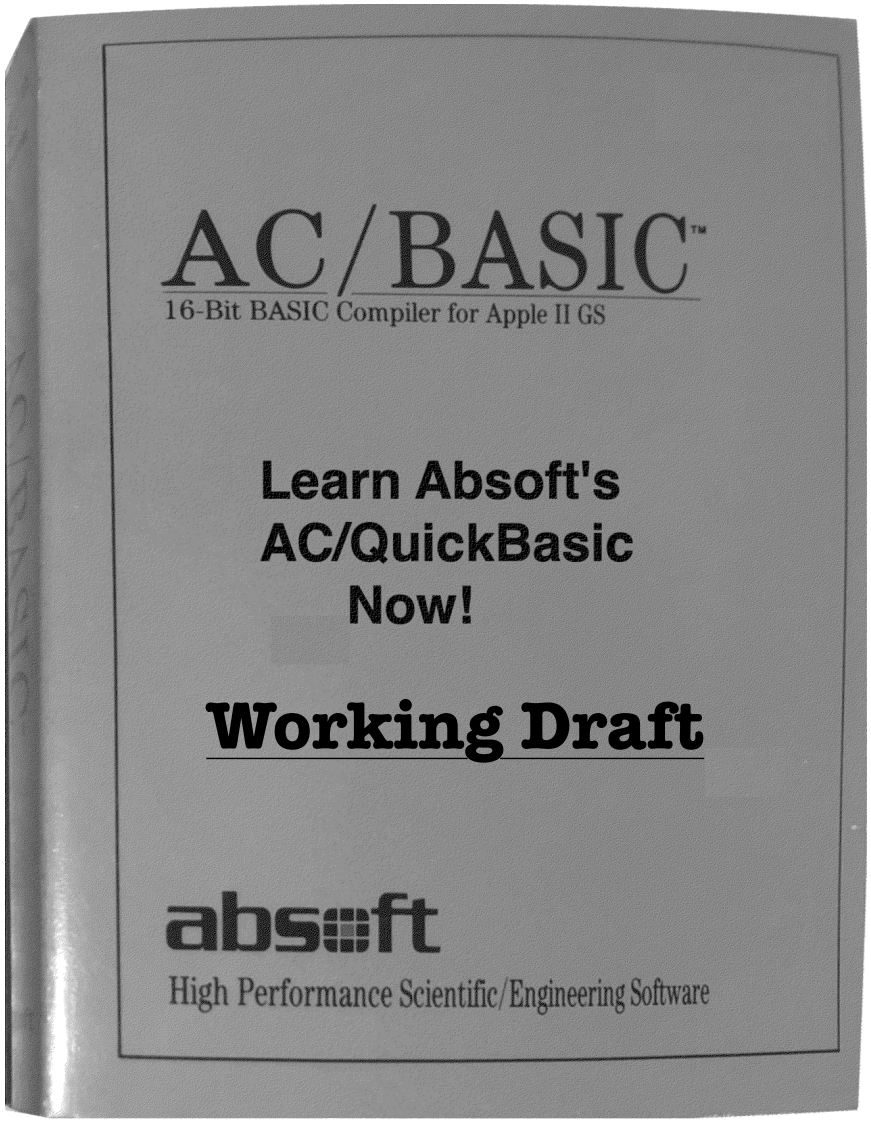


This is a COPY/PASTE/EDIT of
Michael Halvorson & David Rygmyr Book
ISBN 1-55615-314-7

Called

Learn Basic For The Macintosh NOW.

Dozens of Sample Programs
The Fastest, Easiest Way to Explore
Absoft's AC/QuickBasic Compiler.

WHY ?
Because I have not been able to Get
the Absoft Apple][GS
Reference/User's Manual.

If you have it! PLEASE, Share It.

Introduction to the BASIC Language. Learn AC/QuickBasic For the Apple][gs NOW

ANATOMY OF A BASIC INSTRUCTION
Each line in a AC/QuickBasic program is simply an instruction that the computer carries out when you run the program.
AC/QuickBasic recognizes two types of instructions: statements and functions.

Statements and functions look very much alike and are equally easy to use. They differ primarily in purpose:

• A statement is usually straightforward.
  It simply does what you tell it to do when you run the program.
  When your program executes a statement, the result is usually apparent.
  The PRINT statement, for example, puts characters in the Output window,
  And the CLS statement clears the Output window.

• A function returns a value that your program can use.
  A function usually works in a less obvious way than a statement does.
  A function appears within a statement and performs its work when the statement is executed.

Each statement and function has a syntax. The syntax dictates the rules for writing the statement or function.

AC/QuickBasic Syntax
The syntax of an AC/QuickBasic instruction, whether a statement or a function.
This is simply a keyword followed by the information that the instruction needs to do its work. The syntax of some
AC/QuickBasic instructions, such as the BEEP statement, consists of only the name of the statement or function.

 Start the AC/QuickBasic Interpreter/Compiler (if you haven't already done so),
 Now Type in and compile the following single-line program:

```
    BEEP
```

Compile and Run the program and see what happens.
The BEEP statement causes the speaker inside your computer to emit a brief sound.
 (The sound you hear depends on your system configuration.)

NOTE: If you don't hear a sound, check the Control Panel Volume Setting.
      Use: Control-Option-Command-Esc to end the CDA Menu.
Choose the Control Panel, then the Sound Option. Be sure that the Volume & Pitch slider is set to a value of 4 or higher.

Now let's take a look at some instructions of greater substance.
Throughout this, we'll work with variations of the PRINT statement. Here's the syntax for a PRINT statement:

PRINT [expressionlist][,|;]
The PRINT statement is quite versatile; The PRINT statement has many options.
Let's take a moment to become familiar with the structure of the PRINT (or any) syntax line, as shown in Figure 3-1.

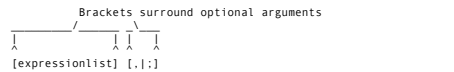


FIGURE [1-1]3-1. Structure of a syntax line.

Each item in the syntax line except the keyword is called an argument.
Each argument provides the instruction with additional information it needs in order to do its work.
Some arguments are optional; in this document optional arguments appear within square brackets.
If you decide to use an optional argument in an instruction, don't type in the square brackets.
Along the same line, don't type in the | symbol - it simply indicates that you are to choose one of the arguments on
either side of the |. In the statement shown in Figure [1-1]3-1, you would type in either the comma or the semicolon.

In a PRINT statement, both arguments are optional: You can use the PRINT keyword by itself.

Practice:
Using a PRINT statement without arguments
1. Pull down the File menu and choose the New command, enter a name for your 'SampleProgram' to enter the Editor.
2. Type in the following program.

```
     CLS

     PRINT "This statement uses an argument."
     PRINT
     PRINT "So does this one."
```

3. SAVE/COMPILE and then RUN the program. Your Output window will look like this:

```
       This statement uses an argument.
       So does this one.
```

As you can see, the second PRINT statement, the one with no arguments, prints a blank line.

Now let's look at three types of arguments you can use with a PRINT statement:
Text, Numeric Expressions, and Functions.

NOTE: We'll rely on CLS in our programs later in the book, and using CLS is a good habit to cultivate.

Use of

```
     CLS
```

Ensures that a program will start with a "clean slate" when it runs.

Using TEXT as an argument
In this chapter, you will use text enclosed by quotation marks ("Live long and prosper.") as an expression-list argument
to the PRINT statement.

Such text is called a string.

You can use any numbers, letters, spaces, and punctuation marks - except double quotation marks - in a string.
AC/QuickBasic perceives a double quotation mark within a string as the end of the string.

Using numeric expressions as arguments Another valid expression-list argument is a numeric expression.
A numeric expression is a number, a mathematical equation, or, as you will learn, a special kind of word called
a numeric variable. A numeric expression is not enclosed in double quotation marks.

Practice:
Using numeric expressions as arguments
1. Modify your program so that it contains only the following PRINT statement:

```
CLS
PRINT 42
```

2. Now compile and run the program. Your Output window will look like this:

```
  42
```

Using functions as arguments A third valid expression-list argument is a AC/QuickBasic function.
A function performs a background task and returns information that your program can use.
A function is not enclosed in double quotation marks. Two AC/QuickBasic functions, named DATE$ and TIME$,
Obtains the current date and time from the clock that is built into your computer.
You can not use these functions by themselves. In other words, You can not simply put DATE$ or TIME$ on a line and expect
something to happen. You can, however, use DATE$ or TIME$ as an argument for a PRINT statement.

1. Pull down the File menu and choose the New command to enter the editor.
2. Type in and compile the following program, then run it:

```
      CLS
     PRINT DATE$
     PRINT TIME$
```

Your Output window will look something like this:

```
     06-18-1991
     21:13:09
```

Although the date and time displayed here are unlikely to match what you see in your Output window,
the underlying principle applies: Combined with a keyword such as PRINT, the DATE$ and TIME$ functions perform their
tasks and return values in an unobtrusive way. That's the beauty of AC/QuickBasic functions:
You simply use them, and they perform useful work without your having to worry about the details.

Printing More than One Item with PRINT
So far, you've used no more than a single argument, such as a single string or a single function, with each PRINT
statement. You can use multiple arguments in a single PRINT statement, but only if you use a special character called a
separator between the arguments. The PRINT statement recognizes two separator characters: the comma and the semicolon.
The comma separator: Think of the comma separator as the programming equivalent of the Tab key.
When PRINT encounters a comma, it prints the value of the next argument at the beginning of the next print zone,
the programming equivalent of a tab stop. A print zone is 14 standard characters long. The ability to print values at
specific locations makes comma separators a natural choice when you want to print information in columns.

Practice:
Using comma separators
1. Pull down the File menu and choose the New command to enter the editor.
2. Type in, compile and run the following program to see how comma separators work:

```
      CLS
      PRINT "Comma", "separators", "separate", "arguments"
      PRINT "They", "also", "align", "arguments"
```

Your Output window looks like this:

```
    Comma      separators    separate      arguments
    They       also          align         arguments
```

You can even use commas by themselves with no argument between them, as shown in the following practice session.

Practice:
Positioning arguments with comma separators
1. Pull down the File menu and choose the New command to enter the editor.
2. Type in, compile and run the following program, which uses commas to position values near the center of the screen:

```
     CLS
     PRINT , , "Use comma separators"
     PRINT , , "to position arguments"
```

Your Output window looks like this:

```
                                  Use comma separators
                                  to position arguments
```

The semicolon separator
When PRINT encounters a semicolon separator, it prints the next argument immediately after the argument it just printed. PRINT places no spaces between arguments separated by a semicolon: You must supply the spaces within the arguments if you want them.

Practice:
Using semicolon separators
1. Pull down the File menu and choose the New command to enter the editor.
2. Type in, compile and run the following program, which uses semicolon separators both with and without added spaces:
```
     CLS
     PRINT "This"; "is"; "what"; "semicolons"; "do"
     PRINT "If "; "you "; "want "; "spaces, "; "add " "them"
```
Your Output window looks like this:
```
                              Thisiswhatsemicolonsdo
                              If you want spaces, add them
```

Using both comma and semicolon separators
Commas and semicolons aren't mutually exclusive - you can use more than one kind of separator on a line.
You can mix commas and semicolons however you like, and they always work as advertised.

Practice:
Using both comma and semicolon separators
1. Pull down the File menu and choose the New command to enter the editor.
2. Type in, compile and run the following program, which uses both commas and semicolons on the same line:
```
     CLS
     PRINT "These"; "words"; "run"; "together", "These", "are", "apart"
```
Your Output window looks like this:
```
                              Thesewordsruntogether             These      are     apart
```
Using a separator at the end of a PRINT statement
When you put a comma or semicolon separator at the end of a PRINT statement,
  You dictate where the result of the subsequent PRINT statement appears in the Output window:
• If you put a comma at the end of a PRINT statement,
  The output of the next PRINT statement appears at the beginning of the next print zone of the first PRINT statement's output line.
• If you put a semicolon at the end of a PRINT statement,
  The output of the next PRINT statement immediately follows the output of the first PRINT statement.

Practice:
Using separators at the ends of PRINT statements
1. Pull down the File menu and choose the New command to enter the editor.
2. Type in, compile and run the following program, which uses a semicolon at the end of the first PRINT statement:
```
     CLS
     PRINT "This is the first line ";
     PRINT "and this is the second"
```
Your Output window will look like this:
```
                              This is the first line and this is the second
```
3. Change the semicolon to a comma and run the program again.
   NOTE: If you use a semicolon at the end of a PRINT statement and no PRINT statements follow, the semicolon has no effect.

SUMMARY
In this chapter, you learned more about AC/QuickBasic program instructions and learned how various syntax options make a AC/QuickBasic statement more versatile. In the next chapter, you'll learn about two important AC/QuickBasic elements that will add even more flexibility to your programs: variables and operators.

QUESTIONS AND EXERCISES
1. Briefly describe the primary difference between a statement and a function.
2. Which of the following AC/QuickBasic keywords are statements and which are functions?
```
     BEEP   CLS   DATE$   PRINT   TIME$
```
3. In a AC/QuickBasic syntax line, what is the significance of square brackets on either side of an item?
   What does the : character signify?
4. What is an argument?
5. What is the difference between a string and a numeric expression?
6. How will the output from the following two PRINT statements differ?
```
     PRINT "Hello", ; "there"
     PRINT "Hello"; , "there"
```
7. What happens when you put a semicolon or a comma at the end of a PRINT statement?

AC/QuickBasic Variables and Operators Learn AC/QuickBasic For the Apple][gs NOW

As you program, you might need to print certain numbers or strings of characters more than once. AC/QuickBasic provides a handy method for doing just that - a method that doesn't require a lot of retyping on your part.

In AC/QuickBasic, you can store data and use it whenever and as often as you like, simply by using the name of the storage location in your program.

You name the storage location, which is known as a variable, according to the type and size of data it contains.

WHAT DO YOU WANT TO STORE?
Variables are of two main types: string and numeric. A string variable is a name representing a storage location for a string of text. A numeric variable is a name representing a storage location for a number. The numeric variable type has several subtypes. Figure 4-1 shows the valid variable types in AC/QuickBasic.

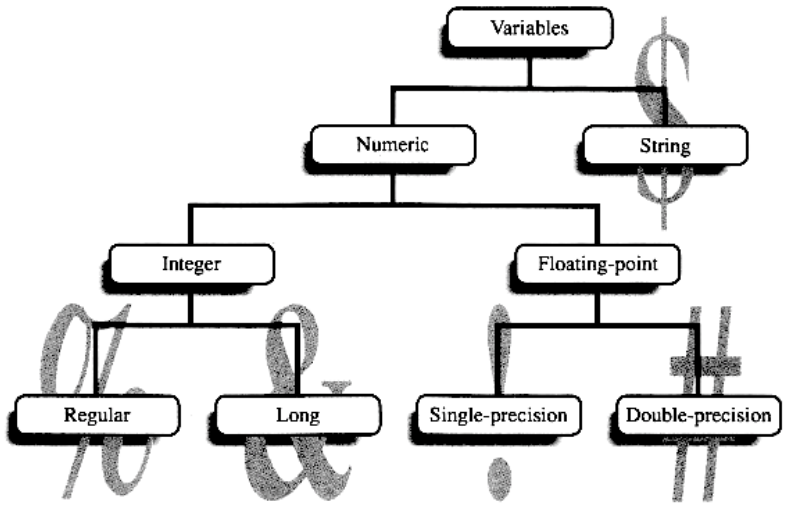


FIGURE 4-1. The variable types used in AC/QuickBasic.

As you read through this chapter, keep the following questions in mind. As you'll see, the answers to these questions help you determine which type of variable is most suited to your purposes.

• Is your data text or a number? Text is stored in a string variable; a number is stored in a numeric variable.

• Does your numeric data contain decimal points?
  An integer- that is, a whole number with no decimal point- is stored in an integer variable.
  A number with a decimal point is stored in a floating-point variable.

• How "large" (in two senses) is the number you want to store?
  The size of the number plays a part when you determine which variable type to use.
  A small whole number is stored in a regular integer variable;
  A large whole number is stored in a long integer variable.
  A decimal number containing relatively few digits is stored in a single-precision floating-point variable;
  A decimal number containing many digits is stored in a double-precision floating-point variable.

USING VARIABLES - AN OVERVIEW
When you decide to use a variable,
You must first declare it in your program; that is, you must inform AC/QuickBasic of three things:

• The variable's name
• The variable's type
• The variable's value

You can see these elements of a variable declaration in Figure 4-2,which shows an example string variable declaration.

```
animals$= "lions and tigers and bears"
   \   \_________ \_____________________
Variable Name    \Type-Declaration Character    \Variable Value
```

FIGURE 4-2. The elements of a variable declaration.

The following sections describe how to use each of these elements.

Naming a Variable
These rules and suggestions will help you choose appropriate names for your AC/QuickBasic variables:

• A variable name can be as many as 40 characters in length.

• A variable name can be any combination of uppercase and lower-case letters.
Keep in mind, though, that all-uppercase names might make it hard to distinguish variables from AC/QuickBasic keywords when you or someone else looks at your program later.

• You can't use AC/QuickBasic keywords, such as PRINT and BEEP, as variable names.

• The last character of the variable name must be the appropriate type-declaration character.
(See "Declaring the Variable Type.")

• The most descriptive variable names are the most useful.
The string variable names firstName$ and lastName$,
for example, leave little doubt in your mind about what data these variable names represent.

NOTE: The AC/QuickBasic Interpreter/Compiler is sensitive to your use of uppercase, lowercase, and mixed case letters. So sensitive, in fact, that if you type in a variable name you've already used but in a different case, AC/QuickBasic adjusts the variable name you've already typed to reflect the new case.

Declaring the Variable Type
If you look again at Figure 4-2, you'll notice that the final character of the variable name is the type-declaration character. The type-declaration character tells AC/QuickBasic what type of variable you are declaring. Each type has its own symbol. When AC/QuickBasic encounters the type-declaration character at the end of a variable name, it knows what kind of data is stored in that variable.

The table shows the variable types and their type-declaration characters.

Variable type	Type-declaration character
String	$
Integer	
Regular	%
Long	&
Floating-point	
Single-precision	!
Double-precision	#

Declaring the Value of a Variable
When you declare the value of a variable, be sure that the type-declaration character and the value type match. An integer variable(%), for example, can't hold a string value ($).

What Next?

Now that you've been introduced to the structure of a variable declaration, you can begin to learn about the different types of variables themselves.

First you will learn how to declare string variables and the various types of numeric variables. Then you will learn how to get information from the keyboard and store it in variables. The final part of this chapter describes how AC/QuickBasic works with numeric variables and mathematical operators.

STRING VARIABLES IN AC/QuickBasic
A string variable name represents a storage location for a string of text that you want to use throughout a program.

To declare a string variable, use the dollar sign type-declaration character ($) at the end of the variable name. Enclose the string in double quotation marks exactly as you would a string in a PRINT statement; for example,

```
beatAuthors$ ="Kerouac, Ginsberg, Ferlinghetti"
```

The size and content of a string variable can change within a program, as you'll learn in Chapter 9.

Practice:
Declaring and using a string variable
1. Choose the New command from the File menu to enter the editor.
2. Type in, compile and run the following program, which assigns a value to a string variable and then prints the string:

```
CLS
tvDocs$ = "Welby, Casey, Kildare, McCoy"
PRINT tvDocs$
```


Your Output window should look like this:

```
Welby, Casey, Kildare, McCoy
```

AC/QuickBasic uses two main types of numeric variables: integer numeric variables and floating-point numeric variables. Each main numeric variable type has subtypes designed for numbers of different sizes. As we mentioned earlier, the size of the number helps you determine which type of variable to use. Figure 4-3 shows the relationships among the numeric variable types in AC/QuickBasic.

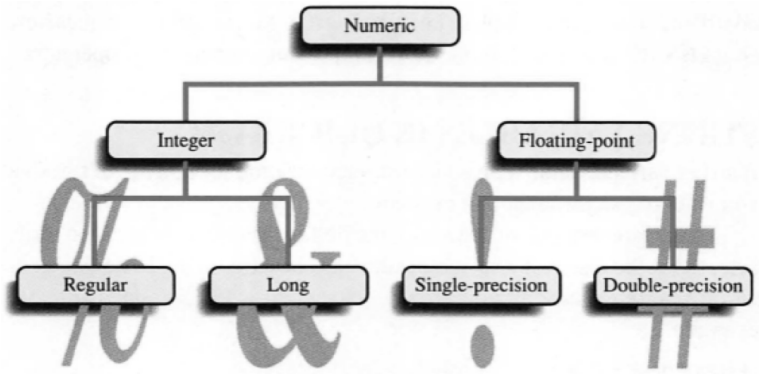


FIGURE 4-3. (Numeric variables in AC/QuickBasic.)
We'll look first at the two types of integer variables and when to use them; then we'll look at the two types of floating-point variables.

Integer Variables
AC/QuickBasic uses two types of integer variables: regular integer variables and long integer variables. The difference between the two lies in the size of the integer number each can represent.

Regular integer variables
A regular integer variable can store any whole number from -32,768 through 32,767.

To declare a regular integer variable, use the percentage sign type-declaration character(%) at the end of the variable name; for example,

```
henrysWives% = 6
```

Declaring and using a regular integer variable

1. Choose the New command from the File menu to enter the editor.

2. Type in, compile and run the following program, which declares a regular integer variable and prints its value:

```
CLS
daysofXmas% = 12
PRINT daysofXmas%
```

Your Output window should look like this:

```
12
```

How AC/QuickBasic Looks at Numbers
Whether you're balancing your checkbook or preparing reports and proposals, you're likely to spend a little time each day working with numbers. What you probably don't realize is that you are actually working with different types of numbers. For a lot of people, the "proper" names for these numbers-whole numbers, decimal numbers, and so on-are only a dim memory from a grade-school math class. But to AC/QuickBasic, the differences between types of numbers are crucial:

• An integer is a number with no decimal point.
• A floating-point number is a number with a decimal point.

Long integer variables
To represent an integer that is outside the range of a regular integer variable, use a long integer variable.

A long integer variable can represent a whole number from -2,147,483,648 through 2,147,483,647.
To declare a long integer variable, use the ampersand type-declaration character(&) at the end of the variable name; for example,

```
cityPopulation& = 175000
```

An Extra Space for Numbers
When you use the PRINT statement in your AC/QuickBasic program to display numbers, you'll notice that every number is, preceded by an extra space. If the number is negative, the AC/QuickBasic Interpreter/Compiler uses this space to display a minus sign. If the number is positive, the AC/QuickBasic Interpreter/Compiler leaves the space blank.

Practice:
Declaring and using a long integer variable
1. Choose the New command from the File menu to enter the editor.
2. Type in, compile and run the following program, which declares a long integer variable and prints its value:

```
CLS
freefall& = 84700
PRINT "The longest free fall, in feet, was"; freefall&
```

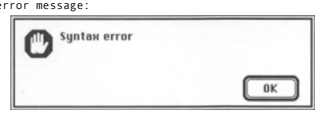
Your Output window should look like this:

```
                    The longest free fall, in feet, was 84700
```

Floating-Point Numbers
So far you've worked only with integers. In real life, you often deal with a floating-point number, which has a fractional part - that is, with a whole number followed by a decimal point and the decimal expression of a fraction.

You Can't Use Commas in Numbers
In everyday life, you use commas to break a large into recognizable segments. The number written 1,653,892,000, for example, is certainly easier to comprehend than the same number written 1653892000. But because commas have a special use as separator characters in AC/QuickBasic, you must learn to get along without them when you enter numbers.

If you forget and try to use a comma as part of a number within your program, the AC/QuickBasic Interpreter/Compiler displays an error message:



In addition, the AC/QuickBasic Interpreter/Compiler draws a box around the comma and the remaining portion of the number to show you the precise location of the error.

Likewise, if the person running your program types in a number with commas, the AC/QuickBasic Interpreter/Compiler asks the user to type the number in again.

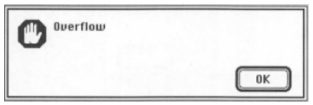
AC/QuickBasic provides two variable types to represent floating-point numbers: single-precision floating-point variables and double-precision floating-point variables. The primary difference between the two variable types lies in how accurately they can represent a given floating-point number.

- A single-precision floating-point variable can represent a number up through 7 digits in length. The decimal point can fall anywhere within those 7 digits.

- A double-precision floating-point variable can represent a number up through 15 digits in length. The decimal point can fall anywhere within those 15 digits.

The table below shows some sample floating-point values.

Using a Variable That's the Wrong Size
As you've seen, working with numbers in AC/QuickBasic requires a lot of attention to the size of the number. If you accidentally assign an out-of-range number to a variable, the AC/QuickBasic Interpreter/Compiler displays an alert box to tell you that the number is invalid for that variable type:



You can help prevent such mistakes by choosing variables of the correct size and displaying the range of valid values for the user.

Single-precision	Double-precision
412.002	1.000032478
19246.34	4280000.0055
.00025	7160.0000000005
657926.3	.10000000001

Single-precision floating-point variables
To declare a single-precision floating-point variable, use the exclamation point type-declaration character (!) at the end of the variable name; for example,

```
lpSpeed! = 33.3333
```

Practice:
Declaring and using a single precision floating-point variable

1. Choose the New command from the File menu to enter the editor.
2. Type in, compile and run the following program, which declares a single-precision floating-point variable and displays its value.

```
CLS
carPrice! = 12999.99
PRINT "This sporty new car costs"; carPrice!
```

Your Output window should look like this:

```
                    This sporty new car costs 12999.99
```

Double-precision floating-point variables
Double-precision floating-point variables, which can be up through 15 digits long, are useful for scientific work that requires precise numbers.

To declare a double-precision floating-point variable, use the pound sign type-declaration character (#) at the end of the variable name; for example,

```
bigFloat# = 5.0000000127
```

Practice:
Declaring and using double-precision floating-point variables

1. Choose the New command from the File menu to enter the editor.
2. Type in, compile and run the following program, which declares a double-precision floating-point variable and displays its value:

```
CLS
pi#= 3.141592653589793
PRINT "The value of pi is"; pi#
```

Your Output window should look like this:

```
                    The value of pi is 3.141592653589793
```

Which Numeric Variable Type Should You Use?
You've learned that AC/QuickBasic provides four types of numeric variables to hold numeric data in your programs: regular integer, long integer, single-precision floating-point, and double-precision floating-point.

When you want to assign a numeric value to a variable in your program, you have to choose the appropriate type of variable. Ask yourself the following questions:

- Is the number an integer? Will the number remain an integer throughout the program? If so, use an integer variable.

- Does the number have a decimal point? Or, more important, might the number have a fractional part later on in the program? If so, use a floating-point variable.

After you've determined the type of variable-integer or floating-point - you have to determine what size variable to use.

Choosing the proper variable size
At first glance you might think that because they can hold large numbers, long integer variables and double-precision floating-point variables are the obvious choices to use at all times. After all, using these larger-size variables would certainly reduce your chances of receiving Overflow error messages.

However (you knew there had to be a catch, right?), these larger variables come with a disadvantage that you need to be aware of. They provide you with greater storage space for your values, but they also take up a lot of memory in your computer. When you use the large variables unnecessarily, it's like using a grain silo to store a five-pound bag of flour. Computer memory, like acreage, is limited. If you build a silo every time you buy a bag of flour, you eventually use up your acreage. Take the time to make wise use of your resources; determine the smallest possible variable type you can use-then use it.

Don't use a larger variable than you need.

Changing the contents of a variable
The word "variable" itself gives you one clue about why variables are so useful: A variable's contents can vary depending on the needs of the program or the needs of the user. You can change the contents of a variable simply by assigning a new value to the variable. Remember, however, that the current value of a variable is the value it was last assigned.

Practice:
Changing the value of a variable

1. Choose the New command from the File menu to enter the editor.
2. Type in, compile and run the following program, which declares and then changes the value of a variable:

```
CLS
fruit$= "apple"
PRINT fruit$
fruit$ = "orange"
PRINT fruit$
fruit$ = "pear"
PRINT fruit$
```

Your Output window should look like this:

```
                        apple
                        orange
                        pear
```

USING USER-SUPPLIED INFORMATION IN A VARIABLE
Now that you've become acquainted with variables, let's look at an area of AC/QuickBasic programming that relies heavily on the use of variables, namely, getting character's from the person using a program. This process, from the program's point of view, is sometimes called "reading characters from the keyboard.''

Reading Characters with the INPUT Statement
The most commonly used AC/QuickBasic statement for reading characters from the keyboard is the INPUT statement.
You can't use an INPUT statement by itself, however. You must follow the INPUT statement with the variable name (including the variable type-declaration character) you want to assign to the characters to be typed by the user.

Practice:
Using the INPUT statement

1. Choose the New command from the File menu to enter the editor.
2. Type in, compile and run the following program:

```
CLS
PRINT "Type in a word and press Return"
INPUT word$
PRINT "The word you typed in was "; word$
```

Your Output window should look like this:

```
    Type a word and press Return
    ? Buckaroo
    The word you typed in was Buckaroo
```

Take a look at what happened:

• First your program cleared the screen, and then it used a PRINT statement to tell the user what to do.

• The INPUT statement displayed a question mark and waited for your response.

• As a user following the PRINT statement's instructions, you typed in a word and pressed Return.
  The INPUT statement assigned the word you typed in to the variable word.$.

• The final PRINT statement used both a string and the word$ variable to show what you typed in.

Asking the user for input
The preceding program used a PRINT statement to ask the user to supply information. You can, however, make the message that asks for user input a part of your INPUT statement. This forces the INPUT statement to do double duty: It both asks the user for input and reads what the user types.

Practice:
Placing a message within INPUT

Change the program you just typed in to this:

```
CLS
INPUT "Type in a word and press Return"; word$
PRINT "The word you typed in was "; word$
```

After you run the program, your Output window will look like this:

```
    Type in a word and press Return? Buckaroo
    The word you typed in was Buckaroo
```

Note the differences between this program and the one you typed in earlier:

• Because the user-friendly request is part of the INPUT statement, the insertion point waits for user input on the same line as the request instead of on a line by itself.

• The INPUT statement displays a question mark at the end of the message. To eliminate the question mark, simply change the semicolon at the end of the INPUT message to a comma and rerun the program.

Practice:
Getting creative with INPUT

Now that you know a little about variables and how to get information from the keyboard using INPUT, you can add a little spice to your programs.

1. Choose the New command from the File menu to enter the editor.
2. Type in, compile and run the following program:

```
CLS
INPUT "Type in your first name and press Return: ", firstName$
PRINT
INPUT "How old are you (I promise not to tell anyone)? ", age%
PRINT
INPUT "How much change do you have in your pocket? $",money!
PRINT
PRINT "Thank you,"; firstName$; ". You said that you are"; age%
PRINT "years old and have$"; money!; "worth of change."
```

After you run the program, your Output window will look similar to this:

```
    Type in your first name and press Return: Pat

    How old are you (I promise not to tell anyone)? 45

    How much change do you have in your pocket? $ 1.16

    Thank you, Pat. You said that you are 45 years old and have $ 1.16 worth of change.
```

AC/QuickBasic MATHEMATICAL OPERATORS
AC/QuickBasic provides several symbols you can use to perform mathematical calculations in your programs. These symbols, called operators, let you perform tasks such as addition, subtraction, multiplication, and division.

Several AC/QuickBasic operators are the same as those you use in every-day life.
For example, in AC/QuickBasic you use + for addition and - for subtraction.
Other AC/QuickBasic operators are represented by special symbols. The following table shows the operators you can use in AC/QuickBasic:

Operator	Mathematical operation
+	Addition
-	Subtraction
*	Multiplication
/	Division
\	Integer division
MOD	Remainder division
^	Exponentiation (raising to a power)

The last three operators (\, MOD, and ") are special-purpose operators that we'll describe shortly.

Working with AC/QuickBasic Operators
To perform calculations in AC/QuickBasic, simply use the operators as you would in "real life."
For example, to add the numbers 12 and 16, you would type

```
    12 + 16
```

You can't just put a mathematical operation like this on a line by itself, however. You must do one of two things:

• Assign the result of the mathematical operation to a numeric variable:

```
  total%= 12 + 16 'Assign the result to a variable
```

Commenting Your Programs
To help you keep track of what your program does, AC/QuickBasic lets you place comments in your programs. Comments are preceded by the REM statement or an apostrophe(') and are for the programmer's use only. Comments do not appear in output produced by the program. Type in, compile and run the following program to see how the REM statement works:

```
REM Sample Program
REM A sample program demonstrating the use of comments.
REM
REM Programmers: Mike and Dave
REM Date: November 1, 1990
CLS
PRINT "This is a program with comments!"
```

The program produces this output:

```
    This is a program with comments!
```

The ' symbol can be used as a less obtrusive equivalent of the REM statement, as shown in the sample program opposite.

• Use the result of the mathematical operation as the argument for a AC/QuickBasic statement:

```
    PRINT 12 + 16 'Print the result directly
```

When you assign a mathematical operation to a variable, you're assigning the result of the operation to the variable, as shown here:

The result is assigned to the variable.

```
          ___________
         /    (28)   \
total% = 12 + 16
```

```
' Sample Program
' A sample program demonstrating the use of comments.
' Programmers: Mike and Dave
' Date: November 1. 1990

CLS

PRINT "This is a program with comments!"
```

When you run this program, it produces the same result as the previous program:

```
    This is a program with comments!
```

We'll use the second commenting style in the programs in this document. As you write your own programs, be sure to include comments not only for yourself but also for others who might need to make sense of your program someday.

When you use a mathematical operation as a statement argument, you're using the result of the operation as the statement argument, as shown here:

```
 ---------\_28 is displayed in the Output window.
 print 12 + 16
 --(28)---\_The result is used as the statement argument.
```

Practice:
Working with AC/QuickBASIC operators

1. Choose the New command from the File menu to enter the editor.
2. Type in, compile and run the following program:

```
PRINT 7 * 7
```

Your Output window will display the result of the PRINT statement:

```
    49
```

1. Choose the New command from the File menu to enter the editor.
2. Type in, compile and run the following program:

```
PRINT 75 / 6
```

The Output window will display the result of the PRINT statement:

```
    12.5
```

The \, MOD, and ^ Operators
AC/QuickBasic has three special-purpose operators: \, MOD, and ^. These operators are useful complements to the more familiar addition, subtraction, multiplication, and division operators.

The \ (integer division) operator
The integer division operator (\) works exactly as the standard division operator(/) does, except that when it divides two numbers it discards any fractional part of the result and returns only the integer part.

Practice:
Using the \ operator

1. Choose the New command from the File menu to enter the editor.
2. Type in, compile and run the following program:

```
CLS
```

This is a quick and easy way to clear the Output window before you enter something new. Because you're printing a lot of numbers, you'll want to do this from time to time.

1. Choose the New command from the File menu to enter the editor.
2. Type in, compile and run the following program:

```
PRINT 5 \ 2
```

Your Output window will display the result of the operation:

```
    2
```

Whereas standard division would provide a result of 2.5, or 2 with a remainder of 1, integer division discards the remainder and gives you the result 2.

The MOD operator
Although MOD doesn't look like a typical single-symbol operator, it is in fact a legitimate AC/QuickBasic operator. The MOD operator is also called the remainder operator. Its result is the opposite of the integer division (\) operator's: When you divide two numbers with the MOD operator, the AC/QuickBasic Interpreter/Compiler returns only the remainder.

Practice:
Using the MOD operator

1. Choose the New command from the File menu to enter the editor.
2. Type in, compile and run the following program:

```
PRINT 5 MOD 2
```

Your Output window will display the result of the operation:

```
    1
```

Again, with standard division the result of this operation would be 2 with a remainder of 1.
Because you used the MOD operator, the AC/QuickBasic Interpreter/Compiler returned only the remainder.

The ^ (exponentiation) operator
The exponentiation operator(") lets you raise a number to a power of itself.
For example, the AC/QuickBasic equivalent of 10^3 (ten to the third power) looks like this:

```
10 ^ 3
```

Different Rules for Division
The standard division operator (/) fully divides one number by another.
For example, the following statement displays the result 2.5 - the same result a calculator provides:

```
PRINT 5 / 2
```

The \ and MOD operators, however, produce results that take into account an evenly divided part and a remainder part.

Practice:
Using the ^ operator

1. Choose the New command from the File menu to enter the editor.
2. Type in, compile and run the following program:

```
PRINT 10 ^ 3
```

Your output window will display the result of the operation:

```
    1000
```

Numeric Expressions
A numeric expression, commonly called a formula, is simply a combination of numbers, numeric variables, numeric operators, and numeric functions that collectively yield a result. The multiplication, division, and exponentiation exercises you just did were all simple examples of numeric expressions.

The AC/QuickBasic Interpreter/Compiler allows you to create a wide range of numeric expressions, from the simple ones you just practiced to elaborate ones. And creating numeric expressions in AC/QuickBasic isn't complicated. All you need to do is to learn a few basic rules and you'll be able to calculate almost anything.

Using more than one operator
The example expressions you just worked with each used only a operator. The AC/QuickBasic Interpreter/Compiler will let you use more than one operator in the same numeric expression, allowing you to do more than one calculation at the same time.

Using more than one operator
1. Choose the New command from the File menu to enter the editor.
2. Type in, compile and run the following program:

```
PRINT 14 + 26 + 15.75 - 33.2
```

The Output window should display the result of the operation:

```
    22.55
```

Order of calculation
The AC/QuickBasic Interpreter/Compiler follows a strict set of rules when it calculates a numeric expression that contains more than one operator. Consider the following numeric expression:

```
3 + 4 * 5
```

What's the result of this expression? Actually, there are two different results, depending on the order in which you calculate it. If you perform addition first, the answer is 35. If you perform multiplication first, the answer is 23.

To help you avoid such confusion, the AC/QuickBasic Interpreter/Compiler calculates operations in the following order:

• Exponentiation (^) is performed first.
• Multiplication and division(*,/,\, and MOD) are performed next.
• Addition and subtraction (+ and -) are performed last.
These rules don't cover all circumstances, however. What about expressions in which operators are all of the same type or "level of importance"? For example,

```
      3 * 5 MOD 2

 or
      100 / 4 * 3
```

When the AC/QuickBasic Interpreter/Compiler encounters such expressions, it calculates them from left to right.

Practice
Working with operator precedence
1. Choose the New command from the File menu to enter the editor.
2. Type in, compile and run the following program:

```
PRINT 3 + 4 * 5
```

Your Output window will display the result of the operation:
```
    23
```

Because the AC/QuickBasic Interpreter/Compiler does the multiplication before the addition, the result is 23.

1. Choose the New command from the File menu to enter the editor.
2. Type in, compile and run the following program:

```
PRINT 100 / 4 * 3
```

Your Output window will display the result of the operation:
```
    75
```

Because the division (/) and multiplication (*) operators have the same weight - that is, the same priority in the list of operator precedence-the AC/QuickBasic Interpreter/Compiler calculates the expression from left to right.

Using parentheses to control the order of calculation

Consider the numeric expression you looked at earlier:

```
3 + 4 * 5
```

As you just learned, the AC/QuickBasic Interpreter/Compiler performs the multiplication before it performs the addition, resulting in the value 23.

But what if you want the addition to be performed before the multiplication?

AC/QuickBasic lets you control the order in which the operations in a numeric expression are calculated. If you put part of a numeric expression inside parentheses, the AC/QuickBasic Interpreter/Compiler performs that part of the calculation before any other. For example, the following numeric expression produces a result of 35 because the AC/QuickBasic Interpreter/Compiler calculates the contents of the parentheses before it calculates the remainder of the expression-even though multiplication has precedence over addition:

```
( 3 + 4 ) * 5
```

If you use two or more operators within the same set of parentheses, the AC/QuickBasic Interpreter/Compiler calculates the contents of the parentheses using the standard rules. For example, the following numeric expression produces a result of 23:

```
(3 + 4 * 5 )
```

Practice:
Using parentheses to control how a numeric expression is calculated

1. Choose the New command from the File menu to enter the editor.
2. Type in, compile and run the following program:

```
PRINT 3 + 4 * 5
```

Your Output window will display the result of the operation:
```
    23
```

1. Choose the New command from the File menu to enter the editor.
2. Type in, compile and run the following program:

```
PRINT ( 3 + 4 ) * 5
```

Your Output window will display the result of the operation:
```
    35
```

Using parentheses within parentheses
If you need to use a numeric expression that requires more control than separate sets of parentheses can provide, you can use nested parentheses, one set of parentheses within another.

Consider the following numeric expression:

```
2 + 3 * 4 ^ 2
```

The AC/QuickBasic Interpreter/Compiler would calculate the exponentiation first, then the multiplication, and finally the addition. But what if you wanted AC/QuickBasic to perform the addition first, then the multiplication, and then the exponentiation? If you used

```
( 2 + 3 * 4 ) ^ 2
```


the AC/QuickBasic Interpreter/Compiler would calculate the contents of the parentheses first and do the exponentiation last, but it would still perform the multiplication before the addition. The answer would be to use a set of nested parentheses, like this:

```
( ( 2 + 3 ) * 4 ) ^ 2
```

When the AC/QuickBasic Interpreter/Compiler encounters nested parentheses, it always calculates the contents of the innermost parentheses before calculating the contents of the outermost parentheses. In the example above, the AC/QuickBasic Interpreter/Compiler would perform the addition first because the addition operation is in the innermost parentheses, then the multiplication, and finally the exponentiation.

AC/QuickBasic's Mathematical Functions
As you will learn in Chapter 3, a function returns a value to a program. AC/QuickBasic provides you with several mathematical functions. (As with any other AC/QuickBasic function, you must use a mathematical function within a AC/QuickBasic statement.) The table below shows some of the AC/QuickBasic mathematical functions and what they do.

The symbol n in the table below represents the number, numeric variable, or expression upon which you want the function to operate. Notice that n is enclosed in parentheses. You must use these parentheses when you provide a value as an argument to a function.

Function	Purpose
ABS(n)	Returns the absolute value of n
ATN(n)	Returns the arctangent, in radians, of n
COS(n)	Returns the cosine of angle n expressed in radians
EXP(n)	Returns the logarithm of n
SGN(n)	Returns -1 if n is less than zero, returns 0 if n is zero, and returns +1 if n is greater than zero
SIN(n)	Returns the sine of angle n expressed in radians
SQR(n)	Returns the square root of n
TAN(n)	Returns the tangent of angle n expressed in radians

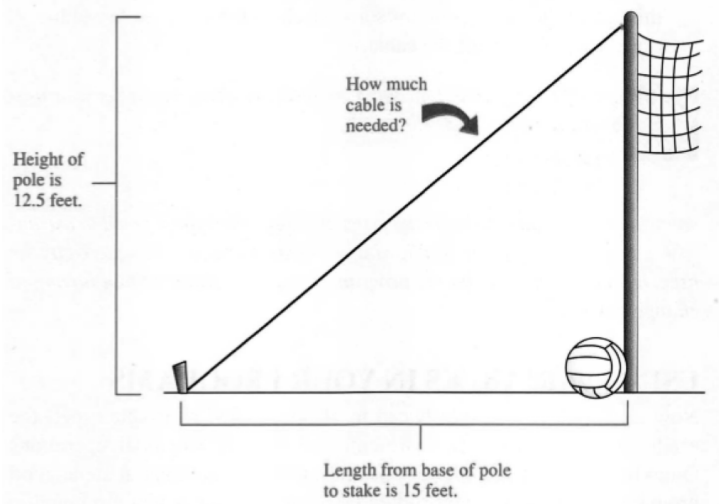
Practice:
With AC/QuickBASIC's mathematical functions

1. Choose the New command from the File menu to enter the editor.
2. Type in, compile and run the following program:

```
PRINT SQR(36)
```

Your Output window should display the result of the operation, in this case the square root of 36:
```
    6
```

Using mathematical functions in numeric expressions
You can use mathematical functions in a numeric expression along with other AC/QuickBasic operators.
In the following example, the SQR function solves a problem involving a right triangle.



Practice:
Using the SQR function
Suppose you have a pole 12 1/2 feet high and a stake in the ground 15 feet from the base of the pole as illustrated in Figure 4-4. You plan to string a cable from the top of the pole to the stake, but you aren't sure how much cable to buy.

You can write a AC/QuickBasic program that uses the SQR function to calculate the length of the cable for you. The formula for this calculation is;

$$\sqrt{height2 + length2}$$

1. Choose the New command from the File menu to enter the editor, type in and compile the following program:

```
CLS
PRINT "This program calculates how much cable you need to"
PRINT "run from the top of a pole to a stake in the ground."
PRINT
INPUT "Enter the height, in feet, of the pole: ",height!
PRINT
PRINT "Enter the length, in feet, from the base"
INPUT "of the pole to the stake in the ground: " length!
cable! = SQR((height! ^ 2) + ( length! ^ 2))
PRINT
PRINT "You need to buy"; cable!; "feet of cable."
```

2. Run the program, and enter the height and length measurements given in Figure 4-4: 12.5 and 15, respectively.
   Your output will look like this:

```
This program calculates how much cable you need to run from the top of a pole to a stake in the ground.

Enter the height, in feet, of the pole: 12.5
Enter the length, in feet. from the base of the pole to the stake in the ground: 15

You need to buy 19.52562 feet of cable.
Height of pole is 12.5 feet.
```

FIGURE 4-4. Finding the length of a cable.

Notice that the program uses single-precision floating-point variables throughout. By using this type of variable, you don't limit the user to using whole numbers. Nor are you using memory-expensive double-precision floating-point variables to calculate a result that in practical use would probably not need to be calculated to 14 or 15 decimal places of precision.

And notice that the line cable! = SQR((height! ^2) +(length! ^2)) uses nested parentheses.
In this case, because of AC/QuickBasic's rules of precedence, the nested parentheses aren't actually needed.

You would get the same result if you didn't use them. But you'll sometimes find that your formulas are easier to read if you use unnecessary parentheses this way.

3. Run the program again, this time using height and length measurements of your own choosing. (If you actually used a program like this, you'd need to allow for some slack in the line and for additional length to tie off the cable.)

NOTE: If you had included the measurements as part of your program, as in

```
height! = 12.5
length! = 15
```

you would have had to change the program itself when you wanted to use another set of measurements.

By allowing the measurements to be entered by the user, however, you can run the program again and again without having to change anything.

USING VARIABLES IN YOUR PROGRAMS
Now that you've been introduced to all the numeric variable types, the kinds of values they can represent, and the mathematical operations AC/QuickBasic can perform, let's tie it all together and look at some good programming practices you should take into account as you use numeric variables in your own programs.

You've learned that it's a good idea to give a variable a descriptive name so that later, when you look back through your program, you'll know exactly what that variable represents. Another aspect of using variables you should consider is where to first declare your variables in your program.
Declaring variables for the first time
Professional programmers always declare their variables near the beginnings of their programs. This provides you (or someone else reading your program) with an easily read list of all variables that will be encountered in the program. And if you need to change initial value of a variable, you won't have to hunt through your program to find the first instance of that variable.

Granted, the example programs you've been typing in so far haven't used many variable names and are only a few lines long. But as your programs grow in size and complexity (as they will continue to do throughout this document), the value of setting up your variables first in a program will become more and more apparent to you.
Let's take a look at a sample program to see how this works.

Practice:

Declaring variables at the beginning of a program

Suppose you want to calculate the real price of one or more items- that is, the price of the item plus the sales tax. You can write a AC/QuickBasic program to help you do this quickly and efficiently.

1. Choose the New command from the File menu to enter the editor.
2. Type in, compile and run the following program:

```
CLS

tax Rate! = .081     ' Sales tax rate in Seattle
price! = 0           ' Price of the item
tax! = 0             ' Amount of tax on price!
total!= 0            ' Total amount you will pay

PRINT "Enter the price of the item. Please do"
INPUT "not use a dollar sign or any commas: $", price!

tax! = price! * taxRate!      ' Calculate the tax amount
total! =price! +tax!          ' Calculate the total cost

PRINT
PRINT "Cost of the item is $"; price!
PRINT "Sales tax would be $"; tax!
PRINT "------------------------------"
PRINT "Total cost would be $"; total!
```

Your Output window should look something like this:

```
    Enter the price of the item. Please do not use a dollar sign or any commas: $ 2500

Cost of the item is $ 2500
Sales tax would be  $ 202.5
------------------------------
Total cost would be $ 2702.5
```

You can run this program as often as you like. If you have a lot of items to calculate, you'll find this program to be much faster and more accurate than a calculator.

Note the list of variable names at the top of the program listing. Probably the most important is taxRate! because if the sales tax rate in your area were to change, it would be a simple matter to change the value in the program line. For purposes of demonstration, all the other variables were listed and initialized with a value of 0. That's probably overdoing it a bit for a program this size, but it gives you a clear idea of how the "pros" do it. It might involve a little extra typing, but once you start writing programs that are several screens long, maintaining such a list can prove invaluable! If you forget a variable name, or what names you've assigned to other numeric variables in your program, all you need to do is check your list at the top.

This program also demonstrates that you can use variables to create new variables. Notice the lines

```
tax! = price! * taxRate!       ' Calculate the tax amount
total! =price! +tax!           ' Calculate the total cost
```

No numbers are involved here - only numeric variables. Because AC/QuickBasic treats numeric variables exactly as it would numbers, you can use variables anywhere you would normally use a number.
Finally, the program uses a good technique to ensure that the user won't enter a dollar sign when asked to enter a dollar amount.

Notice the dollar sign in the line;

```
      INPUT "not use a dollar sign or any commas: $", price!
```

Because you use a dollar sign as part of the message the INPUT statement displays, the user is less likely to enter one when prompted for a dollar amount. Adding the message Please do not use a dollar sign or any commas helps, too!

SUMMARY
Congratulations! You have just taken a major step forward in your journey toward becoming a AC/QuickBasic programmer. Variables (both string and numeric) and operators are important topics - topics that find their way into the very heart of most AC/QuickBasic programs. In the next chapter, you'll add some intelligence to your programs by allowing them to make decisions on their own based on sets of rules you give them.

QUESTIONS AND EXERCISES
1. What are the four types of AC/QuickBasic numeric variables, and how do they differ?
2. Why does AC/QuickBasic put a space in front of a positive number?
3. When dealing with numeric variables,
   What does it mean when the AC/QuickBasic Interpreter/Compiler displays a Syntax error dialog box?
4. What does it mean when the AC/QuickBasic Interpreter/Compiler displays an Overflow dialog box or error message?
5. What type of variable would you use for each of the following numbers?

   a. -32679 d. 14.000001 g. 268110
   b. 12.3774 e. -1286.0 h. -10.222222
   c. 142286.9 f. -268.0005 i. -.0000001

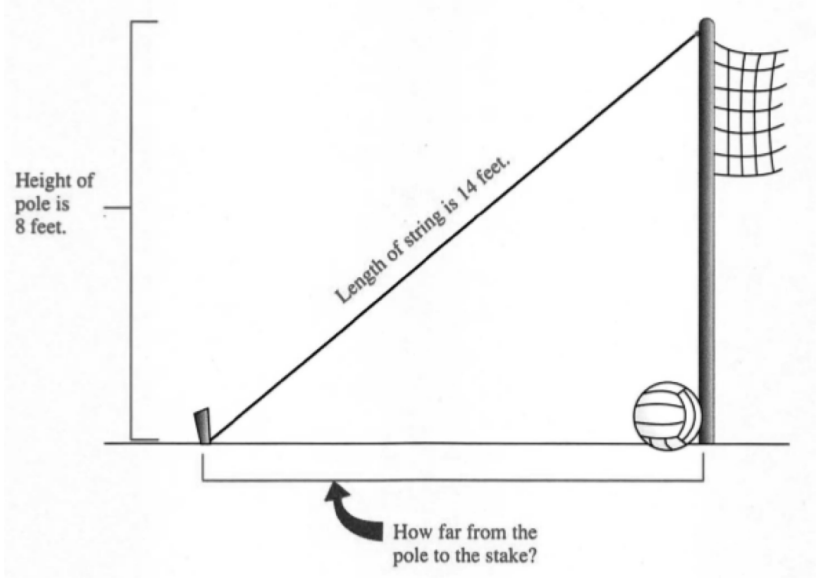
6. What are the differences between regular division with the / operator, integer division with the \ operator, and remainder division with the MOD operator?

7. From highest priority to lowest priority,
What is the order of precedence that AC/QuickBasic assigns to mathematical operators?
8. What is the result of this calculation? (((5 + 8) - (1 + 3) / 4) * ((7 - 2) ^ 2))
9. Write a program (complete with comments) that calculates and prints the following values:
ABS(-10) + 5
SQR(36)
SQR(4) ^ 2
COS(3.141592654)
10. The value of the mathematical constant pi can be approximated as 3.141592654. The formula for the circumference of a circle is 2 x pi x radius. (radius is the distance from the center of the circle to the edge of the circle.)
Write a program that asks the user for the radius of the circle and then displays a message telling the user the circumference of the circle.
11. (BONUS) You're setting up a volleyball net. The top of the pole is 8 feet off the ground, and the string tied to the top of the pole is 14 feet long:

The formula for calculating the distance from the bottom of the pole to where the stake should go is

$$\sqrt{string\_length^2 - height^2}$$

Write a program that asks the user for the height of the pole and the length of the string and then prints out how many feet away from the bottom of the pole the stake should be driven.



Controlling Program Flow Learn AC/QuickBasic For the Apple][gs NOW

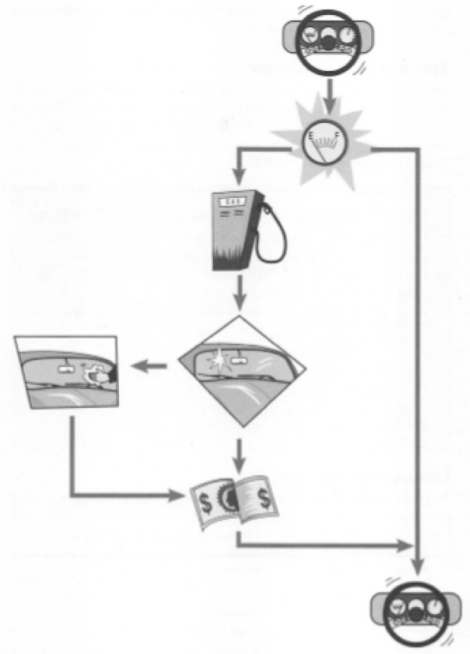
The programs you've written thus far have all run in a straightforward way:
The AC/QuickBasic Interpreter/Compiler runs the first instruction and then works in a straight progression to the final instruction. At times, however, you might want the AC/QuickBasic Interpreter/Compiler to run certain parts of your program under a certain set of circumstances.

This chapter teaches you how to program the AC/QuickBasic Interpreter/Compiler to do this.

INTRODUCTION TO DECISION MAKING
You probably don't need an introduction to decision making. You make thousands of decisions every day. Some decisions require that you weigh your options; others require little or no thought. The decision whether to breakfast on a fruit cup or a jelly doughnut might give you pause, but the decision to reach up and scratch your ear is likely to be spontaneous. Other decisions can lead to separate sets of actions. For example, if you were on a long drive, you would need to be sure you wouldn't run out of gas. Let's examine a typical decision-making process, one you would go through each time you checked your gas gauge.

As the flowchart on the next page shows, you would check the status of your gas tank. If it were full, you'd continue driving. If it were nearly empty, you'd pull into a gas station and fill the tank. You might then check the condition of your windshield. If it were clean, you'd pay the attendant and then drive down the road. If it were dirty, however, you would wash it before paying the attendant and continuing down the road.

It's easy to see how a single decision, based on the answer to the question Am I low on gas?, caused you to do one of two things and then to make further decisions if necessary.



Decision making at work: checking your gas gauge.

Decision Making in AC/QuickBasic
AC/QuickBasic lets you include decision points such as these in your programs.
You place a question like statement in your program, and along with it you write instructions that tell the AC/QuickBasic Interpreter/Compiler what to do given this or that answer.

If the answer is yes, the AC/QuickBasic Interpreter/Compiler takes a particular course of action.
If the answer is no, the AC/QuickBasic Interpreter/Compiler takes a different course of action, based on the instructions you've written.

The answer maybe never occurs in AC/QuickBasic or in any other programming language.
The answer to a question is always either yes or no.

True and False Conditions
You don't actually place yes - or - no questions in a AC/QuickBasic program. Instead, you establish "questions" by placing conditional expressions in your programs. AC/QuickBasic evaluates these conditional expressions - not to determine a yes or a no answer, but to determine whether the conditional expressions are true or false.

IF the Expression's Conditional THEN It's Boolean
Conditional expressions are actually Boolean expressions.

Named after nineteenth-century English mathematician George Boole,
Boolean expressions can be evaluated as true or false.

Here are some examples of Boolean expressions:

Boolean expression	Evaluation
A pint is larger than a gallon.	False
Twelve is greater than ten.	True
Five is less than or equal to six.	True
Eleven inches are equal to one foot.	False

Creating conditional expressions
To create conditional (Boolean) expressions in your program, you must use a relational operator, a logical operator, or a combination of the two kinds. AC/QuickBasic provides the following relational operators:

Relational operator	Meaning
=	Equal to
<>	NOT Equal to
>	Greater THAN
<	Less THAN
>=	Greater than or equal to
<=	Less than or equal to

The table below shows some sample conditional expressions that use the AC/QuickBasic relational operators and their results. You'll learn about logical operators in the next section.

Condition	Result
3<7	True (3 is less than 7)
14 >= 22	False (14 is not greater than or equal to 22)
11 <> 16	True (11 is not equal to 16)
11>=11	True (11 is greater than or equal to 11)
total%< 5	True if the value of total% is less than 5; otherwise, false
num1% = num2%	True if the value of num1% is equal to the value of num2%; otherwise, false

You'll get a chance to work with relational operators and conditional expressions shortly. Right now, let's take a look at some of the AC/QuickBasic statements that allow you to make good use of these conditional expressions in your programs.

Numeric and Conditional
In Chapter 4, you will learn about numeric expressions, which look something like conditional expressions:
Both consist of an operator and data. The difference between a numeric expression and a conditional expression is that a numeric expression uses a numeric operator (such as +, -, /, or MOD) and a conditional expression uses a conditional operator (such as one of the relational operators >=, <, <>,and <=). A numeric expression yields a numeric result, but a conditional expression yields a true or a false result.

Loading Programs from Disk
By now, you've had plenty of practice typing in and running your own programs - one of the best ways to learn to program in AC/QuickBasic. From this point on, the programs become long, so feel free to load the example programs from your hard disk or from your AC/QuickBasic Work Disk.

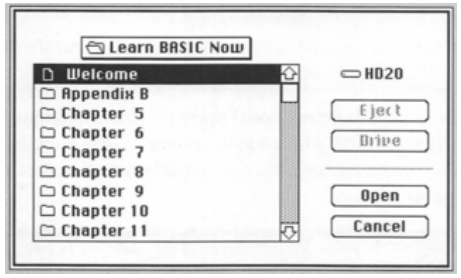
Programs that appear in boxes in this document are listed by name on disk.

• If you have a hard disk, the example programs are located in chapter folders (Chapter 5 through Chapter 13) in the Learn AC/QuickBasic Now folder on your hard disk.

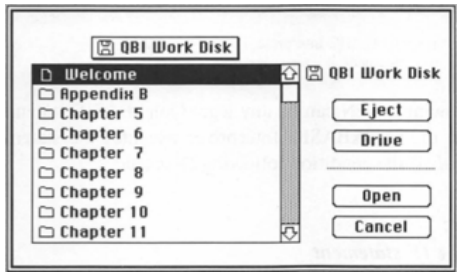
• If you're using a floppy disk drive system,
The example programs are located in chapter folders (Chapter 5 through Chapter 13) on your AC/QuickBasic Work Disk.

[Change this photo to reflect Learn AC/QuickBasic Now.]
The File Open dialog box for a hard disk system.

