN PRINT "THREES" : RETURN
5030 IF DE=4 THEN PRINT "FOURS" : RETURN
5040 IF DE=5 THEN PRINT "FIVES" : RETURN
5050 IF DE=6 THEN PRINT "SIXES" : RETURN
```

```
5060 IF DE=7 THEN PRINT "SEVENS" : RETURN
5070 IF DE=8 THEN PRINT "EIGHTS" : RETURN
5080 IF DE=9 THEN PRINT "NINES" : RETURN
5090 IF DE=10 THEN PRINT "TENS" : RETURN
5100 IF DE=11 THEN PRINT "JACKS" : RETURN
5110 IF DE=12 THEN PRINT "QUEENS" : RETURN
5120 PRINT "KINGS" : RETURN
```

This is a long but simple subroutine. It does nothing but check the value of DE and print an appropriate name for the denomination. Line 5120 assumes that the denomination must be KINGS, if the computer has gotten that far.

The result of calling this subroutine is that line 280, above, will print out something like this:

```
YOU HAVE A BOOK OF JACKS
```

if the value of DE is 11, or

```
YOU HAVE A BOOK OF FOURS
```

if the value of DE is 4, and so forth. Line 290 prints:

```
YOU DISCARD THIS BOOK
```

so that you will know that it is no longer in your hand.

We must also increase the value of variable PB, which counts the number of books that the player has discarded, and decrease the value of PN, which counts the number of cards in the player's hand. If PN equals 0, we must end the game, because the player is out of cards. All of this is done here:

```
300 PB=1 : PN=PN-4 : IF PN=0 THEN PRINT "YOU ARE OUT OF CARDS!"; : GOTO 950
```

This tells us that the player has one more book and four less cards. It is impossible for the game to be over after a single book has been discarded, but this routine will be executed many more times in the course of the game, and so we must check to see if the player has run out of cards, thus ending the game. If so, the computer prints "YOU ARE OUT OF CARDS!," then jumps to line 950. The routine beginning at line 950 will announce that the game is over, then decide who has won and congratulate the winner. We will show you this routine shortly.

Now the player must ask the computer for all of its cards of a certain denomination. Of course, the player can't speak to the computer directly—unless the computer is equipped with voice-recognition equipment. Instead, the program asks the player which denomination he or she desires, like this:

```
310 PRINT "WHICH DENOMINATION WOULD YOU LIKE (1-13)?"
320 INPUT DE
```

The player must type a number from 1 to 13, indicating the desired denomination, with 1 representing the ace, 2 representing the two, and so forth. Line 320 uses an INPUT statement to obtain this number, and stores it in variable DE.

The player requesting a denomination must already have at least one card of that denomination in hand. Therefore, we must scan through the player's hand to see if it contains cards of this denomination.

Here is a subroutine that will look through the hand of player CP for a card of denomination DE. It reports its findings in variable FL (for "flag," since it flags the presence of the cards). If one or more cards of denomination DE are found, the subroutine will set FL to 1. If no cards of that denomination are found, FL will be set to 0. Here is the subroutine:

```
6000 FL=0 : REM CARD NOT FOUND YET
6010 FOR SU=1 TO 4 : REM CHECK ALL FOUR SUITS
```

```
6020 IF DK(SU,DE)=CP THEN FL=1 : REM IF CARD FOUND, FLAG
IT
6030 NEXT SU
6040 RETURN
```

Before calling this subroutine, CP must be set to 1 if the human player's hand is to be searched and 2 if the computer's hand is to be searched, and DE must equal the denomination to be searched for. The subroutine loops through all cards of that denomination in array DK, to see if any are equal to CP, which indicates that the card is in that player's hand.

Since the denomination requested by the human player is already in DE, we can now call this subroutine, like this:

```
330 CP=1 : GOSUB 6000 : REM SCAN PLAYER'S HAND
340 IF FL=0 THEN PRINT "YOU DON'T HAVE ANY IN YOUR HAND!" :
GOTO 310
```

If no cards of that denomination are found in the player's hand, then we have caught the player cheating! (Of course, it was probably an accident. . . .) We then loop back and ask the player again which denomination he or she would like.

On the other hand, if the player is holding at least one card of that denomination, we must search the *computer's* hand to see if it is holding cards of that denomination and, if so, how many. The next few lines of our program do all that, as well as move the cards from the computer's hand to the player's hand:

```
350 NC=0 : REM START COUNT AT 0
360 FOR SU=1 TO 4 : REM CHECK ALL FOUR SUITS
370 IF DK(SU,DE)=2 THEN NC=NC+1 : DK(SU,DE)=1 : REM IF IN
COMPUTER'S HAND (2), COUNT IT, AND MOVE IT TO PLAYER'S
HAND (1)
380 NEXT SU
```

These four lines do a lot. They check through all four suits looking for cards of the requested denomination. The number of cards is counted in variable NC. When a card is found, it is moved from the computer's hand (2) to the player's hand (1).

At the end of the loop, NC is equal to the number of cards that were moved. If NC is still equal to zero, then the computer was holding no cards of that denomination. Effectively, the computer has told the player to "Go Fish!"—and so we skip ahead to the Go Fish routine, like this:

```
390 IF NC=0 THEN 440 : REM IF NO CARDS, THEN "GO FISH"
```

Otherwise, we adjust the number of cards in the player's hand (PN) and the computer's hand (CN) to reflect the transfer of NC cards:

```
400 PN=PN+NC : CN=CN-NC
```

This could cause the computer to run out of cards, in which case the game is over. We check for that like this:

```
410 IF CN=0 THEN PRINT "THE COMPUTER IS OUT OF CARDS!"; :
GOTO 950
```

The routine beginning on 950, you'll recall, will announce the end of the game and check to see who won.

If the game isn't over, we continue. First, we have to announce how many cards were transferred:

```
420 PRINT "THE COMPUTER GIVES YOU";NC; : GOSUB 5000
```

At this point, variable DE still contains the number of the denomination the player requested. The subroutine at line 5000, which we saw earlier, prints out the name of the denomination con-

tained in DE. If the denomination was 7 and the number of cards transferred (NC) was 3, this would print:

```
THE COMPUTER GIVES YOU 3 SEVENS.
```

If you'll recall the rules of Go Fish, a player that is successful in requesting cards from the other player is automatically given a free turn. Therefore, we jump back to the beginning of the main loop:

```
430 GOTO 170 : REM REPEAT HUMAN PLAYER'S TURN
```

A glance at line 170 will tell you that this will cause the player's hand to be printed out again and another search to be conducted for books. (A search for new books should be conducted every time the player puts new cards in his or her hand.)

What if the player was unsuccessful in requesting cards? The computer will have jumped ahead to line 440, as in the instruction in line 390. This is where the computer tells the player to "Go Fish!":

```
440 PRINT "THE COMPUTER SAYS: 'GO FISH!' "
```

"Fishing" is a simple matter of drawing a new card from the deck—and this means that we must call our card-dealing subroutine at line 2000 to deal us a random card, like this:

```
450 PD=DE : CP=1 : GOSUB 2000 : DK(SU,DE)=1 : PN=-PN+1
```

What is this new variable called PD? It stands for "previous denomination." The value of variable DE, which holds the denomination that the player asked for, is about to be changed by the subroutine at line 2000. However, we want to remember what denomination this was, so we save it in PD. The reason for this will become obvious in a moment.

The rest of line 450 sets CP equal to 1 and calls the subroutine at line 2000 to deal a random card to the human player. The instruction PN=PN+1 notes that the number of cards in the player's hand has increased by one.

The next line prints out the name of the card that was drawn:

```
460 PRINT "YOU HAVE DRAWN THE "; : GOSUB 3000 : PRINT"."
```

The subroutine at line 3000 prints the actual name of the card, in our special notation. The statement PRINT"." prints a period (.) at the end of the sentence.

According to the rules, if the card that the player draws from the deck is of the same denomination that he or she just requested from the other player, then the player gets another turn. Thus, we check to see if the denomination of the card that was drawn (DE) is equal to the denomination of the card that was asked for (PD):

```
470 IF DE=PD THEN PRINT "CONGRATULATIONS! YOU GET AN
EXTRA TURN!" : GOTO 170
```

If so, we loop back to the beginning of the main loop, on line 170, and the player's turn is repeated. And now you know why we saved the value of the requested card in variable PD.

If we don't loop back, we still must check to see if the player has any books in his or her hand, now that a new card has been added. This part of the program proceeds much as before:

```
480 CP=1 : GOSUB 4000 : REM LOOK FOR BOOKS IN CP'S HAND
490 IF DE=0 THEN 530 : REM IF NO BOOKS, SKIP AHEAD
500 PRINT "YOU HAVE A BOOK OF "; : GOSUB 5000 : REM ELSE
ANNOUNCE IT
510 PRINT "YOU DISCARD THIS BOOK"
520 PB=PB+1 : PN=PN-4 : IF PN=0 THEN PRINT "YOU ARE OUT
```

