Chapter 5

Strings and Table Construction

G O A L S

After working through this chapter, you will be able to:

- Manipulate strings.
- Use the string comparison operators (=, <, and >) and the **StrComp(str1, str2, n)** function.
- Use the **Ucase**, **Lcase**, and **Trim$** functions appropriately.
- Use multiline scrollable textboxes to build a small text editor.
- Work with **For-Next** statements.
- Use the **Format$** function to format display strings.
- Use nested **For-Next** loops.
- Use scrollbars as a way to enter values into a procedure.
- Use a listbox in an application.
- Use file statements to save and retrieve information from data files.
- Use the properties of points and lines to write an application that uses techniques from algebra.

O V E R V I E W

Visual Basic programs work with strings frequently. Much of the data stored in databases is stored in strings. You can write programs to gather information in the form of strings (such as students' names and favorite occupations). Then you can manipulate these strings, such as by sorting them in a specific order or by selecting only those students who like to hike. You use strings as label captions, to prompt the user for information, or to pass a message.

The second section introduces you to menus, which are a distinctive feature of Windows. With a menu bar, users can quickly see what commands are

available in the program. Typically, commands are placed in logical groups so that they are easy to find. For example, you know that the Save command is in the File menu, no matter what Windows application you are using.

In the MiniEdit project, you will create a menu bar with one item (File). You will also create the drop-down menu for that item. Through the program, you will work with opening, saving, and closing files. These are very important tasks for any Visual Basic program.

Loops are a common way to perform work in a program. Loops let you repeat a series of instructions as many times as you need. When a condition is met (such as all the calculations being complete), the program moves to the next line of code outside the loop. ***For-Next*** *statements are one of the most common types of loops in Visual Basic.*

You can add scrollbars to a textbox on a form through the Properties window or by placing scrollbar objects on the form. In this chapter, you will learn the difference between the two types of scrollbars as well as how to use the scrollbar control.

Strings

Strings are series of characters. These characters may include digits, letters, punctuation, and special characters, including:

Digits = 0,1,2,3,4,5,6,7,8,9
Letters = a,b,...,z,A,B,...,Z
Punctuation = . ? ! ; : , " '
Special characters = ~ @ # $. + | \

When you think of strings, you probably think of text. You may find it surprising that social security numbers, telephone numbers, times, and dates are all usually treated as strings. Can you see the difference between numbers that you multiply, divide, and add, and these "numbers"? Look at the examples below:

331-50-5440, social security number
(708)555-7561, phone number
162-99-2421, driver's license number

These numbers have a specific structure. A social security number, for example, always has three numbers, then two numbers, then four numbers. These numbers also have specific punctuation. The groupings

of numbers in social security numbers are always divided by hyphens. The punctuation in phone numbers is admittedly somewhat more variable. Sometimes you see the area code set off by parentheses, but sometimes people just use two hyphens.

Although these numbers consist of digits and (sometimes optional) punctuation, they are best thought of as names—strings that uniquely identify someone or something. Notice that they lack almost all the properties of actual numbers. For example, it makes no sense at all to add two social security numbers. Whose social security number would you have by performing this addition? Furthermore, the result of the addition may simply have too many digits to be a valid social security number. Similarly, it is meaningless to multiply a phone number by 2.

To work with strings, you need to understand their characteristics. Strings:

- Have a length. The number of spaces and characters in a string is its length.
- Can be joined together.
- Can be searched. How often have you searched for a word in a text file, then replaced it with something else?
- Can be displayed on the screen.
- Can be read from controls on the screen. Users enter strings in textboxes. The program can then read the strings from those controls. Figure 5-1 shows a name being entered into a textbox.

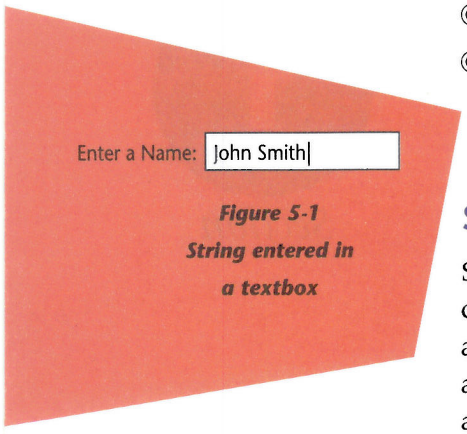


Figure 5-1
String entered in a textbox

String Literals and String Variables

Some numeric values change often, some change seldom, and some change not at all. In the same way, some string variables change values all the time, some change rarely, and some never change. When an assignment statement is used to give a value to a variable, you're probably looking at a variable whose value changes rarely. The string used in the assignment statement, a series of characters within double quotes, is called a string literal. The variable it is assigned to, used exclusively to represent string values, is called a string variable.

STRING LITERALS

You can dictate a specific string by using a string literal. String literals are series of characters between double quotation marks—for example, "Hello there" or "Please enter a positive number". Placing the period at the end of the sentence outside the closing quotation mark shows that the period is not part of the string.

You can use string literals with the **MsgBox** statement to communicate simple messages to the user, for example, or to ask a simple Yes/No question. You can also assign string literals to string variables. Both of these possibilities are illustrated by the following example:

```
Dim strAskAge As String, strName As String
strAskAge = "How old are you?"
strName = "Wendy"
MsgBox "You are standing on my foot!"
```

NOTE:

A string can be empty. The empty string is indicated by two double quotes with nothing in between (""). The length of the empty string is 0, because it contains no characters. Just as a numeric variable is sometimes initialized to 0, it is often useful to initialize a string variable to the empty string.

STRING VARIABLES

If you want to let the value of a string be determined during run-time, you use a string variable. You also use a string variable if you want to define the value of the variable during design, but let the variable change value as the program runs. The program might then change the value of the string variable by reading a string from a textbox or other control.

For example, look at Figure 5-1. You might define a string variable called *Name*, then have the program insert a text string into that variable during run-time. In this instance, the program would make the value of *Name* equal to "John Smith". Now look back at the Primes program in Chapter 4. The variable *Divisor* is a string variable.

You should declare all string variables with a **Dim** statement. For example:

```
Dim strName As String, strUserPrompt As String
```

Joining Strings Together

Joining strings together is a powerful option you will find yourself using frequently. In Visual Basic, this is called concatenation. For example, consider these two assignment statements:

```
Desire = "I would like to learn to "
ObjectOfDesire = "skydive"
```

You can join these variables together by using the ampersand operator:

```
Desire & ObjectOfDesire
```

The result is a concatenated string: "I would like to learn to skydive".

You need to pay attention to the spaces included within the quotation marks of a string literal. Any character within these marks is part of the string. In this case, the value of *strDesire* ends with a trailing space.

This space provides the proper spacing between the end of this string and the beginning of the string joined to it. Otherwise, in the joined string, the words "to" and "skydive" would run together. A leading space is a space at the beginning of a string.

You also need to pay attention to the order of the string variables in the concatenation. **strDesire & strObjectOfDesire** is not at all the same as **strObjectOfDesire & strDesire**. The latter string has the value: "skydiveI would like to learn to ".

You have already had some experience with joining strings together. In the Primes program (Chapter 4), you joined the divisors of a number to build a list of divisors.

NOTE:

In older versions of Basic, the string concatenation operator was the plus sign (+). Visual Basic lets you use the plus sign for joining strings, but, because the plus sign is used to add numbers, you should try to use the ampersand.

Comparing Strings

Besides joining strings, another common way to work with strings is to compare them. For example, imagine that you want to find a specific word in a file. To do so, you need to have the program compare every word in the file against the search word. If the program finds a match, an action occurs, such as a message being displayed. This is how any program performs a search-and-replace task.

Yet another common reason to compare strings is to sort them. For example, you might have a list of students in your school. To sort these names alphabetically, you would compare them to each other. Using ASCII codes, your program can determine if a name is "less than" another and move it further up the list. One string is "less than" another if it precedes it alphabetically—that is, if it would come first in the dictionary or the phone book.

For these kinds of tasks, you use four different mathematical signs: the equal sign, the greater than (>) sign, the less than (<) sign, and the not equal sign (<>). This section provides a couple of examples showing you how to use these signs to compare strings

The following code uses the equal sign in an **If-Then** statement:

```
Dim Fruit1 As String, Fruit2 As String
Dim Outcome As String
Fruit1 = "Apple"
Fruit2 = "apple"
If Fruit1 = Fruit2
   Outcome = " are equal."
Else
   Outcome = " are not equal."
End If
MsgBox "The strings" & Outcome
```

This code compares the two strings *Fruit1* and *Fruit2*, character by character. It then displays a message informing the user of the result of the comparison. If the words differ in even the slightest way, the strings are not equal. Because "Apple" is not equal to "apple", the message "The strings are not equal" will always be displayed.

This code uses less than and greater than signs to compare strings:

```
Dim Word1 As String, Word2 As String, Msg As String
Word1 = "play"
Word2 = "ball"
If Word1 > Word2 Then
   Msg = "The first word is the greater of the two."
Else If Word1 < Word2 Then
      Msg = "The first word is the lesser of the two."
   Else
      Msg = "The two words are equal."
   End If
```

Because "ball" precedes "play" alphabetically, the condition *Word1* > *Word2* is **True**, and *Word1* < *Word2* is **False**. Thus, *Msg* will be set to "The first word is the greater of the two."

Visual Basic compares strings by looking at the values, or the ASCII codes, of the characters. The lengths of the strings does not matter. In the example above, "ball" is less than "play". The ASCII code for "b" is 98, and the ASCII code for "p" is 112.

Characters have different ASCII codes, depending on their case. For example, "ball" is greater than "Play". The ASCII code for "P" is 80, compared to 98 for "b". The ASCII codes for uppercase letters range from 65 for "A" to 90 for "Z"; the ASCII codes for lowercase letters range from 97 for "a" to 122 for "z".

Now imagine that you were comparing the strings "play" and "plays". Visual Basic would find a match for the first four characters. The difference lies in the "s" in "plays". The word "play" is what is called a left substring of "plays". As a result, "play" is less than "plays".

NOTE:

There is a more powerful way to compare strings than using the =, <, >, and <> signs. See the description of the StrComp function in the next section.

Using Functions to Work with Strings

Visual Basic provides some useful tools for working with strings. These tools take the form of string functions (see the text box for a definition of functions). Two of the functions can change the case of the characters in the string. Another function can compare two strings alphabetically. The final function discussed in this section counts the number of characters in a string.

UCASE AND LCASE FUNCTIONS

Sometimes, you will want the program to ignore the case of two strings it is comparing. For example, if you are alphabetizing a list of words, you would want "labor" to be listed before "Materials". Or you may want to find *every* instance of the word "text"—including both "text" and "Text". You can use the **UCase** and **LCase** functions for this purpose.