Follow these instructions to load the Learn AC/QuickBasic Now example programs from disk:



1. Pull down the File menu and choose the Open command.
2. When the dialog box appears, check to be sure that the current folder is Learn AC/QuickBasic Now
   (if you 're using a hard disk) or that the current disk (displayed above the list box) is AC/QuickBasic Work Disk
   (if you 're using a floppy disk).
If necessary, click the Drive button to change to the drive with the Learn AC/QuickBasic Now files.
3. Double-click on the name of the folder you want to open.
   A list of the files in that folder will appear. Scroll the list to see all the files if you need to,
   and double-click on the name of the file you want to open.



[Change this photo to reflect Learn AC/QuickBasic Now.]
The File Open dialog box for a floppy disk system.

THE IF STATEMENT
The IF statement lets you evaluate a condition
And works together with the THEN clause to take a course of action based on the evaluation.

In their simplest combination, IF and THEN make up a single statement.

Here's the syntax for a single-line conditional statement using IF and THEN:

IF condition THEN statement

The condition portion of the statement is one of the conditional expressions you just learned about.
The statement portion is another AC/QuickBasic statement that is executed only if condition is true.

If condition is false,
The AC/QuickBasic Interpreter/Compiler ignores statement and moves on to the next line in your program.

Here's an example:

If this condition is true, the AC/QuickBasic Interpreter/Compiler executes this statement.

```
IF userNum% > 1000 THEN PRINT "The number is too big!"
```

If this condition is false,
The AC/QuickBasic Interpreter/Compiler ignores the rest of the line and executes the next statement in your program.

The statement following THEN can be any legal AC/QuickBasic statement.

But remember that the AC/QuickBasic Interpreter/Compiler executes the statement following THEN only if
    The condition following IF is true.

Practice:
Using the IF statement

1. Load/Compile and run the Rockettes program (Figure 5-1) from the Chapter 5 folder and run it.
2. Enter the number 36. Your Output window will look like this:

```
      How many Rockettes dance at Radio City Music Hall? 36

      That's right!!
      Radio City Music Hall opened on December 27, 1932.
```

```
' Rockettes
' This program demonstrates the IF statement.
CLS
INPUT "How many Rockettes dance at Radio City Music Hall? ", guess%
IF guess%= 36 THEN PRINT "That's right!!"
IF guess% <> 36 THEN PRINT "Sorry! The correct answer is 36!"
PRINT "Radio City Music Hall opened on December 27, 1932."
```

      FIGURE 5-1.   Rockettes: a program that demonstrates use of the IF statement.

Because you entered the value 36, the condition in the first IF statement of the program was true,
so the AC/QuickBasic Interpreter/Compiler executed the PRINT statement at the end of the line.

Also, because the condition in the second IF statement was not true,
The AC/QuickBasic Interpreter/Compiler did not execute the PRINT statement at the end of the second IF statement.

3. Run the program again, entering a value other than 36. Your Output window should look something like this:

```
    How many Rockettes dance at Radio City Music Hall?   4

    Sorry! The correct answer is 36!
    Radio City Music Hall opened on December 27, 1932.
```

This time, the value you entered caused a different set of actions to occur.

Because the value 4 is not equal to the value 36, the condition in the first IF statement was false,
so the AC/QuickBasic Interpreter/Compiler ignored the PRINT statement in that instruction line.

This time, the condition in the second IF statement was true,
so the AC/QuickBasic Interpreter/Compiler executed the PRINT statement in that instruction line.

Notice that in both cases AC/QuickBasic executed the final PRINT statement of the program,
printing the message Radio City Music Hall opened on December 27, 1932.

This demonstrates that even if the condition in an IF statement is false,
the AC/QuickBasic Interpreter/Compiler executes the rest of the program as it normally would.

Only the statement that follows the THEN portion of an IF statement is ignored if the condition is not true.

Using More than One Condition with IF
In the preceding program, AC/QuickBasic evaluated a single condition in each IF statement.

You can specify multiple conditions in an IF statement by using the logical operators AND and OR.
You use logical operators in conditional expressions much as you use math operators in numeric expressions.

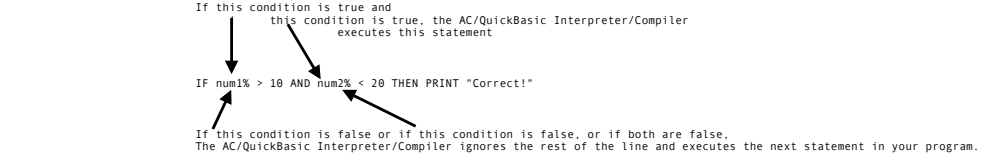
The AND logical operator
The AND operator lets you specify multiple conditions that must be true before an action can be taken.

Here's the syntax line for an IF statement that uses the AND operator:

IF condition1 AND condition2 THEN statement

Both condition1 and condition2 must be true before the AC/QuickBasic Interpreter/Compiler can execute statement.

Here's an example:



Note that the IF statement contains two conditions and that because they're connected by the logical operator AND,

Both num1% > 10 and num2% < 20 must be true before the AC/QuickBasic Interpreter/Compiler can execute the PRINT
statement that follows THEN.

Practice:
Working with the AND operator
1. Load and Compile the Teenagers program (Figure 5-2) from disk and run it.

```
' Teenagers
' This program demonstrates the AND logical operator.
CLS
INPUT "How many teenagers can fit in a phone booth? ", guess%
PRINT
IF guess%> 9 AND guess%< 13 THEN PRINT "That's right!!"
PRINT "Depending on their sizes, approximately 10 to 12"
PRINT "teenagers can fit in a phone booth."
```

    FIGURE 5-2.   Teenagers: a program that demonstrates use of the AND logical operator.

2. Enter the number 10. Your Output window will look like this:

```
    How many teenagers can fit in a phone booth?   10

    That's right!!
    Depending on their sizes, approximately 10 to 12 teenagers can fit in a phone booth.
```

Because you entered the value 10, both conditions in the IF statement were true.
If you run the program again and enter an integer value less than 10 or more than 12,

One of the conditions will be false
And the AC/QuickBasic Interpreter/Compiler will not execute the PRINT statement that follows THEN.

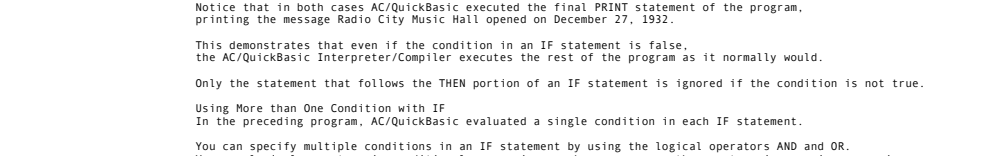
The OR logical operator
The OR operator lets you create a more flexible set of conditions that must be met before an action can take place.

Here's the syntax line for an IF statement that uses the OR logical operator:

   IF condition1 OR condition2 THEN statement

Note that the IF statement contains two conditions and that only one of these conditions need be true before the
AC/QuickBasic Interpreter/Compiler can execute the statement that follows THEN.

The AC/QuickBasic Interpreter/Compiler also executes the statement following THEN if both conditions are true.
You'll see an example below.



Practice:
Working with the OR operator

1. Load and compile the Guess 63 program (Figure 5-3) from disk.
2. Run the program.   Enter the value   63, the number the program is “thinking of.”

   Your Output window will look like this:

```
    I'm thinking of a number between 1 and 100.
    Can you guess what it is?
    Please enter a number between 1 and 100:    63

    That's right!!
    Thanks for playing!
```

```
' Guess 63
' This program demonstrates the OR logical operator.
CLS
PRINT "I'm thinking of a number between 1 and 100."
PRINT "Can you guess what it is?"
INPUT "Please enter a number between 1 and 100:  ", guess%
PRINT
```

```
IF guess% = 63 THEN PRINT "That's right!!"
IF guess% < 53 OR guess% > 73 THEN PRINT "You're way off!"
PRINT "Thanks for playing!"
```

FIGURE 5-3.
Guess 63: a program that demonstrates use of the OR logical operator.

3. Run the program again,
   And enter a number that is less than 53 or greater than 73.
   Your Output window will look something like

```
I'm thinking of a number between 1 and 100.
Can you guess what it is?
Please enter a number between 1 and 100:   76

You're way off!
Thanks for playing!
```

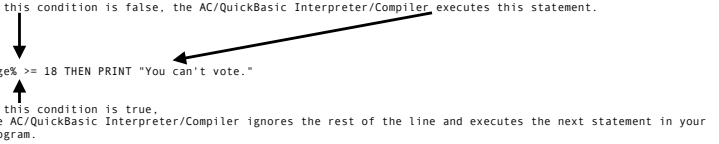
Look at the second IF statement in Guess 63.
By entering a value that was 11, or more less than or 11 or more greater than the number the program; Was "thinking of," you caused the AC/QuickBasic Interpreter/Compiler to print the message You're way off!

The NOT logical operator
The NOT operator lets you negate a condition.
In other words, if a condition is false, the NOT operator makes the condition true; if a condition is true, NOT makes it false. Here's the syntax line for an IF statement that uses the NOT operator:

```
IF NOT condition THEN statement
```

NOT is useful when you want to execute a statement when a condition is not true.

Here's an example:



Using ELSE with IF and THEN
Now you know how to make the AC/QuickBasic Interpreter/Compiler evaluate a condition and take an action if the condition is true or if the condition is false.

But what if you want the AC/QuickBasic Interpreter/Compiler to choose between two actions based on the condition?

When paired with IF and THEN, an ELSE clause;

Lets you specify two separate actions for the AC/QuickBasic Interpreter/Compiler to follow:
One action (following THEN) if the condition is true, and another (following ELSE) if the condition is false.

Here's the syntax for an IF statement that uses THEN and ELSE:

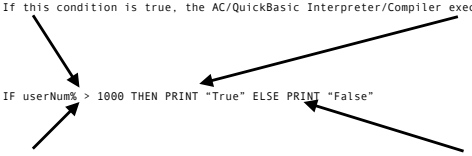
```
IF condition THEN statement1 ELSE statement2
```

condition is the logical condition you want the AC/QuickBasic Interpreter/Compiler to evaluate as true or false,

statement1 is the AC/QuickBasic statement the AC/QuickBasic Interpreter/Compiler executes if condition is true,
statement2 is the AC/QuickBasic statement the AC/QuickBasic Interpreter/Compiler executes if condition is false.

IF, THEN, and ELSE must all appear in the same instruction line.

Here's an example:



Practice:
Working with IF, THEN, and ELSE

1. Load and Compile the Guess 1-5 program (Figure 5-4) from disk.

```
' Guess 1-5
' This program demonstrates the ELSE clause.
CLS
PRINT "I'm thinking of a number between 1 and 5."
PRINT "Can you guess what it is?"
INPUT "Enter a number between 1 and 5:  ", guess%
PRINT
IF guess% = 3 THEN PRINT "That's right!!" ELSE PRINT "Sorry!"
PRINT "Thanks for playing!"
```

FIGURE 5-4. Guess 1-5: a program that demonstrates use of the ELSE clause.

2. Run the program and enter the number 3

   Your Output window will look like this:

```
 I'm thinking of a number between 1 and 5.
 Can you guess what it is?
 Enter a number between 1 and 5:   3

 That's right!!
 Thanks for playing!
```

Because you entered the value 3,
The condition is true, so the AC/QuickBasic Interpreter/Compiler executed the PRINT statement that follows THEN.

3. Run the program again, entering a value other than 3.

   Your Output window will look something like this:

```
 I'm thinking of a number between 1 and 5.
 Can you guess what it is?
 Enter a number between 1 and 5: 4

 Sorry!
 Thanks for playing!
```

This time the condition was not true, so the AC/QuickBasic Interpreter/Compiler executed the PRINT statement that follows ELSE.

Making Longer Conditional Statements with END IF
Using ELSE with IF and THEN is a handy way to give the AC/QuickBasic Interpreter/Compiler two options to take depending on the evaluation of a particular condition. However, as you just saw, packing all that information on a single line doesn't allow much room for creativity.

That's where END IF comes in. By using END IF, you can create longer programs by placing each possible course of action on a separate line. Here's the syntax for an IF statement that uses THEN, ELSE, and ENDIF:

```
IF condition THEN
   statements executed if condition is true
ELSE
   statements executed if condition is false
END IF
```

An IF statement of this type actually consists of several individual lines, which are collectively known as a block.
The condition part of the statement is the conditional expression you want the AC/QuickBasic Interpreter/Compiler to evaluate as true or false.

The THEN keyword ends the line. In a block IF statement, THEN must always appear in the same line as IF.

If condition is true, the AC/QuickBasic Interpreter/Compiler
1. Executes the statements between THEN and ELSE
2. Bypasses the statements between ELSE and END IF
3. Continues to execute the program

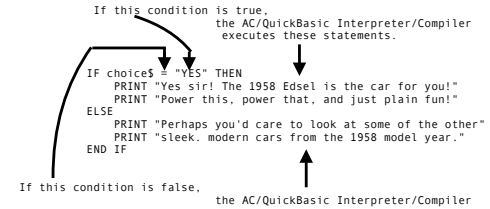
You can include any number of statements between THEN and ELSE, including other IF statements.

If condition is false, the AC/QuickBasic Interpreter/Compiler

1. Bypasses the statements between THEN and ELSE
2. Executes the statements between ELSE and END IF
3. Continues to execute the program

You can include any number of statements between ELSE and END IF, including other IF statements.

Here's an example:



Notice how the indentation of related statements under the clauses that govern them keeps the program easy to read.
You should develop the habit of using indentation in your own programs.

Practice:
Working with IF, THEN, ELSE, and END IF

1. Load and compile the Auto Trivia program (Figure 5-5) from disk.

2. Run the program. Enter the value 1885, which is the value that causes the condition that follows IF to be true.

    Your Output window will look like this:

```
          Welcome to Automobile Trivia!

          In what year did Karl-Friedrich Benz test-drive
          the first successful gasoline-driven automobile?   1885

          That's right! You're quite a car buff!
          Thanks for guessing!
```

```
' Auto Trivia
' This program demonstrates the block IF statement.

CLS

PRINT "Welcome to Automobile Trivia!"
PRINT
PRINT "In what year did Karl-Friedrich Benz test-drive"
INPUT "the first successful gasoline-driven automobile?  ", guess%
PRINT
IF guess% = 1885 THEN
    PRINT "That's right!  You're quite a car buff!"
ELSE
    PRINT "No, he first drove it at Mannheim, Germany,"
    PRINT "in 1885.  (It was patented on January 29, 1886.)"
END IF
PRINT "Thanks for guessing!"
```

FIGURE 5-5.
Auto Trivia: a program that demonstrates use of the block IF statement.

3. Run the program again, but this time enter a value other than 1885.

   Your Output window will look something like this:

```
          Welcome to Automobile Trivia!

          In what year did Karl-Friedrich Benz test-drive the first successful gasoline-driven automobile?    1776

          No, he first drove it at Mannheim, Germany, in 1885. (It was patented on January 29, 1886.)
          Thanks for guessing!
```

As you can see, using END IF with IF, THEN, and ELSE allows you to use entire blocks of statements.

The ELSEIF Keyword
ELSEIF is similar to ELSE in that it provides an alternate course of action if condition is false.
With ELSEIF, however, you supply at least a second condition for the AC/QuickBasic Interpreter/Compiler to evaluate.

Here's the syntax for an IF statement that uses ELSEIF:

```
IF condition1 THEN
      statements executed if condition1 is true
ELSEIF condition2 THEN
      statements executed if condition2 is true
ELSEIF condition3 THEN
      statements executed if condition3 is true
  .
  .
  .
ELSE
      statements executed if all conditions are false
END IF
```

The column of dots between ELSEIF and ELSE indicates
That you can have more ELSEIF statements followed by other conditions if you want to.

AC/QuickBasic places no limit on the number of ELSEIF statements and associated conditions you can use.

NOTE: You must use THEN at the end of an ELSEIF statement.
      The use of ELSE is optional.
 We've shown it so that you'll know where it goes if you decide that your program needs it.

If condition1 is true, the AC/QuickBasic Interpreter/Compiler

1. Executes the statements in the following lines until it encounters the first ELSEIF
2. Jumps down to END IF
3. Continues to execute the program

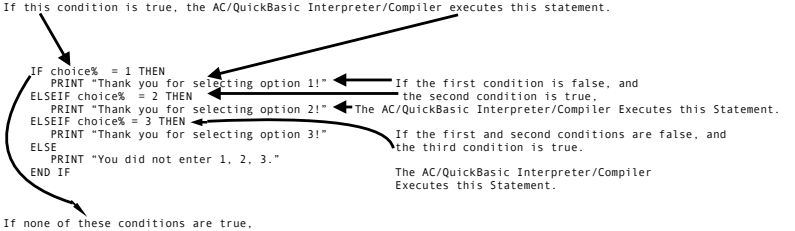
If condition1 is false, the AC/QuickBasic Interpreter/Compiler Jumps down to the first ELSEIF and evaluates
condition2.

If condition2 is true, the AC/QuickBasic Interpreter/Compiler
1. Executes the statements on the following lines until it encounters the next ELSEIF or,
   If you included one, the ELSE clause
2. Jumps down to END IF and continues to execute the program

If the condition associated with an ELSEIF statement is false,
The AC/QuickBasic Interpreter/Compiler jumps down to the next ELSEIF statement and evaluates its condition.

The AC/QuickBasic Interpreter/Compiler continues this process until an ELSEIF statement evaluates as true
or until it encounters an ELSE statement.

Here's an example:



If none of these conditions are true,
The AC/QuickBasic Interpreter/Compiler executes this statement.

Using the ELSEIF keyword allows you to specify different conditions and different courses of action for the
AC/QuickBasic Interpreter/Compiler to take.

Practice:
Working with ELSEIF

1. Load and compile the Movie Trivia program (Figure 5-6) from disk.

```
' Movie Trivia
' This program demonstrates the use of ELSEIF.
CLS
PRINT "Welcome to Motion Picture Trivia!"
PRINT "In what year did the film Ben Hur win"
PRINT "the Academy Award for Best Picture?"
PRINT
INPUT "Please enter a year from 1950 through 1959:  ", year%
IF year% = 1950 THEN
    PRINT "Incorrect.  In 1950, the Academy Award for"
    PRINT "Best Picture went to All About Eve."
ELSEIF year% = 1951 THEN
    PRINT "Incorrect.  In 1951, the Academy Award for"
    PRINT "Best Picture went to An American in Paris."
ELSEIF year% = 1952 THEN
    PRINT "Incorrect.  In 1952, the Academy Award for"
    PRINT "Best Picture went to The Greatest Show on Earth."
ELSEIF year% = 1953 THEN
    PRINT "Incorrect.  In 1953, the Academy Award for"
    PRINT "Best Picture went to From Here to Eternity."
ELSEIF year% = 1954 THEN
    PRINT "Incorrect.  In 1954, the Academy Award for"
    PRINT "Best Picture went to On the Waterfront."
ELSEIF year% = 1955 THEN
    PRINT "Incorrect.  In 1955, the Academy Award for"
    PRINT "Best Picture went to Marty."
ELSEIF year% = 1956 THEN
    PRINT "Incorrect.  In 1956, the Academy Award for"
    PRINT "Best Picture went to Around the World in 80 Days."
ELSEIF year% = 1957 THEN
    PRINT "Incorrect.  In 1957, the Academy Award for"
    PRINT "Best Picture went to The Bridge on the River Kwai."
ELSEIF year% = 1958 THEN
    PRINT "Incorrect.  In 1958, the Academy Award for"
    PRINT "Best Picture went to Gigi."
ELSEIF year% = 1959 THEN
    PRINT "Correct!  Ben Hur, directed by William Wyler,"
    PRINT "won 11 Academy Awards in 1959, including"
    PRINT "Best Picture."
ELSE
    PRINT "You did not enter a number from 1950 through 1959."
    PRINT "Please run the program again and enter a year from"
    PRINT "1950 through 1959."
END IF
```

2. Run the program and enter the value 1959, which is the correct answer to the question.
   Your Output window will look like this:

```
Welcome to Motion Picture Trivia!
In what year did the film Ben Hur win the Academy Award for Best Picture?

Please enter a year from 1950 through 1959: 1959
Correct! Ben Hur, directed by William Wyler, won 11 Academy Awards in 1959, including Best Picture.
```

3. Run the program again, this time entering a value other than 1959.
   Your Output window will look something like this:

```
Welcome to Motion Picture Trivia!

In what year did the film Ben Hur win the Academy Award for Best Picture?
Please enter a year from 1950 through 1959: 1950
Incorrect. In 1950, the Academy Award for Best Picture went to All About Eve.
```

4. Run the program one more time, this time entering a value that is not in the range 1950 through 1959.
   Your Output window will look something like this:

```
Welcome to Motion Picture Trivia!

In what year did the film Ben Hur win the Academy Award for Best Picture?
Please enter a year from 1950 through 1959: 1596
You did not enter a number from 1950 through 1959.
Please run the program again and enter a year from 1950 through 1959.
```


Working with AC/QuickBasic Learn AC/QuickBasic For the Apple][gs NOW

Loops
You've now learned most of the AC/QuickBasic fundamentals.
In this chapter, you'll learn how to use one of AC/QuickBasic's most powerful tools: the loop.
By using loops, you can have the computer perform certain repetitive tasks in a fraction of the time it would take to do them yourself.

Introduction to AC/QuickBasic Loops
A loop is simply one or more AC/QuickBasic statements that you direct the AC/QuickBasic Interpreter/Compiler to repeat.

Loops can repeat in two ways:

- A specific number of times
- As long as a certain condition is met

In AC/QuickBasic for the Apple][gs, two statements - FOR and WHILE - allow you to add loops to your program.

  In the FOR statement, you specify the number of times you want a loop to be executed.
  In the WHILE statement, you specify a condition,

and

  The AC/QuickBasic Interpreter/Compiler evaluates the condition and executes the loop as long as the condition is met.

Although the FOR and WHILE statements perform similar functions, each performs its task in a slightly different way.
As you'll see, their variations make each statement suited for a particular type of work.

THE FOR STATEMENT
To create a loop that executes a specific number of times (the most common type of loop in AC/QuickBasic),

  Use the FOR statement.

The FOR statement always ends with the NEXT statement, as in the following syntax:

```
  FOR variable = start TO end
       statements to be repeated
  NEXT variable
```

variable is a numeric variable that reflects the number of times the AC/QuickBasic Interpreter/Compiler has executed the loop. It's a counter of sorts, and the AC/QuickBasic Interpreter/Compiler increments variable each time it executes the loop.
Notice that variable follows both the FOR statement and the NEXT statement. These variable names must be exactly the same.

start is the numeric value (either a number or a numeric expression) at which you want the AC/QuickBasic Interpreter/Compiler to start counting.

end is a numeric value (either a number or a numeric expression) that tells the AC/QuickBasic Interpreter/Compiler how high it should count - that is, how many times it should repeat the loop.

When you assign values to start and end, keep the following hints in mind:

- You can use both positive and negative values for start and for end.
- You can use integers or floating-point values for start and for end.
- The value of start must be less than or equal to the value of end.
- The value of start doesn't need to be 1.

The statements between the FOR statement and the NEXT statement are the AC/QuickBasic statements that the AC/QuickBasic Interpreter/Compiler executes the specified number of times. There is no limit to the number of statements you can use, and the statements don't need to be of the same type.

The AC/QuickBasic Interpreter/Compiler uses the NEXT statement to count how many times it has executed the statements between the FOR statement and the NEXT statement.

Here's an example of a FOR statement that prints a message five times,
   incrementing the variable i% each time it does so:

```
  FOR i% = 1 TO 5
      PRINT "I am in a loop."
  NEXT i%
```

Each time it completes the loop;
The AC/QuickBasic Interpreter/Compiler jumps back up to the FOR statement to compare the value of i% with the value of end
   (which is 5).

As soon as i% exceeds end,
The AC/QuickBasic Interpreter/Compiler jumps to the statement following NEXT and continues to execute the program.

Working with Full-Size List and Output Windows

The programs in this chapter and throughout the rest of the book are large enough to warrant both a full-size List window and a full-size Output window. Before you load the first program in this chapter from disk, you'll zoom the List window to full size. The first program contains an instruction that will keep the List window from appearing before you've had a chance to look at the program's output.

Practice:
Working with a FOR loop
1. Click on the zoom box in the upper right corner of the List window.
   The List window should expand to fill the entire screen.

2. Load and compile the FOR Loop 1 program (Figure 6-1) from the Chapter 6 folder on disk.

```
' FOR Loop 1
' This program demonstrates a simple FOR loop.

CLS

INPUT "Please enter a number between 2 and 5:  ", times%
PRINT

FOR i% = 1 TO times%
    PRINT "These lines will print"; times%; "times."
    PRINT "This is time"; i%
    PRINT
NEXT i%

INPUT "Press Return to continue…", dummy$
```

3. Run the program. Your Output window will look something like

```
      Please enter a number between 2 and 5: 4

      These lines will print 4 times.
      This is time 1

      These lines will print 4 times.
      This is time 2

      These lines will print 4 times.
      This is time 3

      These lines will print 4 times.
      This is time 4

      Press Return to continue ...
```

4. Press the Return key to return to the List window.

Using INPUT to Make a Program Pause
You've used the INPUT statement many times already to get information from the keyboard. We've put a nifty side effect of the INPUT statement to use in the FOR Loop 1 program to keep the Output window displayed on the screen.
Notice the line INPUT "Press Return to continue ... ", dummy$ at the end of the program.
When the AC/QuickBasic Interpreter/Compiler encounters this line, it pauses and waits for the user to enter the value of dummy$.
The program doesn't actually use the value of dummy$ - dummy$ is merely a placeholder, or "dummy," to satisfy the requirement that you must specify a variable in an INPUT statement.

Because the List window appears as soon as a program has finished running, and because the List window can cover part or all of the Output window, it's handy to use this trick to keep the Output window displayed until you're ready to see the List window again. After the List window is displayed, the information in the Output window it covers up is gone forever - even if you choose the Output command from the Window menu, the information now by the List window is not displayed again unless you run the program again.

Practice:
When start is greater than end
1. Load and compile the FOR Loop 2 program (Figure 6-2) from disk. Notice that start is greater than end.

```
' FOR Loop 2
' This program demonstrates a FOR loop that never loops.

CLS

FOR i% = 6 TO 5
    PRINT "The current value of i% is"; i%
NEXT i%

INPUT "Press Return to continue…", dummy$
```

2. Run the program. Except for the message "Press Return to continue ..." , your Output window will be blank.
   The AC/QuickBasic Interpreter/Compiler saw the counter at 6, a value greater than end, and therefore considered its task complete.

Practice:
Using equal start and end values
1. Change the FOR statement in the FOR Loop 2 program so that start and end have the same value:

```
    FOR i % = 5 TO 5
```

2. Run the program. Your Output window will look like this:

```
    The current value of i% is 5
```

This time, the AC/QuickBasic Interpreter/Compiler executed the body of the FOR statement only once.
Because the start and end values are the same, the AC/QuickBasic Interpreter/Compiler assumed that this was its last "lap" and executed the PRINT statement only one time before moving on.

Looping and the TEXTSIZE Statement
You'll learn more about the TEXTSIZE statement later, but for now just sit back and enjoy this demonstration, which uses both the TEXT-SIZE statement and a FOR loop within a program.

Changing the text size with a FOR loop
1. Load and compile the Textsize program (Figure 6-3 on the next page) from disk.

Why Use i%?
In most of the FOR loops in this book, you'll notice that the loop variable is i%. Why is this?

Before BASIC was invented, many programmers used a language called FORTRAN.

In FORTRAN, the first letter of a variable name specified its type.
Any variable name that started with a letter from i through n, for instance, denoted an integer variable, just as a percent sign (%) appended to a AC/QuickBasic variable name denotes an integer variable.

Most FOR loop counters are integers. Thus, to save typing time, a FORTRAN programmer usually used the variable i as the loop counter. Programmers needing to nest two or more loops (see "Nesting FOR Loops" later in this chapter) would go to j, then k, and so on. Many BASIC programmers have adopted this tradition. Must you use i%, j%, and k%? No. You can use any valid AC/QuickBasic variable name - count%, numLines%, or dayOfMonth%, for example.

```
' Textsize
' This program demonstrates using a FOR loop and the TEXTSIZE
'   statement to show different character heights.

CLS

FOR i% = 10 TO 20
    TEXTSIZE i%
    PRINT "These letters are"; i%; "points tall."
NEXT i%

INPUT "Press Return to continue…", dummy$
```

FIGURE 6-3.
Textsize: using the TEXTSIZE statement in a FOR loop to show character heights.

The TEXTSIZE Statement
With the TEXTSIZE statement, you can tell the AC/QuickBasic Interpreter/Compiler the size in which it should display characters printed with a PRINT statement in the Output window. Here's the syntax for the TEXTSIZE statement:

```
   TEXTSIZE size
```

size is a value representing the height of the characters, in points, that you want to display.
A point is a typographic unit of measurement. There are approximately 72 points in an inch, so the statement

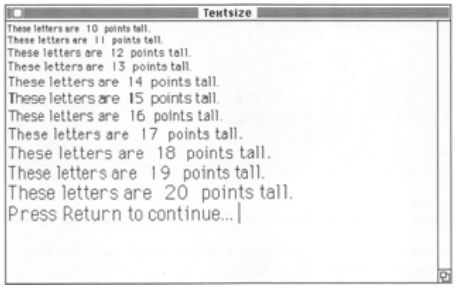
```
   TEXTSIZE 72
```

Causes the AC/QuickBasic Interpreter/Compiler to display the characters printed with a PRINT statement at a height of 1 inch.

The value you use for size can range from 2 on up, although not all values produce good-looking results.
You'll get more details about this later in this document.

The default text size in the AC/QuickBasic Interpreter/Compiler's Output window is 12 points.

2. Run the program. Your Output window will look like this:



Notice that not all sizes produce good-looking text. The 12-point, 14-point, 18-point, and 20-point lines look pretty good, but the others have some contorted-looking letters in them because the Apple][gs tries its best to approximate specified sizes by scaling sizes that are already loaded into the system.

Also notice that the message Press Return to continue ... is displayed in 20-point text.
To display the message in normal-size (12-point) letters, you must precede the INPUT statement with the statement

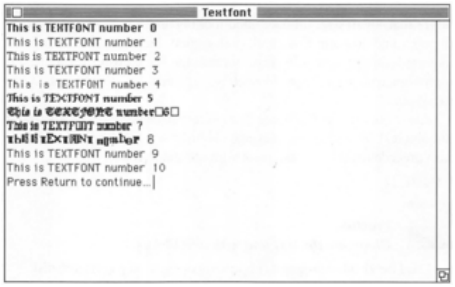
```
TEXTSIZE 12
```

Practice:
Changing the text font with a FOR loop

1. Load and compile the Textfont program (Figure 6-4 on the next page) from disk.

```
' Textfont
' This program demonstrates using a FOR loop and the TEXTFONT
'   statement to show different text faces.

CLS

FOR i% = 0 TO 10
    TEXTFONT i%
    PRINT "This is TEXTFONT number"; i%
NEXT i%

INPUT "Press Return to continue…", dummy$
```

FIGURE6-4.
Textfont: using the TEXTFONT statement in a FOR loop to display different text faces.



2. Run the program. Your Output window will look something like the Output window shown below.

Some font numbers will not produce a different text face; in the example above, font numbers 9 and 10 did not produce a different font (or, more correctly, produced font number 1) because the Apple][gs used to create this example did not have those particular fonts installed.

Apple][gs Fonts and the TEXTFONT Statement
With the TEXTFONT statement, you can tell the AC/QuickBasic Interpreter/Compiler what typeface, or font, it should use to display characters printed with a PRINT statement in the Output window. Here's the syntax of the TEXTFONT statement:

```
TEXTFONT fontnumber
         fontnumber is the number of the font you want to use.
```

The following table shows the numbers and names of the fonts available in AC/QuickBasic: { Correct Values for][GS }

Font number	Font name
0	System font
1	Application font
2	New York
3	Geneva
4	Monaco
5	Venice
6	London
7	Athens
8	San Francisco
9	Toronto
10	Seattle
11	Cairo
21	Helvetica
22	Courier

The system font (0) is the font used for the AC/QuickBasic Interpreter/Compiler's menu and window title text.
The application font (1) is the default font the AC/QuickBasic Interpreter/Compiler uses to display text in the Output window.

Note that, because of the wide variety of fonts available for the Apple][gs, your Apple][gs might contain fewer, more, or some different fonts than our Apple][gs does. Use the Font installer program that came with your Apple][gs to find out which fonts you have on your system. We'll look at fonts again later in this document.
For future reference, make a note of which numbers produced which fonts on your Apple][gs.

Controlling the Count with STEP
As you've seen, when the AC/QuickBasic Interpreter/Compiler executes a FOR loop, it begins by assigning the start value to the variable following FOR and then loops through the block of statements until the value of the counter variable is greater than the value of end. Each time it encounters the NEXT statement, the AC/QuickBasic Interpreter/Compiler increments the value of the counter by 1.

You can tell the AC/QuickBasic Interpreter/Compiler to increment the counter by a value other than 1 if you use the STEP clause.

Here's an example:

```
FOR i% = 5 TO 25 STEP 5
PRINT "This message will print 5 times."
NEXT i %
```

When the AC/QuickBasic Interpreter/Compiler encounters the NEXT statement, it increments the counter by the number you've specified after STEP.

The number in the STEP clause doesn't have to be positive.
If you specify a negative number after STEP, the AC/QuickBasic Interpreter/Compiler decrements the value of the counter each time it encounters the NEXT statement.

The STEP value also affects the values you assign to start and end:

- If the STEP value is positive, the start value must be less than or equal to the end value.
- If the STEP value is negative, the start value must be greater than or equal to the end value.

Think about it: If you use a negative STEP value and end is greater than start, the AC/QuickBasic Interpreter/Compiler will assume that its work is done, jump over the block of statements without executing it, and continue to execute the rest of the program.

Practice:
Using STEP
1. Load and compile the Falling program (Figure 6-6) from disk.

```
' Falling
' This program demonstrates the use of STEP.

CLS

FOR i% = 15 TO 1 STEP -1
    PRINT "Falling!"
    SOUND (50 * i%), 1
NEXT i%

PRINT
INPUT "Press Return to continue…", dummy$
```

FIGURE 6-6.
Falling: using STEP in a FOR statement.

2. Run the program. The AC/QuickBasic Interpreter/Compiler will print the word Falling! 15 times, with accompanying sound to accent the falling effect.

3. Change the duration in the SOUND statement from 1 to 2, and run the program again.

Notice that the AC/QuickBasic Interpreter/Compiler finishes printing Falling!, all 15 times, long before it finishes creating all the sounds. This brings up an important point about using the SOUND statement, one we'll touch on now and explore in greater detail later in this document.

One unique feature of the Apple][gs is that it uses a separate sound generator chip to create the sounds it makes. The nice thing about this is that the Apple][gs can do two things at once:

It can display the AC/QuickBasic Interpreter/Compiler's output at the same time that it plays the sounds you've instructed the AC/QuickBasic Interpreter/Compiler to play. As you've just seen, however, this feature can work against you if you're not careful.

If your program finishes running before all the sounds are played, the AC/QuickBasic Interpreter/Compiler displays its List window while the sounds continue to play - it won't wait for the sounds to finish.

To prevent this from happening, simply remember how the Apple][gs works with sound and plan accordingly in your programs: Don't use duration values that are too large, and avoid using SOUND statements at the very end of your programs.

Nesting FOR Loops
Nesting is the apt term for the practice of putting one loop within another.

Here's an example of a nested FOR loop:

```
FOR i% = 1 TO 4
    FOR j% = 1 TO 4
        PRINT "You'll see this message 16 times."
    NEXT j%
NEXT i%
```

In nesting, indentation really helps you see what's going on.

In a nested FOR loop, the inner FOR loop is actually a statement for the outer loop;
that is, the outer FOR loop executes the inner loop as many times as you specify.

In this example, the outer loop executes the inner loop a total of 4 times.
Because the inner loop executes its statements 4 times,

The AC/QuickBasic Interpreter/Compiler ends up executing the PRINT statement a total of 16 times.

Practice:
Working with nested FOR loops

1. Load and compile the Nested FOR Loops program (Figure 6-7) from disk.

```
' Nested FOR Loops
' This program demonstrates nested FOR loops.

CLS

FOR i% = 1 TO 3
    PRINT "The outer loop value is"; i%
    FOR j% = 1 TO 3
        PRINT , "The inner loop value is"; j%
        SOUND (j% * 400), .5
    NEXT j%
NEXT i%

PRINT
INPUT "Press Return to continue…", dummy$
```

FIGURE 6-7.
Nested FOR Loops: using one FOR loop nested inside a second FOR loop.

2. Run the program. When the program has finished running, your Output window will look like this:

```
   The outer loop value is 1
                  The inner loop value is 1
                  The inner loop value is 2
                  The inner loop value is 3
   The outer loop value is 2
                  The inner loop value is 1
                  The inner loop value is 2
                  The inner loop value is 3
   The outer loop value is 3
                  The inner loop value is 1
                  The inner loop value is 2
                  The inner loop value is 3
```

The SOUND statement within the inner loop creates a sound after each occurrence of the inner loop's PRINT statement.
If you want to, change some of the values. Try to anticipate what the changed program will do before you run it,
and then run the program to see if you were right.

THE WHILE STATEMENT
To design a loop that repeats as long as a specific condition is true, use the WHILE statement.
The WHILE statement always ends with the WEND statement, as shown in the following syntax:

```
WHILE condition
    statements to be repeatedly executed
WEND
```

condition can be any conditional expression (such as wage! = 11.50, year% <= 1959, or temperature%< 32).

When it encounters a WHILE statement, the AC/QuickBasic Interpreter/Compiler evaluates condition:

- If condition is true,
  AC/QuickBasic executes the block of statements between the WHILE and WEND statements and then checks condition again.

- If condition is false,
  AC/QuickBasic jumps to the WEND statement and moves on to the rest of the program.

  To use a WHILE loop successfully,
  You must make provisions inside the loop to alter the condition that follows the WHILE statement.

- NOTE: Be sure that condition eventually becomes false; if you don't, your loop will never stop, and the rest of your program will never run. (A loop that never stops is called an infinite or endless loop.)

Practice:
Working with the WHILE loop
In the next program, all activity takes place within the loop.

Because AC/QuickBasic increments the value of userNum% each time it executes the loop, userNum% eventually becomes greater than 12. This creates a false condition and causes the AC/QuickBasic Interpreter/Compiler to stop executing the loop.

1. Load and compile the WHILE Loop program (Figure 6-8) from disk.

```
' WHILE Loop
' This program demonstrates the WHILE loop.

CLS

INPUT "Please enter a number from 1 through 10:  ", userNum%
PRINT

WHILE userNum% <= 12
    PRINT "The current value of userNum% is"; userNum%
    SOUND ((userNum% * 30) + 300), .5
    userNum% = userNum% + 1
WEND

PRINT
INPUT "Press Return to continue…", dummy$
```

FIGURE6-8.
WHILE Loop: using a WHILE loop with a counter.

2. Run. the program. Your Output window will look something like

```
   Please enter a number from 1 through 10: 9

      The current value of userNum% is 9
      The current value of userNum% is 10
      The current value of userNum% is 11
      The current value of userNum% is 12
```

Nesting WHILE Loops
Just as you can nest FOR statements, you can place one WHILE statement inside another. Each WHILE statement has its own condition and closing WEND statement; the two conditions need not be related to each other.

```
    Here is the syntax for a nested WHILE loop:
  WHILE condition
      WHILE condition
          statement for inner WHILE loop
  WEND
WEND
```

The inner WHILE loop serves as one of the statements of the outer WHILE statement.

• NOTE: To prevent an infinite loop,
Remember to ensure that both the inner condition and the outer condition eventually evaluate as false.

NESTING DIFFERENT KINDS OF LOOPS
So far we've discussed nesting loops of the same kind - a FOR loop inside another FOR loop and a WHILE loop inside another WHILE loop.

You can mix the kinds of loops, too, especially when that will make your program more efficient.

The following program nests a FOR loop inside a WHILE loop.

The WHILE loop lets you execute the FOR loop as many times as you want to without having to start the program each time.

Practice:
Nesting different kinds of loops
1. Load and compile the Nested Loops program (Figure 6-9) from disk.

```
' Nested Loops
' This program demonstrates nesting different kinds of loops.

CLS

numSounds% = 1       ' ensure that WHILE statement doesn't fail the first time

WHILE numSounds% <> 0    ' loop until user enters a zero
    PRINT "How many tones do you want to hear?"
    INPUT "(Enter 0 to end)  ", numSounds%
    PRINT

       FOR i% = 1 TO numSounds%
        SOUND (i% * 100), 1
    NEXT i%
WEND
```

FIGURE 6-9.

2. Run the program. Your Output window will look something like

```
    How many tones do you want to hear?
    (Enter 0 to end) 5

    How many tones do you want to hear?
    (Enter 0 to end) 20

    How many tones do you want to hear?
    (Enter 0 to end) O
```

PRACTICAL USES FOR AC/QuickBasic LOOPS
Now that you've been introduced to loops, it's time to see what they can really do for you.
The following examples present some practical and fun applications for loops.

Using a Loop to Collect Information
Let's say that you want to automate your monthly budget planning.
The philosophy behind budgeting is straightforward: You subtract monthly expenses from monthly income.

Using the tools you've learned about so far, you could write a AC/QuickBasic program to help you do this.
You might use an INPUT statement to get the starting amount and then another for each expenditure.

After some calculations, you could have the AC/QuickBasic Interpreter/Compiler print out how much money you'd have left. The program shown in Figure 6-10 is an example of such a program.

```
' This program helps you figure out how much money is
' left over after you've paid your monthly bills.

CLS

PRINT· "Budget Calculator"
PRINT
INPUT "Enter your total income for this month: $", total!
PRINT

INPUT "Enter expense# 1: $". expense1!
total! = total !-expense1!
INPUT "Enter expense# 2: $", expense2!
total! = total !-expense2!
INPUT "Enter expense # 3: $" J expense3!
total! =total !-expense3!
INPUT "Enter expense # 4: $"' expense4!
total! = total !-expense4!
PRINT
PRINT "You have $"; total!; "left over this month."
```

FIGURE 6-10. A sample budgeting program.

Which Loop Should You Choose?
Keep the following points in mind as you decide which type of loop to use in a program:

• Use a FOR loop when you want to execute a block of statements a specific number of times.
• Use a WHILE loop to execute statements based on the value of a condition.

This program would work fine, but it does have limitations:

• Because your number of expenditures might change from month to month,
  You have to alter the program each time you run it.
• Each expenditure you enter must be immediately subtracted from the running total.

  This isn't a problem, but it is a lot of repetitive calculation.
  Consider how long this program would be if you had 20 or more expenses each month!

Programs with this degree of repetition are ideal candidates for a loop.

The Monthly Budget program in Figure 6-11 does the same job as the program in Figure 6-10 but offers some advantages:

• It's a shorter program.
• It accommodates a changing number of expenditures by asking you for the total number of bills you plan to enter.
• It uses a loop both to ask you for the bill amount and to subtract the amount from the running total.

```
' Monthly Budget
' This program helps you figure out how much money is
'   left over after you've paid your monthly bills.

CLS

PRINT "Budget Calculator"
PRINT
INPUT "Enter your total income for this month:  $", total!
PRINT
INPUT "How many bills will you have this month?  ", bills%

FOR i% = 1 TO bills%
    PRINT "Enter expense #"; i%;
    INPUT ":  $", thisExpense!
    total! = total! - thisExpense!
NEXT i%

PRINT
PRINT "You have $"; total!; "left over this month."

PRINT
INPUT "Press Return to continue…", dummy$
```

Using Loops with Random Numbers
Dice. Keno. Bingo. Lottery. They all make use of random numbers - numbers that occur in no predictable order.
We'll discuss three of the tools that AC/QuickBasic provides for creating random numbers: RND, RANDOMIZE TIMER, and INT.

Creating random numbers
The RND function instructs the AC/QuickBasic Interpreter/Compiler to return a random number to your program. (The number will be a single-precision floating-point number between 0 and 1.)

Because RND is a function, you must use it within a AC/QuickBasic statement.

Here's the syntax for the simplest form of the RND function:

```
    RND
```

By itself, RND returns the same series of numbers whenever you run your program. These are not truly random numbers because the series repeats and is therefore predictable. To create a different series of numbers each time you run your program, use RANDOMIZE TIMER as one of the first statements in your program. The syntax for the RANDOMIZE TIMER statement is as simple as that for the RND function:

```
RANDOMIZE TIMER
```


Practice:
Creating random numbers
The Random Numbers 1 program (Figure 6-12) uses the RANDOMIZE TIMER statement and a FOR loop containing the RND function to create six random numbers.

To create more random numbers, simply change the 6 in the FOR statement to a higher value.

1. Load and compile the Random Numbers 1 program from disk.

```
' Random Numbers 1
' This program generates random numbers between 0 and 1.

CLS

RANDOMIZE TIMER

FOR i% = 1 TO 6
    PRINT RND
NEXT i%

PRINT
INPUT "Press Return to continue…", dummy$
```

FIGURE 6-12.
Random Numbers 1: creating random numbers.

2. Run the program. Your Output window will look something like

```
   .2494013473385104
   .3081127713528831
   3.6702454735834010-02
   .5742615837592013
   .479102663966725
   .149526520761561 7
```

The numbers you see will undoubtedly be different from these. Remember, these are just random numbers.

Customizing the results of RND
Because numbers between 0 and 1 might not always meet your needs, you can use AC/QuickBasic's mathematical operators to change the size of RND's results.

Practice:
Creating larger random numbers
The Random Numbers 2 program (Figure 6-13 on the next page) is identical to Random Numbers l, but it multiplies the result of RND by 100, creating values that range between 0 and 100.

1. Load the Random Numbers 2 program from disk.

```
' Random Numbers 2
' This program generates random numbers between 0 and 100.

CLS

RANDOMIZE TIMER

FOR i% = 1 TO 6
    PRINT RND * 100
NEXT i%

PRINT
INPUT "Press Return to continue…", dummy$
```

FIGURE 6-13.
Random Numbers 2: creating larger random numbers.

2. Run the program. Your Output window will look something like

```
    85.67621396720176
    42.35302294483902
    97.795372984590
    48.80183662881284
    42.533066197375
    4.83424376293510
```

Creating random integers
If you'd like to create random numbers that are integers (numbers without decimal points), use the INT function along with the RND function.

INT discards the fractional portion of a floating-point number, leaving only the integer portion.

Here's the syntax for the INT function:

INT(number)

number is the floating-point number you want to convert to an integer.

As with all functions that return integers, the integer result of INT must be assigned either to a statement that accepts integer values or to an integer variable.

The following practice session demonstrates how to use the INT function with RND to create random integer numbers.

Practice:
Creating a guess-a-number game
Random integers are ideal for guessing games. The program on the next page uses the INT function and a WHILE loop to this.

Very Small and Very Large Numbers in AC/QuickBasic
The values produced by the AC/QuickBasic Interpreter/Compiler's RND function
Are always greater than 0 and less than 1.
Sometimes the values are very small.

When AC/QuickBasic tries to print a value that's too small or too large to display without using a lot of digits, it switches to AC/QuickBasic's version of exponential notation. To put it simply, when you see a D in a number, the decimal point was shifted from its actual location. The number after the D indicates in which direction and by how many places the decimal point was moved. If that number is positive, the decimal point actually resides to the right of its displayed location; if the number is negative, the - decimal point belongs to the left of its displayed location.

For example, in the sample output from Random Numbers 1 you can see the number 3.6702454735834O1D-02.
The D-02 means that the decimal point actually belongs two places to the left. In other words, you'd normally write this long number as .036702454735834O1. '

So why not simply display the number as it usually appears?
Often the number won't fit neatly on the screen. Take the number 4.136111 D+28. If printed in its entirety, it would look like this: 4136111000000000000000000000

Here is the same number in standard scientific notation:

4.136111x10^28

1. Load and compile the Guess A Number program (Figure 6-14) from disk.

```
' Guess A Number
' This program is a guess-a-number game.  The program generates a
'   random number and asks the user to guess what it is.  After the
'   user has guessed the number, the program displays the number
'   of guesses made.

CLS

PRINT "Guess-a-number Game"
PRINT
PRINT "I'm thinking of a number between 1 and 100."
PRINT "Can you guess what it is?"
PRINT

RANDOMIZE TIMER

randNum% = INT(RND * 100) + 1   ' generate random number
numGuesses% = 0                 ' start with a clean slate

WHILE guess% <> randNum%
    INPUT "What is your guess?  ", guess%
    IF guess% = randNum% THEN PRINT "That's right!!!"
    IF guess% < randNum% THEN PRINT "Try a bigger number!"
    IF guess% > randNum% THEN PRINT "Try a smaller number!"

    numGuesses% = numGuesses% + 1        ' chalk up one guess
    PRINT                                ' print a blank line
WEND

PRINT "You guessed the number in"; numGuesses%; "tries!"
INPUT "Press Return to continue…", dummy$
```

FIGURE 6-14.
Guess A Number: a guessing game.

2. Run the program. Your Output window will look something like the output shown at the top of the opposite page.

```
          Guess-a-number Game

          I'm thinking of a number between 1 and 100.
          Can you guess what it is?

          What is your guess? 50
          Try a bigger number!

          What is your guess? 99
          Try a smaller number!

          What is your guess? 65
          Try a bigger number!

          What is your guess? 71
          That's right!!!

          You guessed the number in 4 tries!
          Press Return to continue ...
```

Practice:
Writing a computer simulation
Let's say that you want to write a program that calculates how many times, out of 100 rolls of two dice, the number 7 comes up. The Dice Simulator program (Figure 6-15) does just that, and the RND function comes in handy again.

```
' Dice Simulator
' This program asks the user to enter how many times the AC/QuickBasic
'   Interpreter/Compiler should "roll" two dice and then calculates how many
'   times the number 7 comes up.

CLS

RANDOMIZE TIMER

PRINT "Dice-Simulation Program"
PRINT
numSeven% = 0       ' start with 7 counter equal to zero

INPUT "How many times should I roll the dice?  ", rolls%
PRINT
PRINT "Working…"  ' ensure that the user knows nothing is wrong

FOR i% = 1 TO rolls%
    die1% = INT(RND * 6) + 1      ' "roll" the first die
    die2% = INT(RND * 6) + 1      ' "roll" the second die
    IF die1% + die2% = 7 THEN numSeven% = numSeven% + 1
NEXT i%

PRINT
PRINT "Out of"; rolls%; "rolls, the number 7 came"
PRINT "up"; numSeven%; "times."

PRINT
INPUT "Press Return to continue…", dummy$
```

2. Run the program. You should see output something like this:

```
          Dice-Simulation Program

          How many times should I roll the dice? 50

          Working ...

          Out of 50 rolls, the number 7 came
          up 10 times.

          Press Return to continue ...
```

SUMMARY
Loops let you create powerful programs that repeat a task a specific number of times or until a condition is met. In this chapter, you've learned about FOR and WHILE loops. Combine this knowledge with the TEXT-SIZE, TEXTFONT, and SOUND statements, the RND function, and the skills you've picked up in previous chapters, and you're ready to take on some impressive programming projects. The following chapters will get you started.

QUESTIONS AND EXERCISES
1. What is the purpose of the counter variable in a FOR loop?
2. What types of values can be assigned to the start and end elements of a FOR loop?
3. What number would you specify as an argument to the TEXT-SIZE statement if you wanted the characters displayed in the Output window via a PRINT statement to be roughly 1 inch tall?
4. What does the SOUND statement do?
5. What is a nested loop?
6. What is an infinite loop? How do you stop it?
7. Under what circumstances might you use a FOR loop? Under what circumstances would you use a WHILE loop?
8. Write a program that keeps track of the amount of money spent on gasoline in a week. Use a FOR loop to collect the dollars and cents spent, and keep a running total in a single-precision floating-point variable.
9. Write a program that prompts the user for valid frequency and duration values and then plays them back with the SOUND statement. Use a WHILE loop to collect information until the user types in -999 as the frequency.
10. Write a program that rolls one simulated die 10 times. Print the value of the die after each roll, and display the message Nice Roll! if the die shows 6.

If you've followed the examples and done the exercises in the book so far, you've written relatively short programs - none has been longer than 50 lines. But now that you've learned the basics of AC/QuickBasic, you're ready to write longer programs. In this chapter, you'll learn some techniques that allow you to write longer programs with a minimum of time and effort.

You'll learn about two program structures that handle repetitive tasks and make your programs shorter and easier to read: subprograms and user-defined functions. By the end of this chapter, you'll have all the tools you need to write compact, well-organized programs.

WHY SUBPROGRAMS?
Suppose you want to write a program that prints the lyrics of the traditional American song "Clementine." Using the skills you've learned so far, you'd write the song with a lot of PRINT statements (and a few INPUT statements to make the display pause), as in the Clementine program shown in Figure 7-1.

Running the Clementine program
Load and compile the Clementine program (Figure 7-1) from the Chapter 7 folder on disk and run it.

Clementine is quite straightforward:
The program prints each verse and each chorus of the song pausing after each chorus so that you have time to read the lyrics (and sing along!) before they scroll out of sight. Notice that the Clementine program contains a chorus that is over and over without modification. Such repetitive text not only takes time and effort to type in, but it also clutters your program listing, making it more difficult to work with. Is there an easier way to code a program that has repetitive parts? The answer is Yes!

```
' Clementine
' This program displays the lyrics of the folk song "Clementine."

CLS
PRINT "------------------------- Clementine -------------------------"
PRINT

PRINT "In a cavern, in a canyon,"                  ' first verse
PRINT "Excavating for a mine,"
PRINT "Dwelt a miner, forty-niner,"
PRINT "And his daughter, Clementine."

PRINT
PRINT "Oh my darling, oh my darling,"          ' chorus
PRINT "Oh my darling, Clementine,"
PRINT "You are lost and gone forever,"
PRINT "Dreadful sorry, Clementine."
PRINT

INPUT "Press Return for more…", dummy$    ' pause
PRINT

PRINT "Light she was and like a fairy,"        ' second verse
PRINT "And her shoes were number nine;"
PRINT "Herring boxes without topses,"
PRINT "Sandals were for Clementine."

PRINT
PRINT "Oh my darling, oh my darling,"          ' chorus
PRINT "Oh my darling, Clementine,"
PRINT "You are lost and gone forever,"
PRINT "Dreadful sorry, Clementine."
PRINT

INPUT "Press Return for more…", dummy$    ' pause
PRINT

PRINT "Drove she ducklings to the water,"    ' third verse
PRINT "Every morning just at nine;"
PRINT "Hit her foot against a splinter,"
PRINT "Fell into the foaming brine."

PRINT
PRINT "Oh my darling, oh my darling,"          ' chorus
PRINT "Oh my darling, Clementine,"
PRINT "You are lost and gone forever,"
PRINT "Dreadful sorry, Clementine."
PRINT

INPUT "Press Return for more…", dummy$    ' pause
PRINT

PRINT "Ruby lips above the water,"              ' fourth verse
PRINT "Blowing bubbles soft and fine;"
PRINT "But alas, he was no swimmer,"
PRINT "So he lost his Clementine."
```

```
PRINT
PRINT "Oh my darling, oh my darling,"          ' chorus
PRINT "Oh my darling, Clementine,"
PRINT "You are lost and gone forever,"
PRINT "Dreadful sorry, Clementine."
PRINT

INPUT "Press Return for more…", dummy$    ' pause
PRINT

PRINT "Then the miner, forty-niner,"           ' fifth verse
PRINT "Soon began to peak and pine;"
PRINT "Thought he oughter join his daughter,"
PRINT "Now he's with his Clementine."

PRINT
PRINT "Oh my darling, oh my darling,"          ' chorus
PRINT "Oh my darling, Clementine,"
PRINT "You are lost and gone forever,"
PRINT "Dreadful sorry, Clementine."
PRINT

INPUT "Press Return for more…", dummy$    ' pause
```

The Subprogram Advantage
AC/QuickBasic provides a programming structure called a subprogram that lets you type a block of statements, assign a name to the block, and then call the block by name whenever you want your program to execute it.

Subprograms offer you the following advantages:

• Subprograms eliminate repeated lines.
  You can define a subprogram only once and have your program execute it any number of times.

• Subprograms make programs easier to read.
  You'll find a program divided into a collection of smaller parts easier to take apart and understand.

• Subprograms simplify program development.
  You'll find a program you've separated into logical units easier to design, write, and debug.
  Plus, if you're writing the program with a friend, you can exchange subprograms instead of the entire program.

• Subprograms can be reused in other programs. You can incorporate your subprograms into other programming projects.

• Subprograms extend the AC/QuickBasic language.
  You can often write a subprogram to do a task that you couldn't accomplish directly with built-in AC/QuickBasic statements and functions.

CREATING SUBPROGRAMS
A subprogram is a block of code between SUB and ENDSUB statements. You can have your program call a subprogram as often as you like. When a subprogram finishes running, control returns to the statement that follows the subprogram call in the main program. In AC/QuickBasic for the Apple][gs, you put subprograms at the bottom of the program to keep separate from the main program above.

Syntax of a Subprogram
Here is the syntax for a subprogram:

```
SUB SubprogramName (parameterList) STATIC
    subprogram statements
END SUB
```

• SUB is the AC/QuickBasic statement that marks the beginning of the subprogram definition.

• SubprogramName is the name of the subprogram, which can be up through 40 characters long and is the name by which the main program or another subprogram will call the subprogram.
  The subprogram's name can't be an AC/QuickBasic keyword or be the same as any variable name or function name you use in your program.

• (parameterList) is an optional list of variables that are separated by commas.
  (See "Passing Arguments to a Subprogram" later in this chapter.)
  If you use parameterList, you must enclose it in parentheses.

• STATIC is a keyword indicating that the variables you declare in the subprogram will retain their values between subprogram calls. You must use the STATIC keyword in all your subprogram definitions.

• subprogram statements is the working part of the subprogram.
  You can use almost any AC/QuickBasic statement in a subprogram.
  Note that in a subprogram you can use a variable that has the same name as a variable in your main program.
  A variable declared in a subprogram is valid in that subprogram only - it doesn't affect a variable with the same name elsewhere, in either the main program or in another subprogram.
  (See "Using Variables with Subprograms" later in this chapter.)

• END SUB is the AC/QuickBasic statement that marks the end of the subprogram definition.

To see how these elements work together, examine the following subprogram we've named Chorus. Every time Chorus is executed, it prints the four-line chorus to "Clementine" and the prompt that asks you to Press Return for more ...

```
SUB Chorus STATIC
' The Chorus subprogram prints the chorus of the song "Clementine"
' and waits for the user to press Return.

PRINT
PRINT "Oh my darling, oh my darling," 'chorus
PRINT "Oh my darling, Clementine,"
PRINT "You are lost and gone forever."
PRINT "Dreadful sorry. Clementine."
PRINT

INPUT "Press Return for more ... ". dummy$ ' pause
PRINT

END SUB
```

Calling a Subprogram
After you've added a subprogram to the bottom of your program, you can execute (call) it with a CALL statement in your main program. Here's the syntax for a CALL statement:

```
CALL SubprogramName (argumentlist)
```

SubprogramName is the name of the subprogram that is to be executed, and argumentList is an optional list of variables to be passed to the subprogram (more about this later). This is where space savings come in - you can execute an entire block of subprogram statements with one subprogram call. And you can call a subprogram as often as you like from anywhere within the main program or another subprogram. The Clementine 2 program (Figure 7-2 on the next page) shows how a program containing a subprogram is put together. The main program contains five calls to the Chorus subprogram, each of which displays the chorus and then waits for the user to press Return. The main program ends with an END statement, an instruction that tells AC/QuickBasic the last line in the program has been reached and execution should stop. Although the AC/QuickBasic Interpreter/Compiler doesn't require the END statement, END is a useful visual clue that indicates exactly where the main program ends and the subprogram section begins. We'll use END throughout the book when a program contains one or more subprograms.