```
OF CARDS!"; : GOTO 950 : REM IF PLAYER OUT OF CARDS, END
GAME
```

Either way, it is now the computer's turn. This part of the program is a lot like the last part, except that a few numbers have been changed and a couple of routines added. First, we must announce how many books the computer has discarded (variable CB) and how many cards it holds (variable CN):

```
530 PRINT : PRINT "THE COMPUTER HAS DISCARDED";
CB;"BOOKS"
540 PRINT "IT HAS";CN;"CARDS"
```

When it was the human player's turn, we printed out the contents of the hand. However, it would be cheating to let the human player see the contents of the computer's hand. Nonetheless, you might want to look at the computer's hand occasionally, to check if the program is working correctly, so we will include instructions to print it out. But we will put these instructions in REM statements, so that the computer will think they are remarks and ignore them. If you want the computer to execute these statements and show you the computer's hand, remove the word REM at the beginning of each line:

```
550 REM PRINT "THEY ARE: ";
560 REM FOR DE=1 TO 13
570 REM FOR SU=1 TO 4
580 REM IF DK(SU,DE)=2 THEN GOSUB 2000 : PRINT " ";
590 REM NEXT SU
600 REM NEXT DE
610 REM PRINT
```

Next, we check the computer's hand for books, just as we checked the human player's hand:

```
620 CP=2 : GOSUB 4000 : REM LOOK FOR BOOKS IN CP'S HAND
630 IF DE=0 THEN 670 : REM IF NO BOOKS, SKIP AHEAD
640 PRINT "THE COMPUTER HAS A BOOK OF "; : GOSUB 5000 :
REM ELSE ANNOUNCE IT
650 PRINT "IT DISCARDS THIS BOOK"
660 CB=CB+1 : CN=CN-4 : IF CN=0 THEN PRINT "THE COMPUT-
ER IS"; : GOTO 950
```

The last line adjusts variable CB (the number of books discarded by the computer) and variable CN (the number of cards in the computer's hand). If the computer is out of cards, we jump to line 950, which announces the end of the game.

Now it is time for the computer to ask the human player for cards of a certain denomination. This is a very important part of the program. It is here that we give the computer its game-playing strategy, its method of decision making.

Strategy? Decision making? Those are pretty heavy concepts for a computer program. They smack of artificial intelligence—computer programs that actually think like human beings.

Maybe so, but we will beg this issue by giving the computer a very simple strategy for choosing the denomination it asks for. We will have it choose a denomination at random.

This might sound like a pretty poor strategy, but you'll be surprised at how well it works. In fact, the computer may beat you the first few times you play against it, unless you are a very experienced Go Fish player. A random strategy in Go Fish is fairly effective, though a clever human player will learn to beat the machine—most of the time, anyway. Of course, the computer will occasionally do stupid things like ask for cards that you have just given it and couldn't possibly have drawn in the meantime—but human players do stupid things too.

Here is how the computer picks the random denomination:

```
670 DE=INT(RND(0)*13+1)
```

This sets DE equal to a random denomination betwen 1 and 13. (Remember, if you needed to change the RND(0) function in the subroutine at line 2000 to fit your version of BASIC, you will need to make a similar change here.) If this denomination has already been discarded, then the computer should choose a different denomination. This is easy enough to check:

```
680 IF DK(1,DE) =3 THEN 670 : REM IF IN A BOOK, TRY AGAIN
```

We need only check one card of that denomination to see if it is equal to 3 (that is, if it is in a book), because if any card of that denomination is in a book, then all four are in that book.

Also, the computer is not allowed to ask for a denomination unless it already has at least one card of that denomination in its hand. Hence, we must use the subroutine at line 6000 to search the computer's hand:

```
690 CP=2 : GOSUB 6000 : REM ANY IN THE COMPUTER'S HAND?
```

If there are none in the computer's hand, we must go back for another random denomination:

```
700 IF FL=0 THEN 670 : REM IF NONE, THEN TRY AGAIN
```

Once we come up with a valid denomination, we must ask the player for it:

```
710 PRINT "THE COMPUTER SAYS:"
720 PRINT "GIVE ME ALL OF YOUR "; : GOSUB 5000
```

The subroutine at line 5000 prints the name of denomination DE.

The player doesn't really need to respond, because the computer can check his or her hand automatically and transfer the

cards. However, it would be nice to pause the program and give the player a chance to rest a minute:

```
730 PRINT "PRESS <RETURN> TO CONTINUE"
740 INPUT Q
```

This will pause the program until the RETURN (or ENTER) key is pressed. (If the key on your computer is called ENTER rather than RETURN, you can change this program line accordingly.)

Next, we must search the human player's hand for cards of this denomination:

```
750 NC=0 : REM SET COUNT TO 0
760 FOR SU=1 TO 4 : REM CHECK ALL FOUR SUITS
770 IF DK(SU,DE)=1 THEN NC=NC+1 : DK(SU,DE)=2 : REM IF CARD FOUND, COUNT IT AND TRANSFER IT
780 NEXT SU
```

Just as before, any cards of the requested denomination are automatically transferred, this time from the player's hand to the computer's.

Variable NC will be set equal to the number of cards of denomination DE that are transferred. If this is 0, then we must skip ahead and tell the computer to "Go Fish!":

```
790 IF NC=0 THEN 840 : REM IF NO CARDS, THEN "GO FISH!"
```

Otherwise, we adjust the values of CN (number of cards in computer's hand) and PN (number of cards in player's hand):

```
800 CN=CN+NC : PN=PN-NC : REM ADD TO COMPUTER'S HAND, SUBTRACT FROM PLAYER'S HAND
```

and we announce the transfer:

```
810 PRINT "YOU GIVE THE COMPUTER";NC; : GOSUB 5000
```

Then we check to see if the human player ran out of cards:

```
820 IF PN=0 THEN PRINT "YOU ARE OUT OF CARDS!" : GOTO
950
```

If not, we loop back to give the computer another turn, because it was successful in its request:

```
830 GOTO 530
```

If the computer was unsuccessful in its request:

```
840 PRINT "YOU TELL THE COMPUTER: 'GO FISH!' "
```

We deal the computer a random card from the deck:

```
850 PD=DE : CP=2 : GOSUB 2000 : DK(SU,DE)=2 : CN=CN+1
```

We then announce that the computer has drawn a card, but not what card it has drawn. (That would be cheating!)

```
860 PRINT "THE COMPUTER DRAWS FROM THE DECK" : IF
PD<>DE THEN 890
```

Notice that we skip ahead to line 890 if the computer does not get the denomination it requests—that is, if PD does not equal DE. If it *does* get the denomination it requests, it gets another turn, just as the human player did—but it must show what card it drew, to prove it got the requested denomination:

```
870 PRINT "THE COMPUTER HAS DRAWN THE "; : GOSUB 3000 :
PRINT "."
880 PRINT "IT GETS AN EXTRA TURN!" : GOTO 530
```

The subroutine at line 3000 prints out the name of the denomination. Of course, these lines are skipped if the computer does not get the denomination it asked for.

We must now search the computer's hand a second time for books and announce any books we find:

```
890 CP=2 : GOSUB 4000 : REM LOOK FOR BOOKS
900 IF DE=0 THEN 940 : REM IF NOT FOUND, SKIP AHEAD
910 PRINT "THE COMPUTER HAS A BOOK OF "; : GOSUB 5000
920 PRINT "IT DISCARDS THIS BOOK"
```

All the numbers must be properly adjusted, and we must check to see if the computer has run out of cards:

```
930 CB=CB+1 : CN=CN-4 : IF CN=0 THEN PRINT "THE COMPUTER IS OUT OF CARDS" : GOTO 950
```

Finally, we return to the start of the main loop, so that the human player can have another turn as the game continues:

```
940 PRINT : GOTO 170
```

The PRINT statement merely adds a blank line on the video display, to make it more readable.

We have made several references to a routine on line 950, which ends the game and congratulates the winner. Here it is:

```
950 REM CONGRATULATE THE WINNER
960 PRINT "THE GAME IS OVER!"
970 PRINT "YOU HAVE";PB;"BOOKS"
980 PRINT "THE COMPUTER HAS";CB;"BOOKS"
990 IF CB>PB THEN PRINT "THE COMPUTER HAS WON!" : GOTO 1020
1000 IF PB>CB THEN PRINT "YOU HAVE WON!" : GOTO 1020
1010 PRINT "THE GAME IS A TIE!"
```

We politely ask the player if he or she wants to play again:

```
1020 PRINT "PLAY AGAIN? (1=YES, 2=NO)"
1030 INPUT Q
1040 IF Q=1 THEN 30 : REM IF YES, PLAY AGAIN
1050 IF Q=2 THEN END : REM IF NO, END PROGRAM
1060 GOTO 1020 : REM IF NEITHER, ASK AGAIN
```

And that is the Go Fish program. A complete listing is given at the end of this chapter, for you to type on your computer.

We should mention that this is a stripped-down, no-frills version of Go Fish. There is a lot that can be done with this program, and we encourage you to experiment.

As we noted earlier, there are no graphics in this program, no pictures. It would be very satisfying if the player could see the cards as they are dealt. A lot of computers, like the Commodore 64, the Atari 800, the Radio Shack Model III/4, and the IBM PC, have the heart, spade, club, and diamond symbols built into their character sets. You could use these characters to replace the cryptic abbreviations—Ht, Sp, Cl, and Di—that we use for these suits in the subroutine at line 3000.

And if you feel particularly ambitious, you could even add sound effects, so that the player could hear a whooshing sound as each card is dealt. Many versions of BASIC have a SOUND command that can be used to produce such sound effects.

How else can you improve this program? Well, you will notice several rough edges in the way the game "talks" to the player. We have left these rough edges in, because it would have complicated the program to remove them. However, you might want to add some instructions that will get rid of them.

For instance, if you have discarded only one book in the game, the computer will continually remind you that:

```
YOU HAVE DISCARDED 1 BOOKS
```

Nobody with a decent command of the English language says "1 books"—but the computer doesn't know any better. You can add instructions telling the computer to change the word "books" to "book" when the number of books (in variable NB) is 1.

In the same way, the computer will say things like:

```
THE COMPUTER GIVES YOU 1 ACES
```

Add instructions to tell the computer that "1 aces" should be "1 ace."

Another area you might want to improve is in the strategy by which the computer chooses the denomination it requests of the human player. As explained earlier, this choice is made randomly—but it doesn't have to be. While you play the game, notice the way in which you make your own choices about which cards to ask for.

Do you notice which denomination the computer keeps asking you for (and therefore must have in its hand)—then wait until you draw a card of that denomination so that you can get those cards from the computer? Do you ever ask the computer for a card that you know it cannot possibly have? Are there certain circumstances under which you *must* ask the computer for a card you know it does not have? Once you have figured out a strategy, see if you can program that strategy into the computer.

As you play the game, of course, you will see other areas where it could use improvement. Make all the improvements you want. As written, this game is really a skeleton, waiting for you to flesh it out, to add the extra features that programmers call "bells and whistles."

And when you have put the program together exactly as you want it, try it out in your friends. But don't be surprised if they have a few suggestions of their own!

Here's the complete program listing:

```
10 REM ( Add an instruction on this line to clear the video display of your
computer. See Chapter Two for details.)
20 DIM DK(4,13)
30 ND=52
40 PRINT "DEALING"
50 FOR SU=1 TO 4
60 FOR DE=1 TO 13
70 DK(SU,DE)=0
80 NEXT DE
90 NEXT SU
100 FOR I=1 TO 7
110 GOSUB 2000 : DK(SU,DE)=1
120 GOSUB 2000 : DK(SU,DE)=2
130 NEXT I
150 PN=7 : CN=7
160 PB=0 : CB=0
170 PRINT "YOU HAVE DISCARDED";PB;"BOOKS"
180 PRINT "YOU HAVE";PN;"CARDS"
190 PRINT "THEY ARE: ";
200 FOR DE=1 TO 13
210 FOR SU=1 TO 4
220 IF DK(SU,DE)=1 THEN GOSUB 3000 : PRINT " ";
230 NEXT SU
240 NEXT DE
250 PRINT
260 CP=1 : GOSUB 4000 : REM LOOK FOR BOOKS
270 IF DE=0 THEN 310
280 PRINT "YOU HAVE A BOOK OF "; : GOSUB 5000
290 PRINT "YOU DISCARD THIS BOOK"
300 PB=PB+1 : PN=PN-4 : IF PN=0 THEN PRINT "YOU ARE OUT
OF CARDS!" : GOTO 950
310 PRINT "WHICH DENOMINATION WOULD YOU LIKE (1-13)?"
320 INPUT DE
330 CP=1 : GOSUB 6000
340 IF FL=0 THEN PRINT "YOU DON'T HAVE ANY IN YOUR HAND!" :
```

```
GOTO 310
350 NC=0
360 FOR SU=1 TO 4
370 IF DK(SU,DE)=2 THEN NC=NC+1 : DK(SU,DE)=1
380 NEXT SU
390 IF NC=0 THEN 440
400 PN=PN+NC : CN=CN-NC
410 IF CN=0 THEN PRINT "THE COMPUTER IS OUT OF CARDS!" :
GOTO 950
420 PRINT "THE COMPUTER GIVES YOU";NC; : GOSUB 5000
430 GOTO 170
440 PRINT "THE COMPUTER SAYS: 'GO FISH!' "
450 PD=DE: CP=1 : GOSUB 2000 : DK(SU,DE)=1 : PN=PN+1
460 PRINT "YOU HAVE DRAWN THE "; : GOSUB 3000 : PRINT" "
470 IF DE=PD THEN PRINT "CONGRATULATIONS! YOU GET AN
EXTRA TURN!": GOTO 170
480 CP=1 : GOSUB 4000 : REM LOOK FOR BOOKS
490 IF DE=0 THEN 530
500 PRINT "YOU HAVE A BOOK OF "; : GOSUB 5000
510 PRINT "YOU DISCARD THIS BOOK"
520 PB=PB+1 : PN=PN-4 : IF PN=0 THEN PRINT "YOU ARE OUT
OF CARDS!"; : GOTO 950
530 PRINT : PRINT "THE COMPUTER HAS DIS-
CARDED";CB;"BOOKS"
540 PRINT "IT HAS";CN;"CARDS"
550 REM PRINT "THEY ARE: ";
560 REM FOR DE=1 TO 13
570 REM FOR SU=1 TO 4
580 REM IF DK(SU,DE)=2 THEN GOSUB 2000 : PRINT " ";
590 REM NEXT SU
600 REM NEXT DE
610 REM PRINT
620 CP=2 : GOSUB 4000
630 IF DE=0 THEN 670
640 PRINT "THE COMPUTER HAS A BOOK OF "; : GOSUB 5000
```

```
650 PRINT "IT DISCARDS THIS BOOK"
660 CB=CB+1 : CN=CN-4 : IF CN=0 THEN PRINT "THE COMPUT-
ER IS OUT OF CARDS!"; : GOTO 950
670 DE=INT(RND(0)*13+1)
680 IF DK(1,DE)=3 THEN 670
690 CP=2 : GOSUB 6000
700 IF FL=0 THEN 670
710 PRINT "THE COMPUTER SAYS:"
720 PRINT "GIVE ME ALL OF YOUR "; : GOSUB 5000
730 PRINT "PRESS <RETURN> TO CONTINUE"
740 INPUT Q
750 NC=0
760 FOR SU=1 TO 4
770 IF DK(SU,DE)=1 THEN NC=NC+1 : DK(SU,DE)=2
780 NEXT SU
790 IF NC=0 THEN 840
800 CN=CN+NC : PN=PN-NC
810 PRINT "YOU GIVE THE COMPUTER";NC; : GOSUB 5000
820 IF PN=0 THEN PRINT "YOU ARE OUT OF CARDS!" : GOTO
950
830 GOTO 530
840 PRINT "YOU TELL THE COMPUTER: 'GO FISH!' "
850 PD=DE : CP=2 : GOSUB 2000 : DK(SU,DE)=2 : CN=CN+1
860 PRINT "THE COMPUTER DRAWS FROM THE DECK" : IF
PD<>DE THEN 890
870 PRINT "THE COMPUTER HAS DRAWN THE "; : GOSUB 3000 :
PRINT "."
880 PRINT "IT GETS AN EXTRA TURN!" : GOTO 530
890 CP=2 : GOSUB 4000 : REM LOOK FOR BOOKS
900 IF DE=0 THEN 940
910 PRINT "THE COMPUTER HAS A BOOK OF "; : GOSUB 5000
920 PRINT "IT DISCARDS THIS BOOK"
930 CB=CB+1 : CN=CN-4 : IF CN=0 THEN PRINT "THE COMPUT-
ER IS OUT OF CARDS!" : GOTO 950
940 PRINT : GOTO 170
```

```
950 REM CONGRATULATE THE WINNER
960 PRINT "THE GAME IS OVER!"
970 PRINT "YOU HAVE";PB;"BOOKS"
980 PRINT "THE COMPUTER HAS";CB;"BOOKS"
990 IF CB>PB THEN PRINT "THE COMPUTER HAS WON!" : GOTO
1020
1000 IF PB>CB THEN PRINT "YOU HAVE WON!" : GOTO 1020
1010 PRINT "THE GAME IS A TIE!"
1020 PRINT "PLAY AGAIN? (1=YES, 2=NO)"
1030 INPUT Q
1040 IF Q=1 THEN 30
1050 IF Q=2 THEN END
1060 GOTO 1020
1989 REM
1990 REM *** DEAL A CARD
1991 REM ***
1992 REM *** SUBROUTINE TO DEAL ONE CARD
1993 REM *** FROM A DECK OF 52 CARDS, IN
1994 REM *** ARRAY DK(SUIT, DENOMINATION)
1995 REM ***
1996 REM
2000 IF ND>0 THEN : GOTO 2020
2010 PRINT "THE DECK IS EMPTY!" : END
2020 SU=INT(RND(0)*4+1) : DE=INT(RND(0)*13+1)
2030 IF DK(SU,DE)<>0 THEN 2020
2040 DK(SU,DE)=CP : ND=ND-1
2050 RETURN
2989 REM
2990 REM *** NAME THE CARD
2991 REM ***
2992 REM *** A SUBROUTINE TO WRITE THE NAME
2993 REM *** OF A CARD ON THE VIDEO DISPLAY
2994 REM *** IN ABBREVIATED NOTATION, WHERE
2995 REM *** SU IS THE SUIT AND DE IS THE
2996 REM *** DENOMINATION OF THE CARD
```