VISUAL BASIC FUNCTIONS

A Visual Basic function is a built-in procedure that performs a specific job and returns a value that can be used in expressions. It is "built-in" in the sense that Visual Basic already contains the code that performs the task in question. These functions are shortcuts that you can use instead of writing code from scratch to perform the same task.

For example, a commonly used numeric function is the square root function, which finds the square root of a positive number. Visual Basic contains the **Sqr** function, which computes square roots so that you do not have to write pages of code to perform that computation. The argument *x* that you supply to **Sqr** can be any numeric expression with a value that is nonnegative. The following example illustrates the use of the **Sqr** function:

```
Dim dblNumber As Double, dblRoot As Double
dblRoot = Sqr(dblNumber)
```

The **UCase** function works by converting all the characters of a string to uppercase. The **LCase** function converts the characters of a string to lowercase. Here is a sample of code using the **UCase** function:

```
Dim Word1 As String, Word2 As String, Msg As String
Word1 = "hello"
Word2 = "Today"
If UCase(Word1) < UCase(Word2) Then
   Msg = Word1 & " is less than " & Word2
Else
   Msg = Word2 & " is less than " & Word1
End If
```

The **UCase** function converts any lowercase character to its uppercase equivalent. It leaves unchanged any uppercase character or nonlet-

ter. Similarly, the **LCase** function converts any uppercase character to lowercase, leaving any other character unchanged.

STRCOMP FUNCTION

You have another, more powerful option for comparing strings than using the =, >, <, and <> signs. You can use the **StrComp** function:

```
Dim Word1 As String, Word2 As String
Dim Result As Integer
'...
Result = StrComp(Word1, Word2, 1)
```

This function uses three parameters. For a review of parameters, see the accompanying textbox. You use the first two parameters to specify the strings that are to be compared. In this sample code, the **StrComp** function compares the values of *Word1* and *Word2*. The third parameter is optional. You can place one of two values in this third parameter:

- 1, which makes the comparison case-insensitive. This means that "h" will match with "H".
- 0, which makes the comparison case-sensitive. This means that "h" will *not* match with "H".

If *Word1* is less than *Word2*, the value returned by the function and assigned to the variable *Result* is -1. If the strings are equal, the value assigned to *Result* is 0. If the first string is greater than the second, the value assigned to *Result* is 1.

PARAMETERS

A parameter or argument is a value sent to a procedure or a function. In the KeyPress event procedure, KeyAscii is a parameter. In the square root function, **Sqr(x)**, the value x is a parameter. Many functions take more than one parameter and some have no parameters at all. **Time** is a function with no parameters that returns the current system time.

LEN FUNCTION

With the **Len** function, you can count the number of characters in a string. If **Word1 = "hello"**, for example, then **Len(Word1)** is 5, which is the number of characters in the string.

Knowing the length of a string is important when, for example, you need to write code that edits text. Replacing text may require a character count to delete old characters and replace them with new. Searching for an occurrence of a particular string often uses the **Len** function to control the limits of the search.

Exercise

Experiment with the string comparison operators by putting the following statements into the Form_Load procedure of a new project. The **MsgBox** statement displays the string that follows in a message box on the screen. For the program to continue, press the Enter key or click OK.

This exercise demonstrates the string functions and comparison operators discussed above. It works by assigning string literals, such as "apple" and "aple", to string variables and using the comparison operators to compare their values. Each time you run the program, the MessageBox displays the results of the comparison. After each sample, you change the values or the operators and run the program again.

To complete the exercise:

1. Start Visual Basic. If Visual Basic is already running, select New Project from the File menu.
2. Insert the following code into the Form_Load procedure of the default form.

```
Sub Form_Load ()
   Dim Fruit As String, Word As String
   Dim Word1 As String, Word2 As String, Msg As String
   Fruit = "apple"
   Word = "aple"
   If Fruit = Word Then
       Msg = "The strings are equal."
   Else
       Msg = Fruit & " and " & Word & " are not equal "
   End If
   MsgBox Msg
End Sub
```

3. Run the program. Check the output. Was it what you expected?
4. Stop the program and add this code just before the **End Sub** statement:

```
Word1 = "play"
Word2 = "ball"
If Word1 > Word2 Then
   Msg = Word1 & " is greater than " & Word2
Else
   Msg = Word1 & " is less than " & Word2
End If
MsgBox Msg
```

5. Run the program. Check the output. Was it what you expected?
6. Stop the program and add this code just before the **End Sub** statement:

```
Word1 = "Play"
If Word1 > Word2 Then
   Msg = Word1 & " is greater than " & Word2
Else
   Msg = Word1 & " is less than " & Word2
End If
MsgBox Msg
```

7 Run the program. Check the output. Was it what you expected?

8 Stop the program and add this code just before the **End Sub** statement.

```
If UCase(Word1) > UCase(Word2) Then
   Msg = UCase(Word1) & " is greater than " & UCase(Word2)
Else
   Msg = UCase(Word1) & " is less than " & UCase(Word2)
End If
MsgBox Msg
```

9 Run the program. Check the output. Was it what you expected?

10 Stop the program and add this code just before the **End Sub** statement.

```
Dim Result As Integer
Word1 = "play"
Result = StrComp(Word1, Word2, 1)
If Result = 0 Then
   Msg = Word1 & " is equal to " & Word2
ElseIf Result = -1 Then
   Msg = Word1 & " is less than " & Word2
ElseIf Result = 1 Then
   Msg = Word1 & " is greater than " & Word2
End If
MsgBox Msg
```

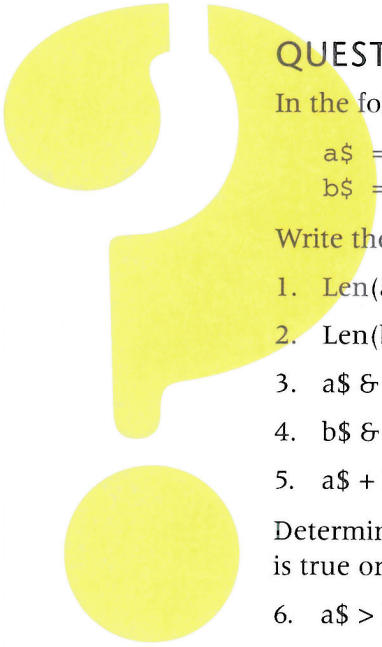
11 Run the program. Check the output. Was it what you expected?

12 Stop the program and add this code just before the **End Sub** statement:

```
Msg = "The length of " & Word1 & " is " & Str$(Len(Word1))
MsgBox Msg
```

13 Run the program.

14 Stop the program.



QUESTIONS AND ACTIVITIES

In the following questions assume these assignments:

```
a$ = "Hello there "
b$ = "California kid "
```

Write the results of the following operations:

1. Len(a$) + Len(b$)
2. Len(b$) + Len(a$)
3. a$ & b$
4. b$ & a$
5. a$ + b$

Determine for each expression in 6 through 9, whether the expression is true or false.

6. a$ > b$
7. a$ > UCase(b$)
8. a$ = b$
9. UCase(a$) < UCase(b$)

What is the result of each of the following?

10. UCase(a$)
11. LCase(b$)
12. StrComp(a$, b$, 1)
13. StrComp(a$, b$, 0)

Section

Menus

In Windows programs, menus are the most common means by which users choose commands. If your program contains many commands, you can see that adding a button for each command would quickly lead to a very cluttered form. In such cases, menus provide an interface that is superior to command buttons.

In other cases, either buttons or a menu bar is an appropriate choice. For example, in the Freefall project (Chapter 3), you could have created a menu to perform the three tasks of calculating, clearing, and quitting. The code you need is not changed; the only question is whether you attach that code to a command button or a menu command.

THE WINDOWS STYLE GUIDELINES FOR MENUS

The Windows style guidelines are a group of recommendations to software developers that put the user's interests first. These recommendations exist not in order to force a bland uniformity on all application programs, or just to make it more difficult to write a program. The guidelines try to benefit users by encouraging consistency among Windows applications. Users can then learn general principles of how to operate Windows programs, rather than having to start from scratch with each new application.

The great majority of Windows application programs work with files. They allow the user to create files of a particular type, edit these files, save them, and open them again later for further editing or inspection. This type of program organization is so common that Microsoft has defined style guidelines to which programs of this type should adhere. According to these guidelines, a program that opens and saves files that the user can edit should provide a main menu whose first two items are the File and Edit submenus. Each of these menus should contain certain commands, and in a recommended order. Another top-level menu item for which standards exist is Help. To see an example of the standard File, Edit, and Help menus, run Notepad and examine its menu structure.

The style guidelines have a great deal more to say on the topic of how menus should be organized and presented. Here, we'll mention just one more recommendation. A menu item should end in an ellipsis (...) if it does not directly perform an action when chosen but instead presents a dialog box to collect further information. For example, standard File menus always let you save the current file under a different name; the text of the menu item that lets you do this is usually Save As.... When you choose Save As..., a dialog box always opens that lets you specify where and under what name to save the file. You can be sure that when you choose a command that ends in an ellipsis, you won't immediately change the data you're working on.

In the programs you've seen and written so far, you've used command buttons instead of menus to let the user perform actions. Menus are neither "better" nor "worse" than command buttons. They're just a another tool that you can use—one that sometimes is more appropriate. Though menus can give your programs the familiar, uncluttered, and

well-organized look-and-feel shared by other Windows programs, keep in mind that it is usually easier and quicker to click a command button than to open a menu and choose a command.

Because users always appreciate programs that let them work rapidly and with a minimum of effort, frequently used commands are excellent candidates for command buttons. When trying to decide whether to use menus or command buttons, keep the user in mind. Put yourself in the position of a user who knows nothing about programming and perhaps not a lot about computers, but who probably knows a great deal about the task that your program is designed to accomplish. Try to create an efficient tool that the user can easily figure out how to operate.

Even after you have created a program, you can change your mind about using command buttons or menus. Simply open the project in design mode and change the interface. In the following steps, you learn how to make that conversion by replacing the three command buttons in the Freefall form with a menu bar containing three commands. You also transfer the code from cmdCalculate to one of these commands.

Like command buttons, menu items have a Click event. When a user clicks on a menu item, the code in the Click event procedure is executed. Just as with objects from the toolbox, you need to name menu commands. For textboxes, you use the "txt" prefix; for labels, the "lbl" prefix; and for command buttons, the "cmd" prefix. For menu commands, you use a "mnu" prefix.

To create a menu bar for the Freefall project, you will add the menu, delete the command buttons, and transfer the code to the menu.

Adding the Menu

You open the Menu Design window by clicking on its button in the toolbar. The button is dimmed unless a form is selected. The menu commands are like the captions of command buttons. Typically, you capitalize the first letter of each caption, and you can designate a character in the caption as a short-cut key by preceding it with the ampersand.