The Chorus subprogram itself appears below the main program, bracketed by SUB and ENDSUB statements. Notice that the Chorus subprogram name begins with an uppercase letter - we'll use this convention throughout the book to distinguish subprogram names from lowercase variable names and uppercase AC/QuickBasic statements and functions. If you compare Clementine 2 to Clementine, you'll find that Clementine 2 is 21 lines shorter than Clementine and much easier to follow. Most programmers find that subprograms justify their memory overhead (the SUB, ENDSUB, and CALL statements) when a block of code is three or more lines long and is used three or more times in a program. That might be a good rule of thumb for you to adopt in your own programming.

```
' Clementine 2
' This program displays the lyrics of the folk song "Clementine."

CLS
PRINT "------------------------- Clementine -------------------------"
PRINT

PRINT "In a cavern, in a canyon,"                 ' first verse
PRINT "Excavating for a mine,"
PRINT "Dwelt a miner, forty-niner,"
PRINT "And his daughter, Clementine."
CALL Chorus                                            ' call Chorus subprogram

PRINT "Light she was and like a fairy,"         ' second verse
PRINT "And her shoes were number nine;"
PRINT "Herring boxes without topses,"
PRINT "Sandals were for Clementine."
CALL Chorus                                            ' call Chorus subprogram

PRINT "Drove she ducklings to the water,"    ' third verse
PRINT "Every morning just at nine;"
PRINT "Hit her foot against a splinter,"
PRINT "Fell into the foaming brine."
CALL Chorus                                            ' call Chorus subprogram

PRINT "Ruby lips above the water,"              ' fourth verse
PRINT "Blowing bubbles soft and fine;"
PRINT "But alas, he was no swimmer,"
PRINT "So he lost his Clementine."
CALL Chorus                                            ' call Chorus subprogram

PRINT "Then the miner, forty-niner,"           ' fifth verse
PRINT "Soon began to peak and pine;"
PRINT "Thought he oughter join his daughter,"
PRINT "Now he's with his Clementine."
CALL Chorus                                            ' call Chorus subprogram

END
```

```
SUB Chorus STATIC
' The Chorus subprogram prints the chorus of the song "Clementine"
'   and waits for the user to press Return.

PRINT
PRINT "Oh my darling, oh my darling,"          ' chorus
PRINT "Oh my darling, Clementine,"
PRINT "You are lost and gone forever,"
PRINT "Dreadful sorry, Clementine."
PRINT

INPUT "Press Return for more…", dummy$    ' pause
PRINT

END SUB
```

USING VARIABLES WITH SUBPROGRAMS
The programs you've written so far have used only a handful of variables, but when you write larger programs you might need to use lots of them. To help you keep track of large numbers of variables, AC/QuickBasic enforces some special rules that deal with variables in a main program and in subprograms. We'll discuss those rules in this section as we introduce the concept of local vs. shared variables and look at how variables are passed to subprograms.

Local Variables
A local variable is valid only within the module (main program or sub-program) in which it is declared:
A local variable is not affected by changes elsewhere in the program. This means that a variable used in the main program module of a AC/QuickBasic program won't be inadvertently altered by a variable with the same name in one of the subprogram modules. The opposite is also true: A variable used locally within a subprogram won't be updated when a change is made to a variable of the same name in the main program or another subprogram.
The point is that you can use the same name for variables in different parts of your program and these variables won't interfere with each other because each is local to the module in which it is declared. You can worry less about using the same variable name twice in a program and concentrate on writing general-purpose subprograms that you can use again later.

Declaring local variables
Variables are local by default, so you don't have to use any special statement to declare a local variable in your main program or in a subprogram. Actually, you're an old hand at declaring local variables - you've been doing it all along!

Practice:
Using local variables
The Local Variable program (Figure 7-3) demonstrates the exclusivity of a local variable in the main program and a local variable of the same name in a subprogram. Both variables are named game$ and both are string variables, but when the value of one variable changes, the value of the other does not.

Load and compile the Local Variable program from disk and run it. You'll see the following output:

```
    In the main program, game$ = Chess
    In the AddGame subprogram, game$ = and backgammon
    Back in the main program, game$ = Chess
```

As you can see, changes to the variable game$ in the AddGame sub-program do not affect the isolated game$ variable in the main program.

```
' Local Variable
' This program demonstrates the use of two local variables.
CLS
game$ = "Chess"             ' initialize game$ with value "Chess"

PRINT "In the main program, game$ = "; game$          ' display in main prog.

CALL AddGame                                                ' display in subprogram

PRINT "Back in the main program, game$ = "; game$ ' display in main prog.

PRINT
INPUT "Press Return to continue…", dummy$

END

SUB AddGame STATIC

game$ = game$ + " and backgammon"

PRINT "In the AddGame subprogram, game$ = "; game$

END SUB
```

Figure 7-3. Local Variable: a program that demonstrates the use of two isolated local variables.

Shared Variables
A shared variable provides a mechanism for exchanging a local variable between a subprogram and the main program. Such an exchange is useful when the main program needs to pass information to a subprogram so that the subprogram can begin a calculation or when the subprogram needs to return the results of its work to the main program.
(We'll describe another technique for sharing variables, called passing arguments, in the next section.)

You declare shared variables by using the SHARED statement in a subprogram. Here's the syntax for a SHARED statement:

```
SHARED variableList
```

variableList is a list of local variables to be shared that are separated by commas.
If you use the SHARED statement, it must be the first statement inside the relevant subprogram.

Practice:
Using shared variables
The Shared Variable program (Figure 7-4) shows how a local variable named game$ can be passed with the SHARED statement. The game$ variable is displayed in the main program; passed to the AddGame sub-program, where it is modified and displayed; and then returned to the main program, where it is displayed a third time.

Load and compile the Shared Variable program from disk and run it. You'll see the following output: '

```
In the main program, game$ = Chess
In the AddGame subprogram, game$ = Chess and backgammon
Back in the main program, game$ = Chess an&-backgammon
```

Note that the contents of the game$ variable are available by default only to the main program - other subprograms must have their own SHARED statements if they are to take part in this data exchange.

```
' Shared Variable
' This program demonstrates the use of a shared variable.

CLS
game$ = "Chess"               ' initialize game$ with value "Chess"

PRINT "In the main program, game$ = "; game$          ' display in main prog.

CALL AddGame                                                ' display in subprogram

PRINT "Back in the main program, game$ = "; game$  ' display in main prog.

PRINT
INPUT "Press Return to continue…", dummy$

END

SUB AddGame STATIC
SHARED game$                ' share game$ with the main program

game$ = game$ + " and backgammon"
PRINT "In the AddGame subprogram, game$ = "; game$

END SUB
```

Figure 7-4.
Shared Variable: a program that demonstrates the use of a shared variable in a program.

Passing Arguments to a Subprogram
You can make any number of variables accessible by sharing them with the SHARED statement. Although it's convenient, this method of sharing variables increases the chances that you'll lose track of which variables are shared and which are local.

To help you track shared variables, AC/QuickBasic provides a more visual method for passing information (both variables and expressions) to only the subprograms you want the information to go to.
This method of sharing is called passing arguments.

With this method, arguments passed to a subprogram from the main program or another subprogram are received by parameters, which are local variables within the subprogram. You can use these local variables exactly as you would any other local variables in the subprogram.

Arguments vs. parameters
Before we go any further, let's formalize the difference between the terms argument and parameter:

- An argument is a variable or expression that is passed to a sub-program.
  Argument names appear in a CALL statement in parentheses after the subprogram name.
  A collection of argument names is known as an argument list.

- A parameter is a variable that receives a value passed to a subprogram.
  Parameter names appear in a SUB statement and follow the rules that apply to standard data types.
  A collection of parameter names is known as a parameter list.

Each argument must have a corresponding parameter of the same type (but not necessarily of the same name) in the subprogram's parameter list. Figure 7-5 shows the relationship between an argument list and a parameter list.

Figure 7-6 shows some valid argument-parameter pairs.

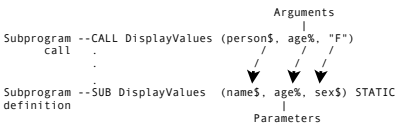


FIGURE 7-5.
The relationship between arguments and parameters.

Modifying arguments
Passing arguments to a subprogram is not a one-way street. Any value that a subprogram assigns to one of its parameters is passed back to the matching argument in the CALL statement when the subprogram has finished executing.

Let's use Figure 7-5 as an example: If the argument person$ in the main program contained the value Elisabeth at the time that the program called the subprogram DisplayValues, the subprogram parameter name$ would receive the value Elisabeth.

```
CALL Getinput (name$, number$, address$)

SUB Getlnput (person$, phone$, address$) STATIC

CALL TranslateToNorsk (pig$, dog$)

SUB TranslateToNorsk (gris$, hund$) STATIC

CALL PrintHeader (TIME$, DATE$, title$, pageNumber%)

SUB PrintHeader (currentTime$, currentDate$, title$, num%) STATIC

CALL MixedCall (testInt%, testString$, VALUE!, "Hello", 747)

SUB MixedCall (a%, b$, c! . d$, e%) STATIC
```

FIGURE 7-6.
Arguments and their matching parameters.

If the subprogram then assigned the value Vivienne to the parameter name$, the value Vivienne would be assigned to the argument person$ upon completion of the subprogram's execution.

Keep in mind, however, that you can't change the value of a number, a numeric expression, or a string literal (a string enclosed by double quotation marks). In Figure 7-5, for example, even if the DisplayValues sub-program modified the value of the parameter sex$, nothing would happen to the matching argument "F" because it is a string literal.

Practice:
Passing arguments to a subprogram
The Argument program (Figure 7-7 on the next page) demonstrates how two arguments are passed to a subprogram called AddInterest. The monthName$ parameter in the subprogram receives the month$ argument, and the amount! parameter receives the balance! argument.

Then the sub-program AddInterest changes the month, multiplies the value of amount! by 1.05, and returns both values to the main program.

```
' Argument
' This program demonstrates passing arguments to a subprogram.

month$ = "January"            ' initialize month$ with value "January"
balance! = 1500                 ' initialize balance! with value 1500

CLS

PRINT "Before subprogram:"       ' display original values
PRINT "  month$ = "; month$
PRINT "  balance! ="; balance!
PRINT

CALL AddInterest (month$, balance!)     ' call subprogram to modify values
```



```
PRINT "After subprogram:"        ' display modified values
PRINT "  month$ = "; month$
PRINT "  balance! ="; balance!

PRINT
INPUT "Press Return to continue…", dummy$

END

SUB AddInterest (monthName$, amount!) STATIC

monthName$ = "February"           ' change name of month
amount! = amount! * 1.05            ' add 5% to amount

END SUB
```

Figure 7-7.
Argument: a program that demonstrates how to pass arguments to a subprogram.

Load the Argument program from disk and run it. You'll see this output:

```
Before subprogram:
  month$ = January
  balance! = 1500

After subprogram:
  month$ = February
  balance! = 1575
```

Both month$ and balance! were modified without resorting to the use of hard-to-track shared variables.

CREATING USER-DEFINED FUNCTIONS
A user-defined function is a one-line expression you define with a DEF FN statement. User-defined functions follow the same general rules that sub-programs do, with one important exception: A function, whether built-in or user-defined, performs a task and returns a value to the main program or calling subprogram. You use a user-defined function in the same way that you use a AC/QuickBasic built-in function- you include it in a AC/QuickBASIC statement.
In this section you'll learn how to create your own user-defined functions with AC/QuickBasic.

Syntax for a User-defined Function
Here's the syntax for a user-defined function:

```
DEF FNFunctionName(parameterList) = functionDefinition
```

• DEF FN is the AC/QuickBasic statement that marks the beginning of the function definition.

• FunctionName is the name of the function and can end with a type declaration character (just as a variable name can).
  The function name can be up through 40 characters long and is the name used to call the function.
  The function name can't be a AC/QuickBasic keyword or the same as any variable name, subprogram name, or other function in your program.

• (parameterList) is an optional list of variables (the variables separated by commas) used in functionDefinition.
  The list of variables is enclosed within parentheses.

• functionDefinition is an expression (typically a mathematical formula) that performs the operation of the function.
  The expression is limited to one logical line; that is, the entire definition must fit on one program line.
  You can use both variables in the function's parameter list and local variables in the main program in functionDefinition.

Calling a User-defined Function
User-defined function calls differ from subprogram calls in that a user-defined function call cannot stand alone. Rather, the result of the function must be assigned to a AC/QuickBasic statement or to a variable of the same type as the function's result. The following user-defined function, for example, accepts three integer parameters and returns a single integer value:

```
DEF FNSumOfTerms%(a%, b%, c%)= a% + b% + c%
```

A call to the FNSumOfTerms% function must include three integer variable or integer expression arguments. The function's result can be assigned to an integer variable or used in a AC/QuickBasic statement, such as PRINT, that accepts an integer value as an argument. A call to FNSumOfTerms%, with the result assigned to an integer variable, might look like this:

```
number%= FNSumOfTerms%(10 + 5, 20, cost%)
```

A call to FNSumOfTerms% in a PRINT statement might look like this:

```
PRINT FNSumOfTerms%(10 + 5, 20, cost%)
```

A user-defined function call always begins with the letters FN and always returns a single value in one of the five AC/QuickBasic data types: integer, long integer, single-precision floating-point, double-precision floating-point, or string.

Positioning a User-defined Function
A user-defined function can appear anywhere in the main program, but it must be defined before it is used.
By convention, user-defined functions appear at the top of the program listing, immediately after any introductory comments. A user-defined function cannot be defined within a subprogram, but it can be called by a subprogram - any subprogram - or by the main program.

Practice:
Using a user-defined function
The Conversion program (Figure 7-8) demonstrates the declaration and call of two user-defined functions that return single-precision floating-point values. Conversion converts ten measurements expressed in centimetres to inches and then to feet and displays them with the PRINT statement.

```
' Conversion
' A program that converts centimetres to inches and then to feet.

' A function to convert centimeters to inches.
DEF FNCentToInch!(cent!) = cent! / 2.54

' A function to round numbers to two decimal places.
DEF FNRound!(num!) = INT(100 * num! + .5) / 100

CLS

PRINT "This program converts centimetres to inches and then to feet."
PRINT
PRINT "Centimetres", "Inches", "Feet"
PRINT

FOR i% = 10 TO 100 STEP 10              ' do 10 conversions
    inches! = FNCentToInch!(i%)             ' convert centimeters to inches
    PRINT i%, FNRound!(inches!), FNRound(inches! / 12)    ' print results
NEXT i%

PRINT
INPUT "Press Return to continue…", dummy$     ' pause screen
```

Figure 7-8.
Conversion: a program that uses functions to convert centimetres to inches and then to feet.

Load and compile the Conversion program from disk and run it. You'll see the following output:

```
This program converts centimetres to inches and then to feet.

Centimetres      Inches         Feet
10                3.94           .33
20                7.87           .66
30               11.81           .98
40               15.75          1.31
50               19.69          1.64
60               23.62          1.97
70               27.56          2.3
80               31.5           2.62
90               35.43          2.95
100              39.37          3.28
```

ORGANIZING YOUR PROGRAM
Figure 7-9 shows the order in a listing of the three types of program modules we've discussed in this chapter: the main program, subprograms, and user-defined functions. You can have any number of subprograms and functions in your program or none at all. Every program, however, must have a main program module that controls the general flow of the program and calls the subprograms and functions. Remember: No matter how many subprograms and functions a program contains, it is still a single AC/QuickBasic program.

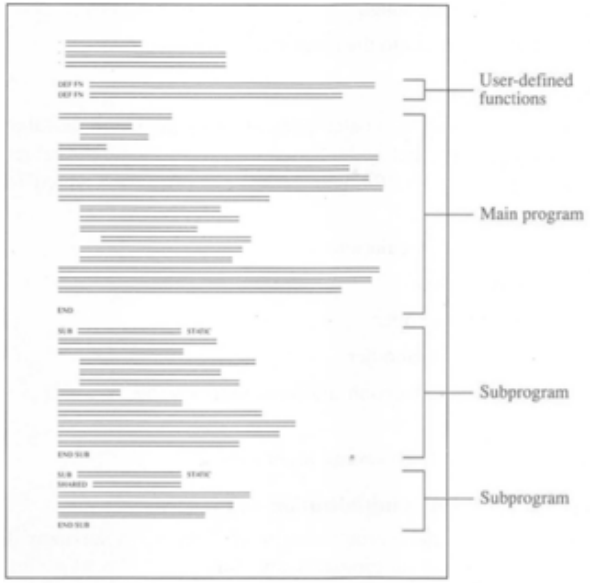
Although it might seem like a little extra work at first, learning to divide your programs into these organizational units will pay off when you start to write longer programs. Debugging, the process of finding and fixing programming errors, will also take more time as your programs get longer, so programming in modules is a good idea.

Which Kind of Program Module Should You Use?
You might still be wondering which kind of program module is best for which situation.

Here are some general guidelines you can follow as you plan your programs.

Figure 7-9.
An illustration of AC/QuickBasic program module organization.

A subprogram is a miniprogram
Think of a subprogram as a small, self-contained program. A subprogram should perform an important task for one program and yet be general purpose enough to be used in other programming projects. A subprogram is ideal for a block of code that will be used more than once in a program. Subprograms are also a good idea for programs that have many variables to manage. The following tasks are often best suited to subprograms:



- Getting input from the user
- Displaying information on the screen
- Processing several numeric values or strings
- Drawing graphic shapes and designs
- Playing musical notes or songs
- Returning multiple values to the main program

A user-defined function excels in calculating and returning a single value to the main program.
A user-defined function won't return multiple values or execute general tasks.
  The following tasks are often best suited to functions:

- Performing a numeric calculation
- Returning a string value
- Generating a random number
- Converting one value to another
- Evaluating a logical expression and returning a value of either true or false
- Calculating one result from several arguments

The main program handles initialization and control
What tasks are left for the main program to handle?
Actually, not many if you make thorough use of subprograms and functions.
The following tasks are often best suited to the main program:

- Introductory comments and explanations
- Initialization of key variables
- Program code that is executed only once
- Flow-control structures that determine the path of program execution

SUMMARY
In this chapter you've learned about subprograms and functions - two programming structures that save you from repetition and extend the AC/QuickBasic language. You define subprograms with SUB and ENDSUB statements, and you define user-defined functions with DEF FN statements. Both structures can be called from either the main program

Creating Your Own Subprograms and Functions

Learn AC/QuickBasic For the Apple][gs NOW

If you've followed the examples and done the exercises in the book so far, you've written relatively short programs - none has been longer than 50 lines. But now that you've learned the basics of AC/QuickBasic, you're ready to write longer programs. In this chapter, you'll learn some techniques that allow you to write longer programs with a minimum of time and effort.

You'll learn about two program structures that handle repetitive tasks and make your programs shorter and easier to read: subprograms and user-defined functions. By the end of this chapter, you'll have all the tools you need to write compact, well-organized programs.

WHY SUBPROGRAMS?

Suppose you want to write a program that prints the lyrics of the traditional American song "Clementine." Using the skills you've learned so far, you'd write the song with a lot of PRINT statements (and a few INPUT statements to make the display pause), as in the Clementine program shown in Figure 7-1.

Running the Clementine program

Load and compile the Clementine program (Figure 7-1) from the Chapter 7 folder on disk and run it.

Clementine is quite straightforward:

The program prints each verse and each chorus of the song pausing after each chorus so that you have time to read the lyrics (and sing along!) before they scroll out of sight. Notice that the Clementine program contains a chorus that is over and over without modification. Such repetitive text not only takes time and effort to type in, but it also clutters your program listing, making it more difficult to work with. Is there an easier way to code a program that has repetitive parts? The answer is Yes!

```
' Clementine
' This program displays the lyrics of the folk song "Clementine."

CLS
PRINT "------------------------- Clementine -------------------------"
PRINT

PRINT "In a cavern, in a canyon,"                  ' first verse
PRINT "Excavating for a mine,"
PRINT "Dwelt a miner, forty-niner,"
PRINT "And his daughter, Clementine."

PRINT
PRINT "Oh my darling, oh my darling,"          ' chorus
PRINT "Oh my darling, Clementine,"
PRINT "You are lost and gone forever,"
PRINT "Dreadful sorry, Clementine."
PRINT

INPUT "Press Return for more…", dummy$    ' pause
PRINT

PRINT "Light she was and like a fairy,"        ' second verse
PRINT "And her shoes were number nine;"
PRINT "Herring boxes without topses,"
PRINT "Sandals were for Clementine."

PRINT
PRINT "Oh my darling, oh my darling,"          ' chorus
PRINT "Oh my darling, Clementine,"
PRINT "You are lost and gone forever,"
PRINT "Dreadful sorry, Clementine."
PRINT

INPUT "Press Return for more…", dummy$    ' pause
PRINT

PRINT "Drove she ducklings to the water,"    ' third verse
PRINT "Every morning just at nine;"
PRINT "Hit her foot against a splinter,"
PRINT "Fell into the foaming brine."

PRINT
PRINT "Oh my darling, oh my darling,"          ' chorus
PRINT "Oh my darling, Clementine,"
PRINT "You are lost and gone forever,"
PRINT "Dreadful sorry, Clementine."
PRINT

INPUT "Press Return for more…", dummy$    ' pause
PRINT

PRINT "Ruby lips above the water,"              ' fourth verse
PRINT "Blowing bubbles soft and fine;"
PRINT "But alas, he was no swimmer,"
PRINT "So he lost his Clementine."
```



```
PRINT
PRINT "Oh my darling, oh my darling,"          ' chorus
PRINT "Oh my darling, Clementine,"
PRINT "You are lost and gone forever,"
PRINT "Dreadful sorry, Clementine."
PRINT

INPUT "Press Return for more…", dummy$    ' pause
PRINT

PRINT "Then the miner, forty-niner,"           ' fifth verse
PRINT "Soon began to peak and pine;"
PRINT "Thought he oughter join his daughter,"
PRINT "Now he's with his Clementine."

PRINT
PRINT "Oh my darling, oh my darling,"          ' chorus
PRINT "Oh my darling, Clementine,"
PRINT "You are lost and gone forever,"
PRINT "Dreadful sorry, Clementine."
PRINT

INPUT "Press Return for more…", dummy$    ' pause
```

The Subprogram Advantage

AC/QuickBasic provides a programming structure called a subprogram that lets you type a block of statements, assign a name to the block, and then call the block by name whenever you want your program to execute it.

Subprograms offer you the following advantages:

- Subprograms eliminate repeated lines.
  You can define a subprogram only once and have your program execute it any number of times.

- Subprograms make programs easier to read.
  You'll find a program divided into a collection of smaller parts easier to take apart and understand.

- Subprograms simplify program development.
  You'll find a program you've separated into logical units easier to design, write, and debug.
  Plus, if you're writing the program with a friend, you can exchange subprograms instead of the entire program.

- Subprograms can be reused in other programs. You can incorporate your subprograms into other programming projects.

- Subprograms extend the AC/QuickBasic language.
  You can often write a subprogram to do a task that you couldn't accomplish directly with built-in AC/QuickBasic statements and functions.

CREATING SUBPROGRAMS

A subprogram is a block of code between SUB and ENDSUB statements. You can have your program call a subprogram as often as you like. When a subprogram finishes running, control returns to the statement that follows the subprogram call in the main program. In AC/QuickBasic for the Apple][gs, you put subprograms at the bottom of the program to keep separate from the main program above.

Syntax of a Subprogram

Here is the syntax for a subprogram:

```
SUB SubprogramName (parameterList) STATIC
    subprogram statements
END SUB
```

- SUB is the AC/QuickBasic statement that marks the beginning of the subprogram definition.

- SubprogramName is the name of the subprogram, which can be up through 40 characters long and is the name by which the main program or another subprogram will call the subprogram.
  The subprogram's name can't be an AC/QuickBasic keyword or be the same as any variable name or function name you use in your program.

- (parameterList) is an optional list of variables that are separated by commas.
  (See "Passing Arguments to a Subprogram" later in this chapter.)
  If you use parameterList, you must enclose it in parentheses.

- STATIC is a keyword indicating that the variables you declare in the subprogram will retain their values between subprogram calls. You must use the STATIC keyword in all your subprogram definitions.

- subprogram statements is the working part of the subprogram.
  You can use almost any AC/QuickBasic statement in a subprogram.
  Note that in a subprogram you can use a variable that has the same name as a variable in your main program.
  A variable declared in a subprogram is valid in that subprogram only - it doesn't affect a variable with the same name elsewhere, in either the main program or in another subprogram.
  (See "Using Variables with Subprograms" later in this chapter.)

- END SUB is the AC/QuickBasic statement that marks the end of the subprogram definition.

To see how these elements work together, examine the following subprogram we've named Chorus. Every time Chorus is executed, it prints the four-line chorus to "Clementine" and the prompt that asks you to Press Return for more ...

```
SUB Chorus STATIC
' The Chorus subprogram prints the chorus of the song "Clementine"
' and waits for the user to press Return.

PRINT
PRINT "Oh my darling, oh my darling," 'chorus
PRINT "Oh my darling, Clementine,"
PRINT "You are lost and gone forever."
PRINT "Dreadful sorry. Clementine."
PRINT

INPUT "Press Return for more ... ". dummy$ ' pause
PRINT

END SUB
```

Calling a Subprogram
After you've added a subprogram to the bottom of your program, you can execute (call) it with a CALL statement in your main program. Here's the syntax for a CALL statement:

```
CALL SubprogramName (argumentlist)
```

SubprogramName is the name of the subprogram that is to be executed, and argumentList is an optional list of variables to be passed to the subprogram (more about this later). This is where space savings come in - you can execute an entire block of subprogram statements with one subprogram call. And you can call a subprogram as often as you like from anywhere within the main program or another subprogram. The Clementine 2 program (Figure 7-2 on the next page) shows how a program containing a subprogram is put together. The main program contains five calls to the Chorus subprogram, each of which displays the chorus and then waits for the user to press Return. The main program ends with an END statement, an instruction that tells AC/QuickBasic the last line in the program has been reached and execution should stop. Although the AC/QuickBasic Interpreter/Compiler doesn't require the END statement, END is a useful visual clue that indicates exactly where the main program ends and the subprogram section begins. We'll use END throughout the book when a program contains one or more subprograms.

The Chorus subprogram itself appears below the main program, bracketed by SUB and ENDSUB statements. Notice that the Chorus subprogram name begins with an uppercase letter - we'll use this convention throughout the book to distinguish subprogram names from lowercase variable names and uppercase AC/QuickBasic statements and functions. If you compare Clementine 2 to Clementine, you'll find that Clementine 2 is 21 lines shorter than Clementine and much easier to follow. Most programmers find that subprograms justify their memory overhead (the SUB, ENDSUB, and CALL statements) when a block of code is three or more lines long and is used three or more times in a program. That might be a good rule of thumb for you to adopt in your own programming.

```
' Clementine 2
' This program displays the lyrics of the folk song "Clementine."

CLS
PRINT "------------------------- Clementine -------------------------"
PRINT

PRINT "In a cavern, in a canyon,"                  ' first verse
PRINT "Excavating for a mine,"
PRINT "Dwelt a miner, forty-niner,"
PRINT "And his daughter, Clementine."
CALL Chorus                                        ' call Chorus subprogram

PRINT "Light she was and like a fairy,"        ' second verse
PRINT "And her shoes were number nine;"
PRINT "Herring boxes without topses,"
PRINT "Sandals were for Clementine."
CALL Chorus                                        ' call Chorus subprogram

PRINT "Drove she ducklings to the water,"   ' third verse
PRINT "Every morning just at nine;"
PRINT "Hit her foot against a splinter,"
PRINT "Fell into the foaming brine."
CALL Chorus                                        ' call Chorus subprogram

PRINT "Ruby lips above the water,"              ' fourth verse
PRINT "Blowing bubbles soft and fine;"
PRINT "But alas, he was no swimmer,"
PRINT "So he lost his Clementine."
CALL Chorus                                        ' call Chorus subprogram

PRINT "Then the miner, forty-niner,"           ' fifth verse
PRINT "Soon began to peak and pine;"
PRINT "Thought he oughter join his daughter,"
PRINT "Now he's with his Clementine."
CALL Chorus                                        ' call Chorus subprogram

END
```



```
SUB Chorus STATIC
' The Chorus subprogram prints the chorus of the song "Clementine"
'   and waits for the user to press Return.

PRINT
PRINT "Oh my darling, oh my darling,"          ' chorus
PRINT "Oh my darling, Clementine,"
PRINT "You are lost and gone forever,"
PRINT "Dreadful sorry, Clementine."
PRINT

INPUT "Press Return for more…", dummy$    ' pause
PRINT

END SUB
```

USING VARIABLES WITH SUBPROGRAMS
The programs you've written so far have used only a handful of variables, but when you write larger programs you might need to use lots of them. To help you keep track of large numbers of variables, AC/QuickBasic enforces some special rules that deal with variables in a main program and in subprograms. We'll discuss those rules in this section as we introduce the concept of local vs. shared variables and look at how variables are passed to subprograms.

Local Variables
A local variable is valid only within the module (main program or sub-program) in which it is declared:
A local variable is not affected by changes elsewhere in the program. This means that a variable used in the main program module of a AC/QuickBasic program won't be inadvertently altered by a variable with the same name in one of the subprogram modules. The opposite is also true: A variable used locally within a subprogram won't be updated when a change is made to a variable of the same name in the main program or another subprogram.
The point is that you can use the same name for variables in different parts of your program and these variables won't interfere with each other because each is local to the module in which it is declared. You can worry less about using the same variable name twice in a program and concentrate on writing general-purpose subprograms that you can use again later.

Declaring local variables
Variables are local by default, so you don't have to use any special statement to declare a local variable in your main program or in a subprogram. Actually, you're an old hand at declaring local variables - you've been doing it all along!

Practice:
Using local variables
The Local Variable program (Figure 7-3) demonstrates the exclusivity of a local variable in the main program and a local variable of the same name in a subprogram. Both variables are named game$ and both are string variables, but when the value of one variable changes, the value of the other does not.

Load and compile the Local Variable program from disk and run it. You'll see the following output:

```
    In the main program, game$ = Chess
    In the AddGame subprogram, game$ = and backgammon
    Back in the main program, game$ = Chess
```

As you can see, changes to the variable game$ in the AddGame sub-program do not affect the isolated game$ variable in the main program.

```
' Local Variable
' This program demonstrates the use of two local variables.
CLS
game$ = "Chess"             ' initialize game$ with value "Chess"

PRINT "In the main program, game$ = "; game$          ' display in main prog.

CALL AddGame                                                  ' display in subprogram

PRINT "Back in the main program, game$ = "; game$ ' display in main prog.

PRINT
INPUT "Press Return to continue…", dummy$

END

SUB AddGame STATIC

game$ = game$ + " and backgammon"

PRINT "In the AddGame subprogram, game$ = "; game$

END SUB
```

Figure 7-3. Local Variable: a program that demonstrates the use of two isolated local variables.
Shared Variables
A shared variable provides a mechanism for exchanging a local variable between a subprogram and the main program. Such an exchange is useful when the main program needs to pass information to a subprogram so that the subprogram can begin a calculation or when the subprogram needs to return the results of its work to the main program.
(We'll describe another technique for sharing variables, called passing arguments, in the next section.)

You declare shared variables by using the SHARED statement in a subprogram. Here's the syntax for a SHARED statement:

```
SHARED variableList
```

variableList is a list of local variables to be shared that are separated by commas.
If you use the SHARED statement, it must be the first statement inside the relevant subprogram.

Practice:
Using shared variables
The Shared Variable program (Figure 7-4) shows how a local variable named game$ can be passed with the SHARED statement. The game$ variable is displayed in the main program; passed to the AddGame sub-program, where it is modified and displayed; and then returned to the main program, where it is displayed a third time.

Load and compile the Shared Variable program from disk and run it. You'll see the following output: '

```
In the main program, game$ = Chess
In the AddGame subprogram, game$ = Chess and backgammon
Back in the main program, game$ = Chess an&·backgammon
```

Note that the contents of the game$ variable are available by default only to the main program - other subprograms must have their own SHARED statements if they are to take part in this data exchange.

```
' Shared Variable
' This program demonstrates the use of a shared variable.

CLS
game$ = "Chess"              ' initialize game$ with value "Chess"

PRINT "In the main program, game$ = "; game$          ' display in main prog.

CALL AddGame                                          ' display in subprogram

PRINT "Back in the main program, game$ = "; game$  ' display in main prog.

PRINT
INPUT "Press Return to continue…", dummy$

END

SUB AddGame STATIC
SHARED game$               ' share game$ with the main program

game$ = game$ + " and backgammon"
PRINT "In the AddGame subprogram, game$ = "; game$

END SUB
```

Figure 7-4.
Shared Variable: a program that demonstrates the use of a shared variable in a program.

Passing Arguments to a Subprogram
You can make any number of variables accessible by sharing them with the SHARED statement. Although it's convenient, this method of sharing variables increases the chances that you'll lose track of which variables are shared and which are local.

To help you track shared variables, AC/QuickBasic provides a more visual method for passing information (both variables and expressions) to only the subprograms you want the information to go to.
This method of sharing is called passing arguments.

With this method, arguments passed to a subprogram from the main program or another subprogram are received by parameters, which are local variables within the subprogram. You can use these local variables exactly as you would any other local variables in the subprogram.

Arguments vs. parameters
Before we go any further, let's formalize the difference between the terms argument and parameter:

- An argument is a variable or expression that is passed to a sub-program.
  Argument names appear in a CALL statement in parentheses after the subprogram name.
  A collection of argument names is known as an argument list.

- A parameter is a variable that receives a value passed to a subprogram.
  Parameter names appear in a SUB statement and follow the rules that apply to standard data types.
  A collection of parameter names is known as a parameter list.

Each argument must have a corresponding parameter of the same type (but not necessarily of the same name) in the subprogram's parameter list. Figure 7-5 shows the relationship between an argument list and a parameter list.

Figure 7-6 shows some valid argument-parameter pairs.

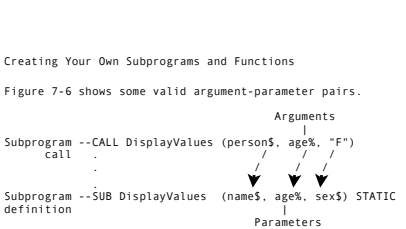


FIGURE 7-5.
The relationship between arguments and parameters.

Modifying arguments
Passing arguments to a subprogram is not a one-way street. Any value that a subprogram assigns to one of its parameters is passed back to the matching argument in the CALL statement when the subprogram has finished executing.

Let's use Figure 7-5 as an example: If the argument person$ in the main program contained the value Elisabeth at the time that the program called the subprogram DisplayValues, the subprogram parameter name$ would receive the value Elisabeth.

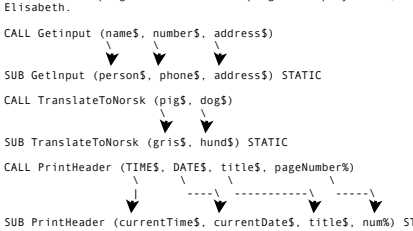


FIGURE 7-6.
Arguments and their matching parameters.

If the subprogram then assigned the value Vivienne to the parameter name$, the value Vivienne would be assigned to the argument person$ upon completion of the subprogram's execution.

Keep in mind, however, that you can't change the value of a number, a numeric expression, or a string literal (a string enclosed by double quotation marks). In Figure 7-5, for example, even if the DisplayValues sub-program modified the value of the parameter sex$, nothing would happen to the matching argument "F" because it is a string literal.

Practice:
Passing arguments to a subprogram
The Argument program (Figure 7-7 on the next page) demonstrates how two arguments are passed to a subprogram called AddInterest. The monthName$ parameter in the subprogram receives the month$ argument, and the amount! parameter receives the balance! argument.

Then the sub-program AddInterest changes the month, multiplies the value of amount! by 1.05, and returns both values to the main program.

```
' Argument
' This program demonstrates passing arguments to a subprogram.

month$ = "January"           ' initialize month$ with value "January"
balance! = 1500                ' initialize balance! with value 1500

CLS

PRINT "Before subprogram:"       ' display original values
PRINT "  month$ = "; month$
PRINT "  balance! ="; balance!
PRINT

CALL AddInterest (month$, balance!)     ' call subprogram to modify values
```

```
PRINT "After subprogram:"        ' display modified values
PRINT "  month$ = "; month$
PRINT "  balance! ="; balance!

PRINT
INPUT "Press Return to continue…", dummy$

END

SUB AddInterest (monthName$, amount!) STATIC

monthName$ = "February"            ' change name of month
amount! = amount! * 1.05             ' add 5% to amount

END SUB
```

Figure 7-7.
Argument: a program that demonstrates how to pass arguments to a subprogram.

Load the Argument program from disk and run it. You'll see this output:

```
Before subprogram:
  month$ = January
  balance! = 1500

After subprogram:
  month$ = February
  balance! = 1575
```

Both month$ and balance! were modified without resorting to the use of hard-to-track shared variables.

CREATING USER-DEFINED FUNCTIONS

A user-defined function is a one-line expression you define with a DEF FN statement. User-defined functions follow the same general rules that sub-programs do, with one important exception: A function, whether built-in or user-defined, performs a task and returns a value to the main program or calling subprogram. You use a user-defined function in the same way that you use a AC/QuickBasic built-in function- you include it in a AC/QuickBASIC statement.
In this section you'll learn how to create your own user-defined functions with AC/QuickBasic.

Syntax for a User-defined Function

Here's the syntax for a user-defined function:

```
DEF FNFunctionName(parameterList) = functionDefinition
```

- DEF FN is the AC/QuickBasic statement that marks the beginning of the function definition.

- FunctionName is the name of the function and can end with a type declaration character (just as a variable name can).
  The function name can be up through 40 characters long and is the name used to call the function.
  The function name can't be a AC/QuickBasic keyword or the same as any variable name, subprogram name, or other function in your program.

- (parameterList) is an optional list of variables (the variables separated by commas) used in functionDefinition.
  The list of variables is enclosed within parentheses.

- functionDefinition is an expression (typically a mathematical formula) that performs the operation of the function.
  The expression is limited to one logical line; that is, the entire definition must fit on one program line.
  You can use both variables in the function's parameter list and local variables in the main program in functionDefinition.

Calling a User-defined Function

User-defined function calls differ from subprogram calls in that a user-defined function call cannot stand alone. Rather, the result of the function must be assigned to a AC/QuickBasic statement or to a variable of the same type as the function's result. The following user-defined function, for example, accepts three integer parameters and returns a single integer value:

```
DEF FNSumOfTerms%(a%, b%, c%)= a% + b% + c%
```

A call to the FNSumOfTerms% function must include three integer variable or integer expression arguments. The function's result can be assigned to an integer variable or used in a AC/QuickBasic statement, such as PRINT, that accepts an integer value as an argument. A call to FNSumOfTerms%, with the result assigned to an integer variable, might look like this:

```
number%= FNSumOfTerms%(10 + 5, 20, cost%)
```

A call to FNSumOfTerms% in a PRINT statement might look like this:

```
PRINT FNSumOfTerms%(10 + 5, 20, cost%)
```

A user-defined function call always begins with the letters FN and always returns a single value in one of the five AC/QuickBasic data types: integer, long integer, single-precision floating-point, double-precision floating-point, or string.

Positioning a User-defined Function

A user-defined function can appear anywhere in the main program, but it must be defined before it is used. By convention, user-defined functions appear at the top of the program listing, immediately after any introductory comments. A user-defined function cannot be defined within a subprogram, but it can be called by a subprogram - any subprogram - or by the main program.

Practice:

Using a user-defined function
The Conversion program (Figure 7-8) demonstrates the declaration and call of two user-defined functions that return single-precision floating-point values. Conversion converts ten measurements expressed in centimetres to inches and then to feet and displays them with the PRINT statement.

```
' Conversion
' A program that converts centimetres to inches and then to feet.

' A function to convert centimeters to inches.
DEF FNCentToInch!(cent!) = cent! / 2.54

' A function to round numbers to two decimal places.
DEF FNRound!(num!) = INT(100 * num! + .5) / 100

CLS

PRINT "This program converts centimetres to inches and then to feet."
PRINT
PRINT "Centimetres", "Inches", "Feet"
PRINT

FOR i% = 10 TO 100 STEP 10              ' do 10 conversions
    inches! = FNCentToInch!(i%)              ' convert centimeters to inches
    PRINT i%, FNRound!(inches!), FNRound(inches! / 12)    ' print results
NEXT i%

PRINT
INPUT "Press Return to continue…", dummy$     ' pause screen
```

Figure 7-8.
Conversion: a program that uses functions to convert centimetres to inches and then to feet.

Load and compile the Conversion program from disk and run it. You'll see the following output:

```
This program converts centimetres to inches and then to feet.

Centimetres      Inches         Feet
10                3.94           .33
20                7.87           .66
30               11.81           .98
40               15.75          1.31
50               19.69          1.64
60               23.62          1.97
70               27.56          2.3
80               31.5           2.62
90               35.43          2.95
100              39.37          3.28
```

ORGANIZING YOUR PROGRAM

Figure 7-9 shows the order in a listing of the three types of program modules we've discussed in this chapter: the main program, subprograms, and user-defined functions. You can have any number of subprograms and functions in your program or none at all. Every program, however, must have a main program module that controls the general flow of the program and calls the subprograms and functions. Remember: No matter how many subprograms and functions a program contains, it is still a single AC/QuickBasic program.

Although it might seem like a little extra work at first, learning to divide your programs into these organizational units will pay off when you start to write longer programs. Debugging, the process of finding and fixing programming errors, will also take more time as your programs get longer, so programming in modules is a good idea.

Which Kind of Program Module Should You Use?

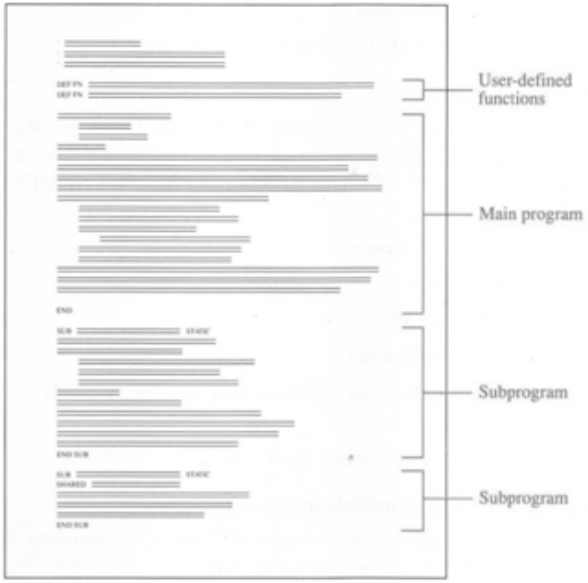
You might still be wondering which kind of program module is best for which situation.

Here are some general guidelines you can follow as you plan your programs.

Figure 7-9.
An illustration of AC/QuickBasic program module organization.

A subprogram is a miniprogram

Think of a subprogram as a small, self-contained program. A subprogram should perform an important task for one program and yet be general purpose enough to be used in other programming projects. A subprogram is ideal for a block of code that will be used more than once in a program. Subprograms are also a good idea for programs that have many variables to manage. The following tasks are often best suited to subprograms:



• Getting input from the user
• Displaying information on the screen
• Processing several numeric values or strings
• Drawing graphic shapes and designs
• Playing musical notes or songs
• Returning multiple values to the main program

A user-defined function excels in calculating and returning a single value to the main program.
A user-defined function won't return multiple values or execute general tasks.
  The following tasks are often best suited to functions:

• Performing a numeric calculation
• Returning a string value
• Generating a random number
• Converting one value to another
• Evaluating a logical expression and returning a value of either true or false
• Calculating one result from several arguments

The main program handles initialization and control
What tasks are left for the main program to handle?
Actually, not many if you make thorough use of subprograms and functions.
The following tasks are often best suited to the main program:

• Introductory comments and explanations
• Initialization of key variables
• Program code that is executed only once
• Flow-control structures that determine the path of program execution

SUMMARY
In this chapter you've learned about subprograms and functions - two programming structures that save you from repetition and extend the AC/QuickBasic language. You define subprograms with SUB and ENDSUB statements, and you define user-defined functions with DEF FN statements. Both structures can be called from either the main program

Working with Large Amounts of Data Learn AC/QuickBasic For the Apple][gs NOW

Now that you've learned how to work with simple data types and variables, it's time to expand your knowledge - to learn how AC/QuickBasic deals with large amounts of data.

In this chapter, you'll learn how to create data structures - collections of individual data items.
Data structures help you organize large amounts of information and speed up operations that involve many variables.

STORING AND RETRIEVING INFORMATION
So far, you've learned two ways to store values in a program:

• Assigning a value to a variable:

```
firstName$ ="Duncan"
```

• Assigning a value to a variable with INPUT:

```
INPUT "Enter number of home runs: ", homers%
```

These are single value assignments. But what if you have many values that you plan to use repeatedly, in a specific order? That's when the READ and DATA statements come in handy.

Using the READ and DATA Statements
The READ and DATA statements work together to let you store and retrieve information within a program.

Values are first stored in one or more DATA statements and are then assigned to variables with one or more READ statements. Here's the syntax for the READ and DATA statements:

```
READ variablelist
DATA constantlist
```

The READ statement's variableList is a list of one or more variables separated by commas.
The DATA statement's constantList is a list of one or more numeric or string values separated by commas.

Each value in the DATA statement must have a corresponding variable of the proper type in the READ statement.
The following pair of READ and DAT A statements illustrates this relationship between the statements.
Note that the DATA statement's corresponding values appear in the same order as the READ statement's variables.

```
READ fullName$, age%
       |          \
       |           |
DATA Beaver Cleaver, 9
```

A simple example best demonstrates how READ and DATA work together.

In the following program, The READ statement assigns the first DATA value to the string variable name$
and the second DATA value to the integer variable age%.

The PRINT statement then makes use of the variables.

Notice that the PRINT statement immediately follows the READ statement,
and that both the READ and the PRINT statements precede the DATA statement.

```
READ fullName$, age%
PRINT fullName$; "is"; age%; "years old."
DATA Beaver Cleaver, 9
```

When you run this program, your Output window displays this result:

```
      Beaver Cleaver is 9 years old.
```

READ and DATA statements: Helpful hints
Keep the following points in mind as you begin to use the READ and DATA statements to store and retrieve information:

• You can create multiple READ and DATA statements;
  just be sure that each READ variable has a matching DATA value,
  As in the example of multiple READ and DATA statements that follows:

```
   READ month$, numberOfDays%
   READ holidays%

   DATA November
   DATA 30, 3
```

• If a value in the DATA statement contains a comma, a colon, or significant leading or trailing spaces,
  you must enclose the entire value in quotation marks, as in the following example:

```
    DATA "     Total  ", "Redmond, Washington"
          /          \            \
         /            \            |
    Leading      Trailing        Included
    spaces       spaces          comma
```

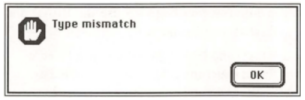
• DATA statements must appear within the main program.
  Programmers usually put DATA statements at the bottom of the main program.
  READ statements can appear anywhere in the program, including in subprograms.

What if the types don't match?
Each value in a DATA statement must be assigned to a corresponding variable of the same type in a READ statement.

If the types don't match,
AC/QuickBasic makes the following conversions:

DATA value type	READ variable type	Result type
string	integer/floating-point	error message
integer/floating-point	string	string
integer	floating-point	floating-point
floating-point	integer	rounded to integer

A type mismatch error is indicated by the following dialog box:



The program that follows demonstrates a type mismatch between a string DATA value and an integer READ variable.

```
READ fullName$
PRINT "Name: "; fullName$
READ address$
PRINT "Address: "; address$
READ age%
PRINT "Age: " age%

DATA Beaver Cleaver, "211 Pine Street, Mayfield", nine
```

When you run the program, your Output window briefly displays the following result:

```
Name: Beaver Cleaver
Address: 211 Pine Street, Mayfield
```

Then the Type mismatch error message appears because the READ statement cannot assign a string value (nine)
to an integer variable (age%).

To fix the program,
Change nine to 9 to accommodate the integer variable age%,
or change age% to age$ to accommodate the string value nine.

Typical DATA Statement Values
READ and DATA statements are most appropriate if you know the values of the variables ahead of time and if you know that the values will always appear in the same order. The following kinds of values are tailor-made for the READ and DATA method of storage and retrieval:

• Days of the week (Sunday through Saturday)
• Months of the year (January through December)
• Names of persons, places, or organizations
• Numeric data for calculation or analysis
• Numeric values for musical notes

Practice:
Storing several values
' The Add Them program (Figure 8-1) shows how to use a READ statement within a FOR loop to assign several DATA values to variables. Note that the integer variable items% controls the number of items read from the DATA statements.
By changing items% you can change the number of items that will be read.

1. Load and compile the Add Them program from the Chapter 8 folder on disk and run it.

```
' Add Them
' This program reads and adds values stored in DATA statements.

items% = 20                         ' set the number of items to be read

CLS

FOR i% = 1 TO items%                ' for each item to be read
    READ number!                    '   assign the next DATA item to number!
    sum! = sum! + number!           '   add the item to the running total
NEXT i%

PRINT "The sum of the"; items%; "numbers is"; sum!
```



```
PRINT
INPUT "Press Return to continue…", dummy$

DATA 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
DATA 88.2, 25, 3.3, 100, -74.2, 0, 20, 0.34, -89, 5.4567
```

FIGURE 8-1.
Add Them: a program that uses READ to assign 20 data items to variables.

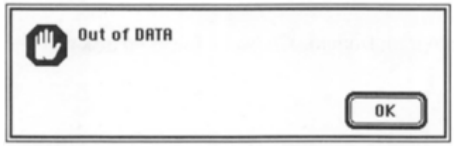
   Your Output window displays this result:

```
   The sum of the 20 is 134.0967
```

2. Change the value of the items% variable to 4 and run the program again. Your output screen displays this result:

```
   The sum of the 4 numbers is 10
```

The end-of-data marker
Unless your program can identify the last DATA value, it might keep executing READ statements-trying to assign a value a variable. The result is this error message, which indicates that no more data is available for assignment:



Tracking DATA Values: The Data Pointer
To help the READ statement assign values to variables, AC/QuickBasic uses a data pointer to point to the next DATA statement value to be assigned. When a program begins to execute, the data pointer points to the first value in the first DATA statement. As program execution continues and each DATA statement value is assigned, the data pointer points to the next unassigned DATA statement value.

To prevent this error, use an end-of-data marker as the final entry in your final DATA statement. The end-of-data marker is a value that your program tests against to determine the end of the DATA values. When it reads this final value, it moves on to the next task. This technique is useful when you have a long list of values in DATA statements that will be processed only once. The Add Them program works around this potential problem by using a variable that contains the exact number of values as the upper limit for the FOR loop. Many times you won't have such a luxury.
In the following program, Add Them is revised to check for an end-of-data marker.

Practice:
Checking for an end-of-data marker
The Add Until Marker program (Figure 8-2) uses an end-of-data marker (-9999) as the last DATA entry in the program.

Load and compile the Add Until Marker program from disk and run it.

```
' Add Until Marker
' This program reads and adds values stored in DATA statements
'   until it detects an end-of-data marker (-9999).

CLS

WHILE (number! <> -9999)                ' loop until end-of-data marker is read
    READ number!                        ' assign the next DATA item to number!
    IF (number! <> -9999) THEN          ' if not end-of-data marker, then
        sum! = sum! + number!           '   keep a running total
        items% = items% + 1             '   count the number of values read
    END IF
WEND

PRINT "The sum of the"; items%; "numbers is"; sum!
PRINT
INPUT "Press Return to continue…", dummy$

DATA 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
DATA 88.2, 25, 3.3, 100, -74.2, 0, 20, 0.34, -89, 5.4567
DATA -9999
```

Your Output window displays this result:

```
The sum of the 20 numbers is 134.0967
```

Rereading DATA values with the RESTORE statement
At times you might want to read repeatedly through the list of DATA values in a program. You might want to perform a number of different calculations on the same set of values, for instance. At the point at which you want to return to the first DATA value, use the RESTORE statement, which resets the data pointer to the first DATA statement value in the program.

You can use RESTORE as often as you like in a program. Here's the syntax for the RESTORE statement:

```
RESTORE
```

Practice:
Using RESTORE to repeat a list of values
The TV Hours program (Figure 8-3) uses RESTORE, DATA, and READ statements to track how many hours a person watches television during a three-week period. A pair of nested FOR loops cycles through the days of the week three times (to simulate three weeks), tracking the total number of viewing hours. The RESTORE statement at the end of each cycle returns the data pointer to Monday.

1. Load and compile the TV Hours program from disk and run it.

```
' TV Hours
' This program uses DATA, READ, and RESTORE statements to track
'   the number of TV-viewing hours over a three-week period.

CLS

PRINT "How many hours of TV did you watch during the past three weeks?"
PRINT

FOR i% = 1 TO 3       ' for each of the last three weeks
    FOR j% = 1 TO 7   '   and for each day in the week
        READ day$                     ' read day name from DATA list
        PRINT day$; ", Week"; i%;    ' prompt with day and week
        INPUT "--> ", hours!           ' get TV hours for that day
        totalHours! = totalHours! + hours!   ' total all the hours
    NEXT j%
    PRINT                ' print a blank line after each week
    RESTORE           ' move data pointer to Monday for next iteration
NEXT i%

PRINT "You watched"; totalHours!;           ' display total number of hours
PRINT "hours of television during the past three weeks!"
PRINT
INPUT "Press Return to continue…", dummy$

DATA Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday
```

FIGURE 8-3.
TV Hours: a program that demonstrates use of the RESTORE statement.

You see three sets of input prompts similar to this one:

```
   How many hours of TV did you watch during the past three weeks?

   Monday, Week 1 --> 0
   Tuesday, Week 1 --> 1
   Wednesday, Week 1 --> 0
   Thursday, Week 1 --> 2
   Friday, Week 1 --> O
   Saturday, Week 1 --> 2.5
   Sunday, Week 1 --> 2
```

2. Respond to the prompts. After you enter all 21 values, the Output window displays the final result:

```
   You watched 23.5 hours of television during the past three weeks!
```

Now that you've learned how to store and retrieve multiple values in your program, it's time to learn an efficient way work with multiple values. In the following section we'll introduce you to the array, a powerful data structure that can help you handle large amounts of data of the same type.

WORKING WITH ARRAYS
If you want to organize multiple variables of the same type under one name, use an array. Much as an egg carton organizes a number of individual eggs, an array lets you organize many values under one name.

An array can contain any one of the data types you've worked with so far in this book; that is, you can have

- String arrays
- Integer arrays
- Long integer arrays
- Single-precision floating-point arrays
- Double-precision floating-point arrays

But you cannot mix data types within an array; a string array can contain only strings, an integer array can contain only integers, and so on.

Let's look at an example that organizes data into three types of arrays:
string arrays, integer arrays, and single-precision floating-point arrays.

Tracking Information with Arrays: Megan's Bike Market
Megan's Bike Market, a large downtown bicycle shop, has a staff of seven salespersons. Megan wants to track two values every month:

- The number of bikes sold by each salesperson
- The number of dollars brought in by each salesperson

When she writes down this information, it falls naturally into three lists, as shown in Figure 8-4:
- A list of string values (salespersons' names)
- A list of integer values (number of bicycles sold by each salesperson)
- A list of floating-point values (total dollar sales for each salesperson)

By making these lists, Megan has already worked with arrays. She has created three of them: a string array for the salespersons' names, an integer array for the number of bicycles sold by each, and a single-precision floating-point array for the total dollar sales. Each list, with its distinct type of information, qualifies as an array. And each array contains seven elements, or individual values. Now that Megan has organized her arrays on paper, she can begin to convert them into a program with the help of the DIM statement.

Salesperson	Bikes sold	Total sales
Erin	6	$1,350.12
Eric	5	$1,578.55
Ron	12	$2,343.84
MaryAnn	7	$1,256.36
JoAnne	11	$2,613.79
Jack	2	$ 489.00
Nancy	5	$1,356.03

FIGURE 8-4.
A sample of salesperson data from Megan's Bike Market.

The DIM Statement: Making Reservations for Your Array
When you make a reservation at a restaurant, you provide particular information in exchange for a guaranteed table. By giving your name, specifying the meal you intend to eat (breakfast, lunch, or dinner), and indicating the number of people in your party, you ensure that space will be available for you. In the world of arrays, the DIM statement serves the same purpose as a reservation. It reserves memory space for an array on the basis of three pieces of information:

- The name you've selected for the array
- The type of data you plan to store in the array
- The maximum number of elements the array will contain

This process of space allocation is called dimensioning. Here's the syntax for a DIM statement:

```
DIM arrayName(subscript)
```

arrayName is the name you select for the array, and subscript is the highest-numbered element in the array.
The final character of arrayName must be the type-declaration character that identifies the data type of the array:
% for integers, & for long integers, ! for single-precision floating-point numbers,
# for double-precision floating-point numbers, or $ for strings.

The following DIM statement,
for example, dimensions a string array that can contain up to SO elements numbered 0 through 49:

```
DIM stateCaps$ (49)
    |      | | |__|
    |      | |   \ The maximum number of elements in your array
    \      / |
     \____/  \---- The data type of your array (%, &, ! , #, or$)
       |
        \--------- The name of your array
```

When you execute this DIM statement, AC/QuickBasic reserves memory space for a 50-element array to contain the names of the state capitals.

NOTE: A typical computer has room for many arrays, each containing thousands of elements. But because computer memory is not unlimited, you shouldn't set aside memory space for more array elements than you think you'll need.

Megan would create the three arrays she needs by using these DIM statements:

- For the names of the salespersons, a seven-element string array called salesGroup$:
  `DIM salesGroup$(6)`

- For the number of bicycles each has sold, a seven-element integer array called bikesSold%:
  `DIM bikesSold%(6)`

- For the total sales for each salesperson, a seven-element single-precision floating-point array called totalSales!:
  `DIM totalSales!(6)`

Note that in all three arrays the number 6 sets aside memory space for seven elements numbered 0 through 6.

Each element in the array is associated with a number. By default, AC/QuickBasic associates the first element of an array with the number 0, the second element of an array with the number 1, and so on, as shown in Figure 8-5.

	salesGroup$ array		bikesSold% array		totalSales! array
0		0		0	
1		1		1	
2		2		2	
3		3		3	
4		4		4	
5		5		5	
6		6		6	

FIGURE 8-5.
The association of each element of a dimensioned array with a number.

The OPTION BASE Statement
To make your program conceptually easier to work with, make the first element in each array 1 instead of 0 by using the OPTION BASE statement. The OPTION BASE statement associates the first element-or base-of all arrays in a program with the number 1.

To use the OPTION BASE statement, simply place the following statement near the top of your program, before any DIM statements:

```
OPTION BASE 1
```

The programs throughout this chapter use the OPTION BASE statement this way.

Working with Array Elements
After you've dimensioned an array with the DIM statement, it's easy to refer to any of the elements within the array.
To refer to an element of an array, you use the array name and an array index enclosed in parentheses.
The index must be an integer value; it can be a simple number or an integer variable, for example.
The following statement assigns the string value Tomato to element 3 in the shoppingList$ array:

```
   shoppinglist$(3) ="Tomato"
```

Practice:
Storing values in an array
The Get Names program (Figure 8-6) demonstrates; How information is stored in an array and printed out using two FOR loops. Get Names uses the OPTION BASE statement to set the first element of all arrays in the program to 1.

The salesGroup$ array is dimensioned at 7 to hold the names of the seven salespersons in Megan's bicycle shop.