```
2997 REM ***
2998 REM
3000 IF DE>1 AND DE<10 THEN PRINT [CHR$(DE+ASC("0"))]; :
GOTO 3060
3010 IF DE=10 THEN PRINT "10"; : GOTO 3060
3020 IF DE=1 THEN PRINT "A"; : GOTO 3060
3030 IF DE=11 THE PRINT "J"; : GOTO 3060
3040 IF DE=12 THEN PRINT "Q"; : GOTO 3060
3050 IF DE=13 THEN PRINT "K"; : GOTO 3060
3060 PRINT"-";
3070 IF SU=1 THEN PRINT "Ht"; : RETURN
3080 IF SU=2 THEN PRINT "Cl"; : RETURN
3090 IF SU=3 THEN PRINT "Di"; : RETURN
3100 IF SU=4 THEN PRINT "Sp"; : RETURN
3989 REM
3990 REM *** CHECK FOR A BOOK
3991 REM ***
3992 REM *** A SUBROUTINE TO CHECK THE DECK
3993 REM *** TO SEE IF THE CURRENT PLAYER (CP)
3994 REM *** HAS FOUR CARDS OF THE SAME
3995 REM *** DENOMINATION (DE)—A "BOOK," IN
3996 REM *** THE PARLANCE OF THE GAME. RETURNS
3997 REM *** THE DENOMINATION (OR 0) IN DE
3998 REM ***
3999 REM
4000 FOR DE=1 TO 13
4010 NC=0
4020 FOR SU=1 TO 4
4030 IF DK(SU,DE)=CP THEN NC=NC+1
4040 IF NC=4 THEN 4090
4050 NEXT SU
4060 NEXT DE
4070 DE=0
4080 RETURN
```

```
4090 FOR SU=1 TO 4
4100 DK(SU,DE)=3
4110 NEXT SU
4120 RETURN
4989 REM
4990 REM *** NAME THE DENOMINATION
4991 REM ***
4992 REM *** A SUBROUTINE TO PRINT THE
4993 REM *** NAME OF DENOMINATION DE TO
4994 REM *** THE VIDEO DISPLAY AS A WORD
4995 REM ***
4996 REM
5000 IF DE=1 THEN PRINT "ACES" : RETURN
5010 IF DE=2 THEN PRINT "TWOS" : RETURN
5020 IF DE=3 THEN PRINT "THREES" : RETURN
5030 IF DE=4 THEN PRINT "FOURS" : RETURN
5040 IF DE=5 THEN PRINT "FIVES" : RETURN
5050 IF DE=6 THEN PRINT "SIXES" : RETURN
5060 IF DE=7 THEN PRINT "SEVENS" : RETURN
5070 IF DE=8 THEN PRINT "EIGHTS" : RETURN
5080 IF DE=9 THEN PRINT "NINES" : RETURN
5090 IF DE=10 THEN PRINT "TENS" : RETURN
5100 IF DE=11 THEN PRINT "JACKS" : RETURN
5110 IF DE=12 THEN PRINT "QUEENS" : RETURN
5120 PRINT "KINGS" : RETURN
5989 REM
5990 REM *** CHECK HAND FOR DENOMINATION
5991 REM ***
5992 REM *** A SUBROUTINE TO CHECK THE
5993 REM *** HAND OF THE CURRENT PLAYER (CP)
5994 REM *** DENOMINATION DE. THE VARIABLE
5996 REM *** FL IS SET EQUAL TO 1 IF SO,
5997 REM *** AND 0 IF NOT
5998 REM ***
```

```
5999 REM
6000 FL=0
6010 FOR SU=1 TO 4
6020 IF DK(SU,DE)=CP THEN FL=1
6030 NEXT SU
6040 RETURN
```

CHAPTER FOUR

SOMETHING COMPLETELY DIFFERENT

A variable array seems ideally suited for representing a deck of cards, as we saw in Chapter Two. The truth is, variable arrays are ideally suited for representing almost anything. In this chapter, we will show how the data structure that we call a two-dimensional array can be used to represent something completely different—the game structure that we call a tic-tac-toe board.

Actually, a tic-tac-toe board is nothing more than a pattern of lines drawn on a sheet of paper, and the "pieces" with which the game is played are nothing more than Xs and Os drawn with a pencil or pen. Nonetheless, we can represent these things much as we represented the playing cards in Go Fish.

We'll assume that everyone reading this book has played tic-tac-toe at some time or another. To loosely summarize the game: two players take turns placing Xs and Os in the nine squares of a crosshatched "board" drawn on a sheet of paper. The first player who can get three Xs or Os in a row, either diagonally, vertically, or horizontally, is the winner.

The strategy of tic-tac-toe is more complex than it might at first seem, so we are not going to write a program that pits the

player directly against the computer. Rather, we are going to write a program that will let two human players play against each other, using the computer screen as a playing field instead of a piece of paper. Of course, if you feel that you are up to the job, you can rewrite the program so that the game is actually played against the computer. Good luck!

The tic-tac-toe board will be simulated by a two-dimensional array called TB, which will be dimensioned like this:

```
20 DIM TB(3,3)
```

Each element of array TB will represent one square of the tic-tac-toe board, in this pattern:

TB(1,1)	TB(1,2)	TB(1,3)
TB(2,1)	TB(2,2)	TB(2,3)
TB(3,1)	TB(3,2)	TB(3,3)

If we think of the tic-tac-toe board as being made up of three horizontal (left-to-right) *rows* and three vertical (up-and-down) *columns*, then the first subscript after TB stands for the row and the second stands for the column. If the value of an element of TB is 0, then the square where that row and column meet is empty. If it is equal to 1, then the first player has his or her mark (the X) in that square. If it is equal to 2, then the second player has his or her mark (the O) in that square.

We must now set up some variables:

```
30 CP=0 : NM=0
```

CP equals the number of the current player, 1 for player one and 2 for player 2. Player 1 will use the X to mark squares and player 2

will use the O. NM equals the number of markers—both Xs and Os—that have been placed on the board.

At the start of the game, we must set every element of the array TB to 0, to indicate that no squares have been marked with either Xs or Os:

```
40 FOR ROW = 1 TO 3 : REM GET EACH ROW . . .
50 FOR COL = 1 TO 3 : REM . . . AND EACH COLUMN
60 TB(ROW,COL) = 0 : REM SET THEM TO 0
70 NEXT COL
80 NEXT ROW
```

Then we must "draw" the tic-tac-toe board. Although we are not using graphics in this game, we will depict the board as a series of letters and symbols. Hyphens ('-') will represent blank squares—that is, squares that have not yet been marked. Xs and Os will fill the remaining squares. The job of drawing this board is important enough that we will put it in a subroutine, like this:

```
1000 REM ** DRAW TIC-TAC-TOE BOARD **
1010 PRINT " "; : REM BLANK SPACE
1020 FOR COL = 1 TO 3 : REM FOR ALL 3 COLUMNS . . .
1030 PRINT COL; : REM . . . PRINT NUMBERS ALONG TOP
1040 NEXT COL
1050 PRINT : REM ADD A BLANK LINE AT TOP
1060 FOR ROW = 1 TO 3 : REM FOR ALL 3 ROWS . . .
1070 PRINT ROW; : REM . . . PRINT NUMBERS ALONGSIDE
1080 FOR COL = 1 TO 3 : REM FOR ALL 3 COLUMNS . . .
1090 IF TB(ROW,COL) = 0 THEN PRINT " - "; : REM FILL IN . . .
1100 IF TB(ROW,COL) = 1 THEN PRINT " X "; : REM EACH
SQUARE . . .
1100 IF TB(ROW,COL) = 2 THEN PRINT " O "; : REM WITH SYM-
BOLS
1120 NEXT COL
```