To add the menu:

1. Open the Freefall project by selecting Open Project from the File menu. Click on the View Form button in the Project window to see the form.

2. Select the form, if necessary. In the toolbar, click on the icon for the Menu Design window. This button is dimmed unless the form is selected. The Menu Design window opens.

3 In the Caption textbox, enter the caption of the first menu item, **&Calculate**. Tab to the Name textbox, and enter the name **mnuCalculate** (Figure 5-2).

4 Click on the Next button to enter the next menu command. Enter the caption **C&lear** and the name **mnuClear**.

5 Click on the Next button to enter the next command. Enter the caption **&Quit** and the name **mnuQuit**. The commands appear at the bottom of the window as you enter them.

6 Click OK to close the Menu Design window. The form now has both commands in a new menu bar and the original command buttons (Figure 5-3).

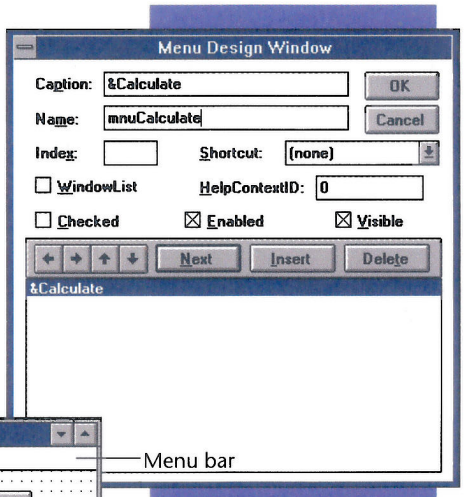


Figure 5-2
Entering information in the Main Design window

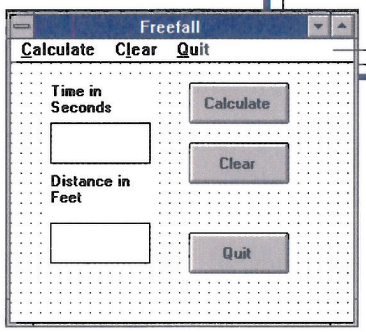


Figure 5-3
Freefall program with both menu bar and command buttons

Deleting the Command Buttons

The menu bar is in place, though clicking the menu items still does nothing. Although some programs allow the user to perform actions either by clicking buttons or by clicking menu items, you will remove the command buttons so there is no redundancy.

To delete the buttons:

1 Highlight each button in turn.

2 Press the Delete key.

The code attached to the command buttons is not lost. Visual Basic moves the code, which is now no longer connected with an event procedure of a screen control. The code is placed in the general area of the Code window.

Transferring the Code to the Menu

To work, the menu items need code attached to their event procedures. In this series of steps, you use the code formerly attatched to the command buttons to bring functionality to the menu.

To transfer the code:

1 Open the Code window by clicking on the View Code button in the Project window, or by double-clicking on one of the controls on the screen.

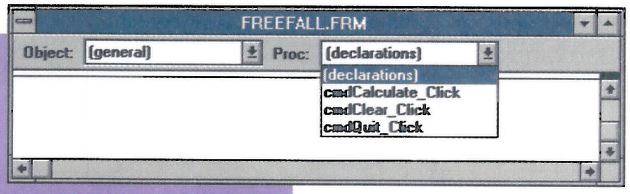


Figure 5-4
Selection of available procedures

2 Open the Object drop-down list by clicking on the downward-pointing arrow. Select the top entry, which is *general*. Open the Proc drop-down list by clicking on the arrow (Figure 5-4). The code of the deleted command buttons is here in the general section.

3 Open an event procedure, such as cmdCalculate_Click. Select and cut the code (not the top or bottom lines).

4 Select the corresponding command from the Object drop-down list, mnuCalculate. Paste the code you just cut into this event procedure (Figure 5-5). Perform this operation for each of the three command buttons you are deleting.

Code from	*Move to*
cmdCalculate	mnuCalculate
cmdClear	mnuClear
cmdQuit	mnuQuit

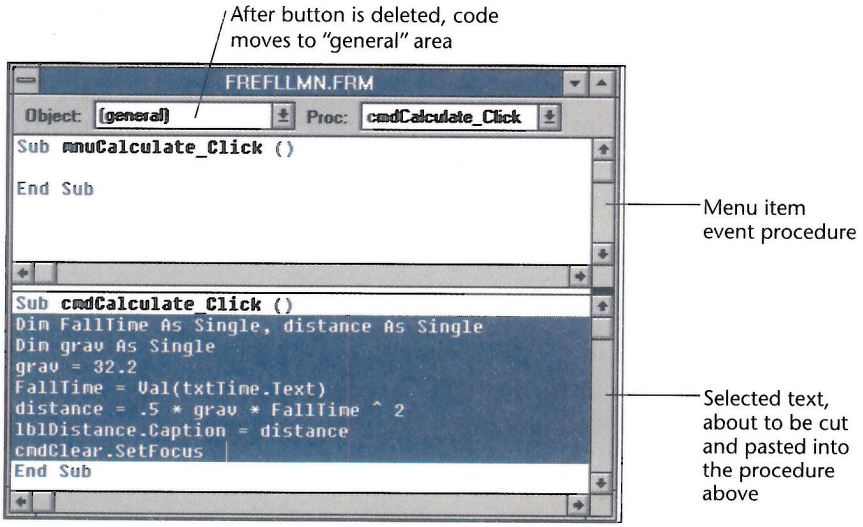


Figure 5-5
Cutting and pasting code

NOTE:

Another approach would be to cut and paste the code first, then delete the buttons.

Finishing Up

Now you are ready to test your changes:

1 Run the program. The program halts before it begins. Can you figure out what the problem is?

2 One line of code from the original cmdCalculate button uses the **SetFocus** method to shift the focus to the Clear button. That button no longer exists. Delete that line of code.

3 Run the program. Test each menu item.

4 Save the project and the file with a new name to preserve the command button version of the program.

QUESTIONS AND ACTIVITIES

1. Replace the command buttons of the Cereal project with a menu bar.

The MiniEdit Project

The MiniEdit program provides the functions of a text editor. Within the Edit window of the application, the user can create and edit text. With the menu choices you will provide, the user can open, save, and close files.

This project introduces more complicated menus as well as reading and writing files to disk. After working through this project, you will be able to create programs that are similar to professional applications. You want your programs to conform to users' expectations of how a Windows program works. Users will then find your programs easier to understand quickly.

All programs except the simplest utilities save and retrieve information from files. This saving and retrieving is the most common use of computers. Companies, for example, use files to save customer and employee information. All kinds of organizations, from a local karate club to an international wildlife federation, maintain mailing lists in files.

State, local, and federal governments keep files of resident information. Schools keep files of students, classes, teachers, expenditures, and revenues. Word processors use files to store and retrieve documents. If you could not store and open files, you could not start working on a letter or term paper at the beginning of the week, then come back to finish the project several days later.

Game programs save high and low scores as well as complete games on disk. Scientists store files containing information on stars, dinosaurs, and DNA sequences. Communications programs transfer disk files all over the world. These are just a few of the ways we have become dependent on reading and writing computer files.

Starting Out

Each project starts by setting up a directory for the project's files. Once a disk directory has been prepared, you set up the user interface. For

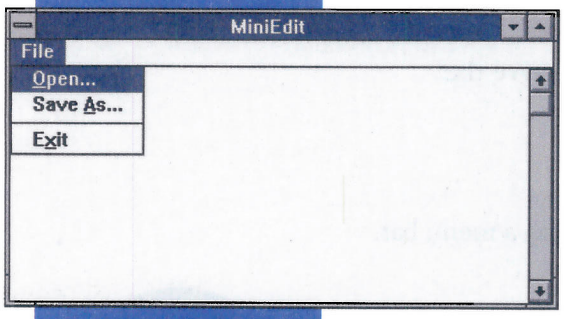


Figure 5-6
The complete MiniEdit project

MiniEdit, you start by defining the menu structure in the Menu Design window. Besides the menu, the multiline text box is the most important feature of the user interface.

One the form has been designed, code is added to provide functionality. You'll start by writing code to resize the edit space to fill the form. Then you'll spend a lot of time working with the code to open, load, and save files. Figure 5-6 shows a shot of the final product.

Creating the Menu and Edit Space

The user interface is one of the most important parts of a program. If the program is not attractive and easy to use, chances are no one will want to use it. This stage of the program development creates a standard-looking user interface. Specifically, you create a File menu.

The body of the form is taken up by a multiline textbox. Remember, a multiline box will automatically wrap text that is too long for the line—just what you want in a text editor. A "form-filling" box just means to size the box to fill the form. Later, in the code, you will adjust the size of the box to fill the form when the program is running and the user resizes the form with the mouse. A vertical scrollbar, provided by the textbox, takes care of scrolling to text that is not visible.

The menu bar contains a separator. Many Windows applications use separators to visually separate related, though distinct, groups of commands. In MiniEdit, the Exit command is set apart from the other file commands with a separator. A single hyphen in the Menu Design window displays a solid bar separating the first three menu items from the last. Although no code is executed by clicking the separator bar on the menu, the bar must have a name. Each item defined in a Menu Design window must have a name and a caption.

Follow these steps to create a menu and edit space.

1. Create a new subdirectory for the MiniEdit project:
   a) Open the File Manager.
   b) Select a directory in which to create the new subdirectory.
   c) Select Create Directory from the File menu and follow the instructions in the window. Create the **Medit** directory.
2. Start Visual Basic. If Visual Basic is already running, select New Project from the File menu.
3. Change the caption of the form to MiniEdit.

4 Click on the Menu Design window button in the toolbar. This opens the Menu Design window.

5 Create a menu bar with a single item on it (File). When the user clicks on this menu item, a drop-down menu opens. See Figure 5-7 for the entries in this menu.

a) In the Caption textbox, enter **File**. In the Name textbox, enter **mnuFile**.

b) Click on the Next button to enter the next menu item.

c) Use the right arrow button in the Menu Design window to create a submenu. The items in a submenu are indented in the window. Enter captions and names for each item in the submenu, as shown below. After entering this information for each item, click on the Next button.