Load and compile the Get Names program from disk and run it.

```
' Get Names
' This program reads string information into an array and prints it.

OPTION BASE 1           ' set base of all arrays to 1

DIM salesGroup$(7)      ' dimension salesGroup$ string array

CLS

FOR i% = 1 TO 7         ' use i% to access array elements
    INPUT "Enter salesperson name:  ", salesGroup$(i%)
NEXT i%

PRINT
PRINT "You entered the following names:"
PRINT

FOR i% = 1 TO 7         ' print entire contents of array
    PRINT salesGroup$(i%)
NEXT i%

PRINT
INPUT "Press Return to continue…", dummy$
```

FIGURE 8-6.
Get Names: a program that demonstrates loading and printing an array.
You'll be prompted to supply names for the salespeople. You'll see output similar to this:


```
    Enter salesperson name: Erin
    Enter salesperson name: Eric
    Enter salesperson name: Ron
    Enter salesperson name: Mary Ann
    Enter salesperson name: JoAnne
    Enter salesperson name: Jack
    Enter salesperson name: Nancy
```

Formatting Your Output: The PRINT USING Statement
The PRINT USING statement lets you design your output's appearance on the screen. PRINT USING is particularly helpful if you need to display large amounts of data in tabular form.
To create the design you want, you use a template string.
Use a monospace font such as Monaco or Courier for effective alignment.
Here's the syntax for the PRINT USING statement:

```
   PRINT USING template; argumentList
```

template is a string, and argumentlist is a collection of one or more values to be displayed, with the values separated by semicolons. template specifies how the values in argumentlist should be displayed.
The formatting characters in template must match up one for one with the characters of the values in argumentlist.
The following table describes a few of the formatting characters that you might find useful in template:

Character(s)	Description
#	Represents one digit of a numeric value
.	Represents the decimal point in a numeric value
$$	Cause a dollar sign to be displayed with a number
\\	Represent one or more spaces that can be filled with string data

The following program creates a formatting template named tmp$ and uses it in a PRINT USING statement to display a string variable and a dollars-and-cents single-precision floating-point value:

```
    tmp$ = "Name: \                 \   Payment: $$###.##"
    fullName$ ="Jack Ryan"
    payment! = 2496.33
    PRINT USING tmp$; fullName$; payment!
```

When you execute the statements, you see this output:

```
   Name: Jack Ryan          Payment: $2496.33
```

```
You entered the following names:
   Erin
   Eric
   Ron
   Mary Ann
   JoAnne
   Jack
   Nancy
```

Practice:
Using multiple arrays in a program
The Bike Info program (Figure 8-7) demonstrates how a number of arrays can be used together in a program to track related information. Bike Info is a revision of the Get Names program. This version tracks the salesperson names with the salesGroup$ array, the number of bikes each person sold with the bikesSold% array, and the total value of each person's sales with the totalSales! array. The three arrays are designed to be used together - each contains a piece of information about the salespersons in the bike shop as shown in the lists in Figure 8-4. If you refer to the array indexes together in a FOR loop, your program can access related items at the same time. The PRINT USING statement and the template tmp$ display the data in each array using font 4 (Monaco), a monospace font.

Load the Bike Info program from disk and run it.

```
' Bike Info
' This program reads information into three arrays and prints it.

OPTION BASE 1          ' set base of all arrays to 1

DIM salesGroup$(7)     ' dimension salesGroup$ string array
DIM bikesSold%(7)      ' dimension bikesSold% integer array
DIM totalSales!(7)     ' dimension totalSales! floating-point array

CLS

FOR i% = 1 TO 7         ' get salesperson name and sales data
    INPUT "Enter salesperson name:  ", salesGroup$(i%)
    INPUT "   Bikes sold:  ", bikesSold%(i%)
    INPUT "   Total sales:  $", totalSales!(i%)
    PRINT
NEXT i%

PRINT "You entered the following sales data:"
PRINT

TEXTFONT 4
```

```
PRINT "Salesperson     Bikes sold     Total sales"
PRINT "--------------------------------------------"

' Initialize tmp$, a formatting template for PRINT USING.
tmp$ = "\                \ ###          $$####.##"

FOR i% = 1 TO 7         ' print contents of each array
    PRINT USING tmp$; salesGroup$(i%); bikesSold%(i%); totalSales!(i%)
NEXT i%

TEXTFONT 1

PRINT
INPUT "Press Return to continue…", dummy$
```

You'll see output similar to this:

```
Enter salesperson name: Erin
  Bikes sold: 6
  Total sales: $ 1350.12

Enter salesperson name: Eric
  Bikes sold: 5
  Total sales: $ 1578.55

Enter salesperson name: Ron
  Bikes sold: 12
  Total sales: $ 2343.84

Enter salesperson name: Mary Ann
  Bikes sold: 7
  Total sales: $ 1256.36

Enter salesperson name: JoAnne
  Bikes sold: 11
  Total sales: $ 2613.79

Enter salesperson name: Jack
  Bikes sold: 2
  Total sales: $ 489

Enter salesperson name: Nancy
  Bikes sold: 5
  Total sales: $ 1356.03

You entered the following sales data:

    Salesperson        Bikes sold         Total sales
        Erin               6                $1350.12
        Eric               5                $1578.55
        Ron               12                $2343.84
        Mary Ann           7                $1256.36
        JoAnne            11                $2613.79
        Jack               2                 $489.00
        Nancy              5                $1356.03
```

Filling Part of an Array
You don't need to fill an array to the brim - allowing a little room for growth is often a good idea.
But then you need a way to signal AC/QuickBasic that the user has finished entering data. One way to do this is by using an end-of-data marker. You can have the user add elements to the array as part of a WHILE loop that continually checks for the end-of-data marker.
As soon as AC/QuickBasic sees the marker, it exits the loop and executes the rest of the program.
The end-of-data marker should have the same data type as the array being filled, and it should be a value the user is unlikely to type during normal execution of the program.

A typical end-of-data marker for a string array is QUIT or END.
A typical end-of-data marker for a numeric array is -9999.

Practice:
Using an end-of-data marker
The Fill Array program (Figure 8-8) demonstrates how to fill an array with different amounts of data.
The program fills and prints three arrays again, but this time dimensions the arrays with 50 elements each and uses END as an end-of-data marker to indicate that the user is finished entering data.
(Note that END must be entered in all capital letters.)

Load and compile the Fill Array program from disk and run it.

```
' Fill Array
' This program reads information into three arrays and prints it.
'   The maximum number of names that can be entered is 50; fewer
'   can be entered by typing "END" for the salesperson name.

OPTION BASE 1                ' set base of all arrays to 1

DIM salesGroup$(50)         ' dimension salesGroup$ string array
DIM bikesSold%(50)           ' dimension bikesSold% integer array
DIM totalSales!(50)           ' dimension totalSales! floating-point array

CLS

PRINT "Follow prompts to enter bike shop data.  Type END to quit."
PRINT

count% = 1                          ' initialize an array counter variable

WHILE (salesGroup$(count%) <> "END")  ' continue until name = "END"
    INPUT "Enter salesperson name:  ", salesGroup$(count%)

    IF (salesGroup$(count%) <> "END") THEN
        INPUT "   Bikes sold:  ", bikesSold%(count%)
        INPUT "   Total sales:  $", totalSales!(count%)
        PRINT
        count% = count% + 1          ' increment the array counter
    END IF
WEND

PRINT
PRINT "You entered the following sales data:"
PRINT

TEXTFONT 4

PRINT "Salesperson     Bikes sold     Total sales"
PRINT "--------------------------------------------"

' Initialize tmp$, a formatting template for PRINT USING.
tmp$ = "\                \ ###          $$####.##"

FOR i% = 1 TO count% - 1         ' print contents of each array
    PRINT USING tmp$; salesGroup$(i%); bikesSold%(i%); totalSales!(i%)
NEXT i%

TEXTFONT 1

PRINT
INPUT "Press Return to continue…", dummy$
```

You'll see output similar to this:

```
Follow prompts to enter bike shop data. Type END to quit.

    Enter salesperson name: Nancy
     Bikes sold: 5
     Total sales: $ 1356.03

   Enter salesperson name: JoAnne
     Bikes sold: 11
     Total sales: $ 2613.79

   Enter salesperson name: END

   You entered the following sales data:

   Salesperson       Bikes sold            Total sales
   Nancy                  5                  $1356.03
   JoAnne                11                  $2613.79
```

Creating Flexible Arrays
As you've learned, when you use the DIM statement you must inform AC/QuickBasic of the number of elements that will go into your array so that it can reserve an adequate amount of space in memory. But what if you're not sure how many elements your array will contain? If your program is dependent on user input for the contents of the array, for instance, the number of elements entered might vary each time the program is run.

How can you tell AC/QuickBasic how much memory to reserve?

The DIM statement is actually quite accommodating. It lets you use an integer variable containing input from the user to dimension the array. This flexibility means that AC/QuickBasic arrays are dynamic - they can be sized on the fly to meet the needs of the person using the program. To create an array whose size is determined by the user:

1. Use the INPUT statement to prompt the user for the number of elements.
2. Assign the value entered by the user to an integer variable.
3. Use the integer variable with the DIM statement to dimension the dynamic array.

Practice:
Using a variable to set array size
The Dynamic Array program (Figure 8-9) uses the persons% variable to dimension the three dynamic sales arrays.

Note the IF statement that checks the value of persons%:

If persons% is less than or equal to 0, no arrays are dimensioned.
Load and compile the Dynamic Array program from disk and run it.

```
' Dynamic Array
' This program reads information into three dynamic arrays and prints it.

OPTION BASE 1                       ' set base of all arrays to 1

CLS

INPUT "How many salesperson names would you like to enter?  ", persons%

IF (persons% > 0) THEN              ' must be at least one salesperson

    DIM salesGroup$(persons%)       ' dimension salesGroup$ string array
    DIM bikesSold%(persons%)        ' dimension bikesSold% integer array
    DIM totalSales!(persons%)       ' dimension totalSales! floating-point array

    PRINT

    FOR i% = 1 TO persons%          ' get salesperson name and sales data
        INPUT "Enter salesperson name:  ", salesGroup$(i%)
        INPUT "   Bikes sold:  ", bikesSold%(i%)
        INPUT "   Total sales:  $", totalSales!(i%)
        PRINT
    NEXT i%

    PRINT "You entered the following sales data:"
    PRINT

    TEXTFONT 4

    PRINT "Salesperson     Bikes sold     Total sales"
    PRINT "--------------------------------------------"

    ' Initialize tmp$, a formatting template for PRINT USING.
    tmp$ = "\               \ ###           $$####.##"

    FOR i% = 1 TO persons%     ' print contents of each array
        PRINT USING tmp$; salesGroup$(i%); bikesSold%(i%); totalSales!(i%)
    NEXT i%

    TEXTFONT 1

END IF

PRINT
INPUT "Press Return to continue…", dummy$
```

```
       You'll see output similar to this:

       How many salesperson names would you like to enter? 2

       Enter salesperson name: Erin
         Bikes sold: 6
         Total sales: $ 1350.12

       Enter salesperson name: Eric
         Bikes sold: 5
         Total sales: $ 1578.55

       You entered the following sales data:

       Salesperson      Bikes sold      Total sales
      ------------      ----------      -----------
       Erin                  6            $1350.12
       Eric                  5            $1578.55
```

Searching for an Element in an Array
At times you might want to perform a search within an array. You generally do this to find a specific array element or to find an array element based on comparison. Both operations involve stepping through an array one element at a time and keeping track of matches to a search string.

• In a search for a specific array element, AC/QuickBasic compares a search string with each element of the array until a match is found or until all array elements have been examined.

• In a comparison search, you use one or more temporary variables to track the progress of the comparison. Comparisons usually take one of the following forms:

• Find the largest number in the array.
• Find the smallest number in the array.

In the next practice, you'll use a typical method for finding a specific array element.
In the practice after that, you'll find the largest number in an array.
Practice:
Finding an array element
The Search Array program (Figure 8-10) demonstrates how to search an array for a specific element.

The program prompts the user for salesperson data and then asks which salesperson's data the user would like to examine.

A FOR loop steps through each element of the salesGroup$ array until a match is found or all the array entries have been examined:

• If the program finds a match, the corresponding elements from the bikesSold% and totalSales! arrays are displayed and then the program exits the loop.

• If the program doesn't find a match, it displays the message Name not found.

Load and compile the Search Array program from disk and run it.

Exiting a FOR Loop Early
From time to time you'll want to exit a FOR loop early. One effective way to do this is to assign the loop counter variable a value beyond the limit of the loop (established in the original FOR statement). Exiting a loop early is useful when you want to loop a specific number of times unless a certain condition is met.

The following program demonstrates this technique.
It asks for names until the user has entered 10 names or until the user enters QUIT, whichever occurs first:

```
FOR i% = 1 TO 10
INPUT "Enter a name: ", firstName$
IF ( firstName$ = "QUIT") THEN i % = 11 ELSE PRINT firstName$
NEXT i%

' Search Array
' This program reads data into three arrays and searches for a name.
'   The maximum number of names that can be entered is 50; fewer
'   can be entered by typing "END" for the salesperson name.

OPTION BASE 1                ' set base of all arrays to 1

DIM salesGroup$(50)          ' dimension salesGroup$ string array
DIM bikesSold%(50)           ' dimension bikesSold% integer array
DIM totalSales!(50)          ' dimension totalSales! floating-point array

CLS

PRINT "Follow prompts to enter bike shop data.  Type END to quit."
PRINT

count% = 1                   ' initialize an array counter variable

WHILE (salesGroup$(count%) <> "END")  ' continue until name = "END"
    INPUT "Enter salesperson name:  ", salesGroup$(count%)

    IF (salesGroup$(count%) <> "END") THEN
        INPUT "   Bikes sold:  ", bikesSold%(count%)
        INPUT "   Total sales:  $", totalSales!(count%)
        PRINT
        count% = count% + 1  ' increment the array counter
    END IF
WEND

PRINT                              ' prompt user for search string
INPUT "What name would you like to search for?  ", search$
PRINT

' Initialize tmp$, a formatting template for PRINT USING.
tmp$ = "\               \ ###          $$####.##"

' Compare each array element with search string until a match is
'   found, and then display the record and exit the loop; display
'   message if search string is not found.

FOR i% = 1 TO count% - 1     ' count% - 1 is the last array element
```

```
    IF (salesGroup$(i%) = search$) THEN
        TEXTFONT 4
        PRINT "Salesperson     Bikes sold     Total sales"
        PRINT "----------------------------------------"
        PRINT USING tmp$; salesGroup$(i%); bikesSold%(i%); totalSales!(i%)
        i% = count%           'exceeding the loop limit ends the FOR loop
    END IF
    IF (i% = count% - 1) THEN PRINT "** Name not found **"
NEXT i%

TEXTFONT 1

PRINT
INPUT "Press Return to continue…", dummy$

        You'll see output similar to this:

       Follow prompts to enter bike shop data. Type END to quit.

       Enter salesperson name: Jack
         Bikes sold: 2
         Total sales: $ 489.00

       Enter salesperson name: Ron
         Bikes sold: 12
         Total sales: $ 2343.84

       Enter salesperson name: Mary Ann
         Bikes sold: 7
         Total sales: $ 1256.36

       Enter salesperson name: END

         What name would you like to search for? Ron

         Salesperson      Bikes sold       Total sales
         -----------      ----------       -----------
             Ron              12            $2343.84
```

Practice:
Finding the largest number in an array
The Find Highest Sales program (Figure 8-11) demonstrates how the largest element in an array can be extracted by use of a FOR loop. The program prompts the user for salesperson data and then examines each element of the totalSales! array. The largest sales figure is stored in the variable largest! and is compared with each element of totalSales!. If an array element is larger than largest!, that array element becomes the new largest!. (The lg% variable stores the array index associated with largest!.) After it examines all array elements, the program displays the largest sales figure it has found along with the related elements in the salesGroup$ and bikesSold% arrays.

Load and compile the Find Highest Sales program from disk and run it.

```
' Find Highest Sales
' This program reads salesperson data into three arrays and displays
'   the name of the salesperson with the highest total sales and the
'   number of bicycles that person has sold.  The maximum number of
'   names that can be entered is 50; fewer can be entered by typing
'   "END" for the salesperson name.

OPTION BASE 1           ' set base of all arrays to 1

DIM salesGroup$(50)     ' dimension salesGroup$ string array
DIM bikesSold%(50)      ' dimension bikesSold% integer array
DIM totalSales!(50)     ' dimension totalSales! floating-point array

CLS

PRINT "Follow prompts to enter bike shop data.  Type END to quit."
PRINT

count% = 1                  ' initialize an array counter variable

WHILE (salesGroup$(count%) <> "END")  ' continue until name = "END"
    INPUT "Enter salesperson name:  ", salesGroup$(count%)
    IF (salesGroup$(count%) <> "END") THEN
        INPUT "   Bikes sold:  ", bikesSold%(count%)
        INPUT "   Total sales:  $", totalSales!(count%)
        PRINT
        count% = count% + 1   ' increment the array counter
    END IF
WEND

largest! = totalSales!(1)    ' first array element is largest so far
lg% = 1                      ' save array index
```



```
' Compare remaining array elements for something bigger -- if one is
'   found, assign it to largest! and save the array index in lg%; if
'   there is a tie for the largest, return the first element found.

FOR i% = 2 TO count% - 1
    IF (totalSales!(i%) > largest!) THEN
        largest! = totalSales!(i%)  ' save new largest value
        lg% = i%                    ' save array index
    END IF
NEXT i%

' Initialize tmp$, a formatting template for PRINT USING.
tmp$ = "\               \ ###          $$####.##"

PRINT
PRINT "** "; salesGroup$(lg%); " has the highest total sales **"
PRINT

TEXTFONT 4

PRINT "Salesperson     Bikes sold     Total sales"
PRINT "----------------------------------------"
PRINT USING tmp$; salesGroup$(lg%); bikesSold%(lg%); totalSales!(lg%)

TEXTFONT 1

PRINT
INPUT "Press Return to continue…", dummy$

       You'll see output similar to this:
       Follow prompts to enter bike shop data. Type END to quit.

       Enter salesperson name: Erin
         Bikes sold: 6
         Total sales: $ 1350.12

       Enter salesperson name: Eric
         Bikes sold: 5
         Total sales: $ 1578.55

       Enter salesperson name: Nancy
         Bikes sold: 5
         Total sales: $ 1356.03

       Enter salesperson name: Mary Ann
         Bikes sold: 7
         Total sales: $ 1256.36

       Enter salesperson name: END

      ** Eric has the highest total sales **

       Salesperson      Bikes sold        Total sales
       -----------      ----------        -----------
         Eric               5             $1578.55
```

Two-Dimensional Arrays
AC/QuickBasic also lets you declare arrays of two dimensions. By using a two-dimensional array, you can represent a table of values with rows and columns, such as a scoreboard, an accounting ledger, or a gameboard. In this section we'll discuss how to declare and use a two-dimensional array in a AC/QuickBasic program.

Let's start with an example. Sam, the local soda distributor, wants to track sales for his top four brands over the last 12 months. He wants to put the information in a table that uses brand names for row titles and months for column titles, as shown in Figure 8-12.

```
                  Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec
                  --- --- --- --- --- --- --- --- --- --- --- ---
       Orca Spray  64 63  58  45  36  32  41  39  50  67  69  103
       Fizzy Delite  35 41  60  57  38  29  25  19  26  37  43  36
       Alki Seltzer  15  9  12  21  24  32  46  42  37  22  18  13
       Schpritz    30 30  30  35  42  44  49  48  38  35  31  30
```

FIGURE 8-12.

A table of values showing soda sales over the last 12 months.
Tabular information is perfectly suited to a two-dimensional array.
In this example, one dimension of the array corresponds to the brand-name rows and the other dimension corresponds to the month columns. Items in a two-dimensional array are identified with row and column subscripts. Figure 8-13 shows how row and column subscripts would be assigned to each dimension if the base of the array were set at 1.

A one-dimensional array requires one subscript to identify each array element. A two-dimensional array requires two subscripts to identify each array element. In Figure 8-13, for example, the number of cases of Orca Spray sold in May would be identified by row 1, column 5.

Row subscripts Column subscripts

		1	2	3	4	5	6	7	8	9	10	11	12
1	Orca Spray	64	63	58	45	36	32	41	39	50	67	69	103
2	Fizzy Delite	35	41	60	57	38	29	25	19	26	37	43	36
3	Alki Seltzer	15	9	12	21	24	32	46	42	37	22	18	13
4	Schpritz	30	30	30	35	42	44	49	48	38	35	31	30

FIGURE 8-13.
Assigning two-dimensional array subscripts to the soda sales table.

Using Arrays with Subprograms
In Chapter 7 you learned how variables can be declared as local or shared and how variable arguments are passed to subprograms. A similar set of rules applies to arrays used in programs that contain subprograms:

• By default, an array is local to the main program or subprogram in which it is declared.

• An array can be shared throughout an entire program if you use the SHARED keyword with the DIM statement when you dimension the array in the main program - for example:
```
DIM SHARED stores$(20)
```

• A single array element can be passed to a subprogram as an argument - for example:
```
CALL Getinput (stores$(12))
```

A subprogram receives a array element as a simple variable parameter - for example:
```
SUB Getinput (store$) STATIC
```

• An entire array can be passed as an argument to a subprogram if you specify no subscript - for example:
```
CALL TallyProfits (stores$())
```

A subprogram receives an entire array through an array parameter with no subscript - for example:
```
SUB TallyProfits (groups$()) STATIC
```

As the programs you write become larger, you'll want to use arrays right along with subprograms to keep your code organized and efficient.

Declaring a two-dimensional array
Here's the syntax for dimensioning a two-dimensional array:

```
DIM arrayName(rows. columns)
```

arrayName is the name of the array, rows is the number of rows (the first dimension), and columns is the number of columns (the second dimension). rows and columns must be integers and can be expressed as numbers, variables, or expressions. The final character of arrayName must be the type-declaration character that identifies the data type of the array: % for integers, & for long integers, ! for single-precision floating-point numbers, # for double-precision floating-point numbers, or$ for strings.

As in one-dimensional arrays, every element in a two-dimensional array must be of the same data type .

NOTE: The first element in each dimension of the array is numbered 0 unless you use the OPTION BASE 1 statement before you dimension the array.

The following statements dimension an array named sodaSales% with 4 rows and 12 columns:

```
OPTION BASE 1
DIM sodaSales%(4, 12)
```

The following statements ask the user to enter the dimensions of the same two-dimensional array:

```
OPTION BASE 1
INPUT "Enter number of soda brands sold: ", brands%
INPUT "Enter number of months to track: ", months%
DIM sodaSales%(brands%, months%)
```


Practice:
Building a table of values
The 2-D Sales Table program (Figure 8-14) demonstrates how sodaSales%, a two-dimensional array, is filled and displayed on the screen. 2-D Sales Table gets the numbers of rows and columns in the soda sales table from the user, stores those numbers in the brands% and months% integer variables, and uses them to dimension the sodaSales% array. The program uses two nested FOR loops to fill and display the array and uses READ and DATA statements to store and retrieve the months of the year. You can use this program as a guide for filling and printing any two-dimensional array.

Load and compile the 2-D Sales Table program from disk and run it.

```
' 2-D Sales Table
' This program tracks the sales of soda over a given number of months
'   with a two-dimensional array.

CLS

PRINT "** Soda Sales Tracking Program **"
PRINT
WHILE (brands% < 1)
   INPUT "How many brands of soda do you sell?  ", brands%
WEND

WHILE (months% < 1) OR (months% > 12)
   INPUT "How many months would you like to record (1-12)?  ", months%
WEND
PRINT
OPTION BASE 1                              ' set first array element at 1
DIM sodaSales%(brands%, months%)           ' dimension soda sales array
DIM brandNames$(brands%)                   ' dimension brand names array

' Get names of soda brands sold.

PRINT "Enter the"; brands%; "brands of soda you sell."
PRINT
FOR i% = 1 TO brands%
   INPUT "Brand name:  ", brandNames$(i%)
NEXT i%

' Get soda sales for each month.

PRINT
PRINT "Enter soda sales in cases."
PRINT

FOR i% = 1 TO brands%                      ' for each brand of soda…
   PRINT "** "; brandNames$(i%); " **"
   PRINT                                   '   print name of brand
   FOR j% = 1 TO months%                   ' for each month…
      READ mo$                             '   read month name from DATA list
      PRINT "  "; mo$;                     '   print month name and prompt for input
      INPUT ":  ", sodaSales%(i%, j%)      ' store input in array
   NEXT j%
   PRINT
   RESTORE                                 ' rewind DATA list to first month
NEXT i%

' Print out soda sales table.

CLS

PRINT "Soda sales in cases for"; months%; "months:"
PRINT

TEXTFONT 4

PRINT "-------------------------------------------------------------"
PRINT "Brand/Month   ";

FOR i% = 1 TO months%
   PRINT " ";
   READ mo$                    ' read month name from DATA list
   PRINT mo$;                  ' print month names across top of table
NEXT i%

PRINT
PRINT "-------------------------------------------------------------"
```

```
' Initialize templates for PRINT USING.
nameTmp$ = "\              \"    ' for brand name (up to 12 characters)
salesTmp$ = " ###"              ' for cases sold (up to 3 digits)

FOR i% = 1 TO brands%           ' fills in the table
    PRINT USING nameTmp$; brandNames$(i%);
    FOR j% = 1 TO months%
        PRINT USING salesTmp$; sodaSales%(i%, j%);
    NEXT j%
    PRINT
NEXT i%

PRINT "------------------------------------------------------------"

TEXTFONT 1

PRINT
INPUT "Press Return to continue…", dummy$

DATA Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec
```

When you run 2-D Sales Table, you are prompted for input:

```
** Soda Sales Tracking Program**

   How many brands of soda do you sell? 4
   How many months would you like to record (1-12)? 12

   Enter the 4 brands of soda you sell.

   Brand name: Orea Spray
   Brand name: Fizzy Delite
   Brand name: Alki Seltzer
   Brand name: Sehpritz

   Enter soda sales in cases.

** Orea Spray **

   Jan: 64
   Feb: 63
   Mar: 58
```

After you've entered the data for each brand name and month, you'll see output similar to this:

```
   Soda sales in cases for 12 months:
   ------------------------------------------ ----------------------------
   Brand/Month Jan  Feb  Mar  Apr  May  Jun  Jul  Aug  Sep  Oct  Nov  Dec
   ------------------------------------------------------------------------
   Orca Spray   64   63   58   45   36   32   41   39   50   67   69  103
   Fizzy Delite 35   41   60   57   38   29   25   19   26   37   43   36
   Alki Seltzer 15    9   12   21   24   32   46   42   37   22   18   13
   Schpritz     30   30   30   35   42   44   49   48   38   35   31   30
```

Array Troubleshooting
Although arrays are a great boon to your programs, they also increase the potential for error. Let's look at some typical programming errors associated with arrays and with processing large amounts of data and then look at some ways to avoid the errors.

Mistake 1: Not using an integer to define a subscript
The number of elements and dimensions in an array is determined by the integer subscripts in the DIM statement. Be sure that the array subscripts are integer values or variables.

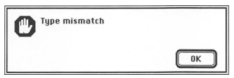
The following array declaration, for example, is valid:

```
DIM grades%(students%, 15)
```

The subscripts students% and 15 are both integer values.
The following declaration is not valid because items$ and quantity$ are string values:

```
DIM costTable!(items$, quantity$)
```

When you try to dimension an array with invalid subscripts, you see this error message:



If you supply a floating-point number as a subscript, AC/QuickBasic rounds the value to the nearest integer value and then dimensions the array. The following statement, for example, awkwardly but successfully dimensions an array with 6 elements (0 through 5):

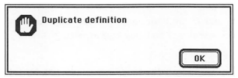

```
DIM values%(4.6)
```

NOTE: Using a floating-point value as an array subscript will be confusing to someone reading your program. Stick to integer subscripts.

Mistake 2: Using a DIM statement in a loop
Be sure that DIM statements are outside any loops in your program. The following program fragment demonstrates the incorrect use of a DIM statement, inside a FOR loop, to declare an array:

```
   OPTION BASE 1

   size%= 5

   FOR i% = 1 to size%
       DIM names$(size%)
       INPUT "Enter a name: ", names$(i%)
   NEXT i%
```

If you run the program, you see this error message when the DIM statement is executed a second time:



To avoid this error, dimension the names$ array before the loop, as follows:

```
   OPTION BASE 1

   size%= 5
   DIM names$(size%)

   FOR i% = 1 to size%
       INPUT "Enter a name: " names$(i%)
   NEXT i%
```

Mistake 3: Confusing the array index with the array value
It's easy to confuse the value used to index an array element with the value in the array element. Remember that the array index is always in parentheses and that it follows the array name that describes the location of the array value (Figure 8-15).

```
   DIM roomRate!(roomStyles%, daysRented%1
   .
   .
   .
   roomRate!( 5, 3 ) = 45.85
              \___/     \___/
                |         |
              Array     Array
              index     value
```

FIGURE 8-15.

Array index and array value.
The following statement tries to assign the string value Orcas Hotel to the fifth element in the vacationSpots$ array but fails because the array index and the array value are in the wrong locations:

```
      vacationSpots$("Orcas Hotel") = 5

      The correct array assignment is:
      vacationSpots$(5) = "Orcas Hotel"
```

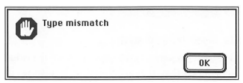
Mistake 4: Mismatching array types

Every element in an array must be of the same data type. If you try to assign a variable or value to an array that does not match the array type specified in the DIM statement, you'll receive an error message.

For example, the following assignment statement generates an error message because cost! is a single-precision floating-point array and the value "57.36" is a string data type.
(A correct assignment would be 57.36 without the quotation marks.)

```
cost!(i%) = "57.36"
```

This assignment produces the following error message:



A similar type mismatch occurs while the program is running if the user enters a value that does not match the array element that receives the entry. The following loop, for example, prompts the user to enter several integers.

```
DIM values%(5)
FOR i% = 1 TO 5
    INPUT "Enter a number: ", values%(i%)
NEXT i%
```

If you enter a string value, you'll see this in your Output window:

```
       Enter a number: ten
      ?Redo from start
       Enter a number:
```

?Redo from start is AC/QuickBasic's way of prompting the user for the correct type of data. Usually, this message is enough to get users back on track. As we've mentioned before, however, the best way to avoid input problems is to spell out with your input prompt exactly what you want.

Mistake 5: Making out-of-range errors
Attempting to refer to an element that does not exist in an array generates an out-of-range error message when the program is run. For example, AC/QuickBasic generates an error message when the third of the following statements is executed because testScores% can contain only 25 elements and a reference is made to a 30th element.

```
OPTION BASE 1
DIM testScores%(25)
testScores%(30) = 92
```

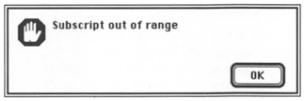
The out-of-range error is most common in a looping structure, as shown in the following program fragment:

```
OPTION BASE 1
DIM colors$(4)
count% = 1

WHILE (colors$(count%) <> "QUIT")
    INPUT "Enter a color name: " colors$(count%)
    count% = count% + 1
WEND
```

When you run the program fragment, you see output something like this:

```
       Enter a color: yellow
       Enter a color: red
       Enter a color: blue
       Enter a color: green
```

After the fourth color is entered (filling the last element in the array), the count% loop counter is incremented and the WHILE statement (now referencing an element beyond the end of the array) triggers the Subscript out of range error message.



The solution for out-of-range problems is to determine the upper and lower bounds (the limits) of the array and not go past them. AC/QuickBasic provides the UBOUND and LBOUND functions to return the bounds of any array in your program so that you can handle this potential problem.

The UBOUND and LBOUND functions
The UBOUND and LBOUND functions return integer values corresponding to the upper and lower bounds of an array.
 Here's the syntax for the UBOUND and LBOUND functions:

```
UBOUND(arrayName, dimension)
LBOUND(arrayName, dimension)
```

arrayName is the name of the array you want to determine bounds for, and dimension is the dimension you want to check. (dimension isn't required if you're evaluating a one-dimensional array.) Both functions return integer values that can be assigned to variables or used in expressions.

The following program fragment uses the UBOUND and LBOUND functions to check a one-dimensional array named players$.

```
OPTION BASE 1
DIM players$(9)
PRINT "Upper bound is"; UBOUND(players$)
PRINT "Lower bound is"; LBOUND(players$)
```

When you run the program fragment, you see this output:

```
       Upper bound is 9
       Lower bound is 1
```


When you check the bounds of a two-dimensional array, you must supply dimension numbers:
1 for the first dimension and 2 for the second dimension.

The following program fragment uses UBOUND and LBOUND to check the bounds of both dimensions in a two-dimensional array named janSales%:

```
OPTION BASE 1
DIM janSales%(3, 4)
PRINT "Upper bound of first dimension is"; UBOUND(janSales%, 1)
PRINT "Lower bound of first dimension is"; LBOUND(janSales%, 1)
PRINT "Upper bound of second dimension is"; UBOUND(janSales%, 2)
PRINT "Lower bound of second dimension is"; LBOUND(janSales%, 2)
```

When you run the program fragment, you see this output:

```
       Upper bound of first dimension is 3
       Lower bound of first dimension is 1
       Upper bound of second dimension is 4
       Lower bound of second dimension is 1
```

Practice:
Avoiding the subscript-out-of-range message
The Array Bounds program (Figure 8-16) demonstrates how to use UBOUND and LBOUND in a WHILE loop to check the bounds of an array and thus avoid out-of-range errors. A one-dimensional string array named party$ holds party game ideas. Array Bounds fills the array and prints it.

Load and compile the Array Bounds program from disk and run it.

```
' Array Bounds
' This program loads data into the party$ array until one of the array's
'   boundaries is exceeded.

OPTION BASE 1              ' set array base to 1
DIM party$(5)              ' dimension array with 5 elements

count% = 1                 ' initialize loop counter to 1

CLS
PRINT "Enter 5 party game ideas."
PRINT

' Read input into the array while count% is within the array bounds.

WHILE (count% >= LBOUND(party$)) AND (count% <= UBOUND(party$))
    INPUT "Party game:  ", party$(count%)
    count% = count% + 1
WEND

PRINT
PRINT "You entered the following games:"
PRINT

FOR i% = 1 TO count% - 1    ' count% - 1 is the number of array elements
    PRINT party$(i%)
NEXT i%

PRINT
INPUT "Press Return to continue…", dummy$
```

   FIGURE 8-16.
   Array Bounds: a program that uses LBOUND and UBOUND to avoid referencing an array element that is out of bounds.

You'll see output similar to this:

```
   Enter 5 party game ideas.

     Party game: Blindman's buff
     Party game: Bingo
     Party game: Pin the tail on the donkey
     Party game: Spin the bottle
     Party game: Moonlight bowling

     You entered the following games:

     Blindman's buff
     Bingo
     Pin the tail on the donkey
     Spin the bottle
     Moonlight bowling
```

SUMMARY
In this chapter, you've worked with the important structures, functions, statements, and techniques AC/QuickBasic provides for working with large amounts of data in a program:

• Using READ, DATA, and RESTORE to assign data stored in a program to variables
• Creating and using a one-dimensional array
• Searching for elements in an array
• Formatting tabular information with PRINT USING
• Creating and using a two-dimensional array
• Troubleshooting common array-related programming errors

In the next chapter, you'll learn about the AC/QuickBasic functions and techniques especially designed for working with strings.

QUESTIONS AND EXERCISES
1. Which comes first in a program, a DATA statement or a READ statement?
2. What type of information is well suited to the DATA-READ-RESTORE approach to working with data?
3. Write a program that displays each of the values in the following DATA statement with a FOR loop:
   DATA Beaver, Wally, Lumpy, Whitey, Gus, Eddie, Larry
4. True or False: An array can store more than one type of data.
5. Write a statement that sets aside memory for an array of 100 single-precision floating-point values.
6. What is an end-of-data marker?

7. Write a program that declares, fills, and then prints a dynamic one-dimensional array containing the major characters of your favorite television show or movie.
8. What is an out-of-range error?
9. Write a program that uses a two-dimensional array to keep score for a nine-inning baseball game. Your program should initialize the array, prompt the user for the baseball teams' names and mascots, get the runs scored for each inning, determine the final score and winner of the game, and display the results.

Working with Strings Learn AC/QuickBasic For the Apple][gs NOW

In Chapters 3 and 4, you met up with the string data type and practiced creating strings and displaying them on your screen. In this chapter we'll continue that discussion and introduce many of the functions available in AC/QuickBasic for working with strings. We'll describe

• Assigning user input to a string
• Combining strings
• Selecting characters from a string
• Comparing strings
• Sorting strings

When combined with the skills you've developed in preceding chapters, these functions can help you perform a wide variety of tasks.

STRINGS: AN OVERVIEW
A string is a series of consecutive characters that you use as a unit. Typically, you store information as a string when you can't easily store it as a numeric data type. For example, because of its textual nature, the title of a book --- let's say The Original Mother Goose --- would be stored as a string rather than as a numeric data type.

You might find it helpful to think of a string as occupying a series of memory locations in a computer.
You can think of the memory locations as little boxes set side by side,
--- each box holding one character of the string --- as shown in Figure 9-1.
The memory locations, or boxes, are fixed in place, but you can move the characters among the boxes at will.
You can also add characters to or remove characters from the string as the need arises.

  You can use the following characters in a string:
• Uppercase letters of the alphabet (A through Z)
• Lowercase letters of the alphabet (a through z)
• Numerals (0 through 9)
• Punctuation symbols(. , ; : ' ? !)
• Mathematical symbols (# % ( ) - + = \ / < >)
• Miscellaneous symbols and foreign language characters

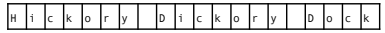


FIGURE 9-1.
A string is a series of characters.

TWO TYPES OF STRINGS
AC/QuickBasic supports two types of strings for use in your programs:

literal strings and string variables. Let's review how you use these types of strings in AC/QuickBasic.

Literal Strings
A literal string is a series of consecutive characters enclosed in double quotation marks.
You generally assign literal strings to a variable or use them as arguments to a statement or function.

For example, the following statements contain literal strings:

```
cheer$= "Go Seahawks!"

PRINT "1313 Mockingbird Lane"

goal$= FNCenterString$("Today I shall fly a kite.")

birthDate$ = "11-19-63"
```

Literal strings are also known as string values.

String Variables
The contents of a string variable can change at any time during the execution of a program. We've been using string variables throughout this document to obtain, store, and pass information in our code.
A string variable can contain from 0 through 32,767 characters and its length can grow or shrink during the execution of a program.

You can declare a string variable in two ways:

• By appending the string type-declaration character($) to the variable name.
  The following statement, for example, gets a string from the user and assigns it to the string variable firstName$.

```
  INPUT "Enter your first name: ", firstName$
```

As you examine the examples in this book, you'll see many instances in which INPUT statements contain string variables that are meaningful names for particular kinds of words and phrases. We also use string variables as arguments to statements and functions such as PRINT.

• By using the DIM statement to declare an array of string variables.
  We described this method in detail in Chapter 8.
  The following statement is an example that declares an array of 10 string variables
   (assuming that the statement OPTION BASE 1 appears first):

```
        DIM names$(10)
```

Practice Using string variables
The Phone Variables program (Figure 9-2) uses INPUT statements and a two-dimensional string array named contacts$ to store a list of friends and their telephone numbers. (Because telephone numbers often contain characters that are not numerals, such as dashes and letters, it's a good idea to store them as strings.)

Load and compile the Phone Variables program from the Chapter 9 folder on disk and run it.

```
' Phone Variables
' This program uses a string array to record names and telephone numbers.

OPTION BASE 1              ' set lower bound of array to 1

CLS

INPUT "How many names would you like to enter?  ", names%
PRINT

DIM contacts$(names%, 2)  ' declare array for names and phone numbers

FOR i% = 1 TO names%      ' read names into contacts$ array
    INPUT "Enter name:  ", contacts$(i%, 1)
    INPUT "Enter phone number:  ", contacts$(i%, 2)
    PRINT
NEXT i%

PRINT "You entered the following contact list:"
PRINT

FOR i% = 1 TO names%      ' print contents of array
    PRINT "Name:  "; contacts$(i%, 1), "Phone:  "; contacts$(i%, 2)
NEXT i%

PRINT
INPUT "Press Return to continue…", dummy$
```

You'll see output similar to the following:

```
How many names would you like to enter? 3

Enter name: Little Bo-Peep
Enter phone number: 555-LAMB

Enter name: Little Jack Horner
Enter phone number: 555-PLUM

Enter name: Little Boy Blue
Enter phone number: 555-HORN

You entered the following contact

Name: Little Bo-Peep             Phone: 555-LAMB
Name: Little Jack Horner         Phone: 555-PLUM
Name: Little Boy Blue            Phone: 555-HORN
```

Note that you achieve the alignment of the phone number column by using a comma in the PRINT statement to form two columns. In this case, advancing to the next print zone is enough to align the items.

Combining Strings
One of the simplest things you can do with strings is to combine them to form longer strings. This process is called concatenation. You concatenate strings primarily to prepare text for output on the screen or a printer or to prepare them for storage in an array or a file. You can concatenate both literal strings and string variables in any combination, and you can assign the result to a string variable or supply the result as an argument to a statement or function that expects string values (such as PRINT).

For example,
The following statement uses the concatenation operator (+) to combine the literal strings Microsoft, AC/QuickBasic, and Interpreter/Compiler and assigns the result to the string variable language$:

```
language$ = "Microsoft" + "AC/QuickBasic" + "Interpreter/Compiler"
```

The concatenation operator combines the three literal strings to form one string. The assignment operator(=) assigns the result to the string variable language$. Note that there are no spaces in the literal strings in this statement. (The spaces surrounding the operators don't count.) If you were to print out the value of language$ by means of the PRINT statement, you'd see the following output:

```
MicrosoftAC/QuickBasicInterpreter/Compiler
```

The concatenation operator combines the strings exactly as they are, without adding spaces. To include spaces, you must add spaces to the literal strings themselves; that is, a space needs to appear within the quotation marks.

Any of the following statements would accomplish this:

```
• PRINT "Microsoft" + " AC/QuickBasic" + " Interpreter/Compiler"
• PRINT "Microsoft " + "AC/QuickBasic " + "Interpreter/Compiler"
• PRINT "Microsoft" + " " + "AC/QuickBasic" + " " +"Interpreter/Compiler"
```

You can also supply the result of a concatenation as an argument to some AC/QuickBasic statements directly - without assigning the result to an intermediate variable such as language$:

```
PRINT "Microsoft" + "AC/QuickBasic" + "Interpreter/Compiler"
```

The result of the preceding statement is the same as the result of

```
   language$ = "Microsoft" + "AC/QuickBasic" + "Interpreter/Compiler"
   PRINT language$
```

Practice:
Concatenating strings
The Add Strings program (Figure 9-3),
Demonstrates a number of the options available to you through string concatenation.

Load and compile the Add Strings program from disk and run it.

```
' Add Strings
' This program demonstrates string concatenation.

structure$ = "Bridge"        ' initialize string variables
action$ = "is falling"

OPTION BASE 1
DIM direction$(5)            ' dimension string array with 5 elements
direction$(1) = "down"       ' put "down" in first array location

CLS

INPUT "Please enter the name of a city:  ", city$

' Print the news flash.
PRINT
PRINT "News Flash:  ";
PRINT city$ + " " + structure$ + " " + action$ + " " + direction$(1) + "!"

PRINT
INPUT "Press Return to continue…", dummy$
```

FIGURE 9-3.
Add Strings: a program that demonstrates string concatenation.

If you respond to the city prompt with London, you'll see this output:

```
Please enter the name of a city: London
News Flash: London Bridge is falling down!
```

PUTTING STRING FUNCTIONS TO WORK
So far you've declared string arrays and variables, combined strings in a process called concatenation, and used strings as arguments in INPUT and PRINT statements. In this section you'll learn about AC/QuickBasic functions that are specifically designed to manipulate and return values from literal strings and string variables.

You'll learn how to.

```
• Change a string to uppercase letters
• Determine the length of a string
• Take strings apart
```

Changing a String to Uppercase Letters
It's easy to change a string's letters to all uppercase. Simply use the UCASE$ function. UCASE$ is handy when you want to give text on the screen extra emphasis. It's also useful when you want to convert all user input to the same format. We'll discuss the importance of this when we cover how strings are compared at the end of the chapter.

Here's the syntax for the UCASE$ function:

```
UCASE$(stringexpression)
```

stringexpression is any kind of string.
The value returned by UCASE$ can be assigned to a string variable or supplied as an argument to a statement or function that accepts string values.

UCASE$ affects only the lowercase letters of stringexpression.

Practice:
Using the UCASE$ function
The Uppercase program (Figure 9-4) demonstrates how the UCASE$ function works. Uppercase declares three string variables and then uses the UCASE$ function to display them in uppercase. Note that the UCASE$ function affects only the output of the PRINT statement - it doesn't change the contents of the writer$ and address$ variables.
When the are displayed again at the end of the program, only the borough$ variable retains uppercase letters, by virtue of its original string assignment.

Load and compile the Uppercase program from disk and run it.

```
' Uppercase
' This program demonstrates the UCASE$ function.

writer$ = "Sir Arthur Conan Doyle"
address$ = "1326 Serpentine Avenue"
borough$ = UCASE$("St. John's Wood")

CLS

PRINT UCASE$(writer$)
PRINT UCASE$(address$) + ", " + borough$
PRINT
PRINT writer$
PRINT address$ + ", " + borough$

PRINT
INPUT "Press Return to continue…", dummy$
```

Figure 9-4.
Uppercase: a program that demonstrates use of the UCASE$ function.

You'll see the following output:

```
SIR ARTHUR CONAN DOYLE
1326 SERPENTINE AVENUE, ST. JOHN'S WOOD

Sir Arthur Conan Doyle
1326 Serpentine Avenue, ST. JOHN'S WOOD
```

Determining the Length of a String
Often you'll want to know how many characters are in a string. This knowledge can be particularly handy with strings entered from the keyboard. To determine how many characters (including spaces) are in a string, use the LEN function.

Here's the syntax for the LEN function:

```
LEN(stringexpression)
```

stringexpression once again is any kind of string. The value returned by LEN can be assigned to an integer variable or supplied as an argument to a statement or function that accepts integer values.

The following routine shows how the LEN function determines the number of characters in a string and assigns the number to an integer variable:

```
fullName$ = "Old Mother Hubbard"
nameLength% = LEN(fullName$)
PRINT fullName$; " is"; nameLength%; "characters long."
```

When you execute this routine, you see the following output:

```
Old Mother Hubbard is 18 characters long.
```

Practice:
Using the LEN function
The String Length program (Figure 9-5) demonstrates how the value returned by the LEN function can be used as an argument to a PRINT statement. Load and compile the String Length program from disk and run it.

```
' String Length
' This program demonstrates the LEN function.

CLS

INPUT "What is your favorite meal?  ", meal$
PRINT
PRINT UCASE$(meal$); " is"; LEN(meal$); "characters long."

PRINT
INPUT "Press Return to continue…", dummy$
```

FIGURE 9-5.
String Length: a program that demonstrates use of the LEN function.

You'll see output similar to this:

```
What is your favorite meal? Kim's Thai Stir Fry

KIM'S THAI STIR FRY is 19 characters long.
```


Taking Strings Apart
Earlier in this chapter you learned that AC/QuickBasic lets you combine strings through concatenation. Sometimes you'll want to take a string apart - to get only a person's last name from a variable containing a full name, for example.
AC/QuickBasic provides four functions that allow you to work with parts of strings.
You'll learn how to use these functions to;

- Get the right end of a string (RIGHT$)
- Get the left end of a string (LEFT$)
- Get the middle of a string (MID$)
- Find a string within a string (INSTR)

You'll also learn about statements that let you

- Get an entire line of input (LINE INPUT$)
- Print repeated characters (SPACE$, STRING$)

Getting the ends of a string
The RIGHT$ and LEFT$ functions let you retrieve one or more characters starting from one end of a string. This is useful when you want to display only part of a string or when you want to remove part of a string.

Here's the syntax for the RIGHT$ function:

```
RIGHT$(stringexpression, n)
```

Here's the syntax for the LEFT$ function:

```
LEFT$(stringexpression, n)
```

stringexpression is any kind of string, and n is an integer value ranging from 0 through the length of the string that indicates the number of characters to be returned by RIGHT$ or LEFT$. You can assign the value returned to a string variable or supply the value as an argument to a statement or function that accepts string values.

Practice:
Using the RIGHT$ function
The Get Right program (Figure 9-6) uses the RIGHT$ function to retrieve characters from a variable named alphabet$, which contains the 26 letters of the alphabet. Get Right extracts the requested number of characters and displays them with a character count.

Load and compile the Get Right program from disk and run it.

```
' Get Right
' This program demonstrates the RIGHT$ function.

CLS

alphabet$ = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"   ' declare test string

PRINT "How many characters (from right to left) in the following"
PRINT "string would you like to display?"
PRINT
PRINT alphabet$                            ' display test string
PRINT

' Prompt user for the number of rightmost characters to be displayed.
' Loop until the number is in the proper range (1 through 26).

WHILE (rightNum% < 1) OR (rightNum% > 26)
    INPUT "    Number (1-26):  ", rightNum%
WEND

PRINT
rightChar$ = RIGHT$(alphabet$, rightNum%)  ' display characters
PRINT "You specified"; LEN(rightChar$); "characters:  "; rightChar$

PRINT
INPUT "Press Return to continue…", dummy$
```

Figure 9-6
Get Right: a program that demonstrates use of the RIGHT$function.

You'll see output similar to this:

```
How many characters (from right to left) in the following string would you like to display?

ABCDEFGHJJKLMNOPQRSTUVWXYZ

Number (1-26): 14
You specified 14 characters: MNOPQRSTUVWXYZ
```

Practice:
Using the LEFT$ function
The Get Left program (Figure 9-7) revises the Get Right program to extract characters from the left side of a string with the LEFT$ function. Notice that the variable names have changed slightly (rightNum% becomes leftNum%, and rightChar$ becomes leftChar$) and that the function RIGHT$ has been changed to LEFT$.

Apart from these changes (and a few changes to the prompt and program comments), Get Left is identical to Get Right. Because the operations of the RIGHT$ and LEFT$ functions are so similar, it's quite easy to change a program so that it modifies a string from the opposite end.

Load and compile the Get Left program from disk and run it.

```
' Get Left
' This program demonstrates the LEFT$ function.

CLS

alphabet$ = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"   ' declare test string

PRINT "How many characters (from left to right) in the following"
PRINT "string would you like to display?"
PRINT
PRINT alphabet$                            ' display test string
PRINT

' Prompt user for the number of leftmost characters to be displayed.
' Loop until the number is in the proper range (1 through 26).

WHILE (leftNum% < 1) OR (leftNum% > 26)
    INPUT "    Number (1-26):  ", leftNum%
WEND

PRINT
leftChar$ = LEFT$(alphabet$, leftNum%)     ' display characters
PRINT "You specified"; LEN(leftChar$); "characters:  "; leftChar$

PRINT
INPUT "Press Return to continue…", dummy$
```

You'll see output similar to this:

```
  How many characters (from left to right) in the following string would you like to display?

  ABCDEFGHIJKLMNOPQRSTUVWXYZ

       Number (1-26): 14

  You specified 14 characters: ABCDEFGHIJKLMN
```

Getting the middle of a string
The MID$ function lets you retrieve one or more characters from anywhere within a string-from the left, from the middle, or (with some help from the LEN function) from the right. Its versatility makes the MID$ function one of the useful string functions. And, as we'll see later, it provides the processing power to solve many string-related problems.
Here's the syntax for the MID$ function:

```
MID$(stringexpression, start, length)
```

stringexpression is any kind of string, start is an integer value between 1 and the length of the string (indicating the first character to be returned), and length is an integer value indicating the number of characters to be returned. You can assign the value returned by MID$ to a string variable or supply the value as an argument to a statement or function that accepts string values. Figure 9-8 shows the elements of the MID$ function syntax in detail.

```
  MID$             Position of first
function        character to be returned
   |                    |
MID$(stringexpression,   start,   length
        |                           |
          String to             Number of
         be searched     characters to be returned
```

FIGURE 9-8.
The components of the MID$function.

The following statements show some nifty uses of the MID$ function.
Notice the powerful possibilities that arise when you use the value returned by a function as an argument to MID$ or when you assign the value returned by MID$ to another statement or function.

```
middleName$ = MID$("Queen Victoria Belfield", 7, 8)
Result: middleName$ contains Victoria

address$= "1521 Plumtree Lane #25-K"
streetNameStart% = 6
```



```
length% = 13
PRINT UCASE$(MID$(address$, streetNameStart%, length%))
Result: PLUMTREE LANE

inString$ ="Making it all make sense"
rightmostWord$ = MID$(inString$, LEN(inString$) - 4, 5)
Result: rightmostWord$ contains sense

PRINT "The current year is "; MID$(DATE$, 7, 4)
Result: The current year is 1991
```

Practice
Using the MID$ function
The Get Middle program (Figure 9-9 on the next page) shows how to retrieve characters from the middle of a string with the MID$ function. Get Middle modifies the Get Right and Get Left programs to include a starting point along with the number of characters in the alphabet$ string to be displayed. Get Middle uses two WHILE loops to get integer values in the proper range and then uses the MID$ function to assign the selected characters to the midChar$ variable.

The results of the selection are printed with the following IF statement, which appears near the end of the program:

```
IF (numToDisplay% = LEN(midChar$)) THEN
    PRINT numToDisplay%; "characters displayed: "; midChar$
ELSE
    PRINT numToDisplay%; "characters requested,";
    PRINT LEN(midChar$); "displayed: "; midChar$
END IF
```

The IF statement compares numToDisplay% - the variable containing the number of characters the user asked to see displayed - to the number of characters in midChar$ (returned by the LEN function). If the two values are equal, the value of numToDisplay% is printed along with the contents of midChar$. If the two values are not equal, the LEN function determines the actual number of characters and this value is displayed along with the midChar$ string. Get Middle contains this additional message to notify the user that the display length entered exceeded the number of characters remaining in the string.

Load and compile the Get Middle program from disk and run it.

```
' Get Middle
' This program demonstrates the MID$ function.

CLS

alphabet$ = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"  ' declare test string

PRINT "How many characters (from left to right) in the following"
PRINT "string would you like to display?"
PRINT
PRINT alphabet$                           ' display test string
PRINT

' Prompt user for the number of characters to be displayed.
' Loop until the number is in the proper range (1 through 26).

WHILE (numToDisplay% < 1) OR (numToDisplay% > 26)
    INPUT "    Number (1-26):  ", numToDisplay%
WEND

PRINT                                     ' get starting number...
PRINT "What character would you like to start with?"
PRINT

WHILE (start% < 1) OR (start% > 26)    '   in proper range
    INPUT "    Starting number (1-26):  ", start%
WEND

PRINT                                     ' get characters
midChar$ = MID$(alphabet$, start%, numToDisplay%)

' Compare requested characters with actual characters retrieved
' and print an appropriate message.

IF (numToDisplay% = LEN(midChar$)) THEN
    PRINT numToDisplay%; "characters displayed:  "; midChar$
ELSE
    PRINT numToDisplay%; "characters requested,";
    PRINT LEN(midChar$); "displayed:  "; midChar$
END IF

PRINT
INPUT "Press Return to continue…", dummy$
```

You'll see output similar to this:

```
How many characters (from left to right) in the following string would you like to display?

ABCDEFGHIJKLMNOPQRSTUVWXYZ
Number (1-26): 14
What character would you like to start with?
Starting number (1-26): 4
14 characters displayed: DEFGHIJKLMNOPQ
```

If you specify a number outside the range permitted by Get Middle, you see output similar to this:

```
How many characters (from left to right) in the following string would you like to display?

ABCDEFGHIJKLMNOPQRSTUVWXYZ

    Number (1-26): 0
    Number (1-26): 30
    Number (1-26): 15

What character would you like to start with?

      Starting number (1-26): w
     ?Redo from start
      Starting number (1-26): 20

      15 characters requested, 7 displayed: TUVWXYZ
```

If you type in a non-numeric value (such as a letter) at one of the prompts, AC/QuickBasic prints the message ?Redo from start and redisplays the prompt. Handling this type of response from the user is important in developing "break-proof" programs.

Getting an entire line of input from the user
Throughout this book we've used the INPUT statement to get input from the user. The INPUT statement is quite versatile - it can assign input to one or more variables of different types and supply an optional prompt to spell out exactly what the user should enter.

We haven't discussed how the INPUT statement processes a comma in the input line. Consider the following statement, which prompts the user to enter a name and address:

```
INPUT "Enter name and address: ", mailingAddress$
```

If the user responds to the prompt with a string containing commas, the error message ?Redo from start appears, as shown in the following dialogue:

```
Enter name and address: Jon Victor, 1118 Skyridge, Lacey, WA, 98503
?Redo from start
Enter name and address:
```

As we noted in Chapter 4, the INPUT statement has a special use for the comma character: The comma separates the assigned to variables. But what happens when the user types unexpected commas - as shown above?
You could provide for commas by assigning parts of the input string to different variables.
The following INPUT statement, for example, assigns the string value entered by the user to five string variables:

```
INPUT "Enter name and address: ", cust$, addr$, city$, state$, zip$
```

But there are times when it would be a lot simpler to have only one variable name associated with a line of input. AC/QuickBasic provides a solution to this problem with the LINE INPUT statement. The LINE INPUT statement reads an entire line of text from the keyboard and assigns it to a string variable, regardless of whether commas are present.

Here's the syntax for the LINE INPUT statement:

```
LINE INPUT[;] ["promptstring";] stringvariable
```

promptstring is a literal string that prompts the user for input, and stringvariable is any string variable.

- Placing a semicolon immediately after LINE INPUT keeps the cursor on the same line after the user presses Return.
- If promptstring is included in the statement, a following semicolon is required to separate promptstring from stringvariable.
- Unlike the INPUT statement, the LINE INPUT statement prints no question mark unless the question mark is included in promptstring.

The following statements demonstrate the usefulness of LINE INPUT for long lines of input that contain the comma character:

```
LINE INPUT "Enter name and address: "; mailingAddress$
PRINT mailingAddress$
```


When you execute the statements and enter the information we tried to enter earlier, you see this:

```
Enter name and address: Jon Victor, 1118 Skyridge, Lacey, WA, 98503
Jon Victor, 1118 Skyridge, Lacey, WA, 98503
```

You'll see the LINE INPUT statement from time to time in later chapters.

Printing repeated characters
AC/QuickBasic provides two useful functions that generate strings of repeated characters:
The SPACE$ function, which returns a string of spaces, and the STRING$ function, which returns a string of characters.
Both functions give you fast ways to build strings you can use in formatting and aligning your program's output.

Here's the syntax for the SPACE$ function:

```
SPACE$(n)
```

n is an integer value specifying the number of spaces the string will contain. You can assign the value returned by SPACE$ to a string variable or supply the value as an argument to a statement or function that accepts string values.
The most common use of the SPACE$ function is in formatting output, as in the following routine:

```
blank$= SPACE$(15)
PRINT blank$; "Big sale on bunnies today!"
```

SPACE$ comes in handy whenever you consistently indent text a set number of spaces.

Here's the syntax for the STRING$ function:

```
STRING$(m, stringexpression)
```

m is an integer value that specifies the length of the string to be returned, and stringexpression is the character to be repeated. You can assign the value returned by STRING$ to a string variable or supply the value as an argument to a statement or function that accepts string values.

Practice:
Using the SPACE$ and STRING$ functions
The most common use of the STRING$ function is for headings used in program output.
The Header program (Figure 9-10) uses STRING$ to display a header message in the middle of the screen.
Note that using a variable to specify the number of repeated characters makes it easy to modify the program later and that the variables created by SPACE$ and STRING$ make excellent candidates for concatenation.

Load and compile the Header program from disk and run it.

```
' Header
' This program demonstrates the SPACE$ and STRING$ functions.

length% = 15

fileName$ = "Sweet Pea Inventory"
blank$ = SPACE$(length%)
asterisk$ = STRING$(length%, "*")
banner$ = blank$ + asterisk$ + " " + fileName$ + " " + asterisk$

CLS

PRINT banner$

PRINT
INPUT "Press Return to continue…", dummy$
```

FIGURE 9-10.
Header: a program that demonstrates use of the SPACE$ and STRING$ functions.