```
1130 PRINT : REM CARRIAGE RETURN TO NEXT LINE
1140 NEXT ROW
1150 PRINT : REM BLANK LINE AT BOTTOM
1160 RETURN
```

As you will see, this routine prints out numbers at the head of each column and at the beginning of each row, so that we can easily identify the location of each square. The spacing of the characters is very important here, which may present a problem if you are using certain versions of BASIC. As we have mentioned before, some versions of BASIC print extra blank spaces around numbers and others don't. This subroutine assumes that the spaces are printed. If you are using Applesoft BASIC or Atari BASIC or any other version that does not perform this automatic spacing, the board will look very cramped, and the column numbers will be in the wrong positions. You will have to rewrite this routine so that spaces will be printed between the squares of the board, and between the column numbers.

We'll call the subroutine like this:

```
90 GOSUB 1000 : REM ** DRAW TIC-TAC-TOE BOARD **
```

In tic-tac-toe, tie games are very common. In fact, if two very good tic-tac-toe players are pitted against one another, *every* game will end in a tie, unless one player makes a mistake. So we have to be careful to check to see if the board has been filled up with markers, and stop the game if it is. The number of markers is contained in variable NM. If this number goes over 9 (the number of squares on the board) and nobody has won yet, the game is a tie. Line 100 watches for a tie game, and increases the value of NM by 1:

```
100 NM=NM+ : IF NM>9 THEN PRINT : PRINT "TIE GAME" : GOTO
400
```

Of course, this isn't going to happen on the first move, but this line will be repeated before every move in the game.

Now it is time for a player to put his or her marker on the board. The current player, 1 or 2, is represented by variable CP. We initially set CP equal to 0, which is not the number of either player. So we will now add 1 to that, and make it the first player's turn:

```
110 CP=CP+1
```

The next time this loop executes, this statement will add another 1 to CP, making the current player 2. But the next time, we will want the current player to be 1 again, so we must make sure that the value of CP never becomes greater than 2:

```
120 IFCP>2 THEN CP=1
```

Once we have decided whose turn it is, we must prompt the player to make a move:

```
130 PRINT "PLAYER";CP" 'S MOVE"
```

First, we will ask the player to choose the row on which his or her marker is to be placed:

```
140 PRINT "ROW;"
150 INPUT ROW
```

Then we will ask what column:

```
160 PRINT "COLUMN";
170 INPUT COL
180 PRINT
```

The PRINT statement in line 180 adds a blank line to keep the video display neat.

It would also be nice to give the players a way to end a game in progress—in case it is obvious that nobody is going to win. If the player types −1 when asked for a ROW, we will stop the game, like this:

```
155 IF RO = −1 THEN PRINT "GAME ABORTED" : GOTO 400
```

We should make sure that the row and column that the player asked for fall within the legal bounds—that is, that neither is greater than 3 or less than 1:

```
190 IF COL<1 OR COL>3 OR ROW<1 OR ROW>3 THEN GOTO
130
```

If an illegal column or row has been asked for, we skip back to line 130 and ask again.

Finally, we must mark TB(ROW,COL) as having the current player's marker on it. First, though, we must check to see that it is not already occupied by a marker:

```
200 IF TB(ROW,COL)=0 THEN TB(ROW,COL)=CP : GOTO 230
210 PRINT "THAT SQUARE IS TAKEN. TRY AGAIN"
220 GOTO 130
```

Should it already be occupied, we jump back to line 130 and ask again.

Now comes the interesting part. We must examine the board to see if the current player has gotten three markers in a row. First we'll check the horizontal row:

```
230 FL=0 : REM ASSUME THE BEST
240 FOR I=1 TO 3 : REM CHECK THREE SQUARES
```

```
250 IF TB(I,COL)<>CP THEN FL=1 : REM IF MARKER NOT FOUND
ON ANY SQUARE, ASSUME THE WORST
260 NEXT I
270 IF FL=0 THEN 370 : REM WE'VE GOT A WINNER
```

The loop beginning in line 240 takes us through all three squares on the row where player CP put his or her marker. Variable FL tells us if any of those squares are *not* occupied by CP's marker. If FL equals 1 after the loop, CP did not have a marker on at least one of the squares—and therefore cannot have three markers in a horizontal row. If FL is still equal to 0 after the loop, then CP does have three markers in a row, so we jump to line 370 to offer our congratulations!

We can do the same thing for the vertical column on which CP placed his or her marker:

```
280 FL=0 : REM ASSUME THE BEST
290 FOR I=1 TO 3 : REM CHECK THREE SQUARES
300 IF TB(ROW,I)<>CP THEN FL=1 : REM IF MARKER NOT FOUND
ON ANY SQUARE, ASSUME THE WORST
310 NEXT I
320 IF FL=0 THEN 370 : REM WE HAVE A WINNER
```

Once again, variable FL tells us if CP has three markers in a row vertically.

Now we have to check to see if player CP has three markers in a *diagonal* row. We could do this in a similar way, but we'll try a different method this time:

```
340 IF TB(1,1)*TB(2,2)*TB(3,3)=CP[3 THEN 370
350 IF TB(3,1)*TB(2,2)*TB(1,3)=CP[3 THEN 370
```

The square bracket symbol ([ ) in lines 340 and 350 is a special arithmetic operator. It raises a number to a power—that is, multi-

plies a number by itself a certain number of times. On some versions of BASIC, such as Applesoft, this should be replaced by a circumflex symbol (^) and on others, such as Commodore BASIC, by a symbol that looks like an upward pointing arrow (↑). If player 1 has placed markers on all squares in the diagonal, then the result of multiplying the values in all of those squares together (as we have done above) will be 1 multiplied by itself three times, or 1. If player 2 has placed markers in all three squares, then the result of multiplying the values together will be 2 multiplied by itself three times, or 8. Thus, we can see if any player has occupied all squares in the diagonal by checking if the result of this multiplication is 1 or 8.

Finally, we must loop back, draw the board again, and let the other player make a move:

```
360 GOTO 90 : REM DO IT AGAIN
```

When the game is over, we must print a picture of the board and congratulate the winner:

```
370 GOSUB 1000 : REM ** DRAW TIC-TAC-TOE BOARD **
380 PRINT "CONGRATULATIONS, PLAYER";CP;"—YOU HAVE WON!"
390 PRINT
```

As before, we will let the players decide whether to play again:

```
400 PRINT "PLAY AGAIN? (1=YES, 2=NO)"
410 INPUT Q
420 IF Q=1 THEN GOTO 30 : REM IF YES, PLAY AGAIN
430 IF Q=2 THEN END : REM IF NO, END PROGRAM
440 GOTO 400 : REM IF NEITHER, ASK AGAIN
```

Here is the complete tic-tac-toe program:

```
10 REM Add an instruction here to clear the video display of your com-
puter. See Chapter Two for details.
20 DIM TB(3,3)
30 CP=0 : NM=0
40 FOR ROW = 1 TO 3
50 FOR COL = 1 TO 3
60 TB(ROW,COL) = 0
70 NEXT COL
80 NEXT ROW
90 GOSUB 1000 : REM ** DRAW TIC-TAC-TOE BOARD **
100 NM=NM+1 : IF NM>9 THEN PRINT : PRINT "TIE GAME" : GOTO
400
110 CP=CP+1
120 IF CP>2 THEN CP=1
130 PRINT "PLAYER";CP" 'S MOVE"
140 PRINT "ROW";
150 INPUT ROW
155 IF ROW = -1 THEN PRINT "GAME ABORTED" : GOTO 400
160 PRINT "COLUMN";
170 INPUT COL
180 PRINT
190 IF COL<1 OR COL>3 OR ROW<1 OR ROW>3 THEN GOTO
130
200 IF TB(ROW,COL)=0 THEN TB(ROW,COL)=CP : GOTO 230
210 PRINT "THAT SQUARE IS TAKEN. TRY AGAIN"
220 GOTO 130
230 FL=0
240 FOR I=1 TO 3
250 IF TB(I,COL)<>CP THEN FL=1
260 NEXT I
270 IF FL=0 THEN 370
280 FL=0
290 FOR I=1 TO 3
300 IF TB(ROW,I)<>CP THEN FL=1
310 NEXT I
```