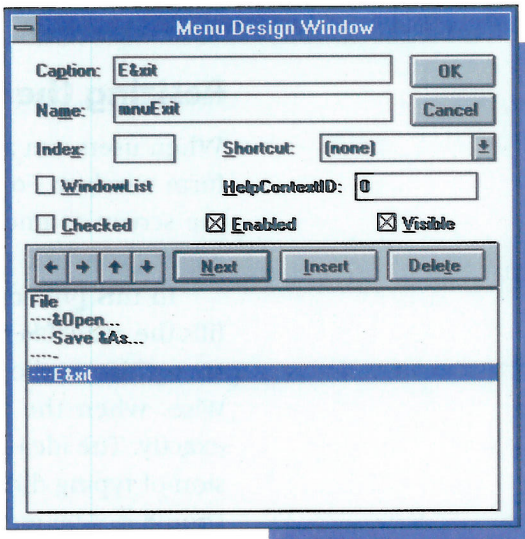


Figure 5-7
Setting up the last entry in the subdirectory

Caption	*Name*
&Open...	mnuOpen
Save &As...	mnuSave
-	mnuSeparator
E&xit	mnuExit

6 Create a form-filling multiline textbox. In the Properties window, be sure that the Top and Left properties of the textbox are both set to 0. Change the Name property to **txtEdit**. Set the MultiLine property to True. Change the Scrollbars property to 2-Vertical. Set the Text property to "". Finally, set the BorderStyle to 0-None (see Figure 5-8).

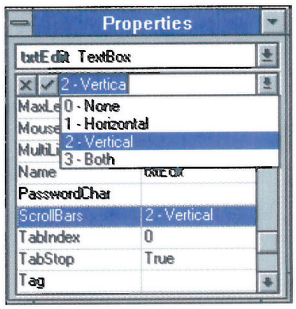


Figure 5-8
Scrollbars property

7 Run the program.

8 Type some text into the textbox. Type enough text to word wrap to the next line.

9 Play with the program. Use the mouse to select text. Press Ctrl+X and Ctrl+V to cut and paste. Try out the scrollbar. Try out the menu choices.

10 Stop the program by pressing Alt+F4 or by clicking on the Stop button on the toolbar.

As you experimented with the program, what did you notice about it? You cannot save your work, for example, nor can you open an existing file to edit. The menu items do nothing yet. The program does not

execute any action if the user clicks on one of the commands. Also, if you resize the form while the program is running, the textbox does not expand or contract to fill the space.

Resizing the Form

When users run applications, they often want to change the size of the form window. For example, a user may want to fit several windows on the screen at once. This can be done by clicking and dragging on the edge of the form to resize it.

In this project, when the program is started, the multiline textbox fills the available space inside the form. To keep it that way, you need to set up the textbox so that it changes size along with the form. Otherwise, when the form is resized, the textbox no longer fits inside it exactly. The idea behind this form is that the user should have the illusion of typing directly in the window, no matter what its size. The user should not be made aware of a separate entity, the textbox, that limits the available typing area.

When a user resizes a form, a Resize event occurs. This is one of many events attached to the form object. As a result of this change, two of the form's properties change:

- ScaleHeight
- ScaleWidth

You need to insert code into the Resize event procedure so that the height and width of the textbox always match the height and width of the form. Then, when the form is resized during run-time, the textbox will be resized as well. The height and width of the textbox are controlled by two properties: Height and Width. You do not have to create any code to keep setting the Top and Left properties of the textbox to 0 whenever the form is resized. These properties retain their initial 0 values.

To keep the textbox in alignment with a resized form:

1. Click on the body of the form, or click on the View Code button in the Project window.
2. Choose the Form object from the Object drop-down list.
3. Choose the Resize event from the Proc drop-down list.
4. Insert the following code in the event procedure:

```
txtEdit.Width = form1.ScaleWidth
txtEdit.Height = form1.ScaleHeight
```

5 Run the program and enter some text. Resize the form and note the change in the edit box. Figure 5-9 shows the running program with text from the Declaration of Independence.
Figure 5-10 shows how the text is rearranged when the form is resized.

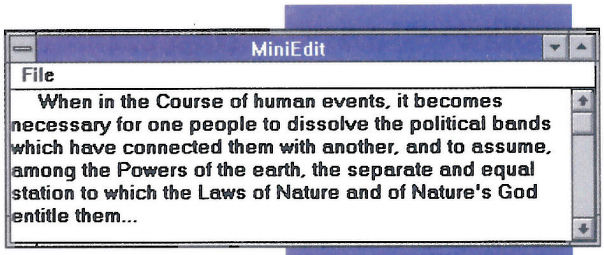


Figure 5-9
Text in the edit box

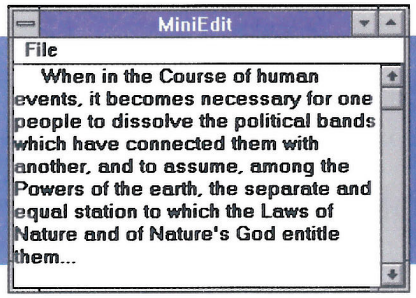


Figure 5-10
Text in a resized edit box

Getting the Exit Command Working

Just like command buttons, menu commands generate a Click event when a user selects them. As a result, you can insert code in the Click event procedure for the menu command. This code will then be executed when a user clicks on the command. You can test this by getting the Exit command working.

To insert code into the mnuExit event procedure:

1 Click on the File menu on the MiniEdit form. This opens the drop-down menu.

2 Double-click on Exit to open the Code window.

3 After the opening line of the mnuExit_Click subroutine, enter **End** (see Figure 5-11).

4 Run the program. Enter text into the textbox. Select Exit. Does anything happen?

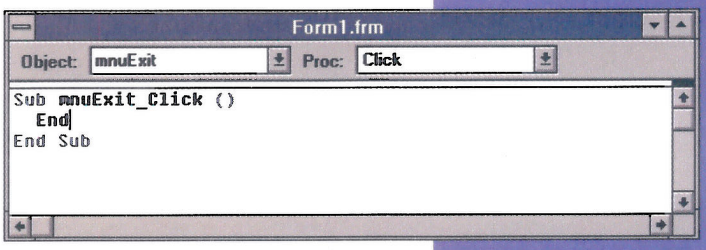


Figure 5-11
Setting up the mnuExit event procedure

The End command works now, but not quite like a professional program. To see where the difference lies, think about a program like Word for Windows. If you type some text, then select Exit, what happens? The program prompts you to save your work. You cannot expect users to remember to save before they exit your program. You will need to add this functionality to the mnuExit_Click subroutine later.

Implementing the Open Command

Opening text files is a basic operation of MiniEdit. Once a file is opened, the contents are put into the textbox for display and editing. Getting your Open command to work correctly takes a few steps. You must write the code so that the program can:

1. Get a filename from the user.
2. Open the file.
3. Move the contents of the file into the textbox.
4. Change the directory in which the program looks for files.

Each of these steps is covered separately in this section.

GETTING A FILENAME FROM THE USER

As you write the code necessary for this step, you will use a number of functions:

- **InputBox**
- **Trim$(str)**
- **UCase$(str)**

These functions are covered in the accompanying box. Read this material before you try to perform the following steps.

To get a filename from the user:

1. Select the MiniEdit form and click on the File menu. Double-click on Open. The Code window opens, showing the mnuOpen_Click() event procedure.
2. Insert the following code between the first and last lines of the subroutine.

```
Dim strFn As String
strFn = UCase$(Trim$(InputBox("Filename", "Open File")))
MsgBox strFn
```

3. Run the program.
4. Select Open from the File menu. Enter filenames with and without uppercase characters. Note the resulting filenames in the message box.

The code you just entered declares *strFn* as a string variable to represent the filename. At the core of the following statement is an **InputBox** statement with two parameters. The first parameter instructs the

user to enter the filename. The second is the caption of the input box (Figure 5-12).

Once the entry has been processed, it is assigned to *strFn*. The **MsgBox** *strFn* statement displays the filename in a message box. You can use this message box to check the program during run-time (Figure 5-13). Later, when you are satisfied that the program is complete, you can remove the statement or comment it out.

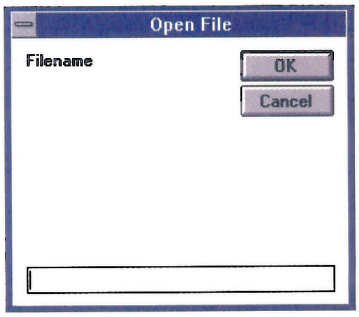


Figure 5-12
Open File InputBox

Figure 5-13
Filename displayed in a message box

FUNCTIONS

The three functions used in this section are introduced here.

INPUTBOX FUNCTION

The **InputBox** function collects information the user enters from the keyboard and returns the entered string as the value of the function call. This returned string can then be used in an expression, which in practice is usually an assignment to a string variable. The function takes three parameters, the last of which is optional. The syntax of the **InputBox** function is:

variable name = **InputBox** *(prompt string, box title string, default value)*

The **InputBox** function creates a window on the screen. The first parameter is an instruction to the user. The second parameter is the caption of the box created. The third parameter is the string that will initially appear in the box in which the user types. If you don't supply a value for the third parameter, the empty string is used by default. The mnuOpen_Click() event procedure that you created uses this default.

If you want to omit a caption, leave that spot blank in the parameter list. The commas must appear:

```
x = InputBox("Enter a number",,"345")
```

This statement creates the box and prints the prompting message "Enter a number". In the textbox that the function provides for the user to type in, the default value 345 appears. When a user enters a value and clicks OK, this value is returned by the function. See Figure 5-14.

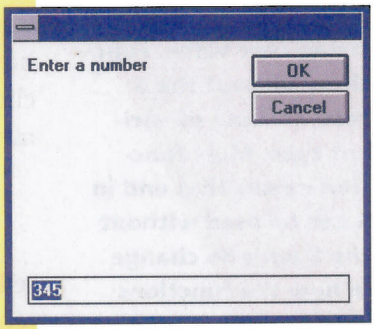


Figure 5-14
Entering a number as prompted

TRIM$(STR) FUNCTION

The **Trim$(str)** function takes a string as a parameter and returns the string with all leading and trailing spaces removed.

This function can be used to "clean up" strings obtained from the user via the **InputBox** function. You can use **Trim$** on the string returned from **InputBox** to remove any initial or trailing spaces that the user may have accidentally typed.

For example, if you use **InputBox** to obtain a person's name, then it is good practice to use **Trim$** on the returned string, because people's names never begin or end with spaces. Furthermore, if you don't remove accidentally typed spaces at the beginning or end of a name string, the results of comparing that string to other names will not be what you might expect. (The ASCII code for the space character is 32, which is less than the code for every letter.)

UCASE$(STR) FUNCTION

The **UCase$(str)** function was introduced earlier in this chapter, in the discussion of strings. You use it to convert characters of a string to uppercase. One use of **UCase$** is to allow users to easily enter information that should be uppercase, without forcing them to hold down the Shift key or turn on Caps Lock. For example, many product model numbers consist of uppercase letters and digits. After the user fills in a textbox containing a model number, you could use **UCase$** to immediately convert the user's input to the proper form and redisplay the result.

NOTE:

Function names that end in a $ return a string type value. Functions without the $ return values of variant type. Most function names that end in $ can be used without the $ with no change in how the functions work.