You'll see the file name Sweet Pea Inventory in the middle of the top line of the screen, surrounded by equal numbers of asterisks and blank spaces.

Finding a string within a string
We've used the RIGHT$, LEFT$, and MID$ functions in this chapter to return characters from the right, left, and middle portions of a string.

These functions are quite effective at extracting characters from a set place in a string, but they are less effective at searching for and extracting a specific pattern from a string. AC/QuickBasic fills this gap with the INSTR function, which searches for a string within another string. INSTR joins RIGHT$, LEFT$, MID$, and the support functions we've discussed in this section (UCASE$, LEN, SPACE$, and STRING$) to round out a complete collection of tools for working with strings.

Here's the syntax for the INSTR function:

```
INSTR([start, ]basestring, searchstring)
```

start is an optional integer value specifying the character at which the search should begin, basestring is the string to be searched, and searchstring is the string to be found within basestring. You can assign the value returned by INSTR to an integer variable or supply the value as an argument to a statement or function that accepts integer values.

The following table lists the values that the INSTR function can return:

Condition	Integer value returned
searchstring found in basestring	Position in basestring at which match is found
searchstring not found in basestring	0
start is greater than length of basestring	0
basestring contains no characters	0
searchstring contains no characters	start (if given); otherwise, 1

Practice:
Using the INSTR/unction

The Find a String program (Figure 9-11) uses the INSTR function to search for the string iddle in the string
 High, diddle, diddle, the cat and the fiddle.

Load and compile the Find a String program from disk and run it.

```
' Find a String
' This program demonstrates the INSTR function.

CLS

baseStr$ = "High, diddle, diddle, the cat and the fiddle"
searchStr$ = "iddle"
strLocation% = INSTR(1, baseStr$, searchStr$)

IF (strLocation% <> 0) THEN
    PRINT searchStr$; " first appears starting at character"; strLocation%
ELSE
    PRINT searchStr$; " not found"
END IF

PRINT
INPUT "Press Return to continue…", dummy$
```

FIGURE 9-11.
Find a String: a program that demonstrates use of the INSTR function.

When you run the program, you see this:

```
iddle first appears starting at character 8
```

Although the pattern iddle appears three times in baseStr$, the INSTR function returns the location of only the first occurrence. To find multiple occurrences of a pattern, you can use INSTR within a loop. Whenever you use the INSTR function, it's a good idea to have your program check the value returned by INSTR. This will allow your program to take appropriate action if the search string is not found, or if either the search string or the base string is empty. In the Find a String program, if iddle were not found, the INSTR function would assign a value of 0 to to strLocation% variable and the IF statement would display the following message to indicate that iddle did not exist in baseStr$:

```
iddle not found
```

Practice:
Finding multiple occurrences of a pattern
The Find Many Strings program (Figure 9-12) uses the INSTR function to find multiple occurrences of a pattern in a series of text lines entered from the keyboard.

The heart of Find Many Strings is the Repeat subprogram. Each time INSTR finds the search string (searchStr$) in the base string (baseStr$), the location of the search string is assigned to currentChar%. Then, when the num% variable is incremented, the currentChar% variable is moved ahead to the position just after the string match in the base string (the new starting place for the next search). This process continues until the end of the base string is reached or the search string is not found in the remainder of the base string.

The Repeat subprogram then passes the total number of matches in the line to the lineRepeats% variable in the main program.

Load and compile the Find Many Strings program from disk and run it.

```
' Find Many Strings
' This program prompts the user for a set number of lines and a search
'   string and then prints the lines and the number of matches found.

' Set maximum number of lines that can be entered and declare string
'   array to hold lines.

maxLines% = 10   ' maxLines% will be shared with the GetText subprogram
DIM inputLines$(maxLines%)

CLS

' Call GetText subprogram to get input from user; at return, the
'   numOfLines% variable will contain number of lines received.

CALL GetText (inputLines$(), numOfLines%)
```



```
' Get pattern to be searched for from user.

PRINT
INPUT "Enter the string to be searched for:  ", pattern$
PRINT

' Call Repeat subprogram to determine the number of matches per line.
' The totalRepeats% variable will accumulate the number of total matches.
FOR i% = 1 TO numOfLines%
    CALL Repeat (pattern$, inputLines$(i%), lineRepeats%)
    totalRepeats% = totalRepeats% + lineRepeats%
NEXT i%

' Display lines entered by the user…

PRINT "You entered the following lines:"
PRINT
FOR i% = 1 TO numOfLines%
    PRINT inputLines$(i%)
NEXT i%

' and the total number of matches.

PRINT
PRINT "The pattern '"; pattern$; "' appears"; totalRepeats%; "times."

PRINT
INPUT "Press Return to continue…", dummy$

END

SUB GetText (strArray$(), count%) STATIC

' The GetText subprogram fills the strArray$ array with text
'   entered at the keyboard.  The number of lines that can be
'   entered is determined by the shared variable maxLines%.
'   Both strArray$ and count% (the number of lines actually
'   entered) are returned to the main program.

SHARED maxLines%    ' get maxLines% from the main program

PRINT "Enter up to"; maxLines%; "lines of text; to end, ";
PRINT "press Return on a new line."
PRINT

count% = 0                    ' initialize variables to 0 and "empty"
inLine$ = "empty"

' Loop until count% = MAXLINES% or an empty line is received.

WHILE (count% < maxLines%) AND (inLine$ <> "")
    LINE INPUT "-> "; inLine$  ' get line from user
    IF (inLine$ <> "") THEN       ' if line is not blank, copy it
        count% = count% + 1          '   to the strArray$ array
        strArray$(count%) = inLine$
    END IF
WEND

END SUB

SUB Repeat (searchStr$, baseStr$, num%) STATIC

' The Repeat subprogram returns the number of times a search string
'   is found in a base string.  The number is returned to the main
'   program through the num% parameter.

searchLength% = LEN(searchStr$)  ' determine length of search string
baseLength% = LEN(baseStr$)        ' determine length of base string
currentChar% = 1                ' character offset in base string
num% = 0                           ' running total of matches found in base

' Loop until entire string is processed or INSTR returns a 0.

WHILE (currentChar% <= baseLength%) AND (currentChar% <> 0)
    currentChar% = INSTR(currentChar%, baseStr$, searchStr$)
    IF (currentChar% <> 0) THEN  ' if not 0, a match was found
        num% = num% + 1                ' increment number of matches
        ' New offset equals current offset plus search-string length.
        currentChar% = currentChar% + searchLength%
    END IF
WEND

END SUB
```

You'll see output similar to this:

```
      Enter up to 10 lines of text; to end, press Return on a new line.

       -> Ten lords a-leaping,
       -> Nine ladies dancing,
       -> Eight maids a-milking,
       -> Seven swans a-swimming,
       -> Six geese a-laying,
       -> Five gold rings,
       -> Four calling birds,
       -> Three French hens,
       -> Two turtledoves, and
       -> A partridge in a pear tree.

      Enter the string to be searched for: ing

      You entered the following lines:

        Ten lords a-leaping,
        Nine ladies dancing,
        Eight maids a-milking,
        Seven swans a-swimming,
        Six geese a-laying,
        Five gold rings,
        Four calling birds,
        Three French hens,
        Two turtledoves, and
        A partridge in a pear tree.

        The pattern 'ing' appears 7 times.
```

COMPARING STRINGS
In Chapter 4 you learned that you can compare one string to another and branch to another place in the program based on the result of the comparison. The following IF statement compares the variable reply$ to the literal string Y and prints a message based on the comparison:

```
reply$ = "Y"
IF (reply$ = "Y") THEN
    PRINT "The two string values are equal."
ELSE
    PRINT "The two string values are not equal."
END IF
```

If you execute the statements as they are above, you see this:

```
      The two string values are equal.
```

If you change the value of reply$ toy and then execute the statement, you see a different response:

```
      The two string values are not equal.
```

Why is this? After all, a Y is a y .... Or is it?
What criteria is AC/QuickBasic using for its string comparisons?

The ASCII Character Set
Before AC/QuickBasic can compare one character to another, it must convert each character into a number by using a translation table called the ASCII character set. AC/QuickBasic then compares the numbers, called ASCII codes, and returns the logical value true if the ASCII codes are equal or false if the codes are not equal.

ASCII Is an Acronym
ASCII stands for American Standard Code for Information Interchange.

The key word here is code:

Like the Morse code used in radio and telegraphy, ASCII is an internationally accepted code for representing characters, but ASCII is used in computers and telecommunication.

Appendix A lists all of the ASCII codes and the character associated with each code in several different fonts.

Each character in the ASCII character set is associated with a unique number;
The set contains 128 characters (codes 0 through 127) in all:

- Control characters (codes 0 through 31),
  including characters that correspond to special keys on your keyboard such as Return, Backspace, and Tab
- Punctuation symbols, numbers, and mathematical symbols (codes 32 through 64)
- Uppercase letters of the alphabet (codes 65 through 90)
- Lowercase letters of the alphabet (codes 97 through 122)
- Miscellaneous symbols (codes 91 through 96 and 123 through 127)

The ASCII code for the uppercase letter A is 65; the ASCII code for the lowercase letter z is 122.
Following this logic, you can see why AC/QuickBasic considers the uppercase letter Y (code 89) and the lowercase letter y (code 121) to be different characters.

Optional Characters
Appendix A also contains a set of characters (codes 128 through 255) known as optional characters.
This set of symbols was developed by Apple for Apple][gs computers and printers and has been adopted by most software publishers writing Apple][gs applications. The optional set (sometimes called upper ASCII characters) contains foreign-language characters and mathematical symbols. Although you can use these symbols in your programs, you can't type them by the usual means - they don't appear on your keyboard!
Instead, you hold down the Shift, Command, or Option key and type in a one-character or two-character code.
(Appendix A lists these codes along with the optional characters.)
For example, to display a bullet (•) on your screen, hold down the Option key and type 8.

The following PRINT statement demonstrates the valid use of an optional character in a AC/QuickBasic program.
The ¡ symbol was entered by holding down the Option key and typing 1.

```
PRINT "¡Vamos amigos!"
```

NOTE: Some characters in the ASCII and optional character sets will vary depending on the font you're using.
The Zapf Dingbats and Symbol fonts, for example, contain completely different graphical shapes and symbols.
Check your font documentation for a list of the character codes and keystrokes you need to use to produce the shapes and symbols of these fonts in your programs.

Converting Codes to Characters
If you know a character's code but you're not exactly sure what the character it represents looks like, you can use the CHR$ function to return the symbol to your screen or to your program.

Here's the syntax for the CHR$ function:

```
CHR$ (code)
```

code is an integer value that specifies an ASCII or optional character code.

You can assign the string value returned by CHR$ to a string variable or supply the value as an argument to a statement or function that accepts string values.

Practice:
Using the CHR$ function
The ASCII Codes program (Figure 9-13) shows how to use the CHR$ function to display the characters in the ASCII and optional character sets. Notice that the program skips the first 32 (control) characters in the ASCII character set and pauses after each multiple of 15 lines so that you can view the results at your own pace.

Load and compile the ASCII Codes program from disk and run it.
(Because of the length of the program's output, we won't show it here.)

```
' ASCII Codes
' This program displays the ASCII character set and the optional
'   Apple ][gs characters.

CLS

FOR i% = 33 TO 255
    PRINT "Code"; i%; "= "; CHR$(i%)
    IF (i% MOD 15 = 0) THEN INPUT "Press Return for more…", dummy$
NEXT i%
```

      FIGURE 9-13.
      ASCII Codes: a program that uses the CHR$ function to display the ASCII and optional character sets.

Converting Characters to Codes
As a complement to the CHR$ function, AC/QuickBasic provides the ASC function, which converts a character to its code in the ASCII or optional character set.

Here's the syntax of the ASC function:

```
ASC(stringexpression)
```

stringexpression is a one-character string. You can assign the integer value returned by ASC to an integer variable or supply the value as an argument to a statement or function that accepts integer values.

Using Relational Operators with Strings
In addition to testing for equivalence of characters, AC/QuickBasic supports string comparisons with the following relational operators:

Operator	Meaning	Operator	Meaning
<>	Not equal	>	Greater than
=	Equal	<=	Less than or equal to
<	Less than	>=	Greater than or equal to

A character is "greater than" another character if its ASCII code number is higher.

For example,
The ASCII value of the letter B is greater than the ASCII value of the letter A,

```
        so the expression
              "A" < "B" is true,
        and the expression
              "A" > "B" is false.
```

When comparing two strings each of which contains more than one character, AC/QuickBasic begins by comparing the first character in the first string to the first character in the other string and then proceeds through the strings character by character until it finds a difference. For example, the strings Mike and Michael are the same up to the third characters (k and c). Because the ASCII value of k is greater than that of c, the expression

```
      "Mike" > "Michael"
                is true.
```

If no differences are found, the strings are equal. If two strings are equal through several characters but one of the strings continues and the other one stops, the longer string is greater than the shorter string.

For example,

```
            the expression
              "AAAAA" > "AAA" is true.
```

Sorting Strings
The Sort Strings program (Figure 9-15) uses the <= relational operator to compare array elements and uses the SWAP statement to switch any elements that are out of order.

Sort Strings declares an array of strings named inputLines$ and calls the GetText subprogram.
The Get Text subprogram in Sort Strings is identical to the GetText subprogram in the Find Many Strings program (Figure 9-12): It reads lines of text into an array and then returns to the main program the array (inputLines$) and the number of elements in the array (numOfElements%).

Sort Strings next calls the ShellSort subprogram. Note the following statements in the heart of ShellSort that compare array elements and then swap the elements if they are out of order:

```
IF strArray$(j%) <= strArray$(j% + span%) THEN
     j% = 1
ELSE    ' swap array elements that are out of order
     SWAP strArray$(j%), strArray$(j% +span%)
END IF
```

The <= relational operator is used to compare the two array elements.

- If the value of the string expression strArray$(j%) is less than or equal to the value of the string expression strArray$(j% + span%), the elements are already in order and the loop counter variable is set to the ending point. This terminates the FOR loop after this iteration.

- If the value of the second string expression is greater than the value of the first, the elements are exchanged by means of the SWAP statement. SWAP, introduced here for the first time, exchanges the values of any two variables of the same type. In this case, two string variables are exchanged. When the array is completely sorted, it is passed back to the main program and printed in its entirety by a FOR loop.

The Shell Sort
The logic in the ShellSort subprogram is based on the popular Shell Sort algorithm for sorting an array of numbers published in 1959 by Donald Shell. The Shell Sort sorts a list of elements by continually dividing the main list into smaller sublists that are smaller by half and comparing the elements at the tops and bottoms of the sublists. If the elements at the tops and bottoms of the lists are out of order, they are exchanged. The end result is an array of items in descending order.

To see other sorting routines in action, load and run the Animated Sort program from the Demonstration Programs folder. The comments in Animated Sort describe the operation of six sorts, including the Shell Sort.

Practice:
Using the <= relational operator and the SWAP Statement

Load and compile the Sort Strings program from disk and run it.

```
' Sort Strings
' This program prompts the user for a list of names and then sorts the
'   names alphabetically.

' Set maximum number of lines that can be entered and declare string
'   array to hold lines.

maxLines% = 15   ' maxLines% will be shared with the GetText subprogram
DIM inputLines$(maxLines%)

CLS

' Call GetText subprogram to get input from user; at return, the
'   numOfElements% variable will contain number of lines received.

CALL GetText (inputLines$(), numOfElements%)
```



```
' Call ShellSort subprogram to put inputLines$ array in
'   alphabetic order.

CALL ShellSort (inputLines$(), numOfElements%)

PRINT
PRINT "Sorting results:"
PRINT

FOR i% = 1 TO numOfElements%    ' print contents of sorted array
    PRINT inputLines$(i%)
NEXT i%

PRINT
INPUT "Press Return to continue…", dummy$

END

SUB GetText (strArray$(), count%) STATIC

' The GetText subprogram fills the strArray$ array with text
'   entered at the keyboard.  The number of lines that can be
'   entered is determined by the shared variable maxLines%.
'   Both strArray$ and count% (the number of lines actually
'   entered) are returned to the main program.

SHARED maxLines%     ' Get maxLines% from the main program

PRINT "Enter up to"; maxLines%; "lines of text; to end, ";
PRINT "press Return on a new line."
PRINT

count% = 0                     ' initialize variables to zero and "empty"
inLine$ = "empty"

' Loop until count% = maxLines% or an empty line is received.

WHILE (count% < maxLines%) AND (inLine$ <> "")
    LINE INPUT "-> "; inLine$  ' get line from user
    IF (inLine$ <> "") THEN    ' if line is not blank, copy it
        count% = count% + 1    '   to the strArray$ array
        strArray$(count%) = inLine$
    END IF
WEND

END SUB

SUB ShellSort (strArray$(), numOfElements%) STATIC


' The ShellSort subprogram sorts the elements of strArray$ and
'   returns strArray$ to the main program.  The numOfElements%
'   argument contains the number of elements in strArray$.
'   ShellSort sorts elements in decending order.

span% = numOfElements% \ 2

WHILE span% > 0
    FOR i% = span% TO numOfElements% - 1
        j% = i% - span% + 1
        FOR j% = (i% - span% + 1) TO 1 STEP -span%
            IF strArray$(j%) <= strArray$(j% + span%) THEN
                j% = 1
            ELSE   ' swap array elements that are out of order
                SWAP strArray$(j%), strArray$(j% + span%)
            END IF
        NEXT j%
    NEXT i%
    span% = span% \ 2
WEND

END SUB
```

You'll see output similar to this:

```
Enter up to 15 lines of text; to end, press .Return on a new line.

->Halvorson, Mike
->Ullom, Kim
-> Halvorson, Ken
->Zell, Linda
-> Halvorson, Victor
->Zell, Ben
->Berquist, Evelyn
-> Gullickson, Emma
->

Sorting results:

   Berquist, Evelyn
   Gullickson, Emma
   Halvorson, Ken
   Halvorson, Mike
   Halvorson, Victor
   Ullom, Kim
   Zell , Ben
   Zell , Linda
```

SUMMARY
You covered a lot of ground in this chapter and added many new functions and statements to your repertoire of programming tools. In all, you were introduced to 12 new statements and functions.

- UCASE$
- LEN
- RIGHT$, LEFT$, MID$
- LINE INPUT
- SPACE$, STRING$
- INSTR
- CHR$, ASC
- SWAP

You also learned how to declare, combine, take apart, compare, and sort strings. AC/QuickBasic provides such a variety of tools for working with strings because so much of what's done with personal computers revolves around working with large amounts of text data. The next chapter discusses efficient ways to manage all that data.

QUESTIONS AND EXERCISES
1. True or False: A literal string is enclosed in double quotation marks.
2. True or False: The following statement correctly dimensions a one-dimensional string array containing 10 strings:

```
DIM trees$(9)
```

3. What output does the following AC/QuickBasic statement produce?

```
PRINT "ONE" + "TWO" + "THREE"
```

4. Which of the following values can be supplied as an argument to the LEN function?
   a. A literal string
   b. A string variable
   c. The result of a string function

5. How should you interpret a call to the INSTR function that returns the value 0?

6. What does the following AC/QuickBasic statement display?
```
PRINT CHR$(ASC("h") - 32)
```

7. What is the ASCII code for the letter M?

8. Write a program that prompts the user for a first name and a last name, converts the names to uppercase, and then prints them in the format Lastname, Firstname.

9. Write a program that gets a string from the user, reverses the order of the characters in the string, and displays the new string.

10. Write a program that gets a full name from the user with one string variable and copies each name in the string to a separate string variable. Your program should be able to process names of any length, provided that the user separates the three subnames with spaces. Output from the program should resemble this:

```
Enter a string: Queen Victoria Belfield

First name: Queen
Middle name: Victoria
Last name: Belfield
```


In Chapters 3 and 4, you met up with the string data type and practiced creating strings and displaying them on your screen. In this chapter we'll continue that discussion and introduce many of the functions available in AC/QuickBasic for working with strings. We'll describe

- Assigning user input to a string
- Combining strings
- Selecting characters from a string
- Comparing strings
- Sorting strings

When combined with the skills you've developed in preceding chapters, these functions can help you perform a wide variety of tasks.

STRINGS: AN OVERVIEW
A string is a series of consecutive characters that you use as a unit. Typically, you store information as a string when you can't easily store it as a numeric data type. For example, because of its textual nature, the title of a book --- let's say The Original Mother Goose --- would be stored as a string rather than as a numeric data type.

You might find it helpful to think of a string as occupying a series of memory locations in a computer.
You can think of the memory locations as little boxes set side by side,
--- each box holding one character of the string --- as shown in Figure 9-1.
The memory locations, or boxes, are fixed in place, but you can move the characters among the boxes at will.
You can also add characters to or remove characters from the string as the need arises.

  You can use the following characters in a string:
- Uppercase letters of the alphabet (A through Z)
- Lowercase letters of the alphabet (a through z)
- Numerals (0 through 9)
- Punctuation symbols(. , ; : ' ? !)
- Mathematical symbols (# % ( ) - + = \ / < >)
- Miscellaneous symbols and foreign language characters

H	i	c	k	o	r	y		D	i	c	k	o	r	y		D	o	c	k

FIGURE 9-1.
A string is a series of characters.

TWO TYPES OF STRINGS
AC/QuickBasic supports two types of strings for use in your programs:

literal strings and string variables. Let's review how you use these types of strings in AC/QuickBasic.

Literal Strings
A literal string is a series of consecutive characters enclosed in double quotation marks.
You generally assign literal strings to a variable or use them as arguments to a statement or function.

For example, the following statements contain literal strings:

```
cheer$= "Go Seahawks!"

PRINT "1313 Mockingbird Lane"

goal$= FNCenterString$("Today I shall fly a kite.")

birthDate$ = "11-19-63"
```

Literal strings are also known as string values.

String Variables
The contents of a string variable can change at any time during the execution of a program. We've been using string variables throughout this document to obtain, store, and pass information in our code.
A string variable can contain from 0 through 32,767 characters and its length can grow or shrink during the execution of a program.

You can declare a string variable in two ways:

- By appending the string type-declaration character($) to the variable name.
  The following statement, for example, gets a string from the user and assigns it to the string variable firstName$.

```
INPUT "Enter your first name: ", firstName$
```

As you examine the examples in this book, you'll see many instances in which INPUT statements contain string variables that are meaningful names for particular kinds of words and phrases. We also use string variables as arguments to statements and functions such as PRINT.

- By using the DIM statement to declare an array of string variables.
  We described this method in detail in Chapter 8.
  The following statement is an example that declares an array of 10 string variables
  (assuming that the statement OPTION BASE 1 appears first):

```
DIM names$(10)
```

Practice Using string variables
The Phone Variables program (Figure 9-2) uses INPUT statements and a two-dimensional string array named contacts$ to store a list of friends and their telephone numbers. (Because telephone numbers often contain characters that are not numerals, such as dashes and letters, it's a good idea to store them as strings.)

Load and compile the Phone Variables program from the Chapter 9 folder on disk and run it.

```
' Phone Variables
' This program uses a string array to record names and telephone numbers.

OPTION BASE 1              ' set lower bound of array to 1

CLS

INPUT "How many names would you like to enter?  ", names%
PRINT

DIM contacts$(names%, 2)  ' declare array for names and phone numbers

FOR i% = 1 TO names%      ' read names into contacts$ array
    INPUT "Enter name:  ", contacts$(i%, 1)
    INPUT "Enter phone number:  ", contacts$(i%, 2)
    PRINT
NEXT i%

PRINT "You entered the following contact list:"
PRINT

FOR i% = 1 TO names%      ' print contents of array
    PRINT "Name:  "; contacts$(i%, 1), "Phone:  "; contacts$(i%, 2)
NEXT i%

PRINT
INPUT "Press Return to continue…", dummy$
```

You'll see output similar to the following:

```
How many names would you like to enter? 3

Enter name: Little Bo-Peep
Enter phone number: 555-LAMB

Enter name: Little Jack Horner
Enter phone number: 555-PLUM

Enter name: Little Boy Blue
Enter phone number: 555-HORN

You entered the following contact

Name: Little Bo-Peep             Phone: 555-LAMB
Name: Little Jack Horner         Phone: 555-PLUM
Name: Little Boy Blue            Phone: 555-HORN
```

Note that you achieve the alignment of the phone number column by using a comma in the PRINT statement to form two columns. In this case, advancing to the next print zone is enough to align the items.

Combining Strings
One of the simplest things you can do with strings is to combine them to form longer strings. This process is called concatenation. You concatenate strings primarily to prepare text for output on the screen or a printer or to prepare them for storage in an array or a file. You can concatenate both literal strings and string variables in any combination, and you can assign the result to a string variable or supply the result as an argument to a statement or function that expects string values (such as PRINT).

For example,
The following statement uses the concatenation operator (+) to combine the literal strings Microsoft, AC/QuickBasic, and Interpreter/Compiler and assigns the result to the string variable language$:

```
language$ = "Microsoft" + "AC/QuickBasic" + "Interpreter/Compiler"
```

The concatenation operator combines the three literal strings to form one string. The assignment operator(=) assigns the result to the string variable language$. Note that there are no spaces in the literal strings in this statement. (The spaces surrounding the operators don't count.) If you were to print out the value of language$ by means of the PRINT statement, you'd see the following output:

```
MicrosoftAC/QuickBasicInterpreter/Compiler
```

The concatenation operator combines the strings exactly as they are, without adding spaces. To include spaces, you must add spaces to the literal strings themselves; that is, a space needs to appear within the quotation marks.

Any of the following statements would accomplish this:

```
• PRINT "Microsoft" + " AC/QuickBasic" + " Interpreter/Compiler"
• PRINT "Microsoft " + "AC/QuickBasic " + "Interpreter/Compiler"
• PRINT "Microsoft" + " " + "AC/QuickBasic" + " " +"Interpreter/Compiler"
```

You can also supply the result of a concatenation as an argument to some AC/QuickBasic statements directly - without assigning the result to an intermediate variable such as language$:

```
PRINT "Microsoft" + "AC/QuickBasic" + "Interpreter/Compiler"
```

The result of the preceding statement is the same as the result of

```
   language$ = "Microsoft" + "AC/QuickBasic" + "Interpreter/Compiler"
   PRINT language$
```

Practice:
Concatenating strings
The Add Strings program (Figure 9-3),
Demonstrates a number of the options available to you through string concatenation.

Load and compile the Add Strings program from disk and run it.

```
' Add Strings
' This program demonstrates string concatenation.

structure$ = "Bridge"        ' initialize string variables
action$ = "is falling"

OPTION BASE 1
DIM direction$(5)            ' dimension string array with 5 elements
direction$(1) = "down"       ' put "down" in first array location

CLS

INPUT "Please enter the name of a city:  ", city$

' Print the news flash.
PRINT
PRINT "News Flash:  ";
PRINT city$ + " " + structure$ + " " + action$ + " " + direction$(1) + "!"

PRINT
INPUT "Press Return to continue…", dummy$
```

FIGURE 9-3.
Add Strings: a program that demonstrates string concatenation.

If you respond to the city prompt with London, you'll see this output:

```
Please enter the name of a city: London
News Flash: London Bridge is falling down!
```

PUTTING STRING FUNCTIONS TO WORK
So far you've declared string arrays and variables, combined strings in a process called concatenation, and used strings as arguments in INPUT and PRINT statements. In this section you'll learn about AC/QuickBasic functions that are specifically designed to manipulate and return values from literal strings and string variables.

You'll learn how to.

```
• Change a string to uppercase letters
• Determine the length of a string
• Take strings apart
```

Changing a String to Uppercase Letters
It's easy to change a string's letters to all uppercase. Simply use the UCASE$ function. UCASE$ is handy when you want to give text on the screen extra emphasis. It's also useful when you want to convert all user input to the same format. We'll discuss the importance of this when we cover how strings are compared at the end of the chapter.

Here's the syntax for the UCASE$ function:

```
UCASE$(stringexpression)
```

stringexpression is any kind of string.
The value returned by UCASE$ can be assigned to a string variable or supplied as an argument to a statement or function that accepts string values.

UCASE$ affects only the lowercase letters of stringexpression.

Practice:
Using the UCASE$ function
The Uppercase program (Figure 9-4) demonstrates how the UCASE$ function works. Uppercase declares three string variables and then uses the UCASE$ function to display them in uppercase. Note that the UCASE$ function affects only the output of the PRINT statement - it doesn't change the contents of the writer$ and address$ variables.
When the are displayed again at the end of the program, only the borough$ variable retains uppercase letters, by virtue of its original string assignment.

Load and compile the Uppercase program from disk and run it.

```
' Uppercase
' This program demonstrates the UCASE$ function.

writer$ = "Sir Arthur Conan Doyle"
address$ = "1326 Serpentine Avenue"
borough$ = UCASE$("St. John's Wood")

CLS

PRINT UCASE$(writer$)
PRINT UCASE$(address$) + ", " + borough$
PRINT
PRINT writer$
PRINT address$ + ", " + borough$

PRINT
INPUT "Press Return to continue…", dummy$
```

Figure 9-4.
Uppercase: a program that demonstrates use of the UCASE$ function.

You'll see the following output:

```
SIR ARTHUR CONAN DOYLE
1326 SERPENTINE AVENUE, ST. JOHN'S WOOD

Sir Arthur Conan Doyle
1326 Serpentine Avenue, ST. JOHN'S WOOD
```

Determining the Length of a String
Often you'll want to know how many characters are in a string. This knowledge can be particularly handy with strings entered from the keyboard. To determine how many characters (including spaces) are in a string, use the LEN function.

Here's the syntax for the LEN function:

```
LEN(stringexpression)
```

stringexpression once again is any kind of string. The value returned by LEN can be assigned to an integer variable or supplied as an argument to a statement or function that accepts integer values.

The following routine shows how the LEN function determines the number of characters in a string and assigns the number to an integer variable:

```
fullName$ = "Old Mother Hubbard"
nameLength% = LEN(fullName$)
PRINT fullName$; " is"; nameLength%; "characters long."
```

When you execute this routine, you see the following output:

```
Old Mother Hubbard is 18 characters long.
```

Practice:
Using the LEN function
The String Length program (Figure 9-5) demonstrates how the value returned by the LEN function can be used as an argument to a PRINT statement. Load and compile the String Length program from disk and run it.

```
' String Length
' This program demonstrates the LEN function.

CLS

INPUT "What is your favorite meal?  ", meal$
PRINT
PRINT UCASE$(meal$); " is"; LEN(meal$); "characters long."

PRINT
INPUT "Press Return to continue…", dummy$
```

FIGURE 9-5.
String Length: a program that demonstrates use of the LEN function.

You'll see output similar to this:

```
What is your favorite meal? Kim's Thai Stir Fry

KIM'S THAI STIR FRY is 19 characters long.
```


Taking Strings Apart
Earlier in this chapter you learned that AC/QuickBasic lets you combine strings through concatenation. Sometimes you'll want to take a string apart - to get only a person's last name from a variable containing a full name, for example.
AC/QuickBasic provides four functions that allow you to work with parts of strings.
You'll learn how to use these functions to;

- Get the right end of a string (RIGHT$)
- Get the left end of a string (LEFT$)
- Get the middle of a string (MID$)
- Find a string within a string (INSTR)

You'll also learn about statements that let you

- Get an entire line of input (LINE INPUT$)
- Print repeated characters (SPACE$, STRING$)

Getting the ends of a string
The RIGHT$ and LEFT$ functions let you retrieve one or more characters starting from one end of a string. This is useful when you want to display only part of a string or when you want to remove part of a string.

Here's the syntax for the RIGHT$ function:

```
RIGHT$(stringexpression, n)
```

Here's the syntax for the LEFT$ function:

```
LEFT$(stringexpression, n)
```

stringexpression is any kind of string, and n is an integer value ranging from 0 through the length of the string that indicates the number of characters to be returned by RIGHT$ or LEFT$. You can assign the value returned to a string variable or supply the value as an argument to a statement or function that accepts string values.

Practice:
Using the RIGHT$ function
The Get Right program (Figure 9-6) uses the RIGHT$ function to retrieve characters from a variable named alphabet$, which contains the 26 letters of the alphabet. Get Right extracts the requested number of characters and displays them with a character count.

Load and compile the Get Right program from disk and run it.

```
' Get Right
' This program demonstrates the RIGHT$ function.

CLS

alphabet$ = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"   ' declare test string

PRINT "How many characters (from right to left) in the following"
PRINT "string would you like to display?"
PRINT
PRINT alphabet$                            ' display test string
PRINT

' Prompt user for the number of rightmost characters to be displayed.
' Loop until the number is in the proper range (1 through 26).

WHILE (rightNum% < 1) OR (rightNum% > 26)
    INPUT "    Number (1-26):  ", rightNum%
WEND

PRINT
rightChar$ = RIGHT$(alphabet$, rightNum%)  ' display characters
PRINT "You specified"; LEN(rightChar$); "characters:  "; rightChar$

PRINT
INPUT "Press Return to continue…", dummy$
```

Figure 9-6
Get Right: a program that demonstrates use of the RIGHT$function.

You'll see output similar to this:

```
How many characters (from right to left) in the following string would you like to display?

ABCDEFGHJJKLMNOPQRSTUVWXYZ

Number (1-26): 14
You specified 14 characters: MNOPQRSTUVWXYZ
```

Practice:
Using the LEFT$ function
The Get Left program (Figure 9-7) revises the Get Right program to extract characters from the left side of a string with the LEFT$ function. Notice that the variable names have changed slightly (rightNum% becomes leftNum%, and rightChar$ becomes leftChar$) and that the function RIGHT$ has been changed to LEFT$.

Apart from these changes (and a few changes to the prompt and program comments), Get Left is identical to Get Right. Because the operations of the RIGHT$ and LEFT$ functions are so similar, it's quite easy to change a program so that it modifies a string from the opposite end.

Load and compile the Get Left program from disk and run it.

```
' Get Left
' This program demonstrates the LEFT$ function.

CLS

alphabet$ = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"   ' declare test string

PRINT "How many characters (from left to right) in the following"
PRINT "string would you like to display?"
PRINT
PRINT alphabet$                            ' display test string
PRINT

' Prompt user for the number of leftmost characters to be displayed.
' Loop until the number is in the proper range (1 through 26).

WHILE (leftNum% < 1) OR (leftNum% > 26)
    INPUT "    Number (1-26):  ", leftNum%
WEND

PRINT
leftChar$ = LEFT$(alphabet$, leftNum%)     ' display characters
PRINT "You specified"; LEN(leftChar$); "characters:  "; leftChar$

PRINT
INPUT "Press Return to continue…", dummy$
```

You'll see output similar to this:

```
  How many characters (from left to right) in the following string would you like to display?

  ABCDEFGHIJKLMNOPQRSTUVWXYZ

       Number (1-26): 14

  You specified 14 characters: ABCDEFGHIJKLMN
```

Getting the middle of a string
The MID$ function lets you retrieve one or more characters from anywhere within a string-from the left, from the middle, or (with some help from the LEN function) from the right. Its versatility makes the MID$ function one of the useful string functions. And, as we'll see later, it provides the processing power to solve many string-related problems.
Here's the syntax for the MID$ function:

```
MID$(stringexpression, start, length)
```

stringexpression is any kind of string, start is an integer value between 1 and the length of the string (indicating the first character to be returned), and length is an integer value indicating the number of characters to be returned. You can assign the value returned by MID$ to a string variable or supply the value as an argument to a statement or function that accepts string values. Figure 9-8 shows the elements of the MID$ function syntax in detail.

```
  MID$             Position of first
function        character to be returned
   |                    |
MID$(stringexpression,   start,   length
        |                           |
          String to             Number of
         be searched     characters to be returned
```

FIGURE 9-8.
The components of the MID$function.

The following statements show some nifty uses of the MID$ function.
Notice the powerful possibilities that arise when you use the value returned by a function as an argument to MID$ or when you assign the value returned by MID$ to another statement or function.

```
middleName$ = MID$("Queen Victoria Belfield", 7, 8)
Result: middleName$ contains Victoria

address$= "1521 Plumtree Lane #25-K"
streetNameStart% = 6
```



```
length% = 13
PRINT UCASE$(MID$(address$, streetNameStart%, length%))
Result: PLUMTREE LANE

inString$ ="Making it all make sense"
rightmostWord$ = MID$(inString$, LEN(inString$) - 4, 5)
Result: rightmostWord$ contains sense

PRINT "The current year is "; MID$(DATE$, 7, 4)
Result: The current year is 1991
```

Practice
Using the MID$ function
The Get Middle program (Figure 9-9 on the next page) shows how to retrieve characters from the middle of a string with the MID$ function. Get Middle modifies the Get Right and Get Left programs to include a starting point along with the number of characters in the alphabet$ string to be displayed. Get Middle uses two WHILE loops to get integer values in the proper range and then uses the MID$ function to assign the selected characters to the midChar$ variable.

The results of the selection are printed with the following IF statement, which appears near the end of the program:

```
IF (numToDisplay% = LEN(midChar$)) THEN
    PRINT numToDisplay%; "characters displayed: "; midChar$
ELSE
    PRINT numToDisplay%; "characters requested,";
    PRINT LEN(midChar$); "displayed: "; midChar$
END IF
```

The IF statement compares numToDisplay% - the variable containing the number of characters the user asked to see displayed - to the number of characters in midChar$ (returned by the LEN function). If the two values are equal, the value of numToDisplay% is printed along with the contents of midChar$. If the two values are not equal, the LEN function determines the actual number of characters and this value is displayed along with the midChar$ string. Get Middle contains this additional message to notify the user that the display length entered exceeded the number of characters remaining in the string.

Load and compile the Get Middle program from disk and run it.

```
' Get Middle
' This program demonstrates the MID$ function.

CLS

alphabet$ = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"  ' declare test string

PRINT "How many characters (from left to right) in the following"
PRINT "string would you like to display?"
PRINT
PRINT alphabet$                           ' display test string
PRINT

' Prompt user for the number of characters to be displayed.
' Loop until the number is in the proper range (1 through 26).

WHILE (numToDisplay% < 1) OR (numToDisplay% > 26)
    INPUT "    Number (1-26):  ", numToDisplay%
WEND

PRINT                                     ' get starting number...
PRINT "What character would you like to start with?"
PRINT

WHILE (start% < 1) OR (start% > 26)    '   in proper range
    INPUT "    Starting number (1-26):  ", start%
WEND

PRINT                                     ' get characters
midChar$ = MID$(alphabet$, start%, numToDisplay%)

' Compare requested characters with actual characters retrieved
' and print an appropriate message.

IF (numToDisplay% = LEN(midChar$)) THEN
    PRINT numToDisplay%; "characters displayed:  "; midChar$
ELSE
    PRINT numToDisplay%; "characters requested,";
    PRINT LEN(midChar$); "displayed:  "; midChar$
END IF

PRINT
INPUT "Press Return to continue…", dummy$
```

You'll see output similar to this:

```
How many characters (from left to right) in the following string would you like to display?

ABCDEFGHIJKLMNOPQRSTUVWXYZ
Number (1-26): 14
What character would you like to start with?
Starting number (1-26): 4
14 characters displayed: DEFGHIJKLMNOPQ
```

If you specify a number outside the range permitted by Get Middle, you see output similar to this:

```
How many characters (from left to right) in the following string would you like to display?

ABCDEFGHIJKLMNOPQRSTUVWXYZ

    Number (1-26): 0
    Number (1-26): 30
    Number (1-26): 15

What character would you like to start with?

      Starting number (1-26): w
     ?Redo from start
      Starting number (1-26): 20

      15 characters requested, 7 displayed: TUVWXYZ
```

If you type in a non-numeric value (such as a letter) at one of the prompts, AC/QuickBasic prints the message ?Redo from start and redisplays the prompt. Handling this type of response from the user is important in developing "break-proof" programs.

Getting an entire line of input from the user
Throughout this book we've used the INPUT statement to get input from the user. The INPUT statement is quite versatile - it can assign input to one or more variables of different types and supply an optional prompt to spell out exactly what the user should enter.

We haven't discussed how the INPUT statement processes a comma in the input line. Consider the following statement, which prompts the user to enter a name and address:

```
INPUT "Enter name and address: ", mailingAddress$
```

If the user responds to the prompt with a string containing commas, the error message ?Redo from start appears, as shown in the following dialogue:

```
Enter name and address: Jon Victor, 1118 Skyridge, Lacey, WA, 98503
?Redo from start
Enter name and address:
```

As we noted in Chapter 4, the INPUT statement has a special use for the comma character: The comma separates the assigned to variables. But what happens when the user types unexpected commas - as shown above?
You could provide for commas by assigning parts of the input string to different variables.
The following INPUT statement, for example, assigns the string value entered by the user to five string variables:

```
INPUT "Enter name and address: ", cust$, addr$, city$, state$, zip$
```

But there are times when it would be a lot simpler to have only one variable name associated with a line of input. AC/QuickBasic provides a solution to this problem with the LINE INPUT statement. The LINE INPUT statement reads an entire line of text from the keyboard and assigns it to a string variable, regardless of whether commas are present.

Here's the syntax for the LINE INPUT statement:

```
LINE INPUT[;] ["promptstring";] stringvariable
```

promptstring is a literal string that prompts the user for input, and stringvariable is any string variable.

- Placing a semicolon immediately after LINE INPUT keeps the cursor on the same line after the user presses Return.
- If promptstring is included in the statement, a following semicolon is required to separate promptstring from stringvariable.
- Unlike the INPUT statement, the LINE INPUT statement prints no question mark unless the question mark is included in promptstring.

The following statements demonstrate the usefulness of LINE INPUT for long lines of input that contain the comma character:

```
LINE INPUT "Enter name and address: "; mailingAddress$
PRINT mailingAddress$
```


When you execute the statements and enter the information we tried to enter earlier, you see this:

```
Enter name and address: Jon Victor, 1118 Skyridge, Lacey, WA, 98503
Jon Victor, 1118 Skyridge, Lacey, WA, 98503
```

You'll see the LINE INPUT statement from time to time in later chapters.

Printing repeated characters
AC/QuickBasic provides two useful functions that generate strings of repeated characters:
The SPACE$ function, which returns a string of spaces, and the STRING$ function, which returns a string of characters.
Both functions give you fast ways to build strings you can use in formatting and aligning your program's output.

Here's the syntax for the SPACE$ function:

```
SPACE$(n)
```

n is an integer value specifying the number of spaces the string will contain. You can assign the value returned by SPACE$ to a string variable or supply the value as an argument to a statement or function that accepts string values.
The most common use of the SPACE$ function is in formatting output, as in the following routine:

```
blank$= SPACE$(15)
PRINT blank$; "Big sale on bunnies today!"
```

SPACE$ comes in handy whenever you consistently indent text a set number of spaces.

Here's the syntax for the STRING$ function:

```
STRING$(m, stringexpression)
```

m is an integer value that specifies the length of the string to be returned, and stringexpression is the character to be repeated. You can assign the value returned by STRING$ to a string variable or supply the value as an argument to a statement or function that accepts string values.

Practice:
Using the SPACE$ and STRING$ functions
The most common use of the STRING$ function is for headings used in program output.
The Header program (Figure 9-10) uses STRING$ to display a header message in the middle of the screen.
Note that using a variable to specify the number of repeated characters makes it easy to modify the program later and that the variables created by SPACE$ and STRING$ make excellent candidates for concatenation.

Load and compile the Header program from disk and run it.

```
' Header
' This program demonstrates the SPACE$ and STRING$ functions.

length% = 15

fileName$ = "Sweet Pea Inventory"
blank$ = SPACE$(length%)
asterisk$ = STRING$(length%, "*")
banner$ = blank$ + asterisk$ + " " + fileName$ + " " + asterisk$

CLS

PRINT banner$

PRINT
INPUT "Press Return to continue…", dummy$
```

FIGURE 9-10.
Header: a program that demonstrates use of the SPACE$ and STRING$ functions.

You'll see the file name Sweet Pea Inventory in the middle of the top line of the screen, surrounded by equal numbers of asterisks and blank spaces.

Finding a string within a string
We've used the RIGHT$, LEFT$, and MID$ functions in this chapter to return characters from the right, left, and middle portions of a string.

These functions are quite effective at extracting characters from a set place in a string, but they are less effective at searching for and extracting a specific pattern from a string. AC/QuickBasic fills this gap with the INSTR function, which searches for a string within another string. INSTR joins RIGHT$, LEFT$, MID$, and the support functions we've discussed in this section (UCASE$, LEN, SPACE$, and STRING$) to round out a complete collection of tools for working with strings.

Here's the syntax for the INSTR function:

```
INSTR([start, ]basestring, searchstring)
```

start is an optional integer value specifying the character at which the search should begin, basestring is the string to be searched, and searchstring is the string to be found within basestring. You can assign the value returned by INSTR to an integer variable or supply the value as an argument to a statement or function that accepts integer values.

The following table lists the values that the INSTR function can return:

Condition	Integer value returned
searchstring found in basestring	Position in basestring at which match is found
searchstring not found in basestring	0
start is greater than length of basestring	0
basestring contains no characters	0
searchstring contains no characters	start (if given); otherwise, 1

Practice:
Using the INSTR/unction

The Find a String program (Figure 9-11) uses the INSTR function to search for the string iddle in the string
 High, diddle, diddle, the cat and the fiddle.

Load and compile the Find a String program from disk and run it.

```
' Find a String
' This program demonstrates the INSTR function.

CLS

baseStr$ = "High, diddle, diddle, the cat and the fiddle"
searchStr$ = "iddle"
strLocation% = INSTR(1, baseStr$, searchStr$)

IF (strLocation% <> 0) THEN
    PRINT searchStr$; " first appears starting at character"; strLocation%
ELSE
    PRINT searchStr$; " not found"
END IF

PRINT
INPUT "Press Return to continue…", dummy$
```

FIGURE 9-11.
Find a String: a program that demonstrates use of the INSTR function.

When you run the program, you see this:

```
iddle first appears starting at character 8
```

Although the pattern iddle appears three times in baseStr$, the INSTR function returns the location of only the first occurrence. To find multiple occurrences of a pattern, you can use INSTR within a loop. Whenever you use the INSTR function, it's a good idea to have your program check the value returned by INSTR. This will allow your program to take appropriate action if the search string is not found, or if either the search string or the base string is empty. In the Find a String program, if iddle were not found, the INSTR function would assign a value of 0 to to strLocation% variable and the IF statement would display the following message to indicate that iddle did not exist in baseStr$:

```
iddle not found
```

Practice:
Finding multiple occurrences of a pattern
The Find Many Strings program (Figure 9-12) uses the INSTR function to find multiple occurrences of a pattern in a series of text lines entered from the keyboard.

The heart of Find Many Strings is the Repeat subprogram. Each time INSTR finds the search string (searchStr$) in the base string (baseStr$), the location of the search string is assigned to currentChar%. Then, when the num% variable is incremented, the currentChar% variable is moved ahead to the position just after the string match in the base string (the new starting place for the next search). This process continues until the end of the base string is reached or the search string is not found in the remainder of the base string.

The Repeat subprogram then passes the total number of matches in the line to the lineRepeats% variable in the main program.

Load and compile the Find Many Strings program from disk and run it.

```
' Find Many Strings
' This program prompts the user for a set number of lines and a search
'   string and then prints the lines and the number of matches found.

' Set maximum number of lines that can be entered and declare string
'   array to hold lines.

maxLines% = 10   ' maxLines% will be shared with the GetText subprogram
DIM inputLines$(maxLines%)

CLS

' Call GetText subprogram to get input from user; at return, the
'   numOfLines% variable will contain number of lines received.

CALL GetText (inputLines$(), numOfLines%)
```



```
' Get pattern to be searched for from user.

PRINT
INPUT "Enter the string to be searched for:  ", pattern$
PRINT

' Call Repeat subprogram to determine the number of matches per line.
' The totalRepeats% variable will accumulate the number of total matches.
FOR i% = 1 TO numOfLines%
    CALL Repeat (pattern$, inputLines$(i%), lineRepeats%)
    totalRepeats% = totalRepeats% + lineRepeats%
NEXT i%

' Display lines entered by the user…

PRINT "You entered the following lines:"
PRINT
FOR i% = 1 TO numOfLines%
    PRINT inputLines$(i%)
NEXT i%

' and the total number of matches.

PRINT
PRINT "The pattern '"; pattern$; "' appears"; totalRepeats%; "times."

PRINT
INPUT "Press Return to continue…", dummy$

END

SUB GetText (strArray$(), count%) STATIC

' The GetText subprogram fills the strArray$ array with text
'   entered at the keyboard.  The number of lines that can be
'   entered is determined by the shared variable maxLines%.
'   Both strArray$ and count% (the number of lines actually
'   entered) are returned to the main program.

SHARED maxLines%    ' get maxLines% from the main program

PRINT "Enter up to"; maxLines%; "lines of text; to end, ";
PRINT "press Return on a new line."
PRINT

count% = 0                    ' initialize variables to 0 and "empty"
inLine$ = "empty"

' Loop until count% = MAXLINES% or an empty line is received.

WHILE (count% < maxLines%) AND (inLine$ <> "")
    LINE INPUT "-> "; inLine$  ' get line from user
    IF (inLine$ <> "") THEN       ' if line is not blank, copy it
        count% = count% + 1          '   to the strArray$ array
        strArray$(count%) = inLine$
    END IF
WEND

END SUB

SUB Repeat (searchStr$, baseStr$, num%) STATIC

' The Repeat subprogram returns the number of times a search string
'   is found in a base string.  The number is returned to the main
'   program through the num% parameter.

searchLength% = LEN(searchStr$)  ' determine length of search string
baseLength% = LEN(baseStr$)        ' determine length of base string
currentChar% = 1                ' character offset in base string
num% = 0                           ' running total of matches found in base

' Loop until entire string is processed or INSTR returns a 0.

WHILE (currentChar% <= baseLength%) AND (currentChar% <> 0)
    currentChar% = INSTR(currentChar%, baseStr$, searchStr$)
    IF (currentChar% <> 0) THEN  ' if not 0, a match was found
        num% = num% + 1                ' increment number of matches
        ' New offset equals current offset plus search-string length.
        currentChar% = currentChar% + searchLength%
    END IF
WEND

END SUB
```

You'll see output similar to this:

```
      Enter up to 10 lines of text; to end, press Return on a new line.

       -> Ten lords a-leaping,
       -> Nine ladies dancing,
       -> Eight maids a-milking,
       -> Seven swans a-swimming,
       -> Six geese a-laying,
       -> Five gold rings,
       -> Four calling birds,
       -> Three French hens,
       -> Two turtledoves, and
       -> A partridge in a pear tree.

      Enter the string to be searched for: ing

      You entered the following lines:

        Ten lords a-leaping,
        Nine ladies dancing,
        Eight maids a-milking,
        Seven swans a-swimming,
        Six geese a-laying,
        Five gold rings,
        Four calling birds,
        Three French hens,
        Two turtledoves, and
        A partridge in a pear tree.

        The pattern 'ing' appears 7 times.
```

COMPARING STRINGS
In Chapter 4 you learned that you can compare one string to another and branch to another place in the program based on the result of the comparison. The following IF statement compares the variable reply$ to the literal string Y and prints a message based on the comparison:

```
reply$ = "Y"
IF (reply$ = "Y") THEN
    PRINT "The two string values are equal."
ELSE
    PRINT "The two string values are not equal."
END IF
```

If you execute the statements as they are above, you see this:

```
      The two string values are equal.
```

If you change the value of reply$ toy and then execute the statement, you see a different response:

```
      The two string values are not equal.
```

Why is this? After all, a Y is a y .... Or is it?
What criteria is AC/QuickBasic using for its string comparisons?

The ASCII Character Set
Before AC/QuickBasic can compare one character to another, it must convert each character into a number by using a translation table called the ASCII character set. AC/QuickBasic then compares the numbers, called ASCII codes, and returns the logical value true if the ASCII codes are equal or false if the codes are not equal.

ASCII Is an Acronym
ASCII stands for American Standard Code for Information Interchange.

The key word here is code:

Like the Morse code used in radio and telegraphy, ASCII is an internationally accepted code for representing characters, but ASCII is used in computers and telecommunication.

Appendix A lists all of the ASCII codes and the character associated with each code in several different fonts.

Each character in the ASCII character set is associated with a unique number;
The set contains 128 characters (codes 0 through 127) in all:

- Control characters (codes 0 through 31),
  including characters that correspond to special keys on your keyboard such as Return, Backspace, and Tab
- Punctuation symbols, numbers, and mathematical symbols (codes 32 through 64)
- Uppercase letters of the alphabet (codes 65 through 90)
- Lowercase letters of the alphabet (codes 97 through 122)
- Miscellaneous symbols (codes 91 through 96 and 123 through 127)

The ASCII code for the uppercase letter A is 65; the ASCII code for the lowercase letter z is 122.
Following this logic, you can see why AC/QuickBasic considers the uppercase letter Y (code 89) and the lowercase letter y (code 121) to be different characters.

Optional Characters
Appendix A also contains a set of characters (codes 128 through 255) known as optional characters.
This set of symbols was developed by Apple for Apple][gs computers and printers and has been adopted by most software publishers writing Apple][gs applications. The optional set (sometimes called upper ASCII characters) contains foreign-language characters and mathematical symbols. Although you can use these symbols in your programs, you can't type them by the usual means - they don't appear on your keyboard!
Instead, you hold down the Shift, Command, or Option key and type in a one-character or two-character code.
(Appendix A lists these codes along with the optional characters.)
For example, to display a bullet (•) on your screen, hold down the Option key and type 8.

The following PRINT statement demonstrates the valid use of an optional character in a AC/QuickBasic program.
The ¡ symbol was entered by holding down the Option key and typing 1.

```
PRINT "¡Vamos amigos!"
```

NOTE: Some characters in the ASCII and optional character sets will vary depending on the font you're using.
The Zapf Dingbats and Symbol fonts, for example, contain completely different graphical shapes and symbols.
Check your font documentation for a list of the character codes and keystrokes you need to use to produce the shapes and symbols of these fonts in your programs.

Converting Codes to Characters
If you know a character's code but you're not exactly sure what the character it represents looks like, you can use the CHR$ function to return the symbol to your screen or to your program.

Here's the syntax for the CHR$ function:

```
CHR$ (code)
```

code is an integer value that specifies an ASCII or optional character code.

You can assign the string value returned by CHR$ to a string variable or supply the value as an argument to a statement or function that accepts string values.

Practice:
Using the CHR$ function
The ASCII Codes program (Figure 9-13) shows how to use the CHR$ function to display the characters in the ASCII and optional character sets. Notice that the program skips the first 32 (control) characters in the ASCII character set and pauses after each multiple of 15 lines so that you can view the results at your own pace.

Load and compile the ASCII Codes program from disk and run it.
(Because of the length of the program's output, we won't show it here.)

```
' ASCII Codes
' This program displays the ASCII character set and the optional
'   Apple ][gs characters.

CLS

FOR i% = 33 TO 255
    PRINT "Code"; i%; "= "; CHR$(i%)
    IF (i% MOD 15 = 0) THEN INPUT "Press Return for more…", dummy$
NEXT i%
```

      FIGURE 9-13.
      ASCII Codes: a program that uses the CHR$ function to display the ASCII and optional character sets.

Converting Characters to Codes
As a complement to the CHR$ function, AC/QuickBasic provides the ASC function, which converts a character to its code in the ASCII or optional character set.

Here's the syntax of the ASC function:

```
ASC(stringexpression)
```

stringexpression is a one-character string. You can assign the integer value returned by ASC to an integer variable or supply the value as an argument to a statement or function that accepts integer values.

Using Relational Operators with Strings
In addition to testing for equivalence of characters, AC/QuickBasic supports string comparisons with the following relational operators:

Operator	Meaning	Operator	Meaning
<>	Not equal	>	Greater than
=	Equal	<=	Less than or equal to
<	Less than	>=	Greater than or equal to

A character is "greater than" another character if its ASCII code number is higher.

For example,
The ASCII value of the letter B is greater than the ASCII value of the letter A,

```
        so the expression
              "A" < "B" is true,
        and the expression
              "A" > "B" is false.
```

When comparing two strings each of which contains more than one character, AC/QuickBasic begins by comparing the first character in the first string to the first character in the other string and then proceeds through the strings character by character until it finds a difference. For example, the strings Mike and Michael are the same up to the third characters (k and c). Because the ASCII value of k is greater than that of c, the expression

```
      "Mike" > "Michael"
                is true.
```

If no differences are found, the strings are equal. If two strings are equal through several characters but one of the strings continues and the other one stops, the longer string is greater than the shorter string.

For example,

```
            the expression
              "AAAAA" > "AAA" is true.
```

Sorting Strings
The Sort Strings program (Figure 9-15) uses the <= relational operator to compare array elements and uses the SWAP statement to switch any elements that are out of order.

Sort Strings declares an array of strings named inputLines$ and calls the GetText subprogram.
The Get Text subprogram in Sort Strings is identical to the GetText subprogram in the Find Many Strings program (Figure 9-12): It reads lines of text into an array and then returns to the main program the array (inputLines$) and the number of elements in the array (numOfElements%).

Sort Strings next calls the ShellSort subprogram. Note the following statements in the heart of ShellSort that compare array elements and then swap the elements if they are out of order:

```
IF strArray$(j%) <= strArray$(j% + span%) THEN
     j% = 1
ELSE    ' swap array elements that are out of order
     SWAP strArray$(j%), strArray$(j% +span%)
END IF
```

The <= relational operator is used to compare the two array elements.

- If the value of the string expression strArray$(j%) is less than or equal to the value of the string expression strArray$(j% + span%), the elements are already in order and the loop counter variable is set to the ending point. This terminates the FOR loop after this iteration.

- If the value of the second string expression is greater than the value of the first, the elements are exchanged by means of the SWAP statement. SWAP, introduced here for the first time, exchanges the values of any two variables of the same type. In this case, two string variables are exchanged. When the array is completely sorted, it is passed back to the main program and printed in its entirety by a FOR loop.

The Shell Sort
The logic in the ShellSort subprogram is based on the popular Shell Sort algorithm for sorting an array of numbers published in 1959 by Donald Shell. The Shell Sort sorts a list of elements by continually dividing the main list into smaller sublists that are smaller by half and comparing the elements at the tops and bottoms of the sublists. If the elements at the tops and bottoms of the lists are out of order, they are exchanged. The end result is an array of items in descending order.

To see other sorting routines in action, load and run the Animated Sort program from the Demonstration Programs folder. The comments in Animated Sort describe the operation of six sorts, including the Shell Sort.