```
320 IF FL=0 THEN 370
340 IF TB(1,1)*TB(2,2)*TB(3,3)=CP[3 THEN 370
350 IF TB(3,1)*TB(2,2)*TB(1,3)=CP[3 THEN 370
360 GOTO 90
370 GOSUB 1000 : REM ** DRAW TIC-TAC-TOE BOARD **
380 PRINT "CONGRATULATIONS, PLAYER";CP;"—YOU HAVE
WON!"
390 PRINT
400 PRINT "PLAY AGAIN? (1=YES, 2=NO)"
410 INPUT Q
420 IF Q=1 THEN GOTO 30
430 IF Q=2 THEN END
440 GOTO 400
1000 REM ** DRAW TIC-TAC-TOE BOARD **
1010 PRINT " ";
1020 FOR COL = 1 TO 3
1030 PRINT COL;
1040 NEXT COL
1050 PRINT
1060 FOR ROW = 1 TO 3
1070 PRINT ROW;
1080 FOR COL = 1 TO 3
1090 IF TB(ROW,COL)=0 THEN PRINT " - ";
1100 IF TB(ROW,COL)=1 THEN PRINT " X ";
1110 IF TB(ROW,COL)=2 THEN PRINT " O ";
1120 NEXT COL
1130 PRINT
1140 NEXT ROW
1150 PRINT
1160 RETURN
```

CHAPTER FIVE

CHECKERS

Now that we know how to set up a game board, it's a simple—well, relatively simple—matter to write programs for more complex board games. In this chapter, we'll write a program for the game checkers.

Like the tic-tac-toe program in the last chapter, our checkers program will not play the game against a human player; rather, it will provide a playing board that two human players can use to play against each other. However, it will check for legal moves and detect when one player or the other has won.

Presumably, you've played a game of checkers at some time in your life. To refresh your memory of the rules, here's a brief summary:

Checkers is played on a board of sixty-four red and black squares, arranged in eight rows of eight squares each, in alternating colors. Each player has twelve pieces, either red or black in color. At the beginning of the game, these pieces are placed in the black squares of the first three rows on either side of the board, black pieces on one side and red pieces on the other. Players alternate moves. Pieces may be moved one square diagonally, but may

not be moved onto an occupied square. All movement must be toward the opposing player's side of the board. When a piece makes it to the far side of the board, it becomes a "king" and may then move toward either side of the board (though it must still move diagonally). A player may jump diagonally over an opponent's piece, if the square beyond that piece is empty, thereby "capturing" the piece and removing it from the board. The winner is the player who captures all of the opposing player's pieces.

We'll call our checkerboard array CB. It can be dimensioned exactly as we dimensioned the tic-tac-toe board, except that the rows and columns will be longer, with eight squares apiece instead of three:

```
20 DIM CB(8,8)
```

Once again, the first subscript will indicate the row of a square and the second subscript will indicate the column. If there is no piece on a square, the element representing that square will be set to 0. If player 1 has a piece on that square, the element will be set to 1. If player 2 has a piece on that square, the element will be set to 2. If player 1 has a *king* on that square, the element will be set equal to 3. If player 2 has a king on that square, the element will be set equal to 4.

The black player always moves first; hence, we'll call this player "player 1." As usual, the variable CP will hold the number of the current player. A new feature of this program is that the variable OP (for "other player" or "opponent") will hold the number of the opposing player. In the beginning, CP is player 1 and OP is player 2:

```
30 CP=1 : OP=2
```

Now we must initialize the checkerboard array. In the tic-tac-toe game, this was a simple matter of setting every element of the array equal to 0. In checkers, however, we start with twenty-four

pieces already on the board, and so we must set the appropriate elements of the array equal to 1 and 2, to indicate which squares are occupied by each player's pieces. To ease our task, we will create a set of DATA statements containing the opening setup of the board, and place these DATA statements at the very end of our program, like this:

```
3000 DATA 0,1,0,1,0,1,0,1
3010 DATA 1,0,1,0,1,0,1,0
3020 DATA 0,1,0,1,0,1,0,1
3030 DATA 0,0,0,0,0,0,0,0
3040 DATA 0,0,0,0,0,0,0,0
3050 DATA 2,0,2,0,2,0,2,0
3060 DATA 0,2,0,2,0,2,0,2
3070 DATA 2,0,2,0,2,0,2,0
```

Each set of eight data items in these statements represents a single row of the board. Zeroes represent empty squares, ones represent squares occupied by player 1, and twos represent squares occupied by player 2. If you look closely at these data statements, in fact, you can almost see a representation of the board as it will look at the beginning of the game. We can move this representation into the actual array CB with a pair of nested FOR-NEXT loops and a READ statement, like this:

```
40 FOR ROW=1 TO 8 : REM LOOP THROUGH EACH ROW . . .
50 FOR COL=1 TO 8 : REM . . . AND EACH COLUMN
60 READ A : REM READ THE DATA FOR THAT SQUARE
70 CB(ROW,COL)=A : REM INITIALIZE THE SQUARE
80 NEXT COL
90 NEXT ROW
```

As in the tic-tac-toe program, we'll include a special subroutine to draw the board. Black squares will be represented by a pair of hyphens (--), red squares by a pair of equals signs (==), black

pieces by the letters BL, red pieces by the letters RD, black kings by the letters BK, and red kings by the letters RK. Here's the subroutine:

```
1000 PRINT " ";
1010 FOR COL=1 TO 8 : REM FOR ALL 8 COLUMNS . . .
1020 PRINT COL;" "; : REM . . . PRINT THE COLUMN NUMBERS
1030 NEXT
1040 PRINT
1050 FOR ROW=1 TO 8 : REM FOR ALL 8 ROWS . . .
1060 PRINT ROW; : REM . . . PRINT THE ROW NUMBERS
1070 FOR COL=1 TO 8 : REM . . . AND PRINT EACH SQUARE
1080 IF INT((ROW+COL)/2)*2=(ROW+COL) THEN PRINT
" == ";:GOTO 1140 : REM PRINT RED SQUARE
1090 IF CB(ROW,COL)=0 THEN PRINT "--"; : GOTO 1140 : REM
PRINT BLACK SQUARE
1100 IF CB(ROW,COL)=1 THEN PRINT " BL "; : GOTO 1140
1110 IF CB(ROW,COL)=2 THEN PRINT " RD "; : GOTO 1140
1120 IF CB(ROW,COL)=3 THEN PRINT " BK "; : GOTO 1140
1130 IF CB(ROW,COL)=4 THEN PRINT " RK ";
1140 NEXT COL
1150 PRINT
1160 NEXT ROW
1170 RETURN
```

This is similar to the subroutine that drew the tic-tac-toe board, but more complicated. Once again, we print out the column numbers at the top of each column and the row numbers at the start of each row; adjust the spacing, if necessary. Lines 1080 and 1090 print out the red and black symbols for the unoccupied squares. The trickiest task performed by this subroutine is figuring out which squares are red and which squares are black by their row and column numbers. This is accomplished in line 1080, by adding together the values of ROW and COLUMN. If the sum of these two variables is odd, then the square is red. If the sum is

even, then the square is black. But how do we tell the difference between an odd and even sum? The simplest way is to divide the sum by 2, chop off any fractional value with the INT function, and multiply the result by 2. If the result of these operation is equal to the original sum, then the sum was even. If it is not equal to the original sum, then the sum was odd. Line 1080 does all of this to see if the square is red, printing the red symbol if it is. If it is not red, and there is no piece on the square, line 1090 assumes it is black and prints the black symbol. If the square is occupied, lines 1100 through 1130 print the symbol for the piece that is on the square. Notice that line 1080 does not check to see if the square is unoccupied before printing the red symbol, since red squares can *never* be occupied.

We'll call this subroutine to draw the initial picture of the board, like this:

```
100 GOSUB 1000 : REM DRAW CHECKERBOARD
```

Finally, it's time for the game to begin. The following instructions will be executed for every move in the game, until someone wins. First, we prompt the current player (CP) to make a move:

```
110 PRINT : PRINT "PLAYER";CP;"- IT'S YOUR MOVE"
```

The extra PRINT statement adds a space between the picture of the board and the prompt, for neatness.

Next, we ask the player which piece he or she wishes to move:

```
120 PRINT "WHICH PIECE DO YOU WISH TO MOVE?"
```

The instructions that input the row and column numbers of the piece will be used a couple of times, so we'll put them in a subroutine, like this:

```
2000 PRINT "ROW"; : INPUT ROW
2010 IF ROW<1 OR ROW>8 THEN PRINT "NO SUCH ROW. TRY
AGAIN" : GOTO 2000
2020 PRINT "COLUMN"; : INPUT COL
2030 IF COL<1 OR COL>8 THEN PRINT "NO SUCH COLUMN. TRY
AGAIN" : GOTO 2020
2040 RETURN
```

This subroutine prompts the user to type in a ROW number, which is stored in variable ROW, followed by a column number, which is stored in variable COL. (Although we have used three-letter variable names here for clarity, most versions of BASIC will see only the first two letters of each name, so that these will be treated as variables RO and CO.) The value of each variable is checked, to make sure that it falls in the range 1 to 8, and the player is given an error message and a second chance to type the number if it does not.

We'll call this subroutine like this:

```
130 GOSUB 2000 : REM GET ROW AND COLUMN NUMBERS
```

We'll be calling the subroutine again in a second, which will change the values of variables ROW and COL. We'll need to save the values of these variables in two other variables, like this:

```
140 R1=ROW : C1=COL
```

Variables R1 and C1 now contain the row and column numbers of the square that holds the piece the current player wants to move. We'll refer to this square as the "source square," because it is the "source" of the piece to be moved. What if the player doesn't have a piece on the source square? That's easy enough to find out, by checking to see if CB(R1,C1) is equal to CP (the current player's identifying number). However, at some later point in the game player CP may also have some kings on the board—that is,

pieces that have made it all the way across the board and gained the power to move in all directions. Kings have an identifying number that is 2 greater than the player's usual identifying number; hence, we also have to check to see if CB(R1,C1) is equal to CP+2. Line 150 does both:

```
150 IF CB(R1,C1)=CP OR CB(R1,C1)=CP+2 THEN 170
160 PRINT "YOU DON'T HAVE A PIECE THERE!" : GOSUB 1000 :
GOTO 120
```

If the source square is not occupied by the current player, we redraw the board (in case the last picture of the board has scrolled off the top of the screen) and repeat the lines that asked the current player which piece to move.

Once we get a good value for R1 and C1, we must ask which square the current player wishes to move to:

```
170 PRINT "WHERE DO YOU WISH TO MOVE THE PIECE?"
```

Once again, we call the subroutine at line 2000 to input the row and column numbers:

```
180 GOSUB 2000 : REM GET ROW AND COLUMN NUMBERS
```

This time, we'll transfer these numbers to variables R2 and C2:

```
190 R2=ROW : C2=COL
```

Variables R2 and C2 now contain the row and column numbers of the square the current player wishes to move to. We'll refer to this square as the "destination square."

At this point, we have a complete description of the move that the current player wishes to make. The piece at R1,C1 is to be moved to R2,C2. Now we begin the most complex part of the program: checking to see if this is a legal move, within the rules of

checkers. And what are those rules? Well, the easiest one, in terms of checking for a legal move, is that the destination square may not already be occupied by a piece. We can check for that possibility like this:

```
200 IF CB(R2,C2)<>0 THEN PRINT "THAT SQUARE IS OCCUPIED" :
GOSUB 2000 : GOTO 170
```

If CB(R2,C2) is not equal to 0, an error message is printed, the board is printed out again, and the player is asked a second time for the row and column numbers of the destination square.

Now things get tougher. The next rule to consider is that we must, under most circumstances, move our piece one square diagonally and one square horizontally. Thus, the row and column numbers of the destination square must differ from the row and column numbers of the source square by exactly 1.

We can test for this by subtracting R2 from R1 and C2 from C1 and checking to see if the result of each subtraction is either 1 or −1 ("negative one"). To simplify this operation, we will use the BASIC ABS, or "absolute value," function. The ABS function will change the number −1 to the number 1, but will leave the number 1 unchanged. Thus, ABS (−1) will be equal to 1 and so will ABS(1). Here is how we check:

```
230 IF ABS(C1−C2)<>1 THEN 270
240 IF ABS(R1−R2)<>1 THEN 270
```

If the difference in either the row or column numbers is not equal to 1 or −1, we jump to line 270. What happens on line 270? There is one exception to the rule that we must always move one square horizontally and one square vertically. If we are capturing one of our opponent's pieces, we must jump over that piece diagonally, and thus move our piece *two* squares horizontally and vertically. Line 270 will check for this possibility. We'll look at line 270 in a minute.

Meanwhile, if the requested move passes the first test, we must also check to see if the piece was moved in the proper direction—that is, in the direction of our opponent's side of the board. If we are playing black, this means that we must move from the low-numbered rows to the high-numbered rows. If we are playing red, we must move from the high-numbered rows to the low-numbered rows. We can check to see which direction we have moved by subtracting the source row (R1) from the destination row (R2). For black (player 1), the result should be 1. For red (player 2), the result should be −1. Since we will be performing this particular check a couple of times (as you will see), we will first set up a variable called LM ("legal move") that will hold the proper result for the current player (CP), like this:

```
210 IF CP=1 THEN LM=1 : KR=8
220 IF CP=2 THEN LM=-1 : KR=1
```

If the current player is 1 (black), then LM is 1. If the current player is 2 (red), then LM is 2. And what is this variable KR that we have also created in this line? It stands for "king row." It is the row that the current player must reach in order for a piece to become a king. We will see in a moment how this variable is used.

Here is how we check to see if our move was legal:

```
250 IF (R2-R1<>LM) AND (CB(R1,C1<3) THEN 460
```

The first part of this statement checks to see if R2−R1 is equal to LM. However, we must also check to make sure that the piece is not a king, in which case it may move in *either* direction. This check is made in the second part of the statement. Since kings are numbered 3 and 4, we simply check to see if the piece on the source square has a number less than 3.

If the move is illegal, we jump to line 460, which will announce that the move is not allowed and then repeat the sequence of instructions prompting the current player to input a

move. If the move is legal, we must jump ahead to the routine that makes the move, like this:

```
260 GOTO 330 : REM MAKE THE MOVE
```

Now, we must consider those moves that failed the test in lines 230 and 240, because they moved more than one square. As we said before, the only case in which a player may move more than one square either horizontally or vertically is when an opponent's piece is being jumped and captured. In this case, the player's piece may be moved two squares horizontally and two squares vertically. Once again, we will use the ABS function to see if C1−C2 and R1−R2 are equal to either 2 or −2:

```
270 IF ABS(C1−C2)<>2 THEN 460
280 IF ABS(R1−R2)<>2 THEN 460
```

If the move fails either test, we jump ahead to line 460, which tells the player that he or she has made an illegal move.

Next, we must check again to see if the player was moving in a legal direction. As before, we test for the direction by subtracting R1 from R2. But now the result can be either 2 or −2, so we must multiply variable LM by 2:

```
290 IF (R2−R1<>LM*2) AND (CB(R1,C1<3) THEN 460
```

Even if the move passes these tests, we must still make sure that one of the opponent's pieces has been jumped and captured. Therefore, we must check to see what was in the square that the player has jumped over.

But how do we determine the number of the square that was jumped? The square will be halfway between the source square and the destination square. Thus, we can take the difference between R1 and R2, and the difference between C1 and C2,

divide each by 2, and add the results to the row and columns numbers of the source square. This gives us the number of the square that was jumped. Then we must check this square to see if it contains the opponent's number (OP), like this:

```
300 IF CB(R1+(R2-R1)/2,C1+(C2-C1)/2)<>OP THEN 460
```

If not, we jump to line 460, to declare an illegal move. Otherwise, we tell the player that he or she has captured a piece:

```
310 PRINT "YOU HAVE CAPTURED A PIECE!"
```

And we must remove the piece from the square that was jumped, by setting that square equal to 0:

```
320 CB(R1+(R2-R1)/2,C1+(C2-C1)/2)=0
```

As we mentioned earlier, the routine that actually moves the piece begins on line 330:

```
330 CB(R2,C2)=CB(R1,C1)
340 CB(R1,C1)=0
```

These lines set the destination square equal to the number of the piece on the source square, then set the source square equal to 0.

If the current player's piece has reached the opposite side of the board for the first time on this move, we must make it into a king. Here we check to see if we have reached the "king row," the number of which is contained in variable KR:

```
350 IF (R2<>KR) OR (CB(R2,C2)>2) THEN 370
360 PRINT "YOU HAVE GAINED A KING!" :
CB(R2,C2)=CB(R2,C2)+2
```

Notice that line 350 not only checks to see if the destination square is on the king row, but also checks to see if the piece on that square is *already* a king (that is, if it has a value greater than 2). If we are not on the king row, or if the piece is already a king, we skip line 360, which announces the conversion of the piece to a king and increases the piece's value by 2, to indicate its new status.

As a last order of business before we go on to the next player's move, we must check to see if we have a winner yet. Obviously, this won't happen on the first time through this loop, but we will check for it after every move just to be safe:

```
370 FL=0
380 FOR ROW=1 TO 8
390 FOR COL=1 TO 8
400 IF CB(ROW,COL)=OP THEN FL=1
410 NEXT COL
420 NEXT ROW
430 IF FL=0 THEN 470
```

This routine first sets variable FL ("flag") equal to 0. Then it loops through all eight rows and all eight columns, checking every square to see if the current player's opponent has a piece on any square. If one of the opponent's pieces is found, FL is set equal to 1. If FL is still equal to 0 after these loops have executed, then the opponent has no pieces left on the board and the current player has won.

We jump to the routine on line 470 to congratulate the winner and end the game. Notice that we don't check to see if the current player has any pieces on the board, since it is impossible for a player to lose while it is his or her turn to move.

Assuming that no winner is found, we must turn the game over to the other player. Lines 440 and 450 swap the values of CP and OP and loop back to the beginning of the main game loop at line 100:

```
440 IF CP=1 THEN CP=2 : OP=1 : GOTO 100
450 CP=1 : OP=2 : GOTO 100
```

We've made several references to line 460, which tells the player that he or she has made an illegal move and repeats the instructions that prompt for the move. Here is line 460:

```
460 PRINT "ILLEGAL MOVE" : GOSUB 1000 : GOTO 120
```

The GOSUB 1000 instruction reprints the game board.

We've also made several references to line 470, which begins the routine that congratulates the winner and ends the game. Here is that routine:

```
470 GOSUB 1000
480 PRINT "CONGRATULATIONS, PLAYER";CP
490 PRINT "YOU HAVE WON!"
500 PRINT "WOULD YOU LIKE TO PLAY AGAIN?"
510 PRINT "(1=YES, 2=NO)"
520 INPUT Q
530 IF Q=1 THEN 30
540 IF Q=2 THEN END
550 GOTO 520
```

As in the tic-tac-toe game, we offer the players a chance for another game. If the offer is declined, the program ends. If it is accepted, we loop back to line 30 to begin again. And if the response makes no sense, we ask the question again.

There are lots of improvements you can add to this game. For instance, this program does not allow multiple jumps. In standard checkers, a player that has just jumped an opponent's piece may jump a second piece without waiting for the next turn, if such a jump is available and can be made by the same piece that made the initial jump. That rule is not included in our game, but can be added with some work.

Another feature that is missing here is the "forced jump." Once again, standard checkers rules require that a player make a jump if it is available, even if the jump will place his or her piece in an undesirable position. This rule is not included here, but could be added.

In fact, you could add both of these features with the addition of a single subroutine. What would this subroutine do? It would look for jumps that the current player could make. First, it would scan the board looking for the current player's pieces. Then, when it found a square containing such a piece, it would scan the squares attached diagonally to that square, to see if the opposing player has pieces on those squares. Finally, it would scan the squares diagonally past the opposing player's pieces to see if they are empty, which would mean that the opposing player's piece could be jumped. If a possible jump is found, this information would be recorded—perhaps by setting a flag variable (such as the FL variable we use to detect a winning move) equal to a special code number.

This subroutine could be called to see if the current player should be forced to make a jump. If a jump (or more than one jump) is available, the program would not accept any move that isn't a jump.

After a jump is made, the subroutine could be called a second time to see if a multiple jump is possible. If so, the player would be called on to make the jump—or the program could make the jump automatically. However, if more than one jump is available, the player would have to choose which jump to make.

And, of course, you could also add graphics to this program—and to any of the programs in this book. Since the drawing of the checkerboard is done entirely by the subroutine at line 1000, you would need only to replace this subroutine with a subroutine of your own, which would use your computer's graphics capabilities to draw a picture of the board. Of course, you would also need to rewrite all statements that PRINT sentences on the screen, so that they don't ruin your neatly planned graphics display.

Here is the complete program:

```
20 DIM CB(8,8)
30 CP=1 : OP=2
40 FOR ROW=1 TO 8
50 FOR COL=1 TO 8
60 READ A
70 CB(ROW,COL)=A
80 NEXT COL
90 NEXT ROW
100 GOSUB 1000
110 PRINT : PRINT "PLAYER";CP;"- IT'S YOUR MOVE"
120 PRINT "WHICH PIECE DO YOU WISH TO MOVE?"
130 GOSUB 2000
140 R1=ROW : C1=COL
150 IF CB(R1,C1)=CP OR CB(R1,C1)=CP+2 THEN 170
160 PRINT "YOU DON'T HAVE A PIECE THERE!" : GOSUB 1000 :
GOTO 120
170 PRINT "WHERE DO YOU WISH TO MOVE THE PIECE?"
180 GOSUB 2000
190 R2=ROW : C2=COL
200 IF CB(R2,C2)<>0 THEN PRINT "THAT SQUARE IS OCCUPIED" :
GOSUB 2000 : GOTO 170
210 IF CP=1 THEN LM=1 : KR=8
220 IF CP=2 THEN LM=-1 : KR=1
230 IF ABS(C1-C2)<>1 THEN 270
240 IF ABS(R1-R2)<>1 THEN 270
250 IF (R2-R1<>LM) AND (CB(R1,C1)<3) THEN 460
260 GOTO 330
270 IF ABS(C1-C2)<>2 THEN 460
280 IF ABS(R1-R2)<>2 THEN 460
290 IF (R2-R1<>LM*2) AND (CB(R1,C1)<3) THEN 460
300 IF CB(R1+(R2-R1)/2,C1+(C2-C1)/2)<>OP THEN 460
310 PRINT "YOU HAVE CAPTURED A PIECE!"
320 CB(R1+(R2-R1)/2,C1+(C2-C1)/2)=0
```

```
330 CB(R2,C2)=CB(R1,C1)
340 CB(R1,C1)=0
350 IF (R2<>KR) OR (CB(R2,C2)>2) THEN 370
360 PRINT "YOU HAVE GAINED A KING!" :
CB(R2,C2)=CB(R2,C2)+2
370 FL=0
380 FOR ROW=1 TO 8
390 FOR COL=1 TO 8
400 IF CB(ROW,COL)=OP THEN FL=1
410 NEXT COL
420 NEXT ROW
430 IF FL=0 THEN 470
440 IF CP=1 THEN CP=2 : OP=1 : GOTO 100
450 CP=1 : OP=2 : GOTO 100
460 PRINT "ILLEGAL MOVE" : GOSUB 1000 : GOTO 120
470 GOSUB 1000
480 PRINT "CONGRATULATIONS, PLAYER";CP
490 PRINT "YOU HAVE WON!"
500 PRINT "WOULD YOU LIKE TO PLAY AGAIN?"
510 PRINT "(1=YES, 2=NO)"
520 INPUT Q
530 IF Q=1 THEN 30
540 IF Q=2 THEN END
550 GOTO 520
1000 PRINT " ";
1010 FOR COL=1 TO 8
1020 PRINT COL;" ";
1030 NEXT
1040 PRINT
1050 FOR ROW=1 TO 8
1060 PRINT ROW;
1070 FOR COL=1 TO 8
1080 IF INT((ROW+COL)/2)*2=(ROW+COL) THEN PRINT " ==";:
GOTO 1140
1090 IF CB(ROW,COL)=0 THEN PRINT " -- "; : GOTO 1140
```

```
1100 IF CB(ROW,COL)=1 THEN PRINT "BL "; : GOTO 1140
1110 IF CB(ROW,COL)=2 THEN PRINT "RD "; : GOTO 1140
1120 IF CB(ROW,COL)=3 THEN PRINT "BK "; : GOTO 1140
1130 IF CB(ROW,COL)=4 THEN PRINT "RK ";
1140 NEXT COL
1150 PRINT
1160 NEXT ROW
1170 RETURN
2000 PRINT "ROW"; : INPUT ROW
2010 IF ROW<1 OR ROW>8 THEN PRINT "NO SUCH ROW. TRY
AGAIN" : GOTO 2000
2020 PRINT "COLUMN"; : INPUT COL
2030 IF COL<1 OR COL>8 THEN PRINT "NO SUCH COLUMN. TRY
AGAIN" : GOTO 2020
2040 RETURN
3000 DATA 0,1,0,1,0,1,0,1
3010 DATA 1,0,1,0,1,0,1,0
3020 DATA 0,1,0,1,0,1,0,1
3030 DATA 0,0,0,0,0,0,0,0
3040 DATA 0,0,0,0,0,0,0,0
3050 DATA 2,0,2,0,2,0,2,0
3060 DATA 0,2,0,2,0,2,0,2
3070 DATA 2,0,2,0,2,0,2,0
```

CHAPTER SIX

ALL THE WORLD'S A GAME

Once you know how data structures such as arrays can be used to simulate decks of cards and game boards, well, the sky's the limit. Just think of the other games you could write programs for.

The same array that we used to create a checkerboard—CB(8,8)—could represent a chessboard, except that a much larger set of numbers would be needed to represent the great variety of pieces in chess.

And there's no reason why you should stop with a chessboard. You could use a two-dimensional array to represent a map of a country, or an entire planet, as part of a political or war game. Each element of the array could be assigned a number indicating what kind of terrain is found at that point on the map. The number 1 could represent grassland, the number 2 could represent arid desert, 3 could represent ocean, 4 could represent mountains, and special numbers could represent major cities, such as New York, Washington, London, Paris, or Moscow—assuming that the planet is Earth.

A three-dimensional array could represent a portion of outer space—or a scaled-down model of the entire universe,—with the subscripts indicating how far up, down, left, right, or sideways each element is from some starting point. Elements could be assigned numbers to indicate whether they represent stars, planets, or just empty space. Players could move through the universe simply by changing elements in the array.

Of course, it would take an imaginative programmer to devise pictures to do justice to such an array. You could fill the video display with stars and planets and spaceships, to show the player what kind of universe the array represents. And that's what computer game simulations are all about—converting your imagination into numbers inside the computer, then converting those numbers back into something that a person can sit down at the computer and play with.

In this book, we've given you a starting point for creating simulations of your own. Now, go ahead and think of new games to simulate, and write the programs yourself.

Even the sky isn't the limit!

INDEX

ABOUT THE AUTHOR

Christopher Lampton's interests and knowledge extend to many areas of science. He is the author of sixteen computer books for Franklin Watts, including the eleven-book series Computer Literacy Skills. Among his other published works for Watts are *Dinosaurs and the Age of Reptiles, Fusion: The Eternal Flame,* and *Meteorology: An Introduction.* Mr. Lampton has just finished a book on mass extinctions—the latest theory on why the dinosaurs disappeared.

Christopher Lampton lives outside of Washington, D.C., with three computers and stacks and stacks of books.