OPENING THE FILE

After the program has prompted the user for a filename, it must open the file requested. To make it possible for a user to open a file, you only have to insert a single line of code. In the MiniEdit File menu, double-click on Open. The Code window opens again, showing the mnuOpen_Click() event procedure. Insert this line:

```
Open strFn For Input As #1
```

This statement opens the file whose pathname is in *strFn*. **For Input** tells Visual Basic the file is opened so its contents can be read. The contents of the file are now available to the program. The phrase **As #1** associates the open file with a file number—in this case, the number 1. After the file is opened, you use the file number to refer to the file. When you ask Visual Basic to read from this file and when you tell Visual Basic that you are finished with it, you refer to the file by using #1 rather than the filename itself.

TRANSFERRING FILE INFORMATION TO MINIEDIT

Once the file is open, the contents need to be read into memory. The information from the file must be transferred to the txtEdit box. In order to test the ability of MiniEdit to open files, you have to be sure that a file exists that it can open. Because you have not yet implemented the Save command, you cannot use MiniEdit to create files that it can later open. Fortunately, MiniEdit uses standard text files, which Windows Notepad also creates, reads, and writes. In this section, you will use Notepad to create a text file that you will then open in MiniEdit.

As you write this code, you will become familiar with two new functions:

- **LOF**, which stands for *l*ength *o*f *f*ile
- **Input$**

To insert a copy of the file's contents into the textbox:

1 If necessary, reopen the Code window displaying the mnuOpen_Click subroutine. After the last line you entered (Open strFn...), insert:

```
Dim FileSize As Integer
FileSize = LOF(1)
txtEdit = Input$(FileSize, #1)
Close #1
```

For information on **LOF** or **Input$()**, see the accompanying box.

LOF AND INPUT$ FUNCTIONS

The **LOF** function takes a single parameter. You use this parameter to send the file number to the function. The function returns the length in bytes of the file at that file number. You could find out this same information about a file by selecting it in the File Manager. The File Manager displays the file size if the proper options are chosen in the View menu.

The **Input$** function returns a string of *FileSize* length, from the file with file number 1. You want to read the entire file into txtEdit, not just a portion of it. That is why you first used **LOF** to obtain the total size of the file and store that number in the variable *FileSize*. You then tell the **Input$** function to read *FileSize* many characters from the file. As a result, the string returned from the **Input$** function is the entire file. This string is assigned to the txtEdit box.

2. Remove the MsgBox statement. You could also make it a comment by inserting a single quote as the first character of the line.

3. Without exiting Visual Basic, press Alt+Tab to move to the Program Manager. From the Program Manager, run NotePad (usually found in the Accessories program group). In NotePad, type:

```
When in the Course of human events, it becomes nec-
    essary for one people to dissolve the political
    bands which have connected them with another,
    and to assume, among the Powers of earth, the
    separate and equal station to which the Laws of
    Nature and of Nature's God entitle them...
```

4. Save the file as **declare.txt** in a directory you will remember (suggestion: **c:\temp**).

5. Close NotePad and return to Visual Basic.

6. Run MiniEdit.

7. Select Open from the File menu and try to open the file you just created (Figure 5-15). Be sure to enter the complete pathname of the file. If Visual Basic reports *File not found* when you run MiniEdit, try to find the file using the File Manager. If you have left **declare.txt** open in NotePad, the file will not be found.

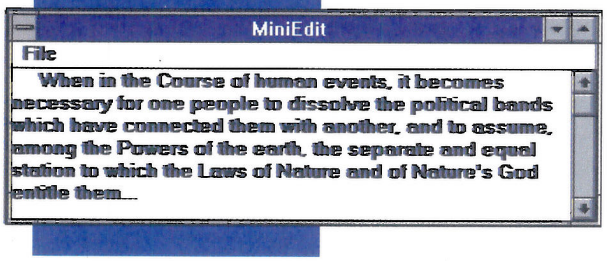


Figure 5-15
Opening the newly created file

8. Exit MiniEdit.

9. Save the project in the Medit directory.

CHANGING THE DEFAULT DIRECTORY

When a user provides the name of a file to open, Visual Basic looks for it in the default directory. Unless you change the default, this is the Visual Basic directory. To open a file that is not in this directory, a user must provide not only the filename, but also the path. For instance, if you saved the file **declare.txt** in the directory **c:\temp**, you could open the file by entering the complete pathname, **c:\temp\declare. txt**, at the prompt.

The directory that an application resides in should contain the files that it needs to operate. The file **declare.txt** that you create for this project is clearly not one which Visual Basic itself will ever use. It is your data file, and should therefore be stored in a different location than the Visual Basic directory.

Changing the directory only requires a couple of lines of code. However, you will be using a number of statements for the first time:

- The **ChDrive** statement, which changes the current drive to the drive indicated in the string that follows.
- The **ChDir** statement, which changes the current directory to the directory indicated in the string that follows.
- The **App** object, which is the currently running application.
- The Path property of the **App** object, which specifies the complete pathname of the application.

When you include these statements in the Form_Load event procedure, the program changes the current directory and drive to match those of the application being run. In this case, the MiniEdit project is the current application. Unless you saved this project in the Medit directory, then Visual Basic is the current application directory.

To change the directory:

1. Double-click on the MiniEdit form to open the Code window. In the Form_Load event procedure, enter these lines.

```
ChDrive App.Path
ChDir App.Path
```

2. Use the File Manager to move **declare.txt** from **c:\temp** to the MiniEdit application directory, Medit.
3. Run MiniEdit. Open the file named **declare.txt**.
4. Exit MiniEdit.

Inserting Code for the Save As Command

You have a final command to implement: Save As. After a user has opened and worked in a file, the work should be saved.

The original file was opened with the **For Input** clause in the **Open** statement:

```
Open strFn For Input As #1
```

You cannot write to a file opened for input. To save the file, query the user for a filename (it could be the old name), and reopen the file **For Output**. Transfer the text into the file and close the file.

To create this code:

1. With MiniEdit halted, click on the Save As command to open the Code window.

2 In the mnuSave_Click procedure, enter the code:

```
Dim strFn As String
strFn = UCase$(Trim$(InputBox("Filename", "Save As...")))
Open strFn For Output As #1
Print #1, txtEdit
Close #1
```

The **Print #** statement writes an expression to a file at the specified file number. In this case, the expression is the entire text of the file. It is written into the file whose name is entered.

The close statement closes the open file. You can continue editing the contents of the editbox, and save it again under the same name or perhaps a different name.

Summary

While it is certainly possible to further enhance and refine MiniEdit, you have already learned how to harness several powerful capabilities of Visual Basic that you will use again and again. This is a good time to step back and take stock of what you have—and what you have *not*—accomplished in MiniEdit.

WHAT MINIEDIT CAN AND CANNOT DO

With MiniEdit, a user can create a text file and save it to a filename. A user can also open an existing text file, edit, and save the changes.

MiniEdit has a number of shortcomings, however. For example, commands in MiniEdit are always enabled, even when that choice might be inappropriate. In professional Windows programs, a menu command is grayed, or dimmed, whenever it is not meaningful for a user to choose it.

MiniEdit doesn't remember filenames. The user enters a filename to open a file and enters the filename again to save the file. MiniEdit also lets you exit the program without asking you to save your work.

All of these shortcomings could be eliminated if MiniEdit had a way of preserving information from event to event. If the filename entered in the Open event could be saved for use in the Save As event, MiniEdit could include the old filename as a default in the **InputBox** function.

At this point, all variables and their values come into existence when an event occurs. They cease to exist when the program exits the event procedure. As you will see in the next chapter, there is indeed a way to make values persist for longer than the handling of single events.

DEFENSIVE PROGRAMMING

Because MiniEdit does not do any error checking, you can easily make the program terminate with a run-time error. Up to this point, all the programs you have built have functioned in their own little world, not touching data outside themselves. As soon as a program has to interact with the file system, though, you must expect the unexpected.

Think of the problems that could occur. A user may type in a bad filename, for example, or the name of a nonexistent file. The file the user is trying to open cannot be opened, either because someone else on your computer's network has it open exclusively, or because of insufficient memory. The user can't save a file because there is not enough space left on the disk. MiniEdit can detect only some of these errors. You need to learn more about error handling before you can write code to detect all of these errors.

Still, as you build programs, you should anticipate that operations may fail. You should think about ways to have the program recover gracefully if a problem occurs. Instead of terminating with a run-time error, the program should alert the user and then continue. Writing code that guards against errors and prevents them from occurring is called "defensive programming."

The For-Next Statement

As you learned in Chapter 3, you use loops to repeat a section of code again and again. This section focuses on the **For-Next** statement, which is one of the principal looping statements of Basic. As you become more familiar with loops, you will find yourself including them in most of your programs. Using loops is an important way to solve problems.

A **For-Next** loop is called a definite loop. In a definite loop, the number of times the code contained in the loop will repeat is generally known when the loop begins. You should consider using a definite loop whenever you know, or can calculate, the exact number of times you want a section of code to be repeated.

The syntax of the **For-Next** loop is:

For *variable* = *start* **To** *limit* **Step** *incr*
.....body of the loop (usually indented for readability)
Next *variable* *'–the same variable from first line*

Here, *variable* represents the name of a variable. Also, *start*, *limit*, and *incr* represent expressions. While an expression is allowed in the places

indicated, more often you will be using single variables or literals. The effect of the **For-Next** loop is to execute the body of the loop repeatedly, with the *variable* taking on a different value each time. The variable runs through all the values *start*, *start* + *incr*, *start* + *incr* + *incr*, ... that are between *start* and *limit*.

The value of the first expression, *start*, defines the starting point for the loop. The starting point is the initial value of the variable when the loop begins. For example, the starting point of the following **For** loop is 1:

```
For i = 1 To 100 Step 1
.....'—body of the loop
Next i
```

The expression following **Step** defines the increment of the loop. As the loop executes, the value of the variable is changed by the increment. If the increment is 1, as in the above code sample, the value of the variable increases by 1 each time through the loop. If the increment is -2, the value of the variable actually *decreases* by 2 each time through the loop.

The value following **To** defines the limit of the loop. In the code sample above, the limit of the loop is 100. The value of the variable is compared to the limit before the body of the loop is executed. If the comparison succeeds, then the body is performed. If it fails, the loop statement ends.

If the increment of the loop is positive, then the comparison checks that the variable is still less than or equal to the limit. If the increment is negative, the comparison checks that the variable is still greater than or equal to the limit. In the above example, *i* starts at 1 and increases through all values up to 100. The body of the loop is performed each time. Finally, *i* is increased to 101 and compared to 100. Because the increment is positive, and *i* is no longer less than or equal to 100, the loop ends.