Practice:
Using the <= relational operator and the SWAP Statement

Load and compile the Sort Strings program from disk and run it.

```
' Sort Strings
' This program prompts the user for a list of names and then sorts the
'   names alphabetically.

' Set maximum number of lines that can be entered and declare string
'   array to hold lines.

maxLines% = 15   ' maxLines% will be shared with the GetText subprogram
DIM inputLines$(maxLines%)

CLS

' Call GetText subprogram to get input from user; at return, the
'   numOfElements% variable will contain number of lines received.

CALL GetText (inputLines$(), numOfElements%)
```



```
' Call ShellSort subprogram to put inputLines$ array in
'   alphabetic order.

CALL ShellSort (inputLines$(), numOfElements%)

PRINT
PRINT "Sorting results:"
PRINT

FOR i% = 1 TO numOfElements%    ' print contents of sorted array
    PRINT inputLines$(i%)
NEXT i%

PRINT
INPUT "Press Return to continue…", dummy$

END

SUB GetText (strArray$(), count%) STATIC

' The GetText subprogram fills the strArray$ array with text
'   entered at the keyboard.  The number of lines that can be
'   entered is determined by the shared variable maxLines%.
'   Both strArray$ and count% (the number of lines actually
'   entered) are returned to the main program.

SHARED maxLines%     ' Get maxLines% from the main program

PRINT "Enter up to"; maxLines%; "lines of text; to end, ";
PRINT "press Return on a new line."
PRINT

count% = 0                     ' initialize variables to zero and "empty"
inLine$ = "empty"

' Loop until count% = maxLines% or an empty line is received.

WHILE (count% < maxLines%) AND (inLine$ <> "")
    LINE INPUT "-> "; inLine$  ' get line from user
    IF (inLine$ <> "") THEN    ' if line is not blank, copy it
        count% = count% + 1    '   to the strArray$ array
        strArray$(count%) = inLine$
    END IF
WEND

END SUB

SUB ShellSort (strArray$(), numOfElements%) STATIC


' The ShellSort subprogram sorts the elements of strArray$ and
'   returns strArray$ to the main program.  The numOfElements%
'   argument contains the number of elements in strArray$.
'   ShellSort sorts elements in decending order.

span% = numOfElements% \ 2

WHILE span% > 0
    FOR i% = span% TO numOfElements% - 1
        j% = i% - span% + 1
        FOR j% = (i% - span% + 1) TO 1 STEP -span%
            IF strArray$(j%) <= strArray$(j% + span%) THEN
                j% = 1
            ELSE   ' swap array elements that are out of order
                SWAP strArray$(j%), strArray$(j% + span%)
            END IF
        NEXT j%
    NEXT i%
    span% = span% \ 2
WEND

END SUB
```

You'll see output similar to this:

```
Enter up to 15 lines of text; to end, press .Return on a new line.

->Halvorson, Mike
->Ullom, Kim
-> Halvorson, Ken
->Zell, Linda
-> Halvorson, Victor
->Zell, Ben
->Berquist, Evelyn
-> Gullickson, Emma
->

Sorting results:

   Berquist, Evelyn
   Gullickson, Emma
   Halvorson, Ken
   Halvorson, Mike
   Halvorson, Victor
   Ullom, Kim
   Zell , Ben
   Zell , Linda
```

SUMMARY
You covered a lot of ground in this chapter and added many new functions and statements to your repertoire of programming tools. In all, you were introduced to 12 new statements and functions.

- UCASE$
- LEN
- RIGHT$, LEFT$, MID$
- LINE INPUT
- SPACE$, STRING$
- INSTR
- CHR$, ASC
- SWAP

You also learned how to declare, combine, take apart, compare, and sort strings. AC/QuickBasic provides such a variety of tools for working with strings because so much of what's done with personal computers revolves around working with large amounts of text data. The next chapter discusses efficient ways to manage all that data.

QUESTIONS AND EXERCISES
1. True or False: A literal string is enclosed in double quotation marks.
2. True or False: The following statement correctly dimensions a one-dimensional string array containing 10 strings:

```
DIM trees$(9)
```

3. What output does the following AC/QuickBasic statement produce?

```
PRINT "ONE" + "TWO" + "THREE"
```

4. Which of the following values can be supplied as an argument to the LEN function?
   a. A literal string
   b. A string variable
   c. The result of a string function

5. How should you interpret a call to the INSTR function that returns the value 0?

6. What does the following AC/QuickBasic statement display?
```
PRINT CHR$(ASC("h") - 32)
```

7. What is the ASCII code for the letter M?

8. Write a program that prompts the user for a first name and a last name, converts the names to uppercase, and then prints them in the format Lastname, Firstname.

9. Write a program that gets a string from the user, reverses the order of the characters in the string, and displays the new string.

10. Write a program that gets a full name from the user with one string variable and copies each name in the string to a separate string variable. Your program should be able to process names of any length, provided that the user separates the three subnames with spaces. Output from the program should resemble this:

```
Enter a string: Queen Victoria Belfield

First name: Queen
Middle name: Victoria
Last name: Belfield
```


In Chapters 8 and 9 you learned how to manage large amounts of data in a program. In this chapter you'll learn how to save the data on disk and get at it whenever you like.

You'll also learn how to print information on the Apple ImageWriter and LaserWriter printers.

CREATING AND USING SEQUENTIAL FILES
A file is simply a collection of data saved on disk. One type of file you can create from within a AC/QuickBasic program is a sequential file. The contents of a sequential file must be used in order, from start to finish. (The contents of the other type of file you can create, a random-access file, can be used in any order, but we won't discuss random-access files in this book.)

You use these statements and functions in your AC/QuickBasic programs to create and work with sequential files:

Statement/Function	Description
OPEN	Opens a file
CLOSE	Closes a file
PRINT#	Prints unformatted data in a file
PRINT# USING	Prints formatted data in a file
WRITE#	Prints data organized into fields in a file
INPUT#	Gets data from a file
EOF	Checks for the end of a file
LINE INPUT#	Gets an entire line of data from a file

In this chapter you'll look at these statements and functions in detail and learn how to use sequential files to store many types of information.

NOTE: In this chapter, we'll be creating files in the folder that contains the AC/QuickBasic Interpreter/Compiler because that's where AC/QuickBasic places files by default. If you know enough about volumes and folders to create files in other folders, feel free to do that.

AC/QuickBasic can take full advantage of the Apple][gs Hierarchical File System (HFS).

Creating and Opening a File
The OPEN statement does double duty: You use OPEN both to create new files and to open existing files. Here's the syntax for the OPEN statement:

```
OPEN filename FOR mode AS #filenumber
```

filename is a valid Apple][gs file name, mode is a word that indicates how the file is to be used, and file number is an integer from 1 through 255 to be associated with the file that is opened.

NOTE: Because you assign a number to your file, you can refer to it within your program simply by calling it by number. You should start numbering files within your programs with 1. The number you assign to a file is valid until you close the file.

Specifying a mode
You can open a file in only one of three possible modes at a time. You use the mode argument to specify how you intend to use the sequential file.

Use one of the following mode arguments with an OPEN statement:

Mode	Description
OUTPUT	Creates and opens a file that will receive output from the program. If the file already exists, its old contents are erased.
APPEND	Opens an existing file to which output from the program is appended; that is, output from the program is placed at the end of the file. The original contents of the file are preserved. If the file can't be located, a dialog box appears to help you find it. If the file still can't be found, OPEN generates an error message.
INPUT	Opens a file that the program can only read. The program cannot change the file, but it can use the file's contents as input. As with APPEND, if the file can't be located, a dialog box appears to help you look for it. OPEN generates an error message if the file can't be found.

Some sample OPEN statements that show the use of each mode follow.
Here's a sample OPEN statement using OUTPUT mode:

```
OPEN "Name Data" FOR OUTPUT AS #1
```

This statement opens the Name Data sequential file, which will store output from the program. If Name Data does not exist, AC/QuickBasic creates it.

If it does exist, AC/QuickBasic erases its existing contents.

Name Data is associated with the number 1.

Here's a sample OPEN statement using APPEND mode:

```
OPEN "Name Data" FOR APPEND AS #1
```

This statement opens the existing Name Data sequential file and associates it with the number 1. The program stores its output at the end of the existing contents of Name Data.

Here's a sample OPEN statement using INPUT mode:

```
OPEN "Name Data" FOR INPUT AS #1
```

This statement opens the existing Name Data sequential file and associates it with the number 1.
The program can use the contents of Name Data as input.

Using a string variable with OPEN
When using the OPEN statement you can even use a string variable in place of filename. You might prompt your user to supply a file name and then assign the supplied name to a variable and use the variable within the OPEN statement, as shown in the following example:

```
INPUT "What file would you like to open? ", filename$
OPEN filename$ FOR OUTPUT AS #1
```

You'll probably want to use some of the string-evaluation skills you learned in Chapter 9 to have the program verify that the user's response (filename$) is a valid Apple][gs file name before the program tries to open the file.

Closing a File
When you've finished working with a file, be sure to close it with the CLOSE statement.
This ensures that all information you have written to the file is actually written on the disk.
After a file is closed, the number associated with it is released and can be assigned to another file.

Here's the syntax for the CLOSE statement:

```
   CLOSE [#filenumber]
```

filenumber is the number associated with the file you want to close. If you omit the #filenumber argument, all open files are closed. After you close a file, you must reopen it before you can use it again.

Let's look at some CLOSE statement examples. The following CLOSE statement closes file 1:

```
CLOSE #1
```

The following CLOSE statement closes all open files:

```
CLOSE
```

NOTE: Although AC/QuickBasic closes all open files when a program terminates, you should develop the habit of closing all files as soon as you are finished with them. If you don't, an unforeseen event such as a power outage can cause your program to lose data.

Storing Data in a File
After a file is opened in OUTPUT or APPEND mode, it can receive data from your program.
Three AC/QuickBasic statements can send data to an open sequential file:

- The PRINT# statement sends unformatted data to a file.
- The PRINT# USING statement sends formatted data to a file.
- The WRITE# statement sends data organized into fields to a file.

The following sections describe each of these file-storage statements in detail.

The PRINT# statement

The PRINT# statement is functionally similar to the PRINT statement, but it sends data to a file rather than to the screen. Here's the syntax for the PRINT# statement:

```
PRINT #filenumber,[ expressionlist][,|;]
```

filenumber is the number of the open file, and expressionlist is the data to be sent to the file. If you omit expressionist, a blank line is sent to the file. If you put more than one item in expressionlist, you must separate the items with semicolons or commas. Semicolons and commas in the PRINT# statement work exactly as they do in the regular PRINT statement. If a semicolon is the last character in the PRINT# statement, the next item sent to the file appears at the end of the line you just sent.

Practice:
Storing unformatted data in a file
The Print to File program (Figure 10-1) demonstrates how to use the PRINT# statement to send three lines of unformatted data to a file named Car Data. Values sent to a file can be literal values, simple variables, or the results of functions or expressions.

Load and compile the Print to File program from the Chapter 10 folder on disk and run it.

```
' Print to File
' This program uses the PRINT# statement to send three lines of
'   car information to a sequential file.

OPEN "Car Data" FOR OUTPUT AS #1  ' open file in QBI folder

CLS
```



```
' Get some car information from the user and write it to the open file.

INPUT "Enter the make of a car in your collection:  ", makeName$
INPUT "What is the model name?  ", modelName$
INPUT "In what year was the car made?  ", year%

PRINT #1, makeName$, modelName$, year%

' Add some literal values to the file.

PRINT #1, "Buick", "Skylark", 1962

' Add a new car to the file.  (RIGHT$ function returns current year
'   from DATE$ function.)

PRINT #1, "Audi", "80 Quattro", " "; RIGHT$(DATE$, 4)

CLOSE #1                            ' close the file

PRINT
PRINT "Information has been successfully written to 'Car Data.'"
PRINT
INPUT "Press Return to continue…", dummy$
```

You'll see output similar to this:

```
      Enter the make of a car in your collection: Ford
      What is the model name? Mustang
      In what year was the car made? 1965

      Information has been successfully written to 'Car Data.'
```

A file named Car Data is also created in the folder Learn BASIC Now (or on the desktop of your QBI Work Disk if you use a floppy disk system). To view the file, exit the AC/QuickBasic Interpreter/Compiler, run a word processor or text editor on your system, and open the Car Data file. If you don't have a word processor handy, you can also open Car Data in the AC/QuickBasic Interpreter/Compiler.

You'll see something similar to this when you open Car Data:

```
   Ford        Mustang     1965
   Buick       Skylark     1962
   Audi        80 Quattro  1991
```

The PRINT# USING statement
The PRINT# USING statement follows the same rules and uses the same special formatting characters as the PRINT USING statement, but it sends formatted data to a file rather than to the screen. PRINT# USING is helpful when you need to send large amounts of tabular data to a file.

Here's the syntax for the PRINT# USING statement:

```
PRINT #filenumber, USING template; [expression list][,|;]
```

filenumber is the number of the open file, template is a string used to format the values in expressionlist, and expressionlist is the data to be formatted and sent to the file. (You can use comma or semicolon separators to separate the values in expressionlist.) The formatting characters in template must match up one-for-one with the characters of the values in expressionlist

NOTE: As we discovered in Chapter 8, PRINT USING is most effective at aligning text when the output font has been set to a monospace font (such as Monaco) before printing. Note, however, that PRINT# USING does not store font information inside a file.

Practice:
Storing formatted data in a file
The Print# Using program (Figure 10-2) demonstrates how to use the PRINT# USING statement to send three lines of formatted data to the Fruit Data file. The formatting template in this program is the string variable tmp$, which uses several special formatting characters to align the data vertically before it is sent to the file.

Load and compile the Print# Using program from disk and run it.

```
' Print# Using
' This program uses the PRINT# USING statement to send three lines
'   of formatted fruit information to a sequential file.

OPEN "Fruit Data" FOR OUTPUT AS #1  ' open file in QBI folder

CLS

' Create a formatting template for the PRINT# USING statement.

tmp$ = "Fruit:  \             \ Cases: ###    Price/pound:$$##.##"
```

```
' Get some fruit information from the user and write it to the open file.

INPUT "Enter your favorite summer fruit:  ", fruit$
INPUT "How many cases would you like to buy?  ", cases%
INPUT "How much does the fruit cost per pound?  $", cost!

PRINT #1, USING tmp$; fruit$; cases%; cost!

' Add some literal values to the file.

PRINT #1, USING tmp$; "Strawberry"; 2; 1.29

PRINT #1, USING tmp$; "Cantaloupe"; 14; .69

CLOSE #1                            ' close the file

PRINT
PRINT "Information has been successfully written to 'Fruit Data.'"
PRINT
INPUT "Press Return to continue…", dummy$
```

Figure 10-2.
Print# Using: A program that demonstrates use of the PRINT# USING statement

You'll see output similar to this:

```
      Enter your favorite summer fruit: Nectarine
      How many cases would you like to buy? 10
      How much does the fruit cost per pound? $ 1.49

     Information has been successfully written to 'Fruit Data.'
```

A file named Fruit Data is also created in the folder that contains the AC/QuickBasic Interpreter/Compiler.
To view the file, exit the AC/QuickBasic Interpreter/Compiler, run a word processing application,
and open the Fruit Data file.

     You'll see something similar to this when you open Fruit Data:

```
     Fruit: Nectarine      Cases: 10     Price/pound: $1.49
     Fruit: Strawberry     Cases: 2      Price/pound: $1.29
     Fruit: Cantaloupe     Cases: 14     Price/pound: $0.69
```

The WRITE# statement.
The WRITE# statement does not format its output as the PRINT# and PRINT# USING statements do - WRITE# creates a file
that will be read by other programs. The information that the WRITE# statement sends to a sequential file is separated
by commas, and it is organized into groups called fields.

Here's the syntax for the WRITE# statement:

```
WRITE #filenumber,[ expressionlist]
```

filenumber is the number of the open file, and expressionlist is the data to be sent to the file. Elements in
expressionlist are separated by commas. If you omit expressionlist, a blank line is written to the file.

Figure 10-3 shows a sample WRITE# statement that sends five types of values to a file. Notice that the commas between
values separate the output into fields. Also notice the string value surrounded by quotation marks: If the string
contained spaces or commas, a non-AC/QuickBasic program would recognize the spaces and commas as part of the string.
If the file created with the WRITE# statement contains multiple lines, each line is called a record.

Sample program statements:

```
OPEN "Test Data" FOR OUTPUT AS #1 <------ OPEN statement

a$ = "Kimberly"
b% = 25
c& = 100000                       <------ Sample variables
d! = 115.5
e# = .012345678911

WRITE #1, a$, b%, c&, d!, e#      <------ WRITE# statement
CLOSE #1                          <------ CLOSE statement

Output sent to file:
      "Kimberly",25,100000&,115.5,.0123456789#
         ^       ^     ^      ^         ^
         |       |     |      |         |
      Field1  Field2 Field3 Field4   Field5
```

FIGURE 10-3.

The WRITE# statement stores data in a file infields and records.
Practice
Storing data in a file with fields
The Write to File program (Figure 10-4) demonstrates how to use the WRITE# statement to store coin-collection
information in a file named Coin Data. Write to File is designed to let the user enter information as many coins as he
or she wants to. To stop entering coin-collection information, the user simply types END when prompted for a country.
Load and compile the Write to File program from disk and run it.

```
' Write to File
' This program uses the WRITE# statement to send coin-collection
'   information to a sequential file in fields.

OPEN "Coin Data" FOR OUTPUT AS #1  ' open file in QBI folder

CLS

PRINT "This program stores coin-collection information on disk in a"
PRINT "file named 'Coin Data.'  Enter coin data and type END to quit."
PRINT

' Until the user types END, get coin-collection info from user and
'   write it to the open file.

WHILE (country$ <> "END")

    INPUT "What country is the coin from?  ", country$
    IF (country$ <> "END") THEN    ' if country$ is END don't write
        INPUT "What is the value of the coin?  ", value$
        INPUT "What is the name of the coin?  ", coinName$
        INPUT "In what year was the coin minted?  ", year%

        WRITE #1, country$, value$, coinName$, year%  ' send fields
    END IF
    PRINT                          ' print blank lines between coins

WEND

CLOSE #1                           ' close the file

PRINT "Information has been successfully written to 'Coin Data.'"
PRINT
INPUT "Press Return to continue…", dummy$
```

FIGURE 10-4.

Write to File: a program that demonstrates use of the WRITE# statement.

You'll see output similar to this:

```
      This program stores coin-collection information on disk in a file named 'Coin Data.'
      Enter coin data and type END to quit.

      What country is the coin from? United States
      What is the value of the coin? 10 cents
      What is the name of the coin? Dime
      In what year was the coin minted? 1980

      What country is the coin from? Canada
      What is the value of the coin? 25 cents
      What is the name of the coin? Quarter
      In what year was the coin minted? 1960

      What country is the coin from? Hungary
      What is the value of the coin? 2 forints
      What is the name of the coin?
      In what year was the coin minted? 1985

      What country is the coin from? Great Britain
      What is the value of the coin? 1 pound
      What is the name of the coin? Pound
      In what year was the coin minted? 1981

      What country is the coin from? END

      Information has been successfully written to 'Coin Data.'
```

A file named Coin Data is also created in the folder that contains the AC/QuickBasic Interpreter/Compiler.

To view the file,
exit the AC/QuickBasic Interpreter/Compiler, run a word processing application, and open the Coin Data file.

You'll see something similar to this when you open Coin Data:

```
"United States","10 cents","Dime",1980
"Canada", "25 cents", "Quarter", 1960
"Hungary", "2 forints", "", 1985
"Great Britain","1 pound","Pound",1981
```

The Coin Data file contains four records of four fields each, as shown in Figure 10-5.

```
Record 1 -- "United States", "10 cents", "Dime", 1980
Record 2 -- "Canada", "25 cents", "Quarter", 1960
Record 3 -- "Hungary", "2 forints", "", 1985
Record 4 -- "Great Britain"," 1 pound"," Pound", 1981
```

FIGURE 10-5.
Each line in the Coin Datafile is a record containing four fields.

Getting Data from a File
Now that you've created three files and examined them with a word processor, you'll learn how to examine and use data files from within a AC/QuickBasic program. You'll look at two statements and one function that help you work with files that have been opened for input:

- The INPUT# statement gets one or more fields of data from a file.
- The EOF function determines whether the end of the file has been reached.
- The LINE INPUT# statement gets an entire line of data from a file.

The INPUT# statement
The INPUT# statement is the primary tool you use to get data from a AC/QuickBasic sequential file.
The INPUT# statement gets input from a sequential file in much the same way that the INPUT statement gets input from the keyboard:
Both statements assign one or more data items to variables of matching types. Here's the syntax for the INPUT# statement:

```
INPUT #filenumber, variablelist
```

filenumber is the number of the open file, and variablelist is one or more variables to be assigned data from the file. Items in variablelist must have the same type as items in the sequential file. Items in the sequential file can be fields of data created with the WRITE# statement or output from the PRINT# statement, from the PRINT# USING statement, or from any program that can create data files.

Practice:
Getting data items from a sequential file.
The Read File program (Figure 10-6 on the next page) demonstrates how to use the INPUT# statement in your AC/QuickBasic program to get data from a sequential file. The program first opens a sequential file in the folder that contains the AC/QuickBasic Interpreter/Compiler for output and sends four strings to it with the WRITE# statement.
The program then closes the file, opens it again for input, and reads the four strings back into the program.

Load and compile the Read File program from disk and run it.

```
' Read File
' This program demonstrates use of the INPUT# statement to get
'   data from a sequential file.

CLS

OPEN "Friend Data" FOR OUTPUT AS #1  ' open file for output

PRINT "Enter the names of four of your friends."
PRINT

FOR i% = 1 TO 4                      ' loop 4 times
    INPUT "Friendly name:  ", pal$   ' each time, get name from user
    WRITE #1, pal$                         '   and write it to disk
NEXT i%

CLOSE #1                               ' close the file

OPEN "Friend data" FOR INPUT AS #1   ' reopen the file for input

PRINT
PRINT "You entered the following names:"
PRINT

FOR i% = 1 TO 4                    ' loop 4 times
    INPUT #1, pal$                 ' each time, get name from file
    PRINT pal$                       '   and display it on screen
NEXT i%

CLOSE #1                             ' close the file
```



```
PRINT
INPUT "Press Return to continue…", dummy$
```

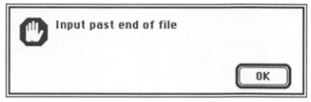
Figure 10-6.
Read File: a program that demonstrates use of the INPUT# statement.

```
      You'll see output similar to this:

      Enter the names of four of your friends.

      Friendly name: Larry
      Friendly name: Whitey
      Friendly name: Gus
      Friendly name: Gilbert

      You entered the following names:

      Larry
      Whitey
      Gus
      Gilbert
```

The Friend Data file is also created on disk and contains the following data records:

```
"Larry"
"Whitey"
"Gus"
"Gilbert"
```

The EOF function
If your AC/QuickBasic program can't tell where your sequential file ends, you might receive an end-of-file error message.
This error occurs when AC/QuickBasic tries to read beyond the end of the file.



To prevent this error, you must explicitly tell AC/QuickBasic to quit reading values after it reaches the end of the file. To do this, use the EOF function.

Here's the syntax for the EOF function:

```
EOF( filenumber)
```

filenumber is the number of the open file you want to check.

EOF returns the logical value true if the next character to be read is past the end of the file, and false otherwise. You can use the EOF function anywhere in your program to check the status of an open file, but it is most effective in a WHILE loop, as shown in the following practice session.

Practice:
Detecting the end of a file
The Write with EOF program (Figure 10-7) is an enhancement of the Write to File coin-collection program we worked with earlier in this chapter: It uses the INPUT# statement to display file data and the EOF function to check for the end of the file. This new version opens the coin-collection file in APPEND mode so that coin-collection information gathered in previous runs of the program isn't overwritten by the new information; instead, all new information is placed at the end of the file. If you have deleted Coin Data, run Write to File again to re-create it.

Load and compile the Write with EOF program from disk and run it.

```
' Write with EOF
' This program uses the WRITE# statement to send coin-collection
'   information to a sequential file and INPUT# to display it.

' Open file in APPEND mode so that previous contents won't be overwritten.

OPEN "Coin Data" FOR APPEND AS #1  ' open file in AC/QuickBasic folder

CLS

PRINT "This program stores coin-collection information on disk in a"
PRINT "file named 'Coin Data.'  Enter coin data and type END to quit."
PRINT
```

```
' Until the user types END, get coin-collection info from user and
'   write it to the open file.

WHILE (country$ <> "END")

    INPUT "What country is the coin from?  ", country$
    IF (country$ <> "END") THEN    ' if country$ is END don't write
        INPUT "What is the value of the coin?  ", value$
        INPUT "What is the name of the coin?  ", coinName$
        INPUT "In what year was the coin minted?  ", year%

        WRITE #1, country$, value$, coinName$, year%  ' send fields
    END IF
    PRINT                         ' print blank lines between coins

WEND

CLOSE #1                        ' close the file

' Wait for the user to press Return to continue.

INPUT "Press Return to see the contents of your collection…", dummy$
CLS                             ' start with a fresh screen
TEXTFONT 4                ' set text font to monospace Monaco for alignment

' Open file in INPUT mode so that contents can be read by program.

OPEN "Coin Data" FOR INPUT AS #1   ' file is in QBI folder

' Display header for tabular coin-collection information.

PRINT "Coin origin      Coin value    Coin name     Year minted"
PRINT "-------------------------------------------------------"

' Initialize formatting template for use with PRINT USING.

tmp$ = "\              \  \          \  \          \  ####"

' While the end of the file has not been reached, assign file
'   items to variables and print them out.

WHILE (NOT EOF(1))
    INPUT #1, country$, value$, coinName$, year%
    PRINT USING tmp$; country$; value$; coinName$; year%
WEND

CLOSE #1                          ' close the file
TEXTFONT 1                       ' reset font to application default (Geneva)

PRINT
INPUT "Press Return to continue…", dummy$
```

```
      You'll see the following prompt for input:

      This program stores coin-collection information on disk in a file named 'Coin Data.'
      Enter coin data and type END to quit.

      What country is the coin from? Netherlands
      What is the value of the coin? 2.5 guilders
      What is the name of the coin? Rijksdaalder
      In what year was the coin minted? 1982

      What country is the coin from? END
      After you've entered the new information, it is displayed along with the information already in the file:

      Coin origin       Coin value    Coin name    Year minted
      United States     10 cents      Dime         1980
      Canada            25 cents      Quarter      1960
      Hungary           2 forints                  1985
      Great Britain     1 pound       Pound        1981
      Netherlands       2.5 guilders  Rijksdaalder 1982
```

The LINE INPUT# statement
The INPUT# statement is an effective way to obtain individual items from files, but what do you do if you need to obtain long strings of textual information? AC/QuickBasic provides the LINE INPUT# statement for this purpose. LINE INPUT# is similar to LINE INPUT, but it obtains input from a file instead of from a keyboard.

Here's the syntax for LINE INPUT#:

```
LINE INPUT #filenumber, stringvariable
```

filenumber is the number of the open file to be read from.
stringvariable is the string variable that is to receive the line of input.

Practice:
Reading a diary with LINE INPUT#
The Diary program (Figure 10-8 on the next page) demonstrates how to use the LINE INPUT and LINE INPUT# statements together to track lines of textual material in a sequential file. Diary first asks the user for the name of the new file and whether the user wants the file to be a diary file. If the file the user names is to be a new diary, a new file is opened in the folder containing the AC/QuickBasic Interpreter/Compiler. If the file the user names is an existing diary, the file is located and opened in APPEND mode. The file is then marked with the current time and date (from the system clock) and prepared for input from the user. The user can enter one or more lines.

The LPRINT Statement
If you have an Apple ImageWriter attached to your Apple][gs, you can send information to it with the LPRINT statement.

LPRINT has a syntax similar to that of the PRINT statement:
LPRINT [expressionlist][,| ;]
expressionlist is a list of numeric or string expressions separated by commas or semicolons.
To send the string Think Snow! to the printer, for example, you would use the following LPRINT statement:

```
LPRINT "Think Snow!"
```

To send a formfeed (page eject) command to your printer when you've finished printing, use the following statement:

```
LPRINT CHR$(12)
```

This sends ASCII character 12 to the printer to advance the paper.

If the user requests a printout, all entries (including their time and date stamps) are sent to the printer by means of LPRINT statements.

NOTE: This program supports the Apple ImageWriter printer only. We describe a technique for sending information to the Apple LaserWriter printer in the next section.

```
' Diary
' This program maintains a computer diary in a sequential file.  The
'   diary can be printed on an Apple ImageWriter printer.

CLS

PRINT "The Secret Diary Program"
PRINT
INPUT "Enter the name of your diary file:  ", diary$
PRINT
INPUT "Is this a new diary file (Y/N)?  ", reply$

IF (UCASE$(reply$) = "Y") THEN
    OPEN diary$ FOR OUTPUT AS #1   ' open new file
 ELSE
    OPEN diary$ FOR APPEND AS #1   ' open existing file
END IF

PRINT
PRINT "Enter your secret thoughts for today; type END to quit."
PRINT

PRINT #1, TIME$; "  "; DATE$        ' write time and date to file
PRINT #1,                                    ' write blank line to file

' Until user types "END", get lines of text and write them to the file.

WHILE (UCASE$(text$) <> "END")
    LINE INPUT text$
    IF (text$ <> "END") THEN PRINT #1, text$
WEND

PRINT #1,                                    ' write blank line to file
CLOSE #1                                     ' close file
```

```
' Find out if user wants an Apple ImageWriter printout.

PRINT
INPUT "Would you like to print out your entire diary (Y/N)?  ", reply$

IF (UCASE$(reply$) = "Y") THEN     ' if yes,
    OPEN diary$ FOR INPUT AS #1  '   open file for input

    LPRINT STRING$(33, "-");         ' print a header at top of page
    LPRINT " My Diary ";
    LPRINT STRING$(33, "-")
    LPRINT                                ' and a blank line

    ' Until end of file is reached, read lines from file and send them
    '   to the printer.

    WHILE (NOT EOF(1))
        LINE INPUT #1, text$
        LPRINT text$
    WEND

    CLOSE #1                              ' close file
    LPRINT CHR$(12)                    ' send formfeed character

    ' Display message indicating diary contents have been sent to printer.

    PRINT
    INPUT "Diary sent to ImageWriter; press Return to continue…", dummy$
END IF
```

FIGURE 10-8.
Diary: a simple diary program that demonstrates use of the LINE INPUT# statement.

When you compile and run the program, you see this prompt for a diary entry:

```
      The Secret Diary Program

      Enter the name of your diary file: First Diary

      Is this a new diary file (Y/N)? y

      Enter your secret thoughts for today; type END to quit.

      After you type in your diary entry and the word END to quit, you see this prompt:

      Would you like to print out your entire diary (Y/N)?

      If you enter Y, the program sends a copy of the diary to the ImageWriter printer.
```

Figure 10-9 shows the contents of a sample printout.

```
-------------------------------- My Diary ---------------------------------
11:43:12 06-07-1991

Today I met a tall, dark stranger in the jungle next to a big papaya tree.
Turned out to be a gorilla in a man suit--just my luck!

09:13:01 06-08-1991

The gorilla and I have turned out to be real chums. He doesn't do
much--just sits around all day eating plants. I think I'll make
him a coconut cream pie this afternoon.

15:25:10 06-09-1991

Pooluseebagumba!--I've been discovered by unfriendly natives from the
other side of the island! This is my last entry. I surely won't be able
to lug this computer around with me as I make my escape to the secret cave.

Signed,

ON THE RUN
```

FIGURE 10-9.
A printout produced by the Diary program.

Using the Apple LaserWriter
Printing on the Apple LaserWriter requires a leap of imagination: You need to think of the LaserWriter itself as a file. Before you can print on the LaserWriter, you must "open" by means of the OPEN statement. And rather than using LPRINT to send information to the LaserWriter, you use the PRINT# statement. Finally, when printing is complete, you close the printer "file" by means of the CLOSE statement. Here's the syntax for this sequence of statements:


```
OPEN "LPT1:" FOR OUTPUT AS #filenumber
PRINT #filenumber, expressionlist
CLOSE #filenumber
```

filenumber is the file number to be associated with the open LaserWriter, and expressionlist is the data to be sent to the LaserWriter. You can include any number of PRINT# statements between the OPEN and CLOSE statements. Note, however, that the printer will "time out" if you wait for longer than two minutes between PRINT# statements. You can work around this limitation by keeping your OPEN, PRINT#, and CLOSE statements together in one block and not prompting the user for input between the PRINT# statements. (A pokey user might take longer than two minutes to reply.)

The LaserWriter "file name" must always be LPT1: because AC/QuickBasic associates the LPT1: keyword with the LaserWriter when the LaserWriter is selected in the Chooser desk accessory.

The Laser Diary program (Figure 10-10) demonstrates how you can modify the Diary program for printing on a LaserWriter.

```
' Laser Diary
' This program maintains a computer diary in a sequential file.  The
'   diary can be printed on an Apple LaserWriter printer.

CLS

PRINT "The Secret Diary Program"
PRINT
INPUT "Enter the name of your diary file:  ", diary$
PRINT
INPUT "Is this a new diary file (Y/N)?  ", reply$

IF (UCASE$(reply$) = "Y") THEN
    OPEN diary$ FOR OUTPUT AS #1   ' open new file
 ELSE
    OPEN diary$ FOR APPEND AS #1   ' open existing file
END IF

PRINT
PRINT "Enter your secret thoughts for today; type END to quit."
PRINT

PRINT #1, TIME$; "  "; DATE$        ' write time and date to file
PRINT #1,                                  ' write blank line to file

' Until user types "END", get lines of text and write them to the file.

WHILE (UCASE$(text$) <> "END")
    LINE INPUT text$
    IF (text$ <> "END") THEN PRINT #1, text$
WEND

PRINT #1,                                  ' write blank line to file
CLOSE #1                                   ' close file

' Find out if user wants an Apple LaserWriter printout.

PRINT
INPUT "Would you like to print out your entire diary (Y/N)?  ", reply$

IF (UCASE$(reply$) = "Y") THEN     ' if yes,
    OPEN diary$ FOR INPUT AS #1  '   open file for input
    OPEN "LPT1:" FOR OUTPUT AS #2    ' open LaserWriter for output

    PRINT #2, "********";              ' print a header at top of page
    PRINT #2, " My Diary ";
    PRINT #2, "********"
    PRINT #2,                                  ' and a blank line

    ' Until end of file is reached, read lines from file and send them
    '   to the printer.

    WHILE (NOT EOF(1))
        LINE INPUT #1, text$
        PRINT #2, text$
    WEND

    CLOSE #1                                ' close file
    PRINT #2, CHR$(12)                ' send formfeed character
    CLOSE #2                                ' close LaserWriter

    ' Display message indicating diary contents have been sent to printer.

    PRINT
    INPUT "Diary sent to LaserWriter; press Return to continue…", dummy$
END IF
```

File-related Operations
AC/QuickBasic provides one function and two statements that unlock the power of the Apple][gs operating system in your programs:

AC/QuickBasic Instruction	Description
FILES$ function	Lists the files in a folder. Lets you search the file system for a file.
NAME statement	Changes the name of a file.
KILL statement	Deletes a file.

The FILES$ function
If you want your program to display a list of files a user can choose, use the FILES$ function. Here's the syntax for using FILES$ to locate a file:

```
stringvariable = FILES$(1, "filetype")
```

stringvariable is the variable that is to receive the file name chosen by the user, and filetype is the type of file to be listed in the list box. Two valid arguments for filetype are TEXT (to display only text files) and APPL to display only application files. If you don't enter a filetype argument in the statement, all files in the folder are listed.

When AC/QuickBasic executes the FILES$ function, it displays a list box containing all the files in the designated mode in the current folder. The user can then select a file in the current folder or a file in another folder by means of the navigation bar at the top of the list box. If the user selects a valid file name, the file name is assigned to stringvariable and can be used later in the program. If the user cancels the list box, a null (empty) string is assigned to stringvariable .

NOTE: You can also use the FILES$ function to get a new file name from the user. To use FILES$ in this way, change the first argument in FILES$ to 0 and substitute an input prompt for the filetype argument. FILES$ then displays a dialog box for user input and returns a string value to stringvariable.

This "new file" feature of FILES$ is useful when you're creating new files in your programs.

Practice:
Viewing a file
The View File program (Figure 10-11) demonstrates how to use the FILES$ function to display a list box and how to process the file name the user selects or types in. View File displays the contents of any text file on your Apple][gs.

Load and compile the View File program from disk and run it.

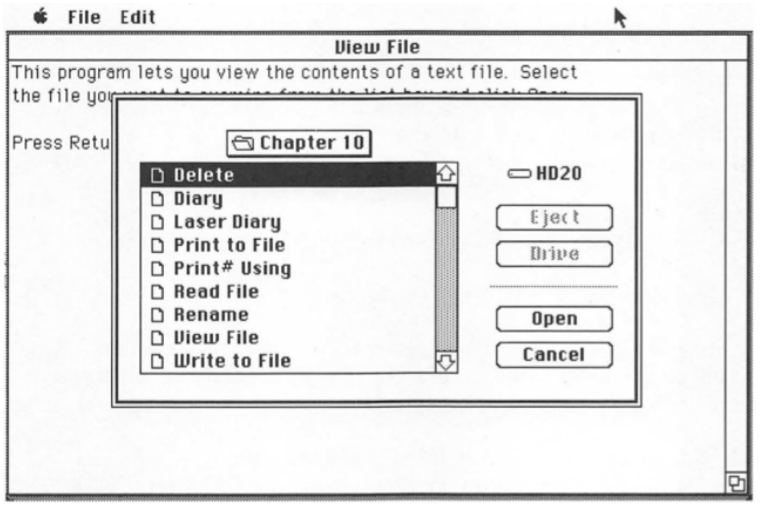
```
' View File
' This program lets you view a data file anywhere on disk.

CLS

PRINT "This program lets you view the contents of a text file.  Select"
PRINT "the file you want to examine from the list box and click Open."
PRINT
INPUT "Press Return to see the list box…", dummy$

filename$ = FILES$(1, "TEXT")   ' display text files only

IF filename$ <> "" THEN
    OPEN filename$ FOR INPUT AS #1

    PRINT
    PRINT "---------- "; filename$; " ----------"
    PRINT

    WHILE (NOT EOF(1))
        LINE INPUT #1, text$
        PRINT text$
    WEND

    PRINT
    INPUT "Press Return to continue…", dummy$
END IF
```

When you run the program, you '11 be prompted to select a file name from a list box:

The file you select is displayed on the screen in its entirety.
The NAME statement
The NAME statement is useful when you want to rename a file. Here's the syntax for the NAME statement:

```
NAME oldFileName AS newFileName
```

oldFileName is the current file name, and newFileName is the new name you'd like the file to have. Folder names are allowed in both oldFileName and newFileName as long as both file names are in the same folder. If the NAME statement cannot rename the files, it generates an error message.



Practice:
Renaming a file

The Rename program (Figure 10-12) demonstrates how to use the NAME statement to change the name of a file in a folder. Note the use of the FILES$ function, which lets you find a file anywhere on disk.

Load and compile the Rename program from disk and run it.

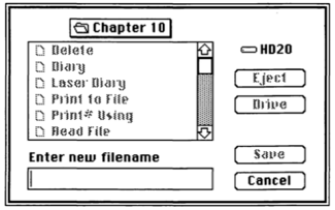
```
' Rename
' This program lets you rename a data file on disk.

CLS

PRINT "This program renames a file on disk."
INPUT "Press Return and find the file you want to rename…", dummy$

oldName$ = FILES$(1, "TEXT")                        ' get old filename
newName$ = FILES$(0, "Enter new filename")    ' get new filename

NAME oldName$ AS newName$  ' try to rename file

' If oldName$ does not exist or newName$ is an invalid name, the
'   NAME statement will generate a run-time error message; otherwise,
'   the following lines will be executed:

PRINT                       ' print success message
INPUT "File renamed successfully; press Return to continue…", dummy$
```

FIGURE 10-12.
Rename: a program that demonstrates use of the NAME statement.

When you run the program, you see output similar to this:

```
      This program renames a file on disk.
      Press Return and find the file you want to rename ...
```

When you press Return, a list box for the current folder appears showing the files you can rename.
You can use the navigation box to find a file in another folder if you like.
When you select a file, another dialog box appears prompting you to enter the new file name. Be sure not to change the folder designation for the new file name; if you do, you '11 receive a run-time error message. (Renamed files must stay in the same folder.)



After you enter a new file name, the following message appears:

```
      File renamed successfully; press Return to continue ...
```

The KILL statement
The KILL statement is both straightforward and permanent: It deletes a file from disk. Once a file is deleted by means of the KILL statement, it cannot be recovered. Programmers commonly use KILL to delete temporary files created during program execution, but you can also use KILL for general disk cleanup.

Here's the syntax for the KILL statement:

```
KILL filename
```

filename is the name of the file to be deleted. You can also use folder and volume names within a KILL statement.

WARNING: Use extreme caution with the KILL statement - you might accidentally delete more files than you intend to. The following practice session shows some of the safety features you should build into a program that uses the KILL statement.

Practice:
Deleting a file
The Delete program (Figure 10-13) demonstrates how to use the KILL statement to delete an unwanted file. The "FILES$ function displays a list box on the screen and asks the user to select the name of the file to be deleted.
An INPUT and IF statement combination then verifies that the user really wants to delete the file.
If so, the file is deleted from disk with the KILL statement.

Load and compile the Delete program from disk and run it.

```
' Delete
' This program lets you delete a file on disk.

CLS

PRINT "This program deletes a text file from disk."
INPUT "Press Return to find the file you want to delete…", dummy$

filename$ = FILES$(1, "TEXT")

CLS

PRINT "File:  "; filename$
INPUT "Are you sure you want to delete this file (Y/N)?  ", reply$

PRINT
IF UCASE$(reply$) = "Y" THEN
    KILL filename$
    PRINT filename$; " has been deleted."
ELSE
    PRINT filename$; " has not been deleted."
END IF

PRINT
INPUT "Press Return to continue…", dummy$
```

Figure 10-13.
Delete: a program that demonstrates use of the KILL statement.

When you run the program, you'll see a list box that prompts you for a file name and a question that verifies whether you really want to delete the file you've selected.

SUMMARY
In this chapter you've started to put your skills together in a really useful way. You've continued to work with large amounts of data by learning how to read and write sequential files. You've also learned how printers work with AC/QuickBasic and how to use some handy file-related AC/QuickBasic statements and functions.

In the next chapter we'll continue our work with sequential files and the Apple][gs interface and produce a database program you can customize to suit your own needs.

QUESTIONS AND EXERCISES
1. What is the difference between opening a file for OUTPUT and opening a file for APPEND?
2. Which of the following statements encloses data items in quotation marks when it sends them to a sequential file?
   a. INPUT#
   b. PRINT# USING
   c. PRINT#
   d. WRITE#
3. True or False: The file name specified in an OPEN statement must be in uppercase letters.
4. When is the LINE INPUT# statement considered more useful than the INPUT# statement?
5. What is wrong with the following AC/QuickBasic statement: filename$ = FILES(1, "TEXT")
6. Write a program that prompts the user for a list of cities and stores the information in a sequential file. Design the program so that the user can view the contents of the file after it has been created.
7. Write a program that prompts the user for a list of names and addresses, stores them in a sequential file, and then sorts the records in the file alphabetically by name.

Hint: The easiest way to solve this problem is to use an array to store the records and sort them. You might want to use the Shell Sort, which you learned about in Chapter 9.

Working with Menus, Windows, and the Mouse Learn AC/QuickBasic For the Apple][gs NOW
The programs you've written so far have asked the user for traditional, character-based input, and they've produced character-based output. Although you've used the AC/QuickBasic Interpreter/Compiler's menus, windows, and buttons -- its graphical interface -- as you wrote your code, you haven't yet given your own programs these attractive, "user-friendly" features.

In this chapter, you'll learn how to add the most popular graphical user-interface components to your programs. You'll learn about the AC/QuickBasic statements used to create menus, windows, buttons, and edit fields. You'll also learn how to use functions to manage input from the user with a technique called event trapping.

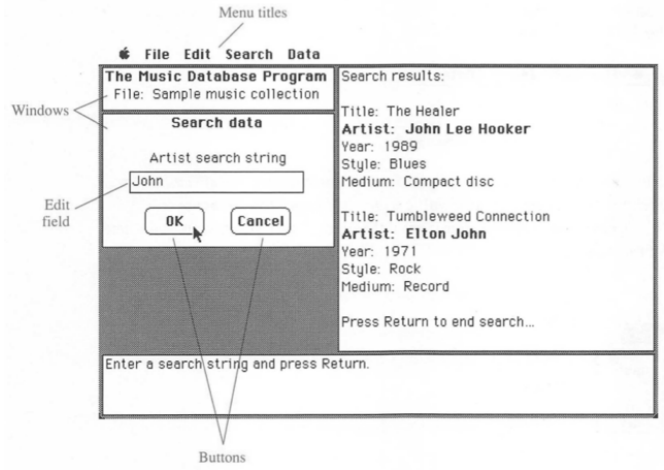
At the end of this chapter, you'll put together what you've learned in this and previous chapters and build a graphical database program that can track an entire home music collection.

THE Apple][gs INTERFACE
If you've been using the Apple][gs for a while, you know that every Apple][gs program follows the same conventions for presenting information and options and getting input from the user. Apple designed the Apple][gs and published the Apple][gs graphical user-interface standard so that once a person learned how to use one Apple][gs program, that person would know how to move around in all Apple][gs programs.

Think of a Apple][gs program you're comfortable using. It probably displays information in one or more windows and lets you use scroll bars to get to information elsewhere in a list or document. You probably move from one window to another by clicking with the mouse. You probably execute commands by pulling down menus and by filling in special windows called dialog boxes that gather input with standard-size buttons and edit fields.

We'll use some of the same graphical elements in a program that this chapter will help you write. Figure 11-1 shows a screen of the music data-base program we create at the end of the chapter. Notice that it contains windows, menu headers, an edit field, buttons, and a mouse pointer.

Let's begin by using the MENU statement to create a menu.



Graphical user-interface elements in the Music Database program.

CREATING YOUR OWN MENUS
AC/QuickBasic Interpreter/Compiler lets you replace the standard AC/QuickBasic menus with your own menus.

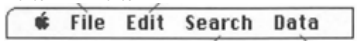
You add each menu title and menu item to the menu bar with the MENU statement.
Here's the syntax for the MENU statement:

MENU menuNumber, itemNumber, status, title

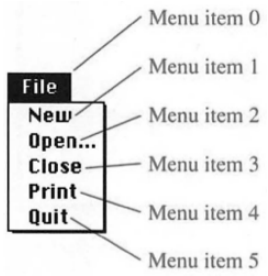
Each menu in AC/QuickBasic is always associated with a menuNumber.
menuNumber must be an integer from 1 through 10.
menuNumber for a menu item will be the same value as menuNumber for the menu title.
itemNumber is an integer from 0 through 20 that you assign to an item in the menu.

If the itemNumber value is 0, the title string will be the menu title on the menu bar. If the itemNumber value is an integer from 1 through 20, the title string will be text for an item underneath the menu title.

status is an integer from 0 through 2 that sets the availability, or state, of the menu item. The table below shows the status values and the menu status associated with each value.



Value	Menu status (Menu Number 3, Menu Number 4)
0	Disabled. The menu item cannot be selected and appears in dimmed type.
1	Enabled. The menu item can be selected and appears in regular type.
2	Selected. The menu item has been selected and appears with a check-mark beside it.



title is a string that is the name of the menu item. You see the title string as a menu title (if itemNumber is 0) or as a menu item (if itemNumber is an integer from 1 through 20) when you run the program.

Practice:
Adding menu items
The Menu Maker program (Figure 11-2 on the next page) demonstrates how to use the MENU statement to add two new menus to the menu bar. The program only creates the menus- it won't do anything special when you select one of the menus. We'll get to that next.

Load and compile the Menu Maker program from the Chapter 11 folder on disk and then run it.
(This is the Menu Maker.bas,bin)

Apple][gs Menu Conventions
By convention, each word in a Apple][gs menu header and menu item has an initial capital letter.
File, Edit, and Search are usually the first three menu headers on the menu bar. Menus to the right of File, Edit, and Search usually contain items that are specific to the application-special commands and options, for example.
The File, Edit, and Search menus usually contain the same items in the same order in all Apple][gs applications.

The File menu, for example, usually contains these items in this order: New, Open, Close, Save, Print, and Quit.
The File menu in some applications (in Apple][gs AC/QuickBasic, for instance)

   includes extras such as Save As and Transfer.

These additional menu items vary from application to application.

As you write "menu-driven" programs, try to use the menu conventions you've seen in popular Apple][gs programs. If your menu style conforms to Apple][gs conventions, users will quickly learn how to use your programs.

```
' Menu Maker
' This program uses the MENU statement to add two menus to the menu
MENU 1, 0, 1, "Time"     ' menu number 1 is the Time menu
MENU 1, 1, 1, "Day"      ' item 1 is Day
MENU 1, 2, 1, "Week"     ' item 2 is Week
MENU 1, 3, 1, "Month"    ' item 3 is Month
MENU 1, 4, 1, "Year"     ' item 4 is Year
MENU 2, 0, 1, "Beverage" ' menu number 2 is the Beverage menu
MENU 2, 1, 1, "Water"    ' item 1 is Water
MENU 2, 2, 1, "Coffee"   ' item 2 is Coffee
MENU 2, 3, 1, "Beer"     ' item 3 is Beer

' Pause so that we can try the new menus.
INPUT "Practice with the menus--press Return to quit ... ", dummy$
```

You'll see this output:

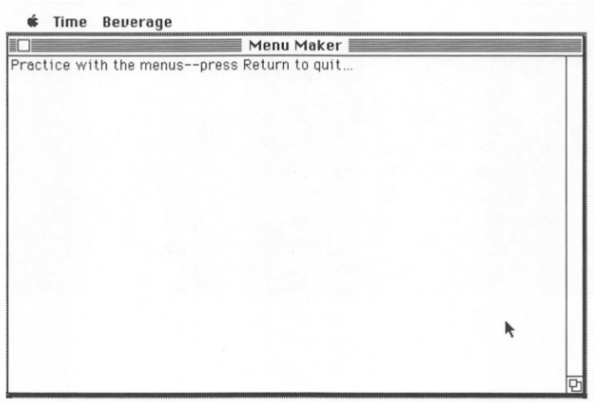


FIGURE 11-2.
Menu Maker: a program that demonstrates use of the MENU statement.

Pull down the Time menu and then the Beverage menu before you press Return to quit.
Notice that these menus work the way any standard Apple][gs menu does even though we haven't written any program code to control the mouse or handle the user's selection of menu items.

Features to control the mouse and handle menu selections are built into AC/QuickBasic and come without any programming
overhead.

Notice that if you select a menu item, the menu title remains highlighted until you pull down and release a menu or press Return to end the program.

We're off to a good start. Now let's see how to "capture" a menu selection.

Waiting for Menu Events
A Apple][gs program that uses menus and other graphical elements requires a technique called; Event Trapping for eliciting input from the user. Rather than demanding input, a program must wait for input in a loop that continues indefinitely until an event occurs that requires the program to stop looping and take action. The loop we use to wait for a menu event is the WHILE loop, and the instruction we use to report a menu event is the MENU function.
(Be careful not to confuse the MENU function with the MENU statement that we just discussed.)
Here's the syntax for the MENU function:

```
MENU(arg)
```

arg is either 0 or 1, depending on whether the event is the selection of a menu or of a menu item within a menu.
The table below shows the two possible arg values and their meanings.

arg	Meaning
0	MENU returns a number corresponding to the last menu selected.
1	MENU returns a number corresponding to the last menu item selected.

MENU returns an integer value depending on the arg integer argument:
The return value 0 means that no event has occurred. A return value other than 0 means that an event has occurred and that the program should trap it and act accordingly.

A simple event-trapping WHILE loop that waits for menu events might look like this:

```
WHILE menuNumber% = 0
    menuNumber% = MENU(0)
WEND
itemNumber% = MENU(1)
```

The WHILE loop cycles continuously until the user selects a menu item.
When the user does, the selection is detected by the MENU function and the menu number is stored in the menuNumber% variable. Because the MENU function returns a value from 1 through 10 when a menu is selected, the test condition in the WHILE loop evaluates as false and the loop ends. Then the next MENU function is executed;

This time with the argument 1 to retrieve the menu item that has been selected.

Always add a MENU statement with no arguments to your program after you have processed the user's menu selection so that the menu bar will return to its normal state.

Adding Status Logic to Menus
It's simple to make your program enable and disable menu items by using an integer variable in the status argument of the MENU statement. This is handy if you want a few of your menu items to be available only some of the time, based on certain program conditions. This is a typical situation in Apple][gs applications.
WordPerfect, for example, enables all 13 items on the File menu when one or more WordPerfect documents are open but enables only 5 items when no WordPerfect document is open. You can't select Print from the File menu, for example, if you haven't opened a document with the New or the Open command first.

The following MENU statements set up a five-item File menu and use a variable named status% to allow for changes in the status of two of the menu items.

```
IF filename$ = "none" THEN status% = O ELSE status% = 1
MENU 1, 0, 1, "File"
MENU 1, 1, 1, "New"
MENU 1, 2, 1, "Open ... "
MENU 1, 3, status%, "Close"
MENU 1, 4, status%, "Print"
MENU 1, 5, 1, "Quit"
```

The filename$ variable is the key to the status switch in this situation.

If filename$ contains the value none (meaning that no file is currently open), status% receives the value 0 and the MENU statements produce a File menu in which some items (Close and Print) are disabled:

```
File
New
Open •••
Close [Grey]
Print [Grey]
Quit
```

If filename$ has a value other than none (meaning that a file is currently open), status% receives the value 1 and the File menu is displayed with all items enabled:

```
New
Open •••
Close
Print
Quit
```

We'll take advantage of this special use of the status argument later, when we create our music database program.

CREATING YOUR OWN WINDOWS
The rectangular Apple][gs window is the key graphical element in many Apple][gs applications. So far, you've been sending program output to the full-size Output window supplied by the AC/QuickBasic Interpreter/Compiler.
But you can create your own windows on the screen and send information to each one.

The WINDOW Statement
In AC/QuickBasic you create a window with the WINDOW statement. You can display as many as 16 windows at the same time. Each window is associated with a number, a dimension, and a type. Here's the syntax for using the WINDOW statement to create a window:

   WINDOW winNum, [title], dimensions, type winNum is an integer from 1 through 16 that identifies the window.

   winNum must be an argument to the WINDOW statement anytime a window is created, activated, or closed.

   title is an optional string that appears in the window's title bar if the window has a title bar. If no title argument is specified, the word Untitled appears in the window title bar.

   dimensions are the x and y (column and row) coordinates of the window rectangle on the screen. The dimensions argument has the format (x1, y1)-(x2, y2).
    where (x1, y1) identifies the integer coordinates of the upper left corner of the window
    and (x2, y2) identifies the integer coordinates of the lower right corner.

The coordinates are absolute; that is,
    (0, 0) specifies the upper left corner of the screen,
and (511, 341) for a Apple][gs screen specifies the lower right corner of the screen.

You'll find that it takes a little time to set up exactly the right window dimensions when you're using more than one window.

type is an integer value that indicates the kind of window the program is creating.
The seven Apple][gs window types are described in the table on the opposite page.

Type	Description
1	Document window with title bar and size box
2	Framed dialog box
3	Simple window with one-line border
4	Simple window with shadow border
5	Document window with title bar and no size box
6	Document window with title bar and rounded corners
7	Document window with title bar, size box, and zoom box

Practice:
Displaying the seven window types

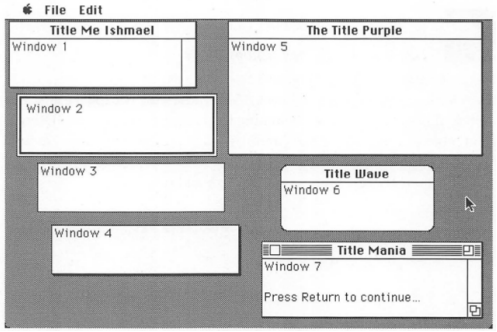
The 7 Windows program (Figure 11-4) demonstrates how to create the seven basic window types in a AC/QuickBasic program. The toughest part of writing this test program is calculating the rectangle coordinates.
Using a good piece of graph paper and carefully marking screen coordinates will help you lay out the windows.

Load and compile the 7 Windows program from disk and run it.
(This is the 7 Windows.bas.bin)

```
' 7 Windows
' This program creates the seven basic window types and displays information in each.
WINDOW 1, "Title Me Ishmael", (5, 40)-(200, 90), 1
PRINT "Window 1"
WINDOW 2, , (20, 105)-(215, 155), 2
PRINT "Window 2"
WINDOW 3, , (35, 170)-(230, 220), 3
PRINT "Window 3"
WINDOW 4, , (50, 235)-(245, 285), 4
PRINT "Window 4"
WINDOW 5, "The Title Purple", (235, 40)-(500, 160), 5
PRINT "Window 5"
WINDOW 6, "Title Wave", (290, 190)-(450, 240), 6
PRINT "Window 6"
WINDOW 7, "Title Mania", (270, 270)-(500, 330), 7
PRINT "Window 7"
PRINT
INPUT "Press Return to continue ... ", dummy$
```

7 Windows: a program that displays the seven basic window types by means of the WINDOW statement.

You'll see this output:



As you can see from the results of the PRINT statements, AC/QuickBasic puts the output of a PRINT statement in the window that was most recently created.

Switching Between Windows
AC/QuickBasic considers the window currently processing input and output to be the active window.

By default, the active window is the window that was last defined by a WINDOW statement,
but you can change the active window by using a simpler form of the WINDOW statement:

WINDOW winNum
winNum is the integer value assigned to the window when it was first created.

To make window 2 the active window in a program, for example, you would use this WINDOW statement:

```
WINDOW 2
```

After this WINDOW statement (and until the next WINDOW statement),
AC/QuickBasic will process all input and output requests in window 2.

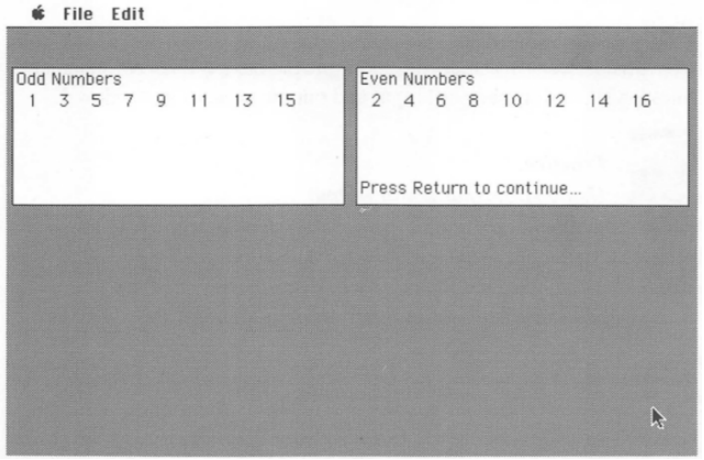
Practice:
Using two windows for output
The Switch Windows program (Figure 11-5) demonstrates how to use the WINDOW statement to switch between two windows.
One window will display odd numbers, and the other will display even numbers.

Load and compile the Switch Windows program from disk and run it.
(This is the Switch Windows.bas.bin)

```
' Switch Windows
' This program displays output in two windows.
WINDOW 1, , (5, 50)-(250, 150), 3
PRINT "Odd Numbers"
WINDOW 2, , (260, 50)-(505, 150), 3
PRINT "Even Numbers"
FOR i% = I to 16
IF (i% MOD 2 <> 0) THEN WINDOW 1 ELSE WINDOW 2
PRINT i%;
NEXT i%
FOR i% = 1 TO 4
PRINT
NEXT i%
INPUT "Press Return to continue ... ", dummy$
```

Switch Windows: a program that switches back and forth between windows by means of the WINDOW statement.

You'll see this output:



Closing Windows
When you've finished using a window, you can make it disappear, or close it. A Apple][gs program often must close a window it has opened temporarily to get user input or display useful information.

Here's the syntax for closing a window with the WINDOW CLOSE statement:

WINDOW CLOSE winNum
winNum is the integer identifier assigned to the window when it was first created.

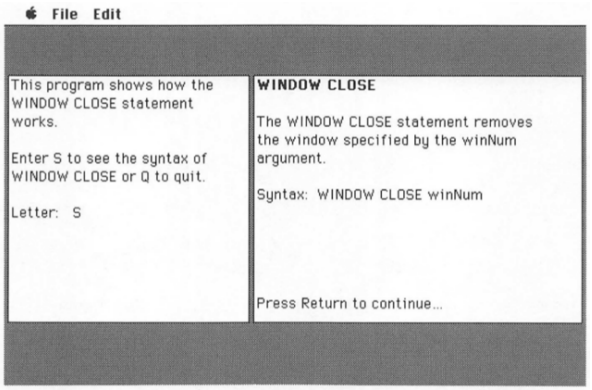
Practice:
Closing a window
The Close Window program (Figure 11-6) demonstrates how the WINDOW CLOSE statement can be used to remove a temporary window from the screen. Close Window displays program information in window 1 and (if the user requests it) displays the syntax of the WINDOW CLOSE statement in window 2.