When the increment is positive, the starting point must be less than or equal to the loop limit for the body of the loop to be performed at all. When the increment is negative, the starting point must be greater than or equal to the loop limit for the body of the loop to be performed at all.

The default increment is 1, by far the most typical case; if you want the loop variable to increase by 1 each time through the loop, you don't even have to write "Step 1". Therefore, the above code that we have used to illustrate the parts of the **For** loop could more easily be written as follows:

```
For i = 1 To 100
.....'—body of the loop
Next i
```

When the loop ends, the program resumes execution with the statement following the **Next** statement (Figure 5-16).

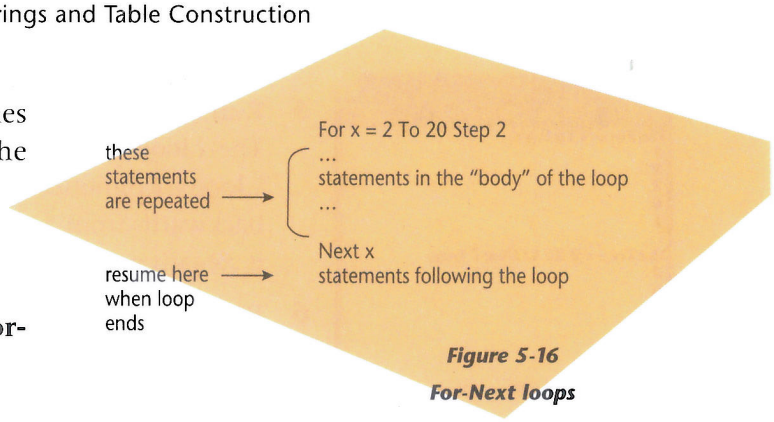


Figure 5-16
For-Next loops

Working with For-Next Loops

The easiest way to become familiar with **For-Next** loops is to write some yourself.

1. Start Visual Basic. If Visual Basic is already running, select New Project from the File menu.
2. Set the AutoRedraw property of the default form to True.
3. Change the caption of the form to **For-Next Examples**.
4. In the Form_Load event procedure, enter the code shown in Figure 5-17.

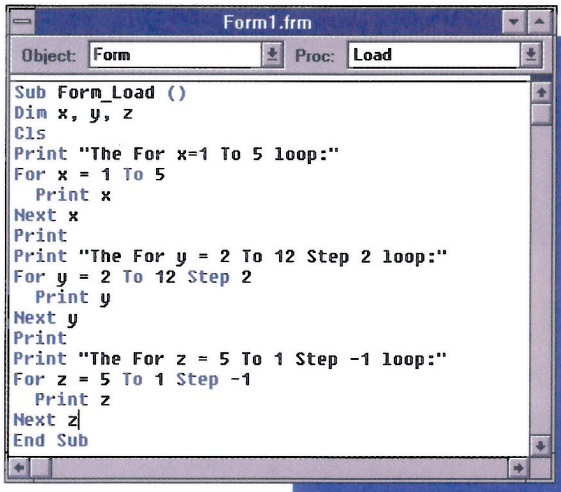


Figure 5-17
Code for the For-Next loop

THE PRINT METHOD

The **Print** method displays expressions on an object, such as a control or a form. The syntax for the **Print** method is:

object.Print expressionlist

If the object name is omitted, expressions are printed on the default form.

Strings and numeric values may be included in the expression list. The **Spc(n)** function may be included to skip *n* spaces in the output. The **Tab(n)** function may be included to skip to print zones in the output. The print zones are equivalent to the tab settings of a text document.

If the **Print** method is used with no *expressionlist*, a blank line is printed.

Do not confuse the **Print** method with the **Print** *statement*. You used the **Print** statement to implement the Save As command in MiniEdit. The **Print** method displays a string in a object; the **Print** statement writes a string to a file.

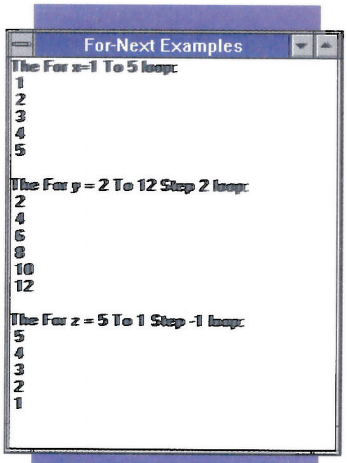


Figure 5-18
Results of running the program

5 Run the program and observe the results (see Figure 5-18).
The *x* loop omits the step value. The variable increments by 1. The *y* loop increments by 2 until the limit is exceeded. The *z* loop steps backwards from 5 to 1, because the increment, defined in the step, is negative.

6 Reenter the Form_Load procedure and replace the **For-Next** statements with these:

```
For x = 0.01 To .35 Step 0.05
For y = -2 To 7 Step 2
For z = 5 To 2*12 Step 5
```

Change the printed labels to match the new **For-Next** statements.

7 Run the program and observe the results (Figure 5-19).

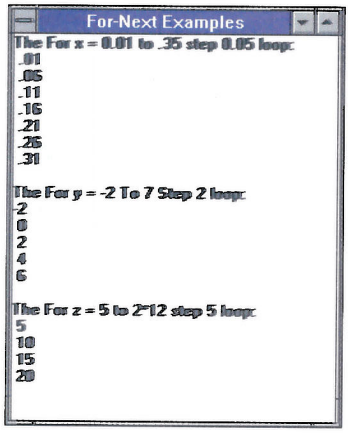
In the first loop, the starting value of *x* is 0.01, one one-hundredth. Each time through the loop, the value is increased by 0.05. The loop ends when the value of *x* exceeds 0.35.

In the second loop, the starting value is a negative number.

In the third loop, the starting value is a calculated value, not a simple constant. The increment is negative. This loop initializes *x* to 2*12 and steps down by 5 each time through. The loop ends when *x* is less than 5.

Try a loop like this: For *w* = 2 To 1. Because the increment is positive, and the starting value is greater than the limit, the body of the loop is never executed.

Figure 5-19
Further results from running the program



Nesting Loops

Just as you can write an **If-Then** statement within another **If-Then** statement, you can also write a loop within another loop. When one loop is inside of another, the loops are said to be nested.

Doing homework is like working through nested loops:

For *ChapterNumber* = *1* **To** *15*
  For *ReviewQuestion* = *1* **To** *20*
    Write answer to question
  Next *ReviewQuestion*
Next *ChapterNumber*

In the example above, the outer loop steps through the 15 chapters of a textbook. The inner loop steps through the 20 review questions for each chapter. The answers to 15*20, or 300, questions will be written. If

your teacher were to assign only the even questions, for example, you would write the loop like this:

```
For ChapterNumber = 1 To 15
    For ReviewQuestion = 2 To 20 Step 2
        Write answer to question
    Next ReviewQuestion
Next ChapterNumber
```

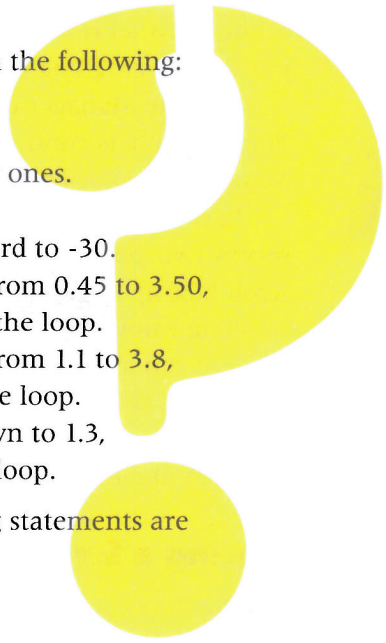
Nested loops cannot overlap like this:

```
For x = 1 To 10
    For y = 1 To 10
        ...
    Next x      '—syntax error!
Next y
```

The example just given will not run, because the **Next** statements are mismatched. Visual Basic will inform you of the syntax error. The **For** *y* loop begins within the **For** *x* loop, and therefore must be completely enclosed within the outer **For** loop.

Using looping statements, a program can repeatedly execute statements far faster than you could. As a result, the computer can mindlessly crank through complicated procedures that would be far too tedious to work through by hand or even with a calculator.

QUESTIONS AND ACTIVITIES



1. Write **For-Next** statements that accomplish the following:
   a) Generate even numbers from 4 to 140.
   b) Generate odd numbers from 3 to 101.
   c) Count backwards from 89 down to 5 by ones.
   d) Generate multiples of 5 from 25 to 130.
   e) Generate multiples of 3 from 15 backward to -30.
   f) Generate a list of numbers that ranges from 0.45 to 3.50, stepping up by 0.05 each time through the loop.
   g) Generate a list of numbers that ranges from 1.1 to 3.8, stepping up by 0.1 each time through the loop.
   h) Generate a list of numbers from 5.3 down to 1.3, stepping by -0.1 each time through the loop.
2. What do you think happens if the following statements are executed? (hint: this is **not** a syntax error)

```
For y = 1 To 10 Step 0
   y = y + 1
Next y
```

3. What numbers are generated by the following:
   a) For x = 3 To 30 Step 3
   b) For y = 22 To 2 Step -2
   c) For z = 11 To 121 Step 11
   d) For a = 1 To -10 Step -1
   e) For b = 7 To 7*14 Step 2

5 Section

Scrollbars

In the MiniEdit application, users edit text in a multiline, scrollable textbox. The vertical scrollbar is an integral part of that textbox and not a separate control. The textbox itself manages its scrollbar. You didn't have to write any code to respond to clicks on the scrollbar; nevertheless, it was fully functional.

In this section, you use the scrollbar controls from the Toolbox. Unlike MiniEdit's scrollbar, every aspect of the scrollbar control is programmed either during program design or at run-time. Scrollbar controls can be placed anywhere on a form, not just along the border of the window or of a textbox.

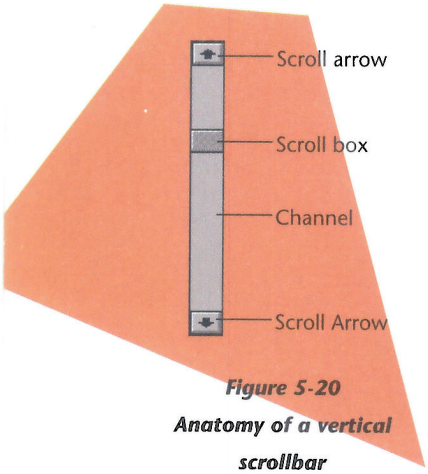


Figure 5-20
Anatomy of a vertical scrollbar

Scrollbars come in two versions: horizontal and vertical. Each has a channel, in which an indicator, called a scroll box, moves. At each end of the channel is an arrow called a scroll arrow. Clicking on this arrow moves the scroll box right or left, up or down (Figure 5-20).

The File Manager uses two vertical scrollbars, one for each window. These scrollbars control the display of directory and file lists. In the File Manager, the scrollbar is used as an input device.

In a word processing program such as MiniEdit, the vertical scrollbar serves two purposes. You can use it as an input control, moving the scroll box up or down. This lets you position the cursor in the text by scrolling (moving) the text up or down within the window.

You can also use a scrollbar as a display indicator. By looking at the position of the scroll box in the scrollbar, you can tell where the cursor is in the document. If the scroll box is halfway down the scrollbar, for example, the cursor is near the middle of the document.

Placing a Scrollbar Control

The Toolbox contains two scrollbar controls: one for a horizontal scrollbar and one for a vertical scrollbar (Figure 5-21). You place a scrollbar control in the exact same way you place any other control.

You can double-click on the control, then click and drag to move the scrollbar to the right place. Or, you can:

1. Click once on the icon, highlighting it.
2. Move the mouse to the form, positioning it where you want the upper-left corner of the scrollbar.
3. Click and drag the mouse to the point at which you want the scrollbar's lower-right corner. Release the mouse. (Figure 5-22).

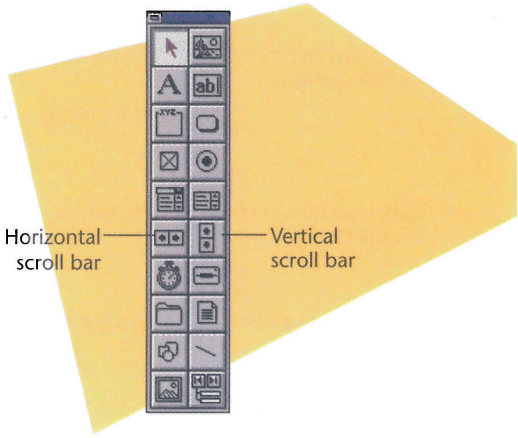
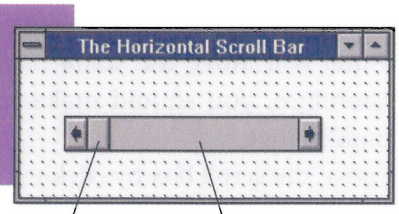


Figure 5-21
Scrollbar controls in the Toolbox

Figure 5-22
Creating a scrollbar on a form



The scrollbox is in the left-hand-most position

The scrollbar channel

Setting the Properties of the Scrollbar

As with any object, you can control the properties of the scrollbar. As the scroll box moves, the Value property of the bar changes. The Value property indicates the current position of the scroll box. The Min property sets the starting value of the Value property. The Max property sets the maximum value of the Value property (Figure 5-23).

As an input device, the scrollbar is used to control displays. The Value property of the scrollbar can be used as a numeric input to a program.

The default settings for a scrollbar control include a minimum value of 0 and a maximum value of 32,767. Before you change these defaults, you should consider the rate at which you want the scroll box to move.

This rate of movement is controlled by the SmallChange property. Clicking either scroll arrow causes a SmallChange. The default value of SmallChange is 1. With Max set to 32,767, each click of a scroll arrow moves the scroll box 1/32,767th of the way from one end of the bar to the other. This is not practical and would be irritating to your users. They would have to click the down scroll arrow 32,767 times to reach the end of a document.

One option is to change the Max property to 10. Then, with every mouse click, the scroll box would move 1/10th of the way from one end

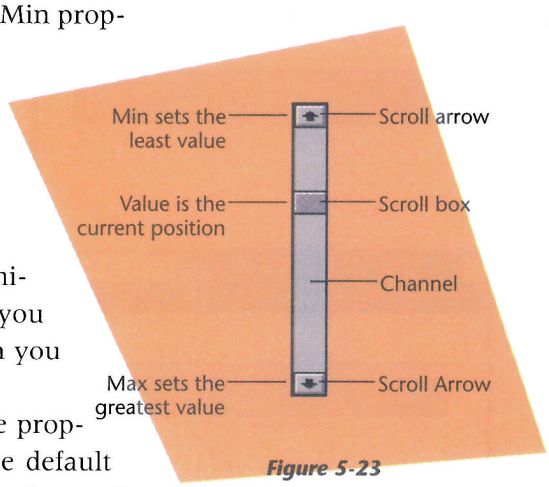


Figure 5-23
Setting different properties for the scrollbar

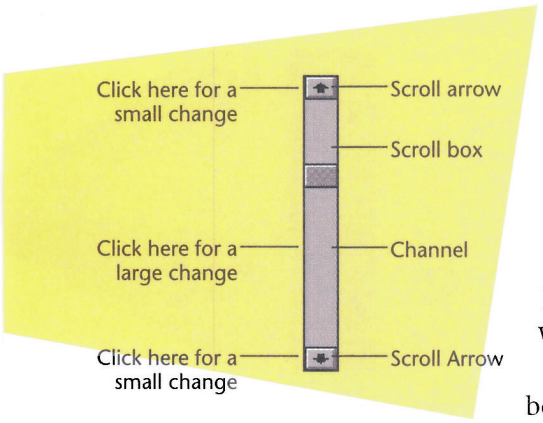


Figure 5-24
Moving the mouse scroll box

of the bar to the other. This movement would be far more noticeable.

Another option is to leave the value of the Max property as 32,767. You can still change the effect of a click on the scroll arrow, by changing the value of the SmallChange property to 1000. Each click on the scroll arrow then changes the Value property by 1000, moving the scroll box 1000/32,767ths of the way from one end of the bar to the other.

If the user clicks the mouse in the channel instead of the scroll box or the scroll arrows, a LargeChange occurs. See Figure 5-24.

The default value of LargeChange is 1. Users expect that a click in the channel will move the scroll box further than a click on a scroll arrow. Therefore, you should make this property's value larger than that of SmallChange.

Users also move the scroll box by clicking and dragging the scroll box along the scrollbar. The Value property changes as the scroll box moves.

Using the Scrollbar as a Progress Indicator

A program can also use a scrollbar to show its progress while performing a lengthy task. It can do this by moving the scroll box from its minimum to its maximum value to indicate the proportion of the task that it has completed. When the program begins its processing, it sets the scroll box to its minimum value. As processing proceeds, the scroll box is set to move toward its maximum value. When the scroll box reaches its maximum, this signals to the user that processing is done.

To use a horizontal scrollbar as a progress indicator, follow these steps.

Figure 5-25
Placing the scrollbar on the form

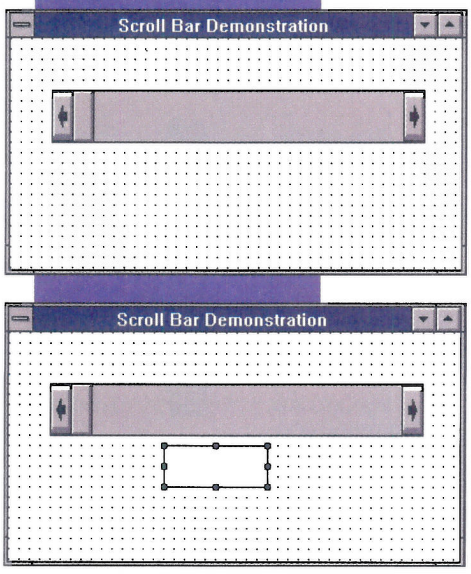


Figure 5-26
Adding a label to the form

1. Start Visual Basic. If Visual Basic is already running, select New Project from the File menu.
2. Change the caption of the form to **Scrollbar Demonstration**.
3. Place a horizontal scrollbar on the form (Figure 5-25).
4. Place a label on the form (Figure 5-26). In the Properties window, make the following changes:
   - Name property: **lblValue**. You will use this label to display the Value property of the scrollbar.
   - BorderStyle: Fixed Single.
   - Caption property: delete the default.
   - Alignment property: Center.

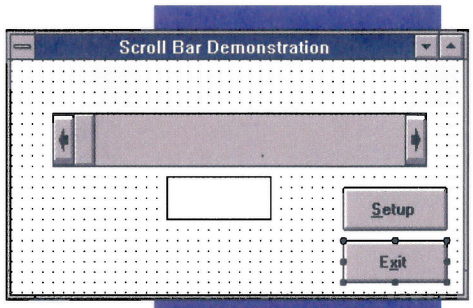


Figure 5-27
Adding command buttons to the form

5 Add two command buttons (Figure 5-27). In the Properties window, make these changes:
   - Caption property: &Setup and E&xit.
   - Name property: cmdSetup and cmdExit.

6 In the Code window, add this code to cmdSetup_Click:

```
hscroll1.Min = 1
hscroll1.Max = 100
hscroll1.SmallChange = 1
hscroll1.LargeChange = 10
```

7 Add this code to cmdExit_Click:

```
End
```

8 Run the program. Before clicking on the Setup button, try the scrollbar. Click in the channel, then click on the scroll arrows. Move the scroll box by dragging it along the scrollbar. Click and hold on the scroll arrows. Eventually, enough SmallChanges will occur and you will see the scroll box move.

9 Click on the Setup button. Test the scrollbar as you did in step 8.

10 Click on the Exit button to stop the program.

11 Whenever you click the scrollbar, a Change event occurs. Code in the Change event handler executes each time the scrollbar is clicked. Insert this code in HScroll1_Change():

```
lblValue.caption = hscroll1.value
```

12 Run the program, click on the Setup button, and repeat the tests of step 8.

13 Exit the program

14 Change the Setup code as follows:

```
Dim x As Integer
hscroll1.Min = 1
hscroll1.Max = 1000
hscroll1.SmallChange = 1
hscroll1.LargeChange = 100
For x = 1 To 1000
   hscroll1.Value = x
Next x
```

15 Run the program. This time, the scroll box is moving under program control. The **For-Next** loop in the code for the Setup button

is changing the value of the scrollbar's Value property. The scroll box moves as the value changes.

16 Exit the program.

QUESTIONS AND ACTIVITIES

1. Describe the significance of the Value property of the scrollbar. Mention both its input role and display functions.
2. How do you set the minimum and maximum values of the Value property?
3. Write the statement needed to use the value of a scrollbar named hscroll1, as the number assigned to the integer variable *Number*.
4. Typically, code attached to a command control is put into its Click event procedure. When attaching code to a scrollbar, you should put it in what event procedure?
5. How would you use the LargeChange event to control the display of a list given that 12 elements of the list fill a full screen?

6

Section

The Who's in Sports? Project

Photographers for school annuals are often assigned to take pictures of members of a specific class. For this project, the photographers are looking for members of one class who play sports. To figure out which students in the class play, they developed this project.

The Who's in Sports? project opens two files: a class list and a list of athletes. Both lists are displayed. The lists are compared and a third list created. The third list contains the names of the athletes also appearing in the class list. The third list shows the names of students who need their picture taken by the class photographer.