Load and compile Close Window from disk and run it.

```
' Close Window
' This program displays syntax information in a temporary window.

WINDOW 1, , (4, 64)-(215, 280), 3              ' create first window
PRINT "This program shows how the"
PRINT "WINDOW CLOSE statement"
PRINT "works."
PRINT
PRINT "Enter S to see the syntax of"           ' display instructions
PRINT "WINDOW CLOSE or Q to quit."
PRINT

WHILE UCASE$(reply$) <> "Q"                    ' get input from user
INPUT "Letter: ", reply$                       ' loop until reply is "q" or "Q"

IF UCASE$(reply$) = "S" THEN                   ' if reply is "s" or "S", display syntax
   WINDOW 2, , (220, 64)-(508, 280), 3         ' create syntax window
   TEXTFACE 1
   PRINT "WINDOW CLOSE"
   TEXTFACE 0
   PRINT ' print syntax
   PRINT "The WINDOW CLOSE statement removes"
   PRINT "the window specified by the winNum"
   PRINT "argument."
   PRINT
   PRINT "Syntax: WINDOW CLOSE winNum"
   FOR i% = 1 TO 5
       PRINT
   NEXT i%
   INPUT "Press Return to continue ... ", dummy$
   WINDOW CLOSE 2                              ' close syntax window
  END IF
 WEND
```

Close Window: a program that closes a temporary window by means of the WINDOW CLOSE statement.

When you type S and then press Return, you'll see this output:



ADDING A BUTTON
The syntax window in the Close Window program looked much like a typical Apple][gs dialog box,
But it lacked an important component: a button to close the window. If you've used Apple][gs applications, you know mouse-activated buttons are an integral part of the user interface. Buttons take the place of the old-fashioned character-based messages and commands programs and users have used to "talk to" each other. Buttons like the ones shown in Figure 11-7 add an intuitive, hands-on feel to the exchange of information in a dialog box window.

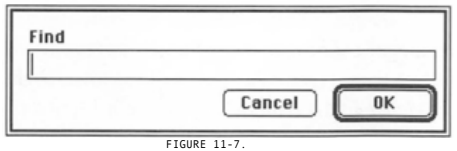


FIGURE 11-7.

A AC/QuickBasic dialog box containing two essential buttons.

The BUTTON Statement
In many ways, Apple][gs buttons and Apple][gs windows are alike:
Their statements have similar syntax lines, with arguments for number, title, dimensions, and type, and buttons and windows can be opened and closed. But buttons appear within windows - most often when a signal is needed to indicate the end of input or output. And the BUTTON statement shares an additional argument, status, with the MENU statement - buttons can be enabled or disabled depending on program conditions.

To create a button, you use the BUTTON statement:
   BUTTON num, status, title, dimensions, type
   num is an integer from 1 through 255 that identifies the button in subsequent BUTTON statements.
   status is an integer argument from 0 through 2 that sets the state of the button. The table below shows the possible status values and the button status associated with each value.

Value	Button status
0	Disabled. The button cannot be selected and appears in dimmed type.
1	Enabled. The button can be selected and appears in regular type.
2	Selected. The button is enabled and has been selected.

   title is an optional string that appears inside or alongside the button.
   dimensions are the x and y (column and row) coordinates of the button on the screen.
The dimensions argument has the format;

 (x1, y1)-(x2, y2)

where (x1, y1) identifies the integer coordinates of the upper left corner of the button and (x2, y2) identifies the integer coordinates of the lower right corner.
The coordinates are relative to the window the button is in; that is, (0, 0) specifies the upper left corner of the window. The dimensions argument takes a little time to master but is flexible enough to help you create the perfect button for the dialog box you're designing.
   type is an integer that indicates the kind of button the program creates.
The three Apple][gs button types are described in the following table:

Type	Description
1	Push button (square box with rounded corners). If type is omitted, push button is the default.
2	Check box (square with "X" inside when selected).
3	Radio button (circle with dot inside when selected).

In the following example statement, an enabled push button containing the title "OK" is added to a window:

```
BUTTON 1, 1, "OK", (116, 180)-(171, 205), 1
```

We'll use this statement in a program soon, but first we'll look at how AC/QuickBasic notifies a program that a button has been selected.

The DIALOG function
The DIALOG function, like the MENU function, notifies a program that an event has occurred in a program.
The MENU function returns information about menu items, and the DIALOG function returns information about items in a window. Depending on the argument supplied before the function call, DIALOG will return information about the status of buttons, edit fields, close boxes, the Return or the Tab key, or mouse clicks in another window.
Like the MENU function, the DIALOG function is used in a WHILE loop that waits for events and acts on them.
Here's the syntax for the DIALOG function:

      DIALOG(num)
   num is an integer argument from 0 through 5 that specifies a particular kind of information about an event.
The following table shows five num arguments and the kind of event information associated with each and describes the significance of the integer value returned by the DIALOG function depending on the argument.

Argument	Meaning
0	What event has taken place
Returns	Description
0	No event has taken place.
1	The user clicked a button in the active window. Use DIALOG(1) to see which one.
2	The user clicked in a new edit field. Use DIALOG(2) to see which one.
3	The user clicked in an inactive window. Use DIALOG(3) to see which one.
4	The user clicked a window close box. Use DIALOG(4) to see which one.
5	A window has been overwritten and needs to be redrawn. Use DIALOG(5) to see which one.
6	The user pressed the Return key.
7	The user pressed the Tab key.

1	What is the most recently pressed button?	
	Returns	Description
	1-255	Number of button.
2	What is the most recently selected edit field?	
	Returns	Description
	1-255	Number of edit field.
3	What is the most recently selected window?	
	Returns	Description
	1-16	Number of window.
4	What is the window with the most recently selected close box?	
	Returns	Description
	1-16	Number of window.
5	What is the window that needs refreshing?	
	Returns	Description
	1-16	Number of window.

The DIALOG function can return quite a bit of information - more than we'll use in this chapter. For now, concentrate on the information in the DIALOG(0) and DIALOG(1) calls. In the next section we'll discuss how the DIALOG(2) call is used with edit fields. We leave the remaining calls, those dealing with selecting different windows and refreshing overwritten ones, for a more advanced text to cover.

A general DIALOG loop
DIALOG can be used in a WHILE loop in many ways. The simplest way is in a WHILE loop that continues until any event occurs in a window:

```
WHILE DIALOG(0) = 0
WEND
```

If the user clicks a button, an edit field, or a close box, presses the Return key, presses the Tab key, or clicks in a different window, the loop ends.

A specific DIALOG loop
A more useful employment of DIALOG is in a loop that waits for a specific event to occur.
The following loop, for example, continues until the user clicks a button in the active dialog box:

```
WHILE DIALOG(0) <> 1
WEND
```

The loop will stop only when the user has selected a button in the active window, which causes the DIALOG function to return a 1. To have your program determine the number of the button pressed, place this line immediately below the WHILE loop:

```
buttonPressed% = DIALOG(1)
```

NOTE: It takes a little patience to use the DIALOG function. The argument and return values can be confusing. Take it slow when you use this function. To help you keep the numbers straight, keep our DIALOG table on page 327 handy.

Checking for multiple events
Checking for more than one dialog-box event requires a slightly more sophisticated loop. In the following routine, the loop continues until the user clicks a button in the active dialog box or presses the Return key:

```
event% = 0
WHILE (event% <> 1) AND (event% <> 6)
    event% = DIALOG(0)
WEND
IF event% = 1 THEN buttonPressed% = DIALOG(1)
```

After the loop, the buttonPressed% variable is updated only if a button was actually pressed.

Practice:
Adding a button to Close Window
The Button Close program (Figure 11-8 on the next page) modifies the Close Window program to include a button at the bottom of the syntax window (window 2). The DIALOG loop near the bottom of the program makes the action pause until the user clicks OK or presses Return. OK and Return are often linked in Apple][gs programs, and we'll use them more or less synonymously throughout this chapter.

Closing a Button
After a button has been created, you can close it with a BUTTON CLOSE statement. Here's the syntax for the BUTTON CLOSE statement:

```
BUTTON CLOSE num
```

num is the integer identifier assigned to the button when it was created. It's often useful to eliminate buttons after they've been used, and the BUTTON CLOSE statement can handle this for you. A BUTTON CLOSE statement with the argument 0 closes all buttons in the active window.

Load and compile the Button Close program from disk and run it.

```
' Button Close
' This program displays syntax information and waits for DIALOG events.
WINDOW 1, , (4, 64)-(215, 280), 3          ' create first window
PRINT "This program shows how the"
PRINT "WINDOW CLOSE statement"
PRINT "works."
PRINT
PRINT "Enter S to see the syntax of"       ' display instructions
PRINT "WINDOW CLOSE or Q to Quit."
PRINT

WHILE UCASE$(reply$) <> "Q"                ' loop until reply is "q" or "Q"
    INPUT "Letter: ", reply$               ' get input from user

IF UCASE$(reply$) = "S" THEN ' if reply is "s" or "S", display syntax
    WINDOW 2, , (220, 64)-(508, 280), 3     ' create syntax window
    TEXTFACE 1
    PRINT "WINDOW CLOSE"
    TEXTFACE 0
    PRINT                                   ' print syntax
    PRINT "The WINDOW CLOSE statement removes"
    PRINT "the window specified by the winNum"
    PRINT "argument."
    PRINT
    PRINT "Syntax: WINDOW CLOSE winNum"
    BUTTON 1, 1, "OK", (116, 180)-(171, 205), 1

    event% = 0
    WHILE (event% <> 1) AND (event% <> 6)      ' wait for button or Return
        event% = DIALOG(0) ' get event status
    WEND

    WINDOW CLOSE 2 ' close syntax window
  END IF
WEND
```

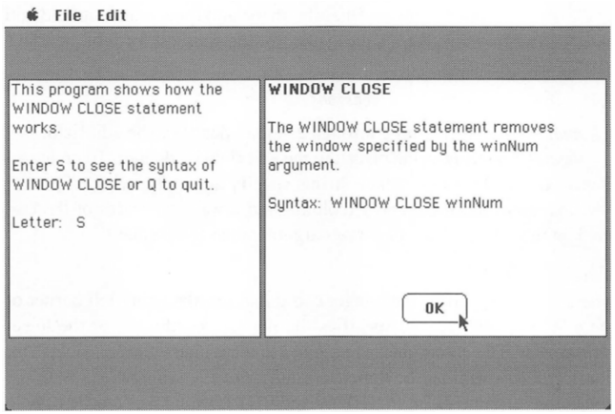
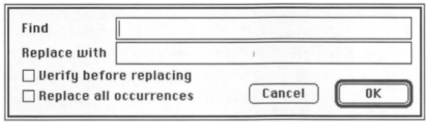


FIGURE 11-8.
Button Close: a program that demonstrates use of the BUTTON statement and the DIALOG function.

Type S and press Return. You'll see this output:

Click OK to close window 2, and enter Q to quit the program.

ADDING AN EDIT FIELD
Another item typically included in a dialog box is an edit field. An edit field is a rectangular text field used to get input from the user. An edit field is usually long enough to hold several words and high enough to hold text and the insertion point without their touching the top or bottom edges of the field. The Change dialog box in the AC/QuickBasic Interpreter/Compiler is a typical example of two edit fields in a window:



The Edit Field Statement
The EDIT FIELD statement defines the shape and type of an edit field and places it in a window.
Here's the syntax for the simplest form of the EDIT FIELD statement:

```
EDIT FIELD num, default, dimensions
```

   num is an integer from 1 through 255 that identifies the edit field.
   default is a string value placed in the edit field by default.
To create an empty edit field with no default string, specify an empty string (" ").
   dimensions are the x and y (column and row) coordinates of the edit field on the screen.
The dimensions argument has the format (x1, y1)-(x2, y2) where (x1, y1) specifies the integer coordinates of the upper left corner of the edit field and (x2, y2) specifies the integer coordinates of the lower right corner.
The coordinates are relative to the window the edit field is in; that is, (0, 0) specifies the upper left corner of the window. As with the dimensions arguments for WINDOW and BUTTON, it takes a little time to master the dimensions argument for EDIT FIELD.

The following statement adds an empty edit field 150 pixels long and 16 pixels high to the active window:

```
EDIT FIELD 1, "", (50, 30)-(200, 46)
```

Closing an Edit Field
After an edit field has been created, it can be closed with an EDIT FIELD CLOSE statement. Here's the syntax for the EDIT FIELD CLOSE statement:

```
EDIT FIELD CLOSE num
```

   num is the integer identifier that was assigned to the edit field when it was created. It's often useful to remove edit fields after they've been used, and the EDIT FIELD CLOSE statement can handle this for you. An EDIT FIELD CLOSE statement with the argument 0 closes all edit fields in the active window.

Getting Input from an Edit Field
We know how to create an edit field in a window. To use an edit field, we need two additional functions:
the DIALOG function to check for window events and the EDIT$ function to return the information in the edit field.
We've seen DIALOG in action with the BUTTON statement. Let's introduce our new player, the EDIT$ function.

The EDIT$ function
The EDIT$ function is straightforward- it returns a string containing the current contents of the specified edit field. Here's the syntax for the EDIT$ function:

```
EDIT$(num)
```

   num is the integer identifier that was assigned to the edit field when it was created.

To get things rolling, let's see how EDIT$ and DIALOG work together in a practice session.

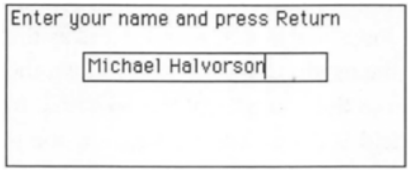
Practice:
Getting input from an edit field
The Edit Field program (Figure 11-9 on the next page) demonstrates how you can create an edit field and get input from it. The WINDOW, PRINT, and EDIT FIELD statements create a simple dialog box with instructions and an active edit field. The DIALOG loop then watches and waits as the user types data in the edit field. When the user presses the Return key, the loop ends and the EDIT$ function returns the contents of the edit field to thefullName$ variable. Then the edit field is closed and the user's name is displayed in the dialog box.

Load and compile the Edit Field program from disk and run it.

```
' Edit field
' This program creates and gets input from an edit field.
WINDOW 1, , (100, 100)-(350, 200), 3
PRINT "Enter your name and press Return"
EDIT FIELD 1, "", (50, 30)-(200, 46)

WHILE DIALOG(0) <> 6
WEND

fullName$ = EDIT$(1)

EDIT FIELD CLOSE 1

PRINT
PRINT "Nice to meet you, " fullName$; "!"
PRINT
PRINT
INPUT "Press Return to continue ... ", dummy$
```

FIGURE 11-9.
Edit Field: a program that uses EDIT$ to get input from an edit field.

When you run Edit Field, you see a prompt that asks you to enter your name. After you type your name, the edit field looks something like this:



When you press Return, the program gets your name, removes the edit field, and displays your name again.
Press Return again to end the program.

Checking for Button Events
Now let's add some buttons to make the dialog box look professional. After all, a "real" Apple][gs dialog box takes edit field input when the Return key is pressed and also when an OK button is selected. And most dialog boxes also allow users to reject the input they've entered, or reverse their decision to enter input at all, by selecting a Cancel button. The next practice session shows how you can modify the Edit Field program to accept input from two buttons in addition to the Return key.

Practice:
Using buttons with an edit field
The Edit Field 2 program (Figure 11-10) modifies the Edit Field program to include support for button events. Again, the program sets up a dialog box with instructions and an edit field, but this time OK and Cancel buttons are shown at the bottom of the box. A WHILE loop waits for a button event (1) or a Return key event (6), and the program branches based on the result.
The value in the edit field is read and displayed only if the OK button or the Return key is pressed.

Load and compile the Edit Field 2 program from disk and run it.

```
' Edit Field 2
' This program gets input from an edit field.
' The user enters data in the edit field and selects OK or Cancel or presses Return.

WINDOW 1, , (100, 100)-(350, 200), 3
PRINT "Enter your name and press Return"
EDIT FIELD 1, "", (50, 30)-(200, 46)
BUTTON 1, 1, "OK", (50, 65)-(105, 90)
BUTTON 2, 1, "Cancel", (145, 65)-(200, 90)

event% = 0
WHILE (event% <> 1) AND (event% <> 6)
    event% = DIALOG(0)
WEND
```

```
BUTTON CLOSE 0
PRINT

IF (DIALOG(1) = 1) OR (event%= 6) THEN
    fullName$ = EDIT$(1)
    EDIT FIELD CLOSE 1
    PRINT "Nice to meet you, "; full Name$; "!"
ELSE
    EDIT FIELD CLOSE 1
    PRINT
END IF

PRINT
PRINT
INPUT "Press Return to continue ... ", dummy$
```

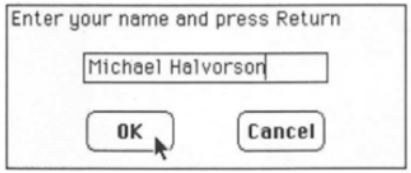


FIGURE 11-10.
Edit Field 2: a program that uses an edit field to get input and supports three dialog box events.

When you run the program you see a prompt that asks you to enter your name. After you type your name, see what happens when you select OK or Cancel or press Return.

This simple dialog box will be a useful addition to many of your programs.

Working with Multiple Edit Fields
To get information from more than one edit field in a window, you must provide a mechanism for switching back and forth among the edit fields. The switching part is easy - a simple form of the EDIT FIELD statement lets you switch from one edit field to another:

```
EDIT FIELD num
```

num is the integer identifier of the edit field you want to switch to. After the program executes the EDIT FIELD statement, the edit field associated with num will be the current edit field until you change it by using another EDIT FIELD statement.

The EDIT FIELD statement is a little deceptive. You might think, for example, that you could get by with the following code fragment in a program that needs input from two edit fields:

```
PRINT "Enter your name and job title"
EDIT FIELD 1, "" (50, 32)-(200, 48)
EDIT FIELD 2, "", (50, 63)-(200, 79)
EDIT FIELD 1
EDIT FIELD 2
fullName$ =EDIT$(1)
job$ = EDIT$(2)
```

Unfortunately, this routine won't work the way you intended it to-the user will have no control over when the first and second edit fields will be activated, and the program as written will activate the first edit field only momentarily.

A standard Apple][gs application is much more comprehensive. The user can move among edit fields by pressing Tab to move to the next edit field or by clicking in an edit field with the mouse. And the user can indicate that he or she has finished by selecting a button or pressing Return. We can add these capabilities to our programs by using the DIALOG function and logic that branches based on the event returned. When called with the argument 0, the DIALOG function returns 7 when the user presses Tab in an edit field and 2 when the user clicks on an edit field with the mouse.

SUMMARY
In this chapter you began to write graphical Apple][gs applications programs that process input and output through menus, windows, buttons, and edit fields. You learned how to monitor events with DIALOG loops and respond to user requests with a well-organized IF structures.

QUESTIONS AND EXERCISES
1. What is the purpose of the status argument of the MENU statement?
2. Write a AC/QuickBasic statement that assigns the menu item selected by the user to the variable itemNumber%.
3. Write a loop that waits for the user to select a menu item and gets the menu number and item number selected.
4. What is wrong with the following WINDOW statement? WINDOW 1. "My Window", (5, 40, 200, 90), 1
5. What statement do you use to make a window disappear?

6. Write a statement that creates a push button named OK that is 50 pixels wide and 20 pixels high.
7. What has happened if the DIALOG function has been called with the argument 0 and returns 6?
   (Consult our DIALOG function table on page 327.)
8. Write a program that generates a three-digit random lottery number each time the user clicks a Roll button.
   Display the output in a window and allow the user to click Quit to stop.
   Hint: Use the RND function we discussed in Chapter 6 to create the random number.
9. Make a copy of the Music Database program, name it Video Data-base, and modify it to track a home video collection.

The database should contain the following fields of information:

- Name of video
- Significant actors/contributors
- Year video was released
- Type of video (comedy, drama, horror, TV show)
- Medium (VHS, Beta, Laserdisc)

AC/QuickBasic Working with Graphics and Sound Learn AC/QuickBasic For the Apple][gs NOW

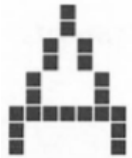
Now that you've learned the basics of AC/QuickBasic, you can enhance your programs by adding graphics and sound to them. In this chapter, you'll learn about the kinds of graphics your computer can create and how to use sound to jazz up your programs.

INTRODUCTION TO GRAPHICS PROGRAMMING
One exciting feature of your Apple][gs is its graphical interface.
Everything you see on your screen - the pull - down menus, the icons, and even the letters and numbers - is actually a graphical shape. You might find it hard to believe that the letters and numbers displayed on your screen are actually graphics, but in truth they are. Each alphanumeric character is stored in your computer's memory in a special form called a bit map. When you use a PRINT statement to instruct the AC/QuickBasic Interpreter/Compiler to display one or more characters in the Output window, it consults the Apple][gs's table of bit maps to find the bit maps for the characters you want to display and uses them as a guide as it "draws" the characters in the Output window.

Bit Maps
A bit map is literally a map of the individual dots that collectively form a character. If you look very closely at your Apple][gs's screen, you'll see that everything displayed there is actually formed from small dots. These dots, called pixels (short for picture elements), are the smallest items your Apple][gs can display. Later in this chapter, we'll work with pixels; for now, you just need to know that pixels collectively form alphanumeric characters.

Recall from Chapter 6 that your Apple][gs has several different fonts. A font is a particular style of character - Courier, Helvetica, and Geneva are the names of some common Apple][gs fonts. Each font has a unique look and style that sets it apart from other fonts.

Fonts come in several standard sizes. In other Apple][gs applications you've probably noticed that, in addition to a choice of fonts, you have a range of sizes to choose from for a particular font. Bit maps come in here, too. A table of bit maps consists of a particular size of a particular font. Your Apple][gs contains a separate bit map for each font - both style and size. Here's an illustration of an enlarged bit map for a capital A character:



The Geneva font {][gs font is Fast Shaston }
The default text font in your Apple][gs is called Geneva. Unless you do some special programming - programming way beyond the scope of this book -

You cannot use the installer to remove the Geneva font. When a program's PRINT statement displays text in the Output window, twelve-point Geneva is the default font.

The Fonts Installed in Your Apple][gs
The disks that come with your Apple][gs contain a program called the installer. Using this program, you can install new fonts or remove fonts already there.

This font installation/deletion program and the large number of fonts available for the Apple][gs make it likely that the fonts you have on your Apple][gs will differ from the fonts we talk about in this book.

Don't worry.

The concepts and examples you find in this book will still apply.

Even if the results you get are slightly different from ours !
You can experiment on your own to find the fonts on your system that yield the best results.

The TEXTFONT statement revisited
In Chapter 6, we used the TEXTSIZE and TEXTFONT statements to show different character heights and to change the font the AC/QuickBasic Interpreter/Compiler used to display characters in the Output window. Now let's combine the statements in a single program.

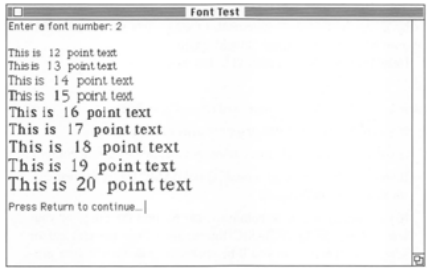
Practice:
Working with the TEXTFONT and TEXTSIZE statements

Load and compile the Font Test program (Figure 12-1) from disk:

```
' Font Test
' This program demonstrates how the AC/QuickBasic Interpreter/Compiler
' approximates font sizes for which it does not have a bit map.

CLS

INPUT "Enter a font number:", font%
TEXT FONT( font%)
PRINT

FOR i% = 12 TO 20
    TEXTSIZE i%
    PRINT "This is"; i% ;"point text"
NEXT i%

TEXTFONT 1            ' restore default text font
TEXTSIZE 12           ' restore original text size

INPUT "Press Return to continue ... ", dummy$
```

FIGURE 12-1.
Font Test: a program that uses the TEXT FONT and TEXTSIZE statements to see how the AC/QuickBasic Interpreter/Compiler approximates font sizes.

Run the program. Your Output window will look something like this:



Now let's take a look at ways to put your knowledge about different fonts and font sizes to work!

The Text Cursor
You'll remember that, as you type a program, the blinking vertical insertion point in the List window indicates where the next character you type will appear. In the Output window, the AC/QuickBasic Interpreter/Compiler uses its own insertion point, called the text cursor, to determine where characters will appear. The text cursor is normally invisible.

The CLS statement we've used in programs throughout this book to clear the Output window serves another purpose, too: CLS sets the text cursor to row 1, column 1 of the Output window. If the AC/QuickBasic Interpreter/Compiler then executes a PRINT statement, the PRINT statement prints its message starting at row 1, column 1. Assuming that the PRINT statement doesn't end in a semicolon, the AC/QuickBasic Interpreter/Compiler then moves the text cursor down to the first column of the next row (row 2, column 1).

The LOCATE Statement
You can override the AC/QuickBasic Interpreter/Compiler's placement of the text cursor by using the LOCATE statement. LOCATE lets you dictate within a program where the text cursor should appear.

Here's the syntax for a LOCATE statement:

```
   LOCATE [row][, column]
```

row is an integer value 1 or more, and column is an integer value 1 or more.
- If you omit row, the text cursor remains in the current row.
- If you omit column, the text cursor remains in the current column.
- If you omit both row and column, AC/QuickBasic leaves the text cursor in its current position.
- If you specify a row or column value beyond the range of your screen size, the AC/QuickBasic Interpreter/Compiler places the text cursor where you specify, but you'll be unable to read the text your program prints. It will be off the screen.
  We'll talk more about this shortly.

Practice:
Working with the LOCATE statement

1. Load and compile the Locate 1 program (Figure 12-2) from the Chapter 1 folder on disk.

```
' Locate 1
' This program demonstrates the LOCATE statement.

CLS

INPUT "Please enter the row coordinate (1-16):  ", rowNum%
PRINT
INPUT "Please enter the column coordinate (1-60):  ", colNum%
PRINT
INPUT "Please enter a message to display:  ", message$

CLS

' Print the column numbers across the top of the Output window.

FOR i% = 0 TO 50 STEP 10
    PRINT "1234567890";
NEXT i%

' Print the row numbers along the left side of the Output window.

FOR i% = 2 TO 16
    LOCATE i%, 1
    PRINT MID$(STR$(i%), 2)
NEXT i%

' Print the message at the user-supplied coordinates.

LOCATE rowNum%, colNum%
PRINT message$

LOCATE 17, 1
INPUT "Press Return to continue…", dummy$
```

FIGURE 12-2.
Locate 1: a program that demonstrates use of the LOCATE statement.

2. Run the program and enter row and column values greater than 2. (You'll see why in a moment.)
   After you run the program, your Output window will look something like the output shown on the next page.

Notice that the program prints column numbers across row 1 and row numbers down columns 1 and 2. The program does this so that you can easily verify that your message begins to print at the coordinates whose values you typed in for the LOCATE statement.

The STR$ Function
The Locate 1 program introduces a new function called STR$. The STR$ function converts a numeric value to a string of numerals.

Here's the syntax for the STR$ function:

STR$(value)

value is any numeric expression.

You can use STR$ together with the MID$ function to get rid of the space that appears to the left of non-negative numbers. This comes in handy when you position a number using the LOCATE statement. For example, these statements display the number 12.34 with no leading space:

```
number! = 12.34
PRINT MID$(STR$(number!), 2)
```

The example output we show for Locate 1 uses the row value 5, the column value 20, and the message Hello!
Run the program a few times so that you can fully understand how the LOCATE statement works.

3. Next, try running the program with a relatively large column value and a fairly long message. Try using the row value 5, for example, and the column value 60, and the message Where will this be displayed? Your Output window should look like this:

Notice that the AC/QuickBasic Interpreter/Compiler displayed the beginning of the message at the requested location (row 5, column 60) and then continued printing the message off the right edge of the window.
The AC/QuickBasic Interpreter/Compiler doesn't regard this as an error and won't display an error message, but you need to keep the boundaries of the window in mind as you design your programs: Know the maximum width and height of the Output\window before you use LOCATE to position text! For a full-size Output window on a Apple][gs computer, using the default Geneva text font and the default 12-point text size, the Output window is roughly 60 characters wide and 17 rows high.

4. Run the program again, and enter the row value 18, the column value 30, and the message Where will this text be displayed? Your Output window should look something like this:

Before printing the message, the AC/QuickBasic Interpreter/Compiler had to scroll the contents of the screen up so that the next row - row 18 - would be displayed. Doing this, as you can see, caused the topmost row to be pushed up and partially out of the window. NOTE: Looking at the previous illustration, you might wonder why the message didn't appear across the bottom of the window. The answer is that the AC/QuickBasic Interpreter/Compiler's Output window won't allow you to display information across the bottom row - the row normally taken up by a horizontal scroll bar. To see the effect of this prohibition, run the program again and use a row value 19 and a column value 30.

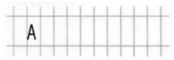
Again, know the size limits of the Output window you're working with, and you can avoid this problem. Unlike using a large column value that harmlessly causes your message to be displayed in an unreadable area, using a large row value can have a destructive side effect!

5. As an exercise, add a TEXTSIZE statement to the program to increase the size of the font the AC/QuickBasic Interpreter/Compiler will use to display your message. You'll have to adjust the maximum value of the second FOR loop (to change the value 16 to 10, perhaps) to prevent the Output window from scrolling, but you'll get a good idea of the effects different font sizes have on your choice of column and row values for the LOCATE statement.

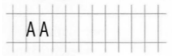
Using LOCATE to Create Animation
When you started reading this book, the first program you loaded from the disk and ran was called Welcome. Welcome caused the words Welcome to learn AC/QuickBasic Now! to "rise" from the bottom of the Output window. This animation was created with the LOCATE statement.

On a computer screen, you create animation by presenting the user with the illusion of movement. One way to do this is by printing a character repeatedly in the direction in which you want the character to move and at the same time replacing the previous occurrence of the character with a space, as shown in Figure 12-3. Although the characters don't actually move, the result provides the illusion of movement.

1. Print a character.



2. Print the same character in an adjacent location.



3. Print a space where the first character was.

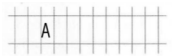


Figure 12-3.
Animating a character on the screen.

Practice:
Working with text-based animation
Let's say that you want to write a program that causes the letter X to "move" across the Output window. Here's how you might accomplish the animation.

1. Load and compile the Locate 2 program (Figure 12-4) from disk.

```
' Locate 2
' This program demonstrates the first step in creating animation.

CLS

FOR i% = 1 TO 50
    LOCATE 10, i%
    PRINT "X"
NEXT i%

INPUT "Press Return to continue…", dummy$
```

FIGURE 12-4.
Locate 2: the first step in animating a character.

2. Run the program. When you do, the program prints a solid row of X's across row 10 of the Output window. This in itself doesn't provide the illusion of movement, but you have accomplished the first step - you've caused the letter X to "move" across the screen.
Because the AC/QuickBasic Interpreter/Compiler executes the program quickly, it prints the row of X's almost instantaneously.
Use the techniques you learned in Chapter 6 to put a delay loop in the program to slow the rate at which the AC/QuickBasic Interpreter/Compiler prints the X's. (The next program includes a delay loop, but try to create one yourself before we move on.)

Next you need to create the illusion of making a single letter X move across the screen, even though the program will actually print 50 separate X's. To do this, you have your program print an X, print a second X immediately to the right of the first, and then erase the first X by overwriting it with two spaces. (It takes two spaces because the width of a space is only half the width of the letter X.)

NOTE: You can also use a standard space - a space the same width as the numeral 0 - instead of two regular spaces. To create a standard space, hold down the Option key and press the spacebar.

3. Load and compile the Locate 3 program (Figure 12-5) from disk.

```
' Locate 3
' This program demonstrates simple animation.

CLS

LOCATE 10, 1            ' position the first X
PRINT "X"
FOR i% = 2 TO 50        ' loop through the rest
    LOCATE 10, i%
    PRINT "X"           ' print an X next to the previous X
    LOCATE 10, i% - 1
    PRINT "  "          ' "erase" the previous X (use two spaces!)

    FOR j% = 1 TO 300   ' delay loop
    NEXT j%

NEXT i%

INPUT "Press Return to continue…", dummy$
```


Figure 12-5.
Locate 3: A program that animates a character.

Notice that the first X is printed before the loop begins. The reason for this lies in how the X's are erased. The LOCATE values for printing the two spaces are 10, i% - 1. If the loop started with the value 1, the first time the program tried to print a space it would try to print the space at row 10, column 0. Because 0 is not a valid value, the AC/QuickBasic Interpreter/Compiler would stop and display an error message.

4. Run the program. When you do, you should see the letter X "move" right across the screen! You might want to stop here and experiment with the program. Including such animation in your own programs can really get the user's attention!

The Welcome program
Now that you've learned the basics of text-based animation, let's re-examine the Welcome program.

Practice:
Working with the Welcome program

Load and compile the Welcome program (Figure 12-6) from disk.

```
' Welcome
'
' Welcome to the Microsoft
' AC/QuickBasic Interpreter/Compiler!
'
' To run this program, choose Run Program
' from the Run menu, or hold down the
' Command key and press R.

CLS

topRow% = 1
FOR i% = 1 TO 5

    READ word$

    FOR row% = 16 TO topRow% STEP -1
        LOCATE row%, 1
        PRINT word$
        LOCATE row% + 1, 1
        PRINT SPACE$(LEN(word$) * 2)
        SOUND (2400 / row%), 1
    NEXT row%

topRow% = topRow% + 1
NEXT i%

LOCATE 1, 1

DATA "Welcome", "to", "Learn", "BASIC", "Now!"
```

Figure 12-6.
Welcome: the introductory program from Chapter 2.

Instead of printing only a single letter, as in the previous example, this program prints entire words. And to "erase" the previous word, this program uses a string of spaces to cover all the letters of the word at the same time. Note the statement

```
PRINT SPACE$(LEN(word$) * 2)
```

Again, because the width of a space is only half the width of an average character, the program must use twice the number of spaces to provide adequate coverage.

The Welcome program uses sound to reinforce the "rising" illusion of each word. Notice how the SOUND statement uses the current value of row% as a divisor of the value 2400. As the value of row% decreases, the result of the division

operation increases and the resulting pitch of the tone becomes higher. (You'll learn more about the SOUND statement later in this chapter.)

A Last Look at Text-based Animation
Using text-based animation can really add excitement to your programs. Important things to remember are the size and other limitations of the window you're working in, and the size and style of the font you're using. With adequate planning, adding text-based animation to your programs is a relatively easy way to really spice up your programs. Text-based animation does have its limitations. You're limited to just alphanumeric characters, and trying to create graphical shapes such as boxes and circles using alphanumeric characters can be difficult if not impossible. Even the picture-style characters of fonts such as Zapf Dingbats and Cairo aren't much help when you want to draw a mountain. As you might expect, the AC/QuickBasic Interpreter/Compiler and the Apple][gs collectively provide you with a large array of tools for performing all kinds of graphical tasks. In the next section, we'll get you started with some of the basics and show you how to put some of these tools to work for you!

INTRODUCTION TO GRAPHICAL SHAPES
Working with different sizes and styles of text can be rewarding, but the real fun of graphics programming comes from creating and animating shapes on the screen. Starting with individual pixels, this section shows you how to create lines, boxes, circles, and complex shapes - even how to create simple animation!

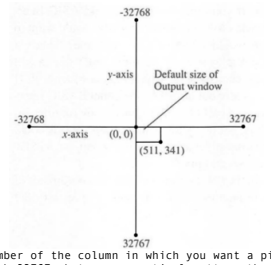
Drawing Individual Points in the Active Window
Recall from Chapter 11 that any statement in your program that causes something to appear in the Output window will actually display its output in the active window. Because the sizes of the Output window and any windows you create can vary, you need to understand how the AC/QuickBasic Interpreter/Compiler views these windows before you begin graphics programming.

Also recall from Chapter 11 that you had to specify coordinates when you wanted your program to create a window on the screen. You must specify coordinates when you create graphical shapes, too: To create a point, line, box, circle, or other shape, you must specify its coordinates. The drawing on the opposite page illustrates the maximum allowable values for coordinates plus the minimum and maximum values for the default - size Output window.

You use two AC/QuickBasic statements to place individual pixels on the screen at specific locations: PSET and PRESET.

The PSET statement
Here's the syntax for the PSET statement in its simplest form:

PSET (xcoordinate, ycoordinate)[, color]



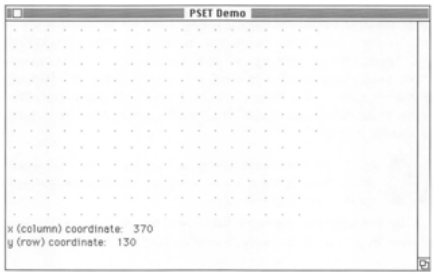
xcoordinate is the number of the column in which you want a pixel to appear. Technically, the value of xcoordinate can be from -32768 through 32767, but as a practical matter, the largest value you can use depends on the size of the window in which you'll display the pixel. If you want to display a pixel at the right edge of the default-size Output window, for example, the largest value you can use is 490.

ycoordinate is the number of the row in which you want a pixel to appear. Again, although the range of values you can use is -32768 through 32767, the size of the window in which you want to display the pixel will limit the maximum value of the pixel. If you want to display a pixel in the bottom of the default-size Output window, for example, the largest value you can use is 296.

NOTE: The parentheses around the xcoordinate and ycoordinate values are not optional. You must use the parentheses around these values in the PSET statement.

color is a value that represents the color of the pixel. The two values for color are 30 and 33:
The value 30 displays the pixel in white; the value 33 displays the pixel in black. If you omit color, the AC/QuickBasic Interpreter/Compiler displays the pixel in black. (Are you wondering why you'd want to display a white pixel, which would effectively be invisible?
That's a handy way to erase a pixel you displayed with a previous PSET statement.) Black and white are your only choices here. If you have a Apple][gs II system that can display color, you're out of luck - the AC/QuickBasic Interpreter/Compiler can't display color. You can call the QuickDraw][routines located in the Apple][gs Toolbox, but that's beyond the scope of this book. A PSET statement sets only one pixel at a time, but you can put a PSET statement inside a loop to draw several pixels in a row.

To erase a pixel drawn with the PSET command, put the coordinates of the pixel you want to erase in another PSET command and specify the background color for color.



Using the PSET statement
1. Load and compile the PSET Demo program (Figure 12-7) from disk.
2. Run the program. Your Output window should look like this:

```
' PSET Demo
' This program demonstrates the PSET statement.

CLS

FOR i% = 10 TO 420 STEP 20        ' values for xcoordinate
    FOR j% = 10 TO 240 STEP 20    ' values for ycoordinate
        PSET (i%, j%)
        LOCATE 16, 1                        ' position text cursor
        PRINT "x (column) coordinate: "; i%
        LOCATE 17, 1
        PRINT "y (row) coordinate: "; j%

        FOR k% = 1 TO 2000                ' delay loop
        NEXT k%

    NEXT j%
NEXT i%
```

Figure 12-7.
PSET Demo: a program that demonstrates use of the PSET statement.

NOTE: If the PSET Demo program (or any similar program later in this chapter) runs too slowly for your tastes, modify the delay loop so that it loops a smaller number of times.

Practice:
Using PSET to erase a pixel
1. Load and compile the PSET Demo 2 program (Figure 12-8 on the next page) from disk.
2. Run the program. This program is identical to PSET Demo except for the addition of the PSET statement that follows the delay loop:

```
PSET (i%, j%), 30
```

Notice that this PSET statement uses the same coordinates as the earlier PSET statement, but with one important difference: This PSET statement uses the color option with the value 30 to display a white pixel. Because the white pixel overlays the black one, it effectively erases the previously displayed pixel from the window.

```
' PSET Demo 2
' This program demonstrates the PSET statement.

CLS

FOR i% = 10 TO 420 STEP 20        ' values for xcoordinate
    FOR j% = 10 TO 240 STEP 20    ' values for ycoordinate
        PSET (i%, j%)
        LOCATE 16, 1                        ' position text cursor
        PRINT "x (column) coordinate: "; i%
        LOCATE 17, 1
        PRINT "y (row) coordinate: "; j%

        FOR k% = 1 TO 2000                ' delay loop
        NEXT k%

        PSET (i%, j%), 30                   ' "erase" the pixel

    NEXT j%
NEXT i%
```

Figure 12-8. SET Demo 2: using the PSET statement to erase a pixel.

The PRESET statement
The PRESET statement works the way the PSET statement does, with one reversal: If you omit the color argument, the AC/QuickBasic Interpreter/Compiler displays the pixel in the background color. This lets you erase pixels simply by omitting color. Here's the syntax for the PRESET statement:

```
PRESET (xcoordinate, ycoordinate)[, color]
```

Note that the PRESET statement erases whatever happens to be located at (xcoordinate, ycoordinate). If you use a PRINT statement to print a message, for instance, and then use a PRESET statement whose coordinates are in the same area as the message, the PRESET statement might "take a bite" out of a letter in the message.

Practice:
Working with the PRESET statement

1. Load and compile the Hailstones program (Figure 12-9) from disk.

```
' Hailstones
' This program demonstrates the PRESET statement.

CLS

PRINT "Press any key to stop…"

WHILE INKEY$ = ""
    RANDOMIZE TIMER
    randCol% = INT(RND(1) * 480)  ' random column number for "hailstones";
                                                '   assumes default-size Output window
    FOR i% = 1 TO 296
        PSET (randCol%, i%)
        PRESET (randCol%, i% - 1)
    NEXT i%
WEND
```

Figure 12-9.
Hailstones: a program that demonstrates use of the PRESET statement.

2. Run the program. As the program runs, you can observe one of the interesting side effects of PRESET. Just before the "hailstones" appear, the program prints the message Press any key to stop. As the program continues to run, you'll see the PRESET statements demolish this message as the "hailstones" pass over it.

Note the use of the INKEY$ function. This is your first encounter with INKEY$, which checks the keyboard to see whether a key has been pressed. If a key has been pressed, INKEY$ returns the ASCII code associated with the key; if not, INKEY$ returns an empty string. Unlike the INPUT and LINE INPUT statements, the IN KEY$ function doesn't wait for the user to press Return before it continues.

Positioning Pixels with Coordinates
You can use two coordinate systems to position pixels in the active window: absolute coordinates and relative coordinates.

Absolute coordinates
The coordinate system you've used so far is the absolute coordinate system. In the absolute system, all coordinate values are based on the upper-left-corner coordinate of the active window- (0, 0). To position a pixel in the active window, you count (or, more likely, guess) the number of columns over and the number of rows down in order to come up with the coordinate values of the pixel position.

As you've seen, the absolute coordinate system works fine, but if you want to use individual pixels to create a particular shape, you're in for a lot of guesswork and counting. The relative coordinate system makes the job of determining coordinate values much easier.

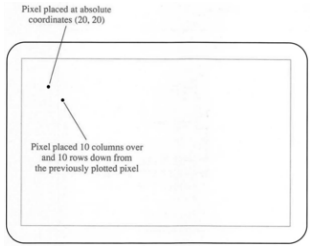
Relative coordinates
With relative coordinates, the coordinates you specify for a specific pixel are relative to the coordinates of the last pixel you placed on the screen. The illustration on the opposite page demonstrates this. To place the first pixel on the screen, you must use absolute coordinates because no relative pixel location exists. For the second and subsequent pixels, however, you can use relative coordinates.

To switch to the relative coordinate system, you use the STEP keyword within a PSET or PRESET statement.

The STEP keyword
Here's the syntax for a PSET statement that contains the STEP keyword:

```
PSET STEP(xcoordinate. ycoordinate)[, color]
```

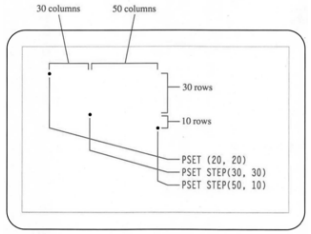
(The syntax is identical for a PRESET statement that contains. the STEP keyword; simply substitute the PRESET keyword for the PSET keyword.)



The xcoordinate and ycoordinate values are positive or negative numbers that tell the AC/QuickBasic Interpreter/Compiler where to position the pixel in relation to the last pixel it positioned with a PSET or PRESET statement. If no previous pixel exists, the AC/QuickBasic Interpreter/Compiler uses the absolute coordinates (0, 0).

The screen illustration at the top of the next page demonstrates this "daisy chain" effect: Each pixel's but the first's location is dependent on the previous pixel's location. Always keep in mind that the STEP keyword causes the AC/QuickBasic Interpreter/Compiler to position a pixel relative to the location of the most recent pixel.

Practice:
Working with the STEP keyword
1. Load and compile the Zoom program (Figure 12-10 on page 385) from disk.



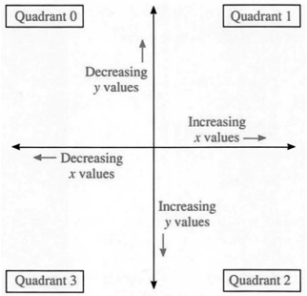
2. Run the program. The program assumes a standard-size Output window. Press any key to stop the program.

The Zoom program provides the illusion of traveling through space by simulating "stars" zooming by. Inside the WHILE loop, the program sets an "invisible" pixel at the center of the screen. This is the reference point on which later PSET statements base their coordinates. Next, the random number quad% determines which quadrant the star will zoom through. The illustration on page 386 shows how the quadrants are numbered.

Notice the two IF statements in Zoom-one for quadrants 0 and 1 and another for quadrants 0 and 3. The values modified in the IF statements affect the PSET coordinates inside the FOR loop-some use positive coordinates, some use negative coordinates, and some use both. (Quadrant 2 uses positive values, so no modification is necessary.)

```
' Zoom
' This program uses the illusion of zooming through space
'   to demonstrate the STEP keyword.

CLS

delay% = 200                            ' controls how fast stars "shoot"

PRINT "Press any key to stop…"

WHILE INKEY$ = ""
    PSET (245, 148), 30             ' set center (for default Output window)
    quad% = INT(RND(1) * 4)       ' random number for quadrant from
                                          '   which the "star" will shoot
    randX% = INT(RND(1) * 10)   ' random number for column movement
    randY% = INT(RND(1) * 10)   ' random number for row movement

    IF quad% = 0 OR quad% = 1 THEN  ' normal movement is down;
        randY% = -randY%                    '   reverse y value to move up
    END IF
    IF quad% = 0 OR quad% = 3 THEN  ' normal movement is to right;
        randX% = -randX%                    '   reverse x value to move to left
    END IF

    FOR i% = 1 TO 12
        PSET STEP(-randX% * i%, randY% * i%)  ' draw "star"

        FOR j% = 1 TO delay%        ' delay loop
        NEXT j%

        PRESET STEP(0, 0)           ' erase "star"
    NEXT i%

WEND
```

FIGURE 12-10.
Zoom: zooming through space using the STEP keyword.

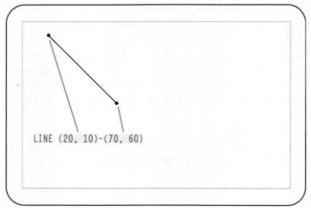


In the PSET statement, a random number is multiplied by the current value of the loop variable i%. With each loop the resulting value increases, causing the "star" to move farther from the center of the Output window. The "star" starts off slowly in the center and accelerates as it gets closer to the edge of the window.

Finally, notice how the PRESET statement is used. The STEP keyword causes the PRESET statement to use relative coordinates - coordinates determined by the PSET statement. Because the coordinates are (0, 0), the PRESET statement causes the AC/QuickBasic Interpreter/Compiler to place the PRESET pixel at the same location as the preceding PSET pixel - effectively erasing it.

Just for fun, think about how you might write this program with absolute coordinates. You can do it - you might even enjoy the challenge - but as you'll soon discover, relative coordinates are better suited to this particular programming feat.

CREATING COMPLEX SHAPES
Plotting individual pixels gives you precise control as you create graphics in the Output window. But to produce complex shapes such as lines, boxes, circles, and polygons, you would have a lot of coordinate calculating to do. For a line, you could use a loop to quickly plot a series of individual points, but you'd still be faced with a lot of calculation. Fortunately, AC/QuickBasic provides a set of tools that make the job of creating complex shapes a snap.



The LINE Statement
The LINE statement offers a single-statement way to create lines. If you use the LINE statement, the AC/QuickBasic Interpreter/Compiler does most of the work for you.

Drawing a simple line
Here's the syntax for the LINE statement in its simplest form:

```
LINE (x1, y1)-(x2, y2)[, color]
```

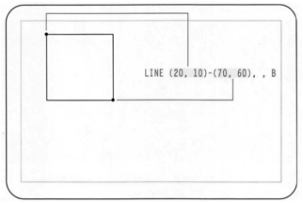
Notice that unlike the PSET and PRESET statements, which use only a single coordinate set, a LINE statement uses two sets of coordinates:

• x1 and y1 are the column and row coordinates of the starting point of the line.

• x2 and y2 are the column and row coordinates of the ending point of the line.

You must enclose the coordinates in parentheses and separate the sets with a hyphen.

The illustration is an example of how the LINE statement works. The starting point needn't be to the left of the ending point. In fact, you can start the line in the lower right corner of the screen and work upward and to the left if you want. Simply specify the coordinates, and the QuickBASIC Interpreter/Compiler "connects the dots."



Practice:
Working with the LINE statement
1. Load and compile the Lines program (Figure 12-11) from disk.

```
' Lines
' This program demonstrates the LINE statement.

CLS

delay% = 100                            ' controls delay between lines drawn

PRINT "Press any key to stop…"

WHILE INKEY$ = ""
    x1pos% = INT(RND(1) * 490)          ' start coordinates
    y1pos% = INT(RND(1) * 296)
    x2pos% = INT(RND(1) * 490)          ' end coordinates
    y2pos% = INT(RND(1) * 296)
    LINE (x1pos%, y1pos%)-(x2pos%, y2pos%)

    FOR i% = 1 TO delay%                    ' delay loop
    NEXT i%

WEND
```

FIGURE 12-11. Lines: drawing random lines with the LINE statement.

2. Run the program. The program prints random lines in the Output window. Press any key to stop the program.

Drawing a box
Drawing a box, hollow or filled, is as easy as drawing a single line. You could use four LINE statements to accomplish the job, but the inventors of BASIC figured that boxes were common enough to warrant their own feature in the BASIC language-saving you considerable time and calculation.

Interestingly enough, you use a LINE statement to create a box:

```
LINE (x1, y1)-(x2, y2)[, [color][, [B[F]]]]
```

This is the same LINE statement you just learned about, with one important difference. If you add the B option at the end of the statement, the AC/QuickBasic Interpreter/Compiler draws a box using the start and end positions as opposite corners of the box. The illustration at the top of the next page shows how this works.
Notice that the AC/QuickBasic Interpreter/Compiler does not draw a line directly between the start and end points as it would in response to a simple LINE statement.

If you choose to include the F option immediately after the B option, AC/QuickBasic fills the box with the current foreground color. You can use the F option only if you use the B option.

Practice:
Working with hollow and filled boxes

1. Load and compile the Boxes program (Figure 12-12) from disk.
2. Run the program. Based on your responses, the program creates either a random pattern of hollow boxes or a random pattern of white-filled or black-filled boxes. As this program shows, the boxes can be any size or shape and you can put them anywhere on the screen.

```
' Boxes
' This program demonstrates the box-drawing capabilities
'   of the LINE statement.

CLS

delay% = 800

INPUT "Hollow or solid boxes (H or S)?  ", box$
PRINT "Press any key to stop..."

WHILE INKEY$ = ""
    IF INT(RND(1) * 2) = 0 THEN
        shade% = 30                                 ' white
    ELSE
        shade% = 33                                 ' black
    END IF
    x1pos% = INT(RND(1) * 490)                  ' start coordinates
    y1pos% = INT(RND(1) * 296)
    x2pos% = INT(RND(1) * 490)                  ' end coordinates
    y2pos% = INT(RND(1) * 296)

    IF UCASE$(box$) = "H" THEN
        LINE (x1pos%, y1pos%)-(x2pos%, y2pos%), shade%, B
    ELSEIF UCASE$(box$) = "S" THEN
        LINE (x1pos%, y1pos%)-(x2pos%, y2pos%), shade%, BF
    END IF

    FOR i% = 1 TO delay%                        ' delay loop
    NEXT i%

WEND
```

FIGURE 12-12.
Boxes: drawing random boxes with the LINE statement.

Drawing complex shapes
The lines, hollow boxes, and filled boxes you just worked with all used absolute coordinates. As with PSET and PRESET you can use the STEP keyword with the LINE statement to specify relative coordinates for lines and boxes. Or you can omit the starting point of the line or box-forcing the AC/QuickBasic Interpreter/Compiler to use the ending point of the
last-drawn object as the starting point for the new object.

Here is the syntax for a LINE statement that uses relative coordinates:

```
LINE [[STEP](x1, y1)]-[STEP](x2, y2)[, [color][, [B[F]]]]
```

Using relative coordinates allows you to create complex line drawings that would take some time to calculate if you used absolute coordinates. In the following table, the sample LINE statements use both absolute and relative coordinates, and you can see the result of each statement.
In each case, assume that the last point was set at (20, 20).

LINE statement	Result
LINE - (30, 50)	Draws from (20, 20) to (30, 50)
LINE -STEP(30, 50)	Draws from (20, 20) to (50, 70)
LINE (40, 40)-STEP(60, 60)	Draws from (40, 40) to (100, 100)
LINE STEP(30, 50)-(70, 70)	Draws from (50, 70) to (70, 70)
LINE STEP(30, 50)-STEP(70, 70)	Draws from (50, 70) to (120, 140)

Here are some conclusions you can draw from the sample LINE statements and their results:

- If you omit the starting coordinates, the AC/QuickBasic Interpreter/Compiler uses the coordinates of the most recent point as the starting coordinates.
- If you omit the starting coordinates and use the STEP keyword with the ending coordinates, the AC/QuickBasic Interpreter/Compiler positions the ending point relative to the most recent point's coordinates.
- If you use starting coordinates and use the STEP keyword with the ending coordinates, the AC/QuickBasic Interpreter/Compiler positions the ending point relative to the specified starting coordinates.
- If you use the STEP keyword with the starting coordinates and do not use the STEP keyword with the ending coordinates, the AC/QuickBasic Interpreter/Compiler positions the starting coordinates relative to the coordinates of the most recent point and positions the ending point at the specified absolute coordinates.
- If you use the STEP keyword with both the starting and the ending coordinates, the AC/QuickBasic Interpreter/Compiler positions the starting coordinates relative to the coordinates of the most recent point and positions the ending point relative to the coordinates of the starting point it just calculated.

Using relative coordinates allows you to "daisy chain" lines and boxes together, simplifying the job of drawing complex shapes.

Practice:
Working with LINE and relative coordinates

1. Load and compile the Simple Sketch program (Figure 12-13) from disk:

```
' Simple Sketch
' This program demonstrates the use of the STEP keyword in
'   the LINE statement.

CLS

blank$ = SPACE$(60)

PRINT "To quit, enter 0 for both horizontal and vertical movement."
PSET (245, 146)                 ' establish a starting point

horiz% = 1                          ' set values to ensure that WHILE loop begins
vert% = 1
WHILE horiz% <> 0 OR vert% <> 0
    LOCATE 16, 1
    INPUT "Horizontal movement (+ or -): ", horiz%
    INPUT "   Vertical movement (+ or -): ", vert%
    LINE -STEP(horiz%, vert%)
    LOCATE 16, 1
    PRINT blank$                    ' erase input prompts
    PRINT blank$
WEND
```

FIGURE 12-13.
Simple Sketch: a simple line-drawing program.

2. Run the program. This program is actually a simple drawing program. The program begins by plotting the starting point for your drawing at (245, 146). The bulk of the work is done in the WHILE loop. The program asks you to enter horizontal and vertical values, which can be either positive or negative, for the coordinates of the next point. After you've entered these values, the program uses the LINE statement together with the STEP keyword to draw the line.

The CIRCLE Statement
The AC/QuickBasic Interpreter/Compiler also lets you easily draw circles. By using the CIRCLE statement, you can draw circles of various sizes that, in conjunction with the other graphics tools at your disposal, allow you to create interesting and useful graphics.

Here's the syntax for the CIRCLE statement in its simplest form;

```
CIRCLE (xcoordinate, ycoordinate), radius[, color]
```

xcoordinate and ycoordinate are the same horizontal and vertical window coordinates you've been working with. In the CIRCLE statement, the xcoordinate and ycoordinate values tell AC/QuickBasic where to position the center point of the circle.
radius is a value that specifies, in pixels, the radius of the circle. (The radius of a circle is one-half its diameter.)
color is an integer value that tells AC/QuickBasic what color to make the circle. The color value 30 draws a white (effectively invisible) circle; the color value 33, the default, draws a black circle.
If you omit color, the AC/QuickBasic Interpreter/Compiler draws a black circle.
As in any other graphics statement, the coordinates you can use in the CIRCLE statement are limited by the size of the active Output window.

Practice:
Working with the CIRCLE statement

1. Load and compile the Circle program (Figure 12-14) from disk.

```
' Circle
' This program demonstrates the CIRCLE statement.

CLS

WHILE UCASE$(dummy$) <> "Q"
    LOCATE 15, 1
    INPUT "Please enter the column number:  ", colNum%
    INPUT "     Please enter the row number:  ", rowNum%
    INPUT "             Please enter the radius:  ", radius%
    CIRCLE (colNum%, rowNum%), radius%
    INPUT "Press Return to continue, or enter Q to quit: ", dummy$
    CLS
WEND
```

FIGURE 12-14.
Circle: a simple circle-drawing program.

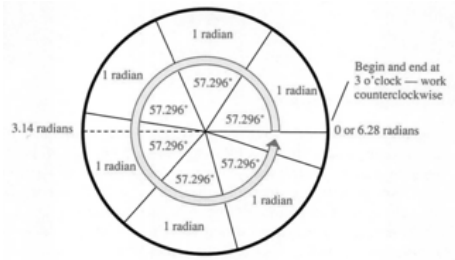
2. Run the program. It asks you to enter a column value, a row value, and a radius value. The CIRCLE statement in the middle of the program draws a circle based on the numbers you entered. Then, when you press Return to continue, the CLS statement "wipes the slate clean," and the process repeats until you enter q or Q to quit. Try entering values greater than the maximum Output window coordinates, or enter a relatively large radius value. You can use this technique to create a particular effect, such as an near one edge of the window.

Drawing arcs
The CIRCLE statement lets you draw arcs-segments of a circle. Use a CIRCLE statement with starting and ending values to draw an arc:

```
CIRCLE (xcoordinate, ycoordinate), radius[, [color][, [start][, end]]]
```

This is the same syntax as that for the simple CIRCLE statement except for the addition of start and end values. The start and end values are the measurements, in radians, of the starting point and the ending point of the arc.

Measuring arcs in radians
Although you might be more accustomed to measuring arcs in terms of degrees, the CIRCLE statement requires that you provide starting and ending measurements in radians. The diagram below illustrates the relationship between the degree and the radian measurement systems. When drawing an arc of your own, simply refer to this illustration to determine the starting and ending points of your own arc. The illustration includes conversion information that you'll find helpful if you know the measurement of your arc in degrees.

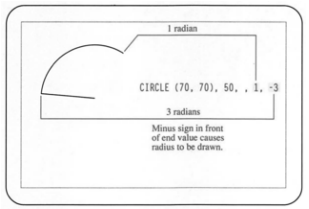


Radians= (3.14159 X(angle in degrees))
1 degree = 0.0174532 radian 180

Measurement starts at 0 (3 o'clock) and progresses counterclockwise to a maximum of 6.28 radians.
The starting point 0 and the ending point 6.28 are the same location.

The values you provide for start and end are numbers from 0 through 6.28.
(Note: The value 6.28 is technically 2 x pi; we're using rounded numbers for simplicity.)

If you put a minus sign in front of the starting or the ending value, the AC/QuickBasic Interpreter/Compiler draws a line (the radius) from that point on the arc to the center of the ''circle.'' The sample arc below shows a line from the ending point to the center of the ''circle'' because of the minus sign in front of the end value in the CIRCLE statement.



Practice:
Working with arcs

1. Load and compile the Circle 2 program (Figure 12-15 on the next page) from disk.

2. Run the program. This program gives you a chance to experiment with creating arcs. Enter starting and ending values from 0 through 6.28. Put a minus sign in front of a number if you want AC/QuickBasic to draw a line from that point to the center of the circle. Note that the starting value can be smaller than the ending value.
Try several combinations of values to see how they affect the arcs the CIRCLE statement creates.

```
' Circle 2
' This program demonstrates how to create arcs with the CIRCLE statement.

CLS

WHILE UCASE$(dummy$) <> "Q"
    LOCATE 14, 1
    PRINT "Enter radian values from 0 through 6.28."
    PRINT "(A negative value draws a radius.)"
    INPUT "Please enter the starting point:  ", starting!
    INPUT "    Please enter the ending point:  ", ending!
    CIRCLE (245, 105), 75, , starting!, ending!
    INPUT "Press Return to continue, or enter Q to quit:  ", dummy$
    CLS
WEND
```

FIGURE 12-15.
Circle 2: a program that creates arcs and pie wedges.

USING BASIC GRAPHICS
We've covered quite a bit of ground here. AC/QuickBASIC offers some powerful and useful drawing tools. Regrettably, we can't cover them all here. But the tools you've become acquainted with in this chapter should keep you busy for quite some time and enable you to add some spiffy graphics to your own programs. Now you can add these enhancements to your programs:

- Bar charts that graphically illustrate data
- Pie charts that show the distribution of data in different categories
- Illustrations that make your programs more interesting
- Screen savers that protect your monitor from burn-in
- Animation that makes games and simulations more realistic

Experiment - it's the best way to learn!

INTRODUCTION TO PROGRAMMING WITH SOUND
You've just learned how to use graphics to enliven your programs. Now you can learn how to make your programs even livelier by using the sound capabilities of your Apple][gs and the AC/QuickBasic Interpreter/Compiler.

The SOUND Statement Revisited
In Chapter 6, where you learned about loops, you had a brief introduction to the SOUND statement.

Now is an appropriate time for a quick review.

Here's the syntax for the SOUND statement:

SOUND frequency, duration

   frequency is an integer value that specifies the frequency, in hertz (cycles per second), of the tone you want. Low frequencies create low tones; high frequencies create high tones. The range of human hearing is approximately 20 to 20,000 hertz, so you'll achieve the best results using a frequency value in that range.
   duration is an integer or floating-point value that tells AC/QuickBasic how long to play the tone.
The duration value specifies for how many "clock ticks" the note should play - there are 18.2 clock ticks per second, so a duration value of 9.1 would cause, the AC/QuickBasic Interpreter/Compiler to play the note for one-half second.

You must specify both a frequency and a duration value in a SOUND statement.

The sound effects in the sample programs you've seen so far were created with no particular tune in mind, but you can choose frequency values that match musical notes. Figure 12-16 on the next page lists the standard frequency values and their associated notes as established by the American Standards Association in 1936.

The subscripted number at the end of each note name indicates the octave of that note. If you study Figure 12-16, you'll notice that to raise a note one octave, you simply double its frequency-approximately. For example, C_2 has a frequency of 65.41 hertz, and C_3 has a frequency of 130.81-almost double the frequency of C_2. This fact can help you when you add music to your own programs-all you need to do is double a particular frequency to raise that note one octave.

Note	Frequency	Note	Frequency	Note	Frequency
C_0	16.35	A_2	110.00	F_5	698.46
$C\#_0$	17.33	$A\#_2$	116.54	$F\#_5$	739.99
D_0	18.35	B_2	123.47	G_5	783.99
$D\#_0$	19.45	C_3	130.81	$G\#_5$	830.61
E_0	20.60	$C\#_3$	138.59	A_5	880.00
F_0	21.83	D_3	146.83	$A\#_5$	932.33
$F\#_0$	23.13	$D\#_3$	155.56	B_5	987.77
G_0	24.50	E_3	164.81	c_6	1046.50
$G\#_0$	25.96	F_3	174.61	$C\#_6$	1108.73
A_0	27.50	$F\#_3$	185.00	D_6	1174.66
$A\#_0$	29.13	G_3	196.00	$D\#_6$	1244.51
B_0	30.87	$G\#_3$	207.65	E_6	1328.51
C_1	32.70	A_3	220.00	F_6	1396.91
$C\#_1$	34.65	$A\#_3$	233.08	$F\#_6$	1479.98
D_1	36.70	B_3	246.94	G_6	1567.98
$D\#_1$	38.89	C_4	261.63	$G\#_6$	1661.22
E_1	41.20	$C\#_4$	277.18	A_6	1760.00
F_1	43.65	D_4	293.66	$A\#_6$	1864.66
$F\#_1$	46.25	$D\#_4$	311.13	B_6	1975.53
G_1	49.00	E_4	329.63	C_7	2093.00
$G\#_1$	51.91	F_4	349.23	$C\#_7$	2217.46
A_1	55.00	$F\#_4$	369.99	D_7	2349.32
$A\#_1$	58.27	G_4	392.00	$D\#_7$	2489.02
B_1	61.74	$G\#_4$	415.30	E_7	2637.02
C_2	65.41	A_4	440.00	F_7	2793.83
$C\#_2$	69.30	$A\#_4$	466.16	$F\#_7$	2959.96
D_2	73.42	B_4	493.88	G_7	3135.96
$D\#_2$	77.78	C_5	523.25	$G\#_7$	3322.44
E_2	82.41	$C\#_5$	554.37	A_7	3520.00
F_2	87.31	D_5	587.33	$A\#_7$	3729.31
$F\#_2$	92.50	$D\#_5$	622.25	B_7	3951.07
G_2	98.00	E_5	659.26	C_8	4186.01
$G\#_2$	103.83				

FIGURE 12-16.
Musical note frequencies ranging over eight octaves.

Practice:
Working with the SOUND statement

1. Load and compile the Scales program (Figure 12-17) from disk.

```
' Scales
' This program demonstrates the SOUND statement.

CLS

INPUT "Please enter an octave value (1-5):  ", octave%

' Play one octave.

FOR i% = 1 TO 8
    READ note%                              ' read frequency of note
    note% = note% * (2 ^ octave%)           ' raise note to desired octave
    SOUND note%, 6                          ' play note
NEXT i%

' Frequencies of the notes of the C major scale in the first octave.

DATA 32.70, 36.70, 41.20, 43.65, 49.00, 55.00, 61.74, 65.41
```

FIGURE 12-17.
Scales: a program that plays a scale.

2. Run the program and then enter an octave value.

Musical Notation and BASIC
Most of us aren't used to dealing with musical notes in terms of each note's frequency, but AC/QuickBasic requires that you tell it what notes to play by using their frequencies.

If you plan to add music to your programs - either a tune you wrote yourself or some sheet music you want to copy - you'll have to convert the notes to frequencies. You might want to make a "cheat sheet" to help you convert the notes more quickly.

Working from sheet music
Although it's beyond the scope of this book to explain musical notation in any detail, you might find the following tips helpful as you add music to your own programs. Take a look at Figure 12-18, which shows the names of the notes and their positions on the musical staff.

If you 're working from a piece of sheet music, you can match the positions of its notes to the notes in Figure 12-18 and look up the appropriate frequency values in Figure 12-16 on page 400. Then you can write beside each note on your sheet music the frequency values you need to use with the SOUND statement.

To come up with duration values for your SOUND statements- that is, how long you want each note to last-simply decide how long you want a whole note to be and then divide that value by 2 for half notes, by 4 for quarter notes, and so on. After you've established these values, you can go through your sheet music again and jot down a duration value beside each note.

After you've determined the frequency values and duration values for each note, you're ready to add a melody to your program.

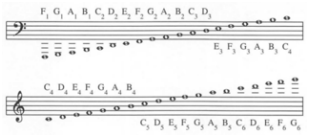


Figure 12-18.
The names of the notes on the musical staff

Practice:
Playing a song with AC/QuickBasic
Figure 12-19 shows the music for the song "My Bonnie Lies Over the Ocean." We calculated the frequency value and duration value for each note and then wrote a short program to play the song.

1. Load and compile the My Bonnie program (Figure 12-20 on the next page) from disk.

2. Run the program, and you will hear the AC/QuickBasic Interpreter/Compiler play the song.
   Notice that the frequency values and the duration values alternate in the DATA statements.
   Inside the FOR loop, the READ statement reads two values-one for the frequency and one for the duration-each time through the loop.

Experiment with the SOUND statement to see what kind of control you have over the values in the DATA statements.

Try
multiplying note% by 2 to raise the notes one octave, for instance, and try dividing duration% by 2 to see the effect on the tempo of the song.

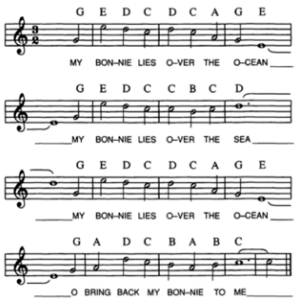


FIGURE 12-19.

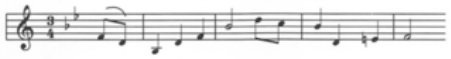
Music for "My Bonnie Lies Over the Ocean."

```
' My Bonnie
' This program demonstrates how to play a song in AC/QuickBasic.

CLS

INPUT "Press Return to begin…", dummy$

FOR i% = 1 TO 34
    READ note%, duration%
    SOUND note%, duration%
NEXT i%

'    My        bon-      nie       lies      o-        ver       the
DATA 392, 8,   659, 8,   587, 8,   523, 8,   587, 8,   523, 8,   440, 8
'    o-        cean,     My        bon-      nie       lies
DATA 392, 8,   330, 32,  392, 8,   659, 8,   587, 8,   523, 8
'    o-        ver       the       sea;      My        bon-      nie
DATA 523, 8,   494, 8,   523, 8,   587, 40,  392, 8,   659, 8,   587, 8
'    lies      o-        ver       the       o-        cean--    O
DATA 523, 8,   587, 8,   523, 8,   440, 8,   392, 8,   330, 32,  392, 8
'    bring     back      my        bon-      nie       to        me!
DATA 440, 8,   587, 8,   523, 8,   494, 8,   440, 8,   494, 8,   523, 32
```

FIGURE 12-20.
My Bonnie: a program that plays "My Bonnie Lies Over the Ocean."

SUMMARY
The AC/QuickBasic Interpreter/Compiler contains a wide range of tools you can use to create graphics and sound - and here we've
just touched on the basics. After you've completed the exercises in the book, take some time to experiment with the programs you've seen in this chapter. Graphics and sound aren't the easiest programming feats to master, but they are certainly two of the most rewarding aspects of AC/QuickBasic programming.

QUESTIONS AND EXERCISES
1. What does the LOCATE statement do?
2. Write a short program that asks the user to enter a name and then displays the name at the left edge of the Output window and "moves" it to the right edge of the Output window.
3. What do the PSET and PRESET statements do, and how do the statements differ?
4. What is the difference between absolute and relative coordinates?
5. True or False: You can use a LINE statement to create a filled box.
6. True or False: You can use a CIRCLE statement to create a filled circle.
7. Write a program that draws a circle near the left edge of the Output window and then "moves" it to the right edge of the Output window. (Hint: Don't forget to erase the previous circle.)
8. Convert the following tune into a AC/QuickBasic program:



Debugging AC/QuickBasic Programs AC/QuickBasic For the Apple][gs NOW

Remember your first AC/QuickBasic program? You've come a long way since your Output window displayed Live long and prosper. Congratulations are definitely in order! But although you're on the road to becoming a BASIC programming whiz, the day will come when you'll find that a program you've written absolutely refuses to run. At such times you need to have confidence in your ability to diagnose and solve the programming problem. This is where debugging skills come in.

Debugging is the process of tracking down and fixing errors in a program. The ability to debug a problem program is one of the most important skills you can develop as a beginning programmer. Debugging requires you to think modularly - to analyze a program as a computer does, monitoring program execution each step of the way - in order to find the flaw that prevents the program from accomplishing its task. Fortunately you're not alone. The AC/QuickBasic Interpreter/Compiler provides several powerful debugging commands that help you examine your programs.
This chapter introduces these commands and describes how to avoid creating bugs in the first place.
You'll also debug a sample program from start to finish to get some experience with the debugging commands.

THE PRACTICE OF DEBUGGING
When you debug a program, you repeat the following steps until your program runs successfully:

1. Run the program.
2. Observe errors and trouble spots.
3. Isolate and study the statements that produced each error, using programming tools as necessary.
4. Fix the errors.

If you have a printer, you'll probably want to print out a copy of any program that needs to be debugged. Studying the code is a major part of debugging, and if you're like most people, you'll find that reading and following a printout is easier than scrutinizing a program on screen.

Two Types of Errors
The errors that can occur in a AC/QuickBasic program break down into two types: run-time errors and logic errors.

- A run-time error is a mistake that causes a program to stop unexpectedly during execution. Run-time errors occur when a syntax error or an outside event forces a program to stop while it is running. Misspelling a keyword, exceeding the bounds of an array, or attempting to open a file with a bad file name are examples of run-time errors.

- A logic error is a human error - a programming mistake that makes the program produce the wrong results. Most debugging efforts are focused on tracking down logic errors introduced by the programmer.

TRACKING VARIABLES WITH PRINT
Another helpful debugging tool is a statement you're already familiar with: the PRINT statement. By placing PRINT statements within a program, or by using them in the Command window, you can examine the contents of variables as a program executes. (You'll probably want to display the PRINT statements in a different typeface - underlined, for instance - to differentiate them from program output.) After you use PRINT statements for debugging, remember to remove them from your program.

The following practice session uses PRINT to find a logic error in a program.

Using PRINT to track a changing variable
The Incorrect Shake program (Figure 13-1 on the next page) is a simple guessing game that asks the user for the size in gallons of the largest milkshake ever made (an impressive record set in Ohio in 1988). Although the program runs without syntax errors, it doesn't operate properly: No matter what number the user enters, the program responds that the guess is too low. Can you find the logic error that causes the program to fail?

Load and compile the Incorrect Shake program from the Chapter 13 folder on disk and run it.

```
' Incorrect Shake
' This program contains a logic error.  Can you find it?

bigShake% = 174        ' size of the largest milkshake ever made

CLS

PRINT "How many gallons were in the largest milkshake ever made?"
PRINT

WHILE (guess% <> bigShake%)                  ' compare guess to largest shake
    INPUT ; "Guess:  ", guess                ' get guess from user

    ' Determine whether guess is high or low and print appropriate clue.

    IF (guess% < bigShake%) THEN
        PRINT , "too low"
    ELSEIF (guess% > bigShake%) THEN
        PRINT , "too high"
    ELSE
        PRINT
    END IF
WEND

PRINT
PRINT "Correct!  A"; bigShake%; "gallon strawberry shake was mixed";
PRINT " in Ohio in 1988."
```

```
PRINT
INPUT "Press Return to continue…", dummy$
```

FIGURE 13-1.
Incorrect Shake: a guessing - game program that contains a logic error.

```
      You'll see output similar to this:

   How many gallons were in the largest milkshake ever made?
   Guess: 100   too low
   Guess: 200   too low
   Guess: 10000 too low
   Guess: 174   too low
   Guess: Command-.
```

No matter what value is entered (including the correct value, 174), the same message appears: too low.
The program is stuck in an endless loop. The only way to terminate the program is to press Command-period(.).

Let's try to track down the error by using a PRINT statement to follow the value of the variable guess%, which changes each time the user enters a new value in the WHILE loop.

1. Place the following statements (which display the value of the guess% variable in underlined type) at the bottom of the WHILE loop, just above the WEND statement:

```
TEXTFACE 4
PRINT "DEBUG: guess%="; guess%
TEXTFACE 0
```

After you add the statements to the WHILE loop, your List window will look like this:

Listing of "Incorrect Shake"

```
 WHILE (guess% <> bigShake%)         ' compare guess to largest shake
  INPUT; "Guess: ", guess            ' get guess from user

' Determine whether guess is high or low end print appropriate clue.

 IF (guess% < bigShake%) THEN
   PRINT, "too low"
 ELSE IF (guess% > bigShake%) THEN
   PRINT, "too high"
 ELSE
   PRINT
 ENDIF

 TEXTFACE 4
 PRINT "DEBUG: guess%="; guess%
 TEXTFACE 0
WEND
```

The PRINT statement is surrounded by two TEXTFACE statements: The first changes the typeface to underline, and the second changes it back to plain text. You might find underlining useful for distinguishing the output of the program from the output of the debugging statements.

2. Compile and run the program again. In the Output window, you'll see output similar to this:

```
Incorrect Shake

      How many gallons were in the largest milkshake ever made?

   Guess: 10    too low
   DEBUG guess%= o
   Guess: 30000 too low
   DEBUG guess%= 0
   Guess: 174   too low
   DEBUG: guess% = 0
   Guess:
```

Press Command-period(.) to break. The result of the PRINT statement is interesting - it shows that for some reason the milkshake guess is not being assigned to the guess% variable. There are two explanations for this: Something is wrong with the INPUT statement, or the guess% variable has been modified somewhere along the line. Can you find the error?

If you examine the INPUT statement in the program closely, you'll discover that the problem is in the guess% variable - in the INPUT statement, the guess% variable is missing its % type - declaration character:

```
INPUT ; "Guess: ", guess
```

As a result, the AC/QuickBasic Interpreter/Compiler considers guess and guess% to be two separate variables.

3. Fix the INPUT statement in the Incorrect Shake program by adding a % symbol after guess:

```
INPUT ; "Guess: ", guess%
```


4. Run the program again to be sure that it's running properly.
   If it is, remove the debugging statements you added in Step 1.
   You'll find a corrected version of the program in the Chapter 13 folder on disk with the file name Correct Shake.

Common Programming Errors
The following logic errors pop up from time to time as the result of typing mistakes or design errors.
Be on the lookout for them!

• Incorrect type assignments. Be sure that data values aren't assigned to variables of the wrong type.
  Incorrect type assignments can slip by the AC/QuickBasic Interpreter/Compiler and cause problems later in the program.
  Double-check the assignments in each INPUT, INPUT#, and READ statement in your program.

• Variable-name confusion. Don't misspell variables or omit type-declaration characters.
  Also be on the lookout for potential confusion between local and shared variables with the same name.
  Remember: You're on your own when you use variable names-the AC/QuickBasic Interpreter/Compiler doesn't check to ensure they're correct.

• Endless loops. Beware of loops that can't end.
  Use the Command window to check the termination conditions of your loops.

• Garbled output.
  Misuse of the PRINT, PRINT USING, and LOCATE statements can produce unclear or confusing screen output.
  Be sure your programs can handle whatever data the user might enter.
  Also be careful when using different fonts and type styles.

• Array troubles.
  Don't confuse an array index number with the value stored in the array.
  And to avoid the Subscript out of range error message, don't read or write past the end of an array.

DEBUGGING A PROGRAM STEP BY STEP
To get some more experience in tracking down and fixing errors, let's debug the Dessert program, which uses an array to record your favourite desserts and the number of calories they contain.

A Bug-Free Dessert
The Dessert program prompts the user for his or her name and the number of favourite desserts that person wants to enter. The number the user enters is used to dimension a one-dimensional array of string values in which the dessert names will be stored. A subprogram then adds data to the array and totals the number of calories the user has entered. The program prints the contents of the array and displays the calorie total.

```
      When Dessert runs properly, it produces output something like this:

      Welcome to the Dessert Program!

      Please enter your name: Mike

      How many desserts would you like to enter, Mike? 2

         Dessert name: Strawberries and cream
         Calories in dessert: 125

         Dessert name: Chocolate fudge cake
         Calories in dessert: 310

      The following is a list of Mike's favorite desserts:

         Strawberries and cream
         Chocolate fudge cake

      The list contains a total of 435 calories!
```

Practice:
Running the Dessert program
But Dessert contains two logic errors. Load and compile the Dessert program (Figure 13-2) from disk and run it to find out just what's wrong.

```
' Dessert
' This program contains two logic errors.  Can you find them?

CLS
                                          ' print welcome message
PRINT "Welcome to the Dessert Program!"
PRINT                                     ' get user's name
INPUT "Please enter your name:  ", userName$
PRINT                                     ' get number of desserts
PRINT "How many desserts would you like to enter, "; userName$;
INPUT num%
PRINT

DIM desserts$(num%)                       ' dimension string array

CALL GetData (desserts$(), num%, totalCalories%)  ' call subprogram
```

```
PRINT "The following is a list of "; userName$; "'s favourite desserts:"
PRINT

FOR i% = 1 TO num%
    PRINT "   "; desserts$(i%)       ' print array contents
NEXT i%

PRINT                                 ' print total number of calories
PRINT "The list contains a total of";
TEXTFACE 1
PRINT totalCalories%;
TEXTFACE 0
PRINT "calories!"

PRINT
INPUT "Press Return to continue…", dummy$

END

SUB GetData (array$(), num%, totalCalories%) STATIC

' The GetData subprogram gets dessert information from the user.

FOR i% = 1 TO num%     ' loop num% times (passed from main program)
    INPUT "   Dessert name:  ", array$(num%)      ' get dessert name
    INPUT "   Calories in dessert:  ", calories%   ' get calories
    PRINT
    totalCalories% = calories%                    ' keep running total
NEXT i%

END SUB       ' return array$ and totalCalories% to main program
```

FIGURE 13-2.
Dessert: a program that contains two logic errors.

```
      You'll see output similar to this:

      Welcome to the Dessert Program!

      Please enter your name: Mike

      How many desserts would you like to enter, Mike? 2

         Dessert name: Strawberries and cream
         Calories in dessert: 125

         Dessert name: Chocolate fudge cake
         Calories in dessert: 310

      The following is a list of Mike's favorite desserts:

         Chocolate fudge cake

      The list contains a total of 310 calories!
```

Note the two obvious problems: The program lists only one of the values stored in the desserts$ array, and the program doesn't display the correct calorie total. With these two observations in mind, you can begin to debug the program step by step. Before you start, we recommend that you resize the List, Output, and Command windows so that each one is visible on the screen without overlapping another. This makes it easier to examine the relationship between your program and the output on the screen.
We also recommend that you select the Trace All command from the Run menu to "box" each statement after it is executed and slow down execution speed a little.

Debugging the Dessert Program
The introductory part of the Dessert program seems to be OK - it displays helpful information on the screen and obtains data from the user. Let's move past these statements by setting a breakpoint near the middle of the program and executing to the breakpoint.

Setting a breakpoint
To set a breakpoint and then run Dessert to the breakpoint:

1. Move the insertion point to anywhere within the 14th line of the program (the line containing the DIM statement), and select the Breakpoint On/Off command from the Run menu (Command-B).

2. Choose Run Program from the Run menu (Command-R) to run Dessert to the breakpoint. Enter your own name at the first input prompt and the number 2 at the second input prompt. The program will stop when it reaches the breakpoint.

3. Choose the Breakpoint On/Off command (Command-B) to clear the breakpoint-it's no longer needed.

NOTE: If the Breakpoint On/Off command is greyed (unavailable), make certain that the List window is active. Breakpoint On/Off works only in the List window.

Stepping through the GetData subprogram
Now that we're in bugland, we'll move through the code at a slower pace. Keep your eyes peeled-something that made sense to the person who wrote the GetData subprogram isn't working now.

1. Choose the Step command (Command-T) from the Run menu to execute the DIM statement that dimensions the desserts$ array. desserts$ should be large enough to hold all the dessert names the user types in- a number we know in advance and have stored in the num% variable. Note that num% appears in the DIM statement and that the DIM statement executes without error-we seem to be OK so far.

2. Choose the Step command (Command-T) six times to get to the GetData subprogram and execute the first INPUT statement in it.

Enter Cherry pie at the input prompt.

3. Maximize the List window by clicking in the box in its upper right corner.
   Take a long look at the contents of the GetData subprogram to see whether anything looks out of order - sometimes a bug will jump out at you. An excellent way to examine the input process up to this point is to enter a debug PRINT statement in the Command window. Let's do this now to see if the first dessert name is stored properly in the first element of array$. Return the List window to its normal size and activate the Command window. Enter the following line:

```
TEXTFACE 4: PRINT "DEBUG: array$(1) = "; array$(1): TEXTFACE 0
```

The colon lets you put more than one statement on a line. This is especially useful in the Command window.
When you press Return, you see the following debugging information in underlined type in the Output window:

```
DEBUG: array$(1) =
```

array$(1), the first element in the array$ array, should contain the string Cherry pie - we just typed it in!
But it doesn't, so thing must be wrong with the INPUT statement; specifically, something must be wrong with the index element of the array assignment-the index element should evaluate as 1 the first time through the loop.

Isolating the first logic error
We've found the symptom of our first logic error. Let's isolate its cause by taking a close look at the INPUT statement and its role in the FOR loop. A basic rule of thumb for array indexes and loops is that if the loop contains a counter variable and each element of the array needs to be accessed, the counter variable (or some derivative of it) should be used as the array index. The GetData subprogram has everything right but the array index:

The INPUT statement should be using the counter variable i% for the array index, not the num% variable.

Fix the error by clicking on the List window and changing the INPUT statement to read as follows:

```
INPUT " Dessert name: ", array$(i%)
```

Testing the bug fix
You've corrected the logic error; now let's test the program again:

1. Run the program again. Enter your name and the number 2 for the number of desserts you want to enter.
   Enter Cherry pie for the first dessert, press Return, and halt execution by choosing Stop from the File menu.

2. To verify that Cherry pie is now stored in the first array element of array$, activate the Command window and type the following debug statement:

```
TEXTFACE 4: PRINT "DEBUG: array$(1) = "; array$(1): TEXTFACE 0
```

This time the correct string value, Cherry pie, should appear in the Output window. Activate the List window again.

3. Continue your testing of the FOR loop by moving the insertion point down to anywhere within the line containing the END SUB statement-it's the last line in the subprogram. Now choose the Breakpoint On/Off command (Command-B) to set a breakpoint on this line, and then choose the Continue command (Command-G) to execute the program up to the breakpoint. As the loop executes, you'll be prompted to enter dessert data.

4. Looks as if our bug fix did the trick-the program looped and stopped at the breakpoint without error.
   Choose the Breakpoint On/Off command (Command-B) to remove the breakpoint.

Searching for the second bug
One down, one to go. Let's take it slow and search for the last bug in the program:

1. Choose the Step command (Command-T) once to return to the main program, and then choose Step ten more times to display a few PRINT statements and the contents of the desserts$ array, which have been passed back to the main program by the GetData subprogram. The correct array elements are all there!

2. Choose the Step command (Command-T) eight more times to execute the statements that display the final calorie count in the Output window and end the program. Look at the Output window for the results. The calorie count number is completely wrong. Let's use a debug PRINT statement to take a look at the totalCalories% variable.

3. Move to the Command window and enter the following debug PRINT statement (all on a single line):

```
TEXTFACE 4: PRINT "DEBUG: total Calories% =": total Calories%:
TEXTFACE 0
```

The PRINT statement displays the last calorie count you entered!
What's the deal here? Hasn't the GetData subprogram been keeping a running total of the calorie counts?

Isolating the second logic error
Let's return to the GetData subprogram to see if we can find the source of the error. An easy first thing to check is spelling-perhaps a variable name has been misspelled in the subprogram. Return to the List window, and then use the Find command (Command-F) from the Search menu to search for the totalCalories% variable. (The Find command is particularly useful when you're working with large programs.) The Find command locates the totalCalories% variable in the parameter list of the GetData subprogram. Select the Find Next command (Command-N) from the Search menu several times to find all occurrences of totalCalories% in the program. It appears five times- three times in GetData and two times in the main program.

Our problem is not a spelling error; these variable names are all spelled the same way. Let's check the statement in the GetData subprogram that keeps a running total of the calorie values the user enters. Scroll to the GetData subprogram near the end of the program. Examine the assignment statement at the bottom of the FOR loop:

```
total Calories% = calories%
```

Does this statement properly update the totalCalories% variable to keep a running total?

Fixing the second logic error
The answer is No! As we've seen throughout this book, in order to update a running total you need to add the current total and the next value entered or read and then assign the result to the running total. To fix this assignment statement, change it to read:

```
total Calories% = total Calories% + calories%
```

After you've made this change, the program is ready to roll. Test the debugged program by running it from start to finish until you're sure it is completely bug free. When you've finished, choose Trace All from the Run menu to return execution speed to normal. If you'd like to double-check your work, check the version of Dessert you've just corrected against the error-free version of the program, Dessert 2, on disk. Dessert 2 contains the corrections you've made to Dessert.

Avoiding Bugs
Now that you've learned how to track and fix bugs, here are some hints that will help you write bug-free programs in the future.

Plan your program carefully
Before you start to type, be sure you understand what you want your program to do and how you plan to write the program in AC/QuickBasic. Think about the algorithms your program will use, how program input and output will be organized, and how data will be stored and manipulated. Start simply, and add complexity as you go.

Work one step at a time
Don't try to write your program all at once. Create and test your program one piece at a time. Isolate different tasks whenever you can by creating subprograms and functions.

Run your program often
When you make a change to your program, compile and re-run the program to ensure that it still works.
By following this rule, you can catch simple programming mistakes early on-before they compound themselves and become major programming problems.

Try out new ideas in the Command window
Use the Command window to test small pieces of program code before you put them in the program.

Test your program each step of the way, and know how your program will respond to any type of input.
Consider the following questions:
• What is the largest number or string this program can handle?
• What is the smallest?
• What will cause this program to "crash"?
• How can I prevent a user from crashing this program?
• Will the user understand what this program does?

SUMMARY
In this chapter you encountered a number of debugging tools, tips, and techniques. Debugging tools have come a long way in a few short years, and we're lucky to have such a powerful collection of them right here in the AC/QuickBasic Interpreter/Compiler. But as a programmer learning how to debug your own programs (and, worst of all, other people's programs), you need more than debugging tools to help you. Solutions come from thinking logically and creatively about your programs and examining program execution step by step. The more you know about AC/QuickBasic and its syntax rules, the better you'll be at detecting program errors and finding solutions for them.

QUESTIONS AND EXERCISES
1. What steps should you follow to isolate a bug in your program?
2. What is the difference between a run-time error and a logic error?
3. When is it useful to set a breakpoint in your program? How do you set a breakpoint?
4. What does the Command-G shortcut key do?
5. What is the purpose of the Trace All menu command?
6. What type of logic error do you find most difficult to track down and fix?
7. The following program (Incorrect Bear) contains two syntax errors. Load the program from the Chapter 13 folder on disk and fix the errors.


```
' Incorrect Bear
' This program contains two syntax errors.  Can you find them?

CLS

DIMM bears$(5)                          ' dimension string array  *** This is the BUG,  should be DIM bears$(5) ***

PRINT "Enter the names of your five favourite bears."
PRINT

FOR i% = 1 TO 5                         ' get 5 strings
    INPUT "Bear:  ", bears$(i%)
NET i%                                  ' *** This is the BUG,  should be NEXT i% ***

PRINT
PRINT "You entered the following bears:"
PRINT

FOR i% = 1 TO 5                         ' print 5 strings
     PRINT bears$(i%)
NEXT i%

PRINT
INPUT "Press Return to continue…", dummy$
```

8. The following program (Incorrect Name) contains two logic errors.
   Load the program from the Chapter 13 folder on disk, and use the debugging tools discussed in this chapter to find and fix them.

```
' Incorrect Name
' This program separates first and last names and prints them.
'   Can you find the two logic errors?

CLS

PRINT "Enter your first and last names in the following format:  ";
PRINT "Last name, First name"
PRINT

INPUT "Name:  ", fullName$

commaLocation% = INSTR(1, fullName$, ",")

IF (commaLocation% < 0) THEN
    lastName$ = LEFT$(fullName$, commaLocation% - 1)
    firstName$ = RIGHT$(fullName$, LEN(fullName$) - commaLocation% - 1)

    PRINT
    PRINT "What a lovely name!  It's so nice to meet you, ";
    PRINT firstName$; " "; lastName$; "!"
ELSE
    PRINT
    PRINT "Name not in Last name, First name format."
END IF

PRINT
INPUT "Press Return to continue…", dummy$
```

Solutions to Questions and Exercises

This appendix contains answers to the questions and exercises at the ends of Chapters 2 through 12.
The programs listed in this appendix are in the Appendix B folder on disk.

CHAPTER 2

1. Double-click on the AC/QuickBasic program icon.
2. False.
3. The List window contains your program.
   The Command window lets you test a program line before you use it in your program.
4. The New command clears the List window and prepares the AC/QuickBasic Interpreter/Compiler for a new program.
   The Open command takes you to the Open dialog box so that you can load an existing program from disk.
5. The List window displays the program you are working on.
   The Output window displays the output of the program when you run the program.
6. The Cut command deletes a selected block of text from the List window and places it in the Clipboard.
   The Copy command places a copy of the selected block of text in the Clipboard without deleting the original
   from the List window.
7. Text in the List window can be selected in the following ways:
   - Double-click on a word (selects one word only)
   - Triple-click on a line (selects one line only)
   - Drag across the text to be selected (no limit)
8. The Save command saves a changed version of a previously saved file under the same file name on disk.
   The Save As command gives you the opportunity to give the file a different file name and, if you want to,
   to save the file on another disk or in another directory.
   Save As also gives you the opportunity to save the file in a different format.
9. Choose the window you want from the Windows menu, or, if a portion of the window is visible on the screen,
   click in it.
10. Choose Quit from the File menu. If you've created a new program or have made changes to a previously saved
    program, a dialog box appears so that you have an opportunity to save the changes, discard the changes,
    or cancel the Quit operation.

CHAPTER 3

1. A statement operates in a straightforward manner, usually producing results that are obvious or tangible
   (a display of characters on the screen, for example). A function generally does its work behind the scenes
   and returns a value that can be used as an argument to a statement.

2. BEEP - statement
   CLS   - statement
   DATE$ - function
   PRINT - statement
   TIME$ - function

3. An item in square brackets is optional.
   The | character means that you can choose only one of the values in the brackets.

4. An argument is a piece of information supplied to a statement or function.

5. A string is a collection of characters (that can include letters, numbers, and symbols) surrounded by double
   quotation marks. A numeric expression is a number or a numeric variable (or any expression that produces a
   numeric result, as discussed in Chapter 4).

6. The output of the two PRINT statements is the same.

7. A semicolon or comma at the end of a PRINT statement causes output from the next PRINT statement to be displayed
   on the same line. A semicolon causes the next PRINT statement's output to appear immediately after the output
   from the current PRINT statement.
   A comma causes the next PRINT statement's output to appear in the next print zone.

CHAPTER 4

1. Regular integer, long integer, single-precision floating-point, and double-precision floating-point.
   They differ in the types and sizes of numbers they can hold and in their type-declaration characters.
   A regular integer variable (%) can hold a whole number from -32,768 through 32,767.
   A long integer variable (&) can hold a whole number from -2,147,483,648 through 2,147,483,647.
   A single-precision floating-point variable (!) can hold a number up through 7 digits in length.
   A double-precision floating-point variable (&) can hold a number up through 15 digits in length.
   In single-precision and double-precision floating-point numbers,
   The decimal point can appear anywhere within the number.

2. To reserve a space for a minus sign if the number had been or could become a negative value.

3. An invalid comma likely appeared within a number.

4. A number outside the range of a variable's numeric data type was assigned to the variable.

5. a. Regular integer
   b. Single-precision floating-point
   c. Single-precision floating-point
   d. Double-precision floating-point.
   e. Regular integer or single-precision floating-point
   f. Single-precision floating-point
   g. Long integer
   h. Double-precision floating-point
   i. Single-precision floating-point

6. Regular division can return a fractional result;
   Integer division returns only an integer, discarding any remainder; remainder division returns only a remainder.

7. Exponentiation(^); multiplication and division(*, /,\, MOD); addition and subtraction (+, -).

8. 300.

9. One possible solution to this problem is the Calculations program:

```
' Calculations
' This program calculates and prints four formulas.

CLS

PRINT "ABS(-10) + 5 ="; ABS(-10) + 5
PRINT "SQR(36) ="; SQR(36)
PRINT "SQR(4) ^ 2 ="; SQR(4) ^ 2
PRINT "COS(3.14) ="; COS(3.141592654#)
```

10. One possible solution to this problem is the Circle Calc program:

```
' Circle Calc
' This program displays the circumference of a circle when the radius is supplied by the user.

pi# = 3.141592654#                          ' use a variable

CLS                                         ' clear screen

PRINT "This program calculates the circumference of a circle ";
PRINT "from its radius."
PRINT                                       ' get input from user
INPUT "Enter the radius of the circle:  ", radius!

circum! = 2 * pi# * radius!                 ' calculate circumference

PRINT                                       ' print result
PRINT "The circumference of the circle is"; circum!
```

CHAPTER 5

1. A numeric expression uses mathematical operators and yields a numeric result.
    A conditional expression uses conditional operators and yields a true or false result.

2. b, c, f, i, j, k.

3. True.

4. The AND logical operator indicates that both conditions must be true before an action will occur.
   The OR logical operator indicates that only one condition must be true before an action will occur.

5. The ELSE keyword lets you execute a block of statements
   When a conditional expression in an IF statement evaluates as false.

   ELSE is the opposite of THEN.

6. The ELSEIF keyword lets you evaluate another condition
   After a previous IF or ELSEIF statement has evaluated as false.
   The THEN keyword must appear on the same line as ELSEIF.

7. One possible solution to this problem is the Question program:

```
' Question
' This program asks the user a question about programming in BASIC.

CLS

INPUT "Do you like programming in BASIC so far (Y/N)?  ", reply$
PRINT

IF (reply$ = "Y") OR (reply$ = "y") THEN
    PRINT "Great!  There's more fun to come!"
ELSEIF (reply$ = "N") OR (reply$ = "n") THEN
    PRINT "Sorry to hear that.  Don't worry--it gets better!"
ELSE
    PRINT "Please run the program again."
    PRINT "Enter 'Y' for Yes or 'N' for No at the prompt."
END IF
```

CHAPTER 6

1. The counter variable identifies the value of the loop-an integer between the start and end limits of the loop.

2. start and end can be numeric constants, numeric variables, or numeric expressions.
   The values can be positive or negative.

3. 72.

4. The SOUND statement causes your computer's speaker to emit a tone of the specified frequency and duration.

5. A nested loop is a FOR or WHILE loop inside another FOR or WHILE loop.

6. An infinite loop is a loop that cycles endlessly.
   A WHILE loop is considered infinite if its logical condition is never met;
   A FOR loop is considered infinite if actions within the loop prevent the counter variable
   from reaching the value of end.
   You stop an endless loop by holding down the Command key and pressing the period key.

7. Use a FOR loop when you want to execute a block of statements a specific number of times.
   Use a WHILE loop to execute a block of statements based on the value of a condition.

8. One possible solution to this problem is the Gasoline Expenses program.

```
' Gasoline Expenses
' This program uses a FOR loop to track weekly gasoline expenses.

CLS

PRINT "For each of the seven days of the week, enter the amount ";
PRINT "you spent on gasoline."
PRINT

FOR day% = 1 TO 7
    PRINT "  Cash spent on day"; day%;
    INPUT "-->  $", dayTotal!
    weekTotal! = weekTotal! + dayTotal!
NEXT day%

PRINT
PRINT "Wow!  $"; weekTotal!; "on gas in one week!"
PRINT
PRINT "Press Return to continue…", dummy$
```

9. One possible solution to this problem is the Sounds program:

```
' Sounds
' This program plays a note based on frequency and duration values
'   entered by the user.

CLS

PRINT "Enter frequency and duration values for the sound you want"
PRINT "to hear.  To quit, enter -999 for frequency."
PRINT

WHILE frequency% <> -999
    INPUT "Frequency (12-32767):  ", frequency%
    IF (frequency% <> -999) THEN
        INPUT "Duration (0-77):  ", duration&
        SOUND frequency%, duration&
        PRINT
    END IF
WEND
```

10. One possible solution to this problem is the Dice Roller program:

```
' Dice Roller
' This program rolls one simulated die 10 times and displays the
'   message "Nice Roll!" if the die shows 6.

CLS

INPUT "Press Return to roll the die 10 times.  Think six…", dummy$
PRINT

RANDOMIZE TIMER

FOR i% = 1 TO 10
    roll% = INT(RND * 6) + 1
    PRINT "Roll: "; roll%,
    IF (roll% = 6) THEN
        PRINT "Nice Roll!"
    ELSE
        PRINT
    END IF
NEXT i%

PRINT
INPUT "Press Return to continue…", dummy$
```


CHAPTER 7
1. All are advantages.

2. The EnterName$ subprogram name contains a string type-declaration character ($).
   This is incorrect - only functions are marked with the type of data they will return.
   The correct SUB would be as follows:

```
SUB EnterName (firstName$, lastName$) STATIC
```

3. Subprograms are located at the bottom of the main program, following its last statement.

4. No, subprogram variables are local to the subprogram in which they are declared
   Unless they are shared by means of the SHARED statement or passed as subprogram parameters.

5. SHARED userName$
   A SHARED statement is located at the top of a subprogram.

6. A subprogram is a multiline block of code that can be called one or more times in a program.
   Subprograms are general purpose in nature and can be used to perform such tasks as setting up the screen,
   initializing variables, getting input from the user, processing data, and handling output.

   A function is a single-line program module that can also be called one or more times in a program.
   Functions are designed to carry out smaller tasks than subprograms do (calculations, usually)
   and return single values to the main program or calling subprogram.

7. The GetCarFacts subprogram can be written as follows:

```
SUB GetCarFacts (make$, model$, year%, paint$) STATIC

PRINT "Please enter information about your car."
PRINT
INPUT" Car make: ", make$
INPUT " Car model: ", model$
INPUT " Model year: ", year%
INPUT" Carcolor: ",paint$

END SUB
```

A statement that calls GetCarFacts can be written as follows:

```
CALL GetCarFacts (carMake$, carModel$, carYear%, carColor$)
```

8. DEF FNPythagorean!(a!, b!) = SQR(a! ^ 2 + b! ^ 2)

   A statement that calls FNPythagorean! can be written as follows:

```
   PRINT "c ="; FNPythagorean!(side1!, side2!)
```

CHAPTER 8
1. READ statements come first by convention.

2. DATA, READ, and RESTORE work best for data in the following categories:

o Data you know about in advance (before the program is run)
o Data that always appears in the same order
o Data that can be cycled through repeatedly, such as days of the week

3. One possible solution to this problem is the Beav DATA program:

```
' Beav DATA
' This program reads the names of Beaver Cleaver and his friends
'   from a DATA statement.

CLS

FOR i% = 1 TO 7
    READ pal$
    PRINT pal$
NEXT i%

PRINT
INPUT "Press Return to continue…", dummy$

DATA Beaver, Wally, Lumpy, Whitey, Gus, Eddie, Larry
```

4. False.

5. DIM numbers!(99)

6. An end-of-data marker is a number or string indicating that no more data items exist in a list.

7. One possible solution to this problem is the Banzai Array program:

```
' Banzai Array
' This program uses a one-dimensional string array to store names
'   of characters from the movie "The Adventures of Buckaroo Banzai."

OPTION BASE 1               ' set array base at 1

CLS

PRINT "** This program collects character names from the film ";
PRINT "'Buckaroo Banzai' **"
PRINT
INPUT "How many names would you like to enter?  ", names%

DIM characters$(names%)   ' dimension array

PRINT                               ' fill dynamic array
FOR i% = 1 TO names%
    INPUT "  Name:  ", characters$(i%)
NEXT i%

PRINT
PRINT "You entered the following names:"
PRINT

FOR i% = 1 TO names%         ' print contents of array
    PRINT characters$(i%)
NEXT i%
```

8. An out-of-range error is a program's attempt to reference an element that does not exist in an array.

9. One possible solution to the problem is the 2-D Baseball program:

```
' 2-D Baseball
' This program keeps score for a nine-inning baseball game with a
'   two-dimensional array named scoreboard%.

OPTION BASE 1            ' set first array element at 1
DIM scoreboard%(2, 9)    ' dimension 2x9 array for baseball scoreboard

' Get team and mascot names.

visitor$ = "Boston"
visitorMascot$ = "Red Sox"
home$ = "Seattle"
homeMascot$ = "Mariners"

CLS

TEXTFONT 4                  ' set font to Monaco for alignment purposes

PRINT "Enter runs scored by each team in a nine-inning baseball game."
PRINT

FOR inning% = 1 TO 9    ' get number of runs scored in each inning
    PRINT "Inning"; inning%; "-->    "; visitor$;
    INPUT ; ":  ", scoreboard%(1, inning%)
    PRINT "    "; home$;
    INPUT ":  ", scoreboard%(2, inning%)
                                    ' …and keep running total for each team
    visitorScore% = visitorScore% + scoreboard%(1, inning%)
    homeScore% = homeScore% + scoreboard%(2, inning%)
NEXT inning%

' Determine the winner of the game and display results.

PRINT

IF (visitorScore% > homeScore%) THEN
    PRINT "News Flash:  "; visitorMascot$; " beat "; homeMascot$;
    PRINT visitorScore%; "to"; homeScore%
ELSEIF (homeScore% > visitorScore%) THEN
    PRINT "News Flash:  "; homeMascot$; " beat "; visitorMascot$;
    PRINT homeScore%; "to"; visitorScore%
ELSE
    PRINT "News Flash:  "; visitorMascot$; " tie "; homeMascot$;
    PRINT visitorScore%; "to"; homeScore%
END IF

' Display the final scoreboard.

PRINT
PRINT "Inning         1    2    3    4    5    6    7    8    9"
```



```
PRINT "-------------------------------------------------------"

FOR team% = 1 TO 2         ' for each team in the game
    IF (team% = 1) THEN PRINT visitor$, ELSE PRINT home$,
    FOR inning% = 1 TO 9   ' …and for each inning in the game…
        PRINT scoreboard%(team%, inning%); "  ";
    NEXT inning%                ' print the number of runs scored
    PRINT
NEXT team%

PRINT
INPUT "Press Return to continue…", dummy$
TEXTFONT 1                      ' restore default application font
```

CHAPTER 9

1. True.

2. True. The array elements are numbered 0 through 9.

3. ONETWOTHREE

4. a, b, c.

5. The value 0 from INSTR means one of the following:
   searchstring was not found in basestring.
   start is greater than the length of basestring.
   basestring contains no characters.

6. H

7. 77.

8. One possible solution to this problem is the Get Names program:

```
' Get Names
' This program gets first and last names from the user and displays
'   them in uppercase.

CLS

INPUT "First name:  ", firstName$
INPUT "Last name:   ", lastName$
PRINT
PRINT UCASE$(lastName$); ", "; UCASE$(firstName$)
PRINT
INPUT "Press Return to continue…", dummy$
```

9. One possible solution to this problem is the Reverse String program:

```
' Reverse String
' This program reverses the order of the characters in a string.

CLS

' Get string from user.

INPUT "Enter a string of characters to be reversed:  ", inString$
numOfChars% = LEN(inString$)              ' find length of string

FOR i% = numOfChars% TO 1 STEP -1   ' step backwards through string
    tempChar$ = MID$(inString$, i%, 1)   ' extract one letter at a time
    reverse$ = reverse$ + tempChar$       ' build new string
NEXT i%

PRINT                                                 ' display new string
PRINT "The characters in reverse order are "; reverse$
PRINT
INPUT "Press Return to continue…", dummy$
```

CHAPTER 10

1. OUTPUT deletes the contents of an existing file; APPEND adds to the end of an existing file.

2. d.

3. False.

4. LINE INPUT# is more useful than INPUT# when you need to read long lines of string data or lines containing commas from a file.

5. The FILES keyword is missing the$ type identifier. The correct statement would be

```
filename$= FILES$(1, "TEXT")
```

6. One possible solution to this problem is the City File program:

```
' City File
' This program stores a list of cities in a sequential file.

OPEN "City Data" FOR OUTPUT AS #1      ' open file in QBI folder

CLS

PRINT "This program stores city names on disk in a file named 'City Data.'"
PRINT "Enter your favorite cities and type END to quit."
PRINT

WHILE (city$ <> "END")                  ' until user enters END
    INPUT "  City name:  ", city$     ' get names from user
    IF (city$ <> "END") THEN PRINT #1, city$   ' write names to file
WEND

CLOSE #1                                    ' close file

PRINT
INPUT "Press Return to see the cities you entered…", dummy$
PRINT

OPEN "City Data" FOR INPUT AS #1       ' open file for input

WHILE (NOT EOF(1))                   ' until end of file is reached
    INPUT #1, city$                    ' get names
    PRINT city$                          ' print names
WEND

CLOSE #1                                  ' close file

PRINT
INPUT "Press Return to continue…", dummy$
```

7. One possible solution to this problem is the Sort List program:

```
' Sort List
' This program gets names and addresses from the user, sorts
'   them alphabetically by last name, and stores them in the
'   sequential file "Name Data."

OPTION BASE 1                 ' set base of arrays to 1

CLS                                   ' clear screen

CALL AddNamesToFile       ' call sub to get input from user

OPEN "Name Data" FOR INPUT AS #1   ' open for input to read file data
numOfNames% = 0              ' counter tracks number of name-and-
                                        '   address items
WHILE (NOT EOF(1))         ' until end of file is reached,
    LINE INPUT #1, fullName$                ' read name and address
    LINE INPUT #1, address$
    numOfNames% = numOfNames% + 1      ' increment item counter
WEND
CLOSE #1

DIM names$(numOfNames%)             ' dimension array to hold names
DIM addresses$(numOfNames%)       ' dimension array to hold addresses

' Copy names and addresses in file to arrays in preparation for sorting.
CALL CopyFileToArrays (names$(), addresses$(), numOfNames%)

' Sort names$ and addresses$ arrays alphabetically by last name.
CALL ShellSort (names$(), addresses$(), numOfNames%)

' Copy sorted arrays back to file.
CALL CopyArraysToFile (names$(), addresses$(), numOfNames%)

' Display new file on screen.
CALL DisplayNewFile

PRINT
INPUT "Press Return to continue…", dummy$
END

SUB AddNamesToFile STATIC
```



```
' Open file for output.
OPEN "Name Data" FOR OUTPUT AS #1

PRINT "This program adds names and addresses to the file 'Name Data'"
PRINT "and then sorts the file alphabetically."
PRINT
PRINT "Enter names in Lastname, Firstname format.  Type END to quit."
PRINT

WHILE (fullName$ <> "END")           ' get names until user enters END
    LINE INPUT "  Name (Last, First):  "; fullName$
    IF (fullName$ <> "END") THEN     ' use LINE INPUT to allow commas
        PRINT #1, fullName$              ' write data to file
        LINE INPUT "  Address:  "; address$
        PRINT #1, address$
    END IF
    PRINT
WEND

CLOSE #1

END SUB

SUB CopyArraysToFile (names$(), addresses$(), numOfItems%) STATIC

' Open for output to overwrite out-of-order entries.
OPEN "Name Data" FOR OUTPUT AS #1

FOR i% = 1 TO numOfItems%
    PRINT #1, names$(i%)             ' write array contents to file
    PRINT #1, addresses$(i%)
NEXT i%

CLOSE #1

END SUB

SUB CopyFileToArrays (names$(), addresses$(), numOfItems%) STATIC

OPEN "Name Data" FOR INPUT AS #1  ' open for input to get file data

FOR i% = 1 TO numOfItems%
    LINE INPUT #1, names$(i%)             ' read file contents
    LINE INPUT #1, addresses$(i%)
NEXT i%

CLOSE #1

END SUB

SUB DisplayNewFile STATIC

INPUT "Press Return to view Name Data…", dummy$
PRINT

OPEN "Name Data" FOR INPUT AS #1   ' open for input to get file data

WHILE (NOT EOF(1))                            ' until end of file is reached,
    LINE INPUT #1, fullName$               ' read file data
    LINE INPUT #1, address$
    PRINT fullName$; " -- "; address$    ' and display on screen
WEND

CLOSE #1

END SUB

SUB ShellSort (names$(), addresses$(), numOfElements%) STATIC
```

```
' You can find a discussion of the Shell Sort in Chapter 9.
'   Note that this version sorts two arrays based on the
'   contents of names$.

span% = numOfElements% \ 2

WHILE (span% > 0)
    FOR i% = span% TO numOfElements% - 1
        j% = i% - span% + 1
        FOR j% = (i% - span% + 1) TO 1 STEP -span%
            IF names$(j%) <= names$(j% + span%) THEN
                j% = 1
            ELSE
                ' Swap array elements that are out of order.
                SWAP names$(j%), names$(j% + span%)
                SWAP addresses$(j%), addresses$(j% + span%)
            END IF
        NEXT j%
    NEXT i%

    span% = span% \ 2
WEND

END SUB
```

CHAPTER 11

1. The status argument determines the availability of a menu item.
   Menu items can be disabled (displayed in dimmed type), enabled (displayed in regular type),
   and selected (displayed with a checkmark beside them).

2. itemNumber% = MENU(1)

3. The following fragment returns the menu number and menu item selected:

```
WHILE menuNumber% = 0
    menuNumber% = MENU(0)
WEND
itemNumber% = MENU(1)
```

4. The window dimensions are specified incorrectly. The correct statement is:

```
WINDOW 1, "My Window", (5, 40)-(200, 90), 1
```

5. WINDOW CLOSE

6. BUTTON 1, 1, "OK", (50, 150)-(100, 170), 1

7. The user has pressed the Return key in the active window.

8. One possible solution to this problem is the Basic Lotto program:

```
' Basic Lotto
' A program that picks random lottery numbers.

CLS
TEXTSIZE 55
LOCATE 1, 3
PRINT "Basic Lotto"                                 ' display title

WINDOW 2, , (40, 130)-(150, 190), 2         ' draw windows
WINDOW 3, , (200, 130)-(310, 190), 2
WINDOW 4, , (360, 130)-(470, 190), 2
WINDOW 5, , (200, 225)-(310, 325), 2

BUTTON 1, 1, "Roll", (15, 15)-(95, 40)        ' draw buttons
BUTTON 2, 1, "Quit", (15, 60)-(95, 85)

WHILE DIALOG(0) <> 1                              ' wait for button click
WEND
```



```
IF DIALOG(1) = 1 THEN                               ' if button is "Roll,"
    notFinished% = 1                                    ' set flag to not finished
    WHILE notFinished%                                ' while we're not finished,
        RANDOMIZE TIMER                             ' get random seed from clock
        FOR i% = 100 TO 600 STEP 10            ' loop 50 times
            SOUND i%, .2                                ' make a clicking sound
            windowNum% = INT(RND * 3) + 2    ' get window to update
            luckyNum% = INT(RND * 10)           ' get lucky number
            WINDOW windowNum%                   ' activate window
            TEXTSIZE 50                                 ' set size and location
            LOCATE 1,1
            PRINT luckyNum%;                        ' display random number
        NEXT i%

        WINDOW 5                                        ' activate dialog box
        WHILE DIALOG(0) <> 1: WEND            ' wait for button click;
        ' if button is "Quit," then set flag to finished
        IF DIALOG(1) = 2 THEN notFinished% = 0
    WEND
END IF
```

9. One solution to this problem is the Video Database program.
   Because of its length, we have not reproduced the program here.
   You can find Video Database on disk in the Appendix B folder.

CHAPTER 12

1. LOCATE lets you position the text cursor in the Output window.

2. One possible solution to this problem is the Name Mover program:

```
' Name Mover
' This program "moves" a name across the Output window.

delay% = 400

CLS

INPUT "Please enter your first name:  ", firstName$
LOCATE 10, 1
PRINT firstName$

FOR i% = 2 TO 60
    LOCATE 10, i% - 1
    PRINT SPACE$(30)
    LOCATE 10, i%
    PRINT firstName$

    FOR j% = 1 TO delay%        ' delay loop
    NEXT j%

NEXT i%
```

3. The PSET and PRESET statements set individual pixels in the Output window at specified locations.
   By default, the PSET statement sets the pixel in black; by default, the PRESET statement sets the pixel in white.

4. Absolute coordinates are calculated using the starting point (0, 0), which is the upper left corner of the
   Output window. Relative coordinates use the last plotted point as the starting point for calculation.

5. True, provided you use the B and F options.

6. False.

7. One possible solution to this problem is the Circle Mover program:

```
' Circle Mover
' This program "moves" a circle across the Output window.

delay% = 50

CIRCLE (20, 100), 20

FOR i% = 21 TO 470
    CIRCLE (i% - 1, 100), 20, 0     ' erase previous circle
    CIRCLE (i%, 100), 20               ' draw new circle

    FOR j% = 1 TO delay%              ' delay loop
    NEXT j%

NEXT i%
```

8. One possible solution to this problem is the Anthem program:

```
' Anthem
' This program plays the opening bars of "The Star-Spangled Banner."

CLS

INPUT "Press Return to begin…", dummy$

FOR i% = 1 TO 12
    READ note%, duration%
    SOUND note%, duration%
NEXT i%

DATA 349, 4, 294, 4, 233, 8, 294, 8, 349, 8, 466, 16
DATA 587, 4, 523, 4, 466, 8, 294, 8, 330, 8, 349, 16
```

CHAPTER 13

1. a. Run your program.
   b. Observe errors and trouble spots in program execution.
   c. Using printouts, programming tools, and your knowledge of BASIC syntax,
      study the statements that produced the error.
   d. Fix the error and test the program.

2. A run-time error is a violation of BASIC syntax during the execution of a program.
   A logic error is a human design error that causes a program to produce unexpected results.

3. A breakpoint is useful when you need to jump over error - free code so that you can debug problem code at a slower pace. You can set a breakpoint by means of the Breakpoint On/Off command on the Run menu or the Breakpoint icon at the lower left corner of the List window.

4. The Command-G shortcut key executes the Continue command on the Run menu. Continue is useful when you want to pick up program execution after setting a breakpoint or using the Step command.

5. The Trace All command lets you run your program in slow motion. Program execution is slower, and each statement is "boxed" after it has been run.

6. There's no one correct answer to this question (especially with so many to choose from!).
   Our picks for the worst are logic errors resulting from incorrect arithmetic or faulty conditional expressions.

7. DIM and NEXT are misspelled. Here is the correct version (Correct Bear):

```
' Correct Bear
' This program is the corrected version of Incorrect Bear.

CLS

DIM bears$(5)                           ' dimension string array

PRINT "Enter the names of your five favorite bears."
PRINT

FOR i% = 1 TO 5                         ' get 5 strings
    INPUT "Bear:  ", bears$(i%)
NEXT i%

PRINT
PRINT "You entered the following bears:"
PRINT

FOR i% = 1 TO 5                         ' print 5 strings
     PRINT bears$(i%)
NEXT i%

PRINT
INPUT "Press Return to continue…", dummy$
```

8. The INPUT statement should be a LINE INPUT statement to handle the comma in the user's input, and the conditional expression in the IF statement should use a greater-than operator (>) instead of a less-than operator (<).
Here is the correct version

(Correct Name):

```
' Correct Name
' This program separates first and last names and prints them.

CLS

PRINT "Enter your first and last names in the following format:  ";
PRINT "Last name, First name"
PRINT
```



```
LINE INPUT "Name:  ", fullName$

commaLocation% = INSTR(1, fullName$, ",")

IF (commaLocation% > 0) THEN
    lastName$ = LEFT$(fullName$, commaLocation% - 1)
    firstName$ = RIGHT$(fullName$, LEN(fullName$) - commaLocation% - 1)

    PRINT
    PRINT "What a lovely name!  It's so nice to meet you, ";
    PRINT firstName$; " "; lastName$; "!"
ELSE
    PRINT
    PRINT "Name not in Last name, First name format."
END IF

PRINT
INPUT "Press Return to continue…", dummy$
```