The Who's in Sports? program uses three listboxes to display a:

- Class list of student names
- List of names of athletes
- List of athletes who are students in the class

Listboxes are not only convenient ways to display lists of strings, they also provide easy access to those strings if further processing is needed.

The program introduces the statements and functions needed to open, process, and save the files used to store the lists described above. The program uses a constant to facilitate changes to the program and simplify program testing. Finally, the program uses nested **For** loops to process the names in the lists.

Starting Out

Before you can consider the design of a form, you need to create a class list as well as a list of student athletes. If you were working on this project professionally, you would be given these lists. A class list, for example, normally comes from the school administration. In this case, you need to create the lists along with a new directory for this project.

The class list and the list of athletes are provided in files on the disk that accompanies the text. The class list is in a file named **Clsslist.txt**. The file of athletes is named **Sptslist.txt**. These lists would normally come from the school administration, but you could use MiniEdit or the Notepad to create "custom" lists. The first step in this project is creating a directory to hold the project files and transferring these lists to that directory.

To prepare the lists and create the directory:

1. Open the File Manager.
2. Find a directory to which to attach your new directory.
3. Create a directory named **Athlete**.
4. Transfer the files **Clsslist.txt** and **Sptrlist.txt** from the disk that accompanies this book, or from a disk or network directory prepared by your instructor, to the **Athlete** directory.
5. Close the File Manager.

Designing the Form

You have three lists of students to display:

- Class list
- Athletes
- Members of class who play on a sports team

You can use listboxes to display each of these lists of names. Although textboxes could be used, they do not lend themselves well to the tasks performed by the Who's In Sports? program. It is very easy to loop through the items in a listbox and use each one as a string; it is clumsier to do this with the lines appearing in a textbox.

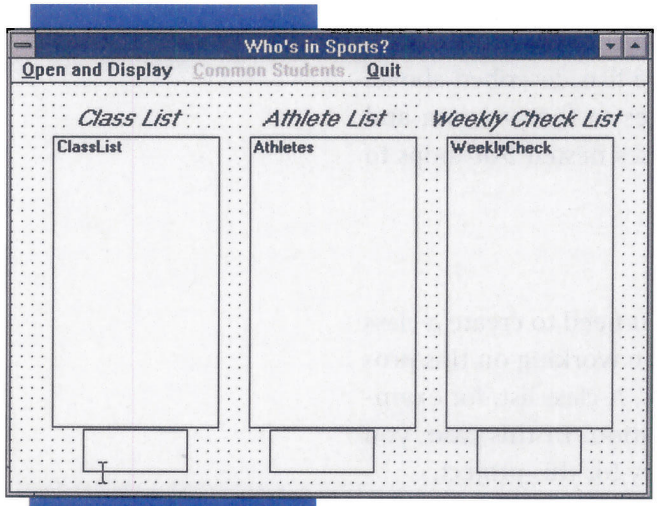


Figure 5-28
Who's in Sports project

To identify the information displayed, you need three labels above the listboxes. In addition, you would like to keep track of the number of student names displayed in each list. For this purpose, you need another three labels. These last labels will display information provided by the program during run-time.

The proposed form is shown in Figure 5-28. As you can see, it also includes a menu bar. With the menu bar, users can display the lists, calculate the overlap between lists, and quit the program.

You are now ready to create the form. To get started:

1. Start Visual Basic, or select New Project from the File menu, if Visual Basic is already running.
2. Change the form's caption to **Who's in Sports?**

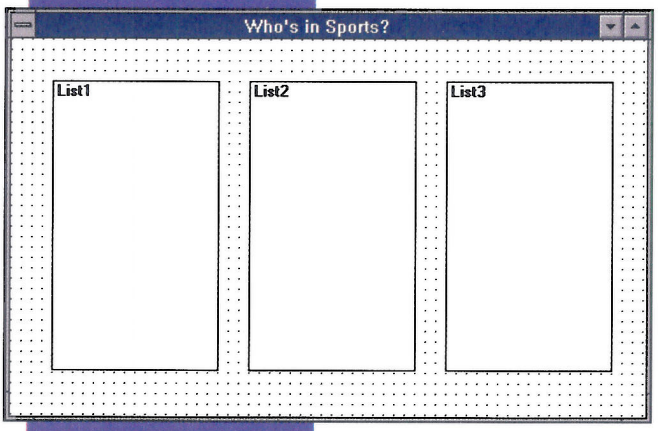


Figure 5-29
Three listboxes on the form

Placing the Objects

To place the objects you need, follow these steps.

1. Place three listboxes, evenly spaced on the form. Leave room above each list for a label (Figure 5-29).
2. Change the names of the lists to:
   a) ClassList
   b) Athletes
   c) WeeklyCheck
3. Place explanatory labels above each list.
4. Below each list, place labels to display the number of people listed. In the Properties window for each control, change these properties:
   - Name property: **lblCLCount**, **lblACount**, and **lblCheckCount**.
   - BorderStyle: Fixed Single.
   - Caption property: Delete default.
   - Alignment: Center.

Sharing Variables and Using Consts

As you will see later, the main work of this program is performed by nested **For** loops. Whenever you want to step through deeply nested code to debug it or verify that it is working properly, your job will be much easier if the loop limits are small. You can achieve this goal in the Who's in Sports? program by imposing an upper bound on the number of names that can appear in the lists. Another way of describing this number is: the number of lines that are read from each file. Clearly, then, this number will be used in more than one procedure of this program.

But what value should be used for the maximum number of names? In order to test the program, you would want this value to be small—for example, 5. Suppose you write the program using the numeric constant 5 everywhere as the maximum number of names. After you test the program and determine that it works as it ought to, you will want it to be able to handle lists of a more realistic size—say, 100. Now you are confronted with an unpleasant task. You have to search for every occurrence of the numeric constant 5 and replace it with the new value.

In this program, doing so would not be hard. But suppose you were working with a program that used 5 in completely unrelated ways. For a concrete example, imagine a program that reads in a list of cities and their temperatures in Farenheit, computes the corresponding Celsius temperatures, and displays a list of the cities and their temperatures expressed in both ways.

The formula to convert from Farenheit to Celsius uses the number 5. In this temperature conversion program, you would not want to change every occurrence of 5 to 100, because doing so would change the conversion formula. To be safe, you would instead have to examine every occurrence of 5, determine whether it means "the maximum number of lines to read," and change it only if it does have that meaning. This process is boring, tedious, and error-prone.

A much better approach is to give a name to the constant value, and use that name in your code. This way, you can use a name like *MAXLINES* wherever you wish to refer to the maximum number of lines to be read. The name tells you its significance immediately; you don't have to guess or hesitate as you do when you see code that contains "magic numbers" like 5 or 100.

Visual Basic lets you declare names for constant values with the **Const** declaration. You declare *MAXLINES* to have the unchanging value 5 by using the following statement:

```
Const MAXLINES = 5
```

To change the value of *MAXLINES*, you only have to edit this statement.

CONSTANTS

A constant names a value that cannot be changed while a program runs. Unlike variables, whose values can be changed, once a constant is set, its value never changes while the program is running. The **Const** statement declares a constant and gives it a value. Here are some examples:

```
Const HousesOfCongress = 2
Const PI = 3.1415927
Const MAXLINES = 5
Const DEFAULT_NAME = "John Doe"
Const ERROR_MESSAGE = "The value that you
enter here must be positive."
```

Constant names are often written in all caps to make them easy to identify. The data type of the constant is the simplest type consistent with the expression used to define the constant. For example, 2 is an integer. Although it *can* be treated as a floating point number, Visual Basic assumes that you intend to define an Integer constant when you initialize a **Const** name with the numeric constant 2.

You can also give names to constants of nonnumeric data types. The last two examples illustrate the declaration of names for string literals.

Using the General Declarations Section of a Form

A final problem remains to be solved: where should the **Const** declaration of *MAXLINES* be put? One option would be to place it in every procedure that used the value. While that would be an improvement over using numeric constants, you would still have to search for all occurrences of **Const MAXLINES** in order to change the value. If you overlook one declaration, different procedures will be using different values of *MAXLINES*—a subtle bug that could be difficult to detect. Ideally, you want to declare *MAXLINES* in one and only one place, and have all procedures use the value it is given there.

If you are going to have only a single declaration of *MAXLINES*, then it cannot be put inside any event procedure. The value of each variable

or constant name used in an event procedure is *local* to that procedure. A variable declared in cmdCalculate_Click is not available in cmdCalculate_KeyPress. When the cmdCalculate_Click procedure is over, any values used within the procedure are lost. Constant names declared within one procedure are not visible to other procedures.

To share a name or variable between more than one event procedure, you declare it in the general declarations section of the form. Names and variables declared in this section are available to every event procedure of every object on the form (Figure 5-30).

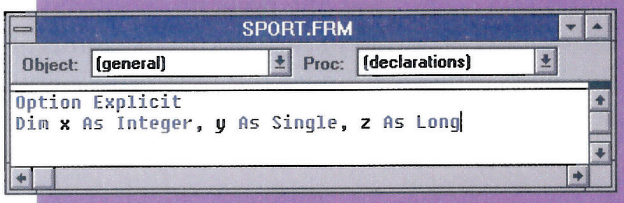


Figure 5-30
Declaring variables in the general declarations section of the form

The values of *x*, *y*, or *z* can be set, accessed, or changed in any event procedure throughout the form.

The Who's in Sports? project uses a constant declaration in the general declarations section of the Code window. Placing the declaration here makes the value of the constant available to every event procedure in the form.

To declare the constant:

1. In the Project window, click on the View Code button to open the Code window.
2. In the Object drop-down list, select the top entry *(general)*. Notice that the Proc drop-down list now displays *(declarations)*.
3. Enter the constant definition shown in Figure 5-31.

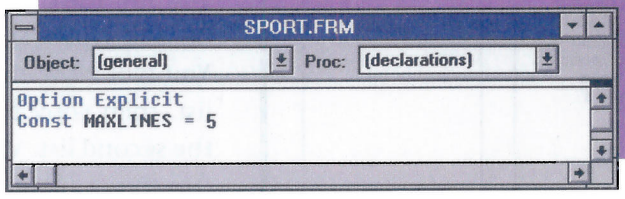


Figure 5-31
Defining a constant in the general declarations section

The **Const** statement defines *MAXLINES* as a constant value. *MAXLINES* is the maximum number of lines that the program will read from each of the two files. The value of 5 makes it easy to test the program; you can increase the value later. *MAXLINES* has the Integer data type.

Designing the Menu

You are going to create a simple menu bar, with three choices and no drop-down menus. You want users to be able to:

- Open and display the ClassList and Athletes lists.
- Display a list of students who are in both lists.
- Quit the program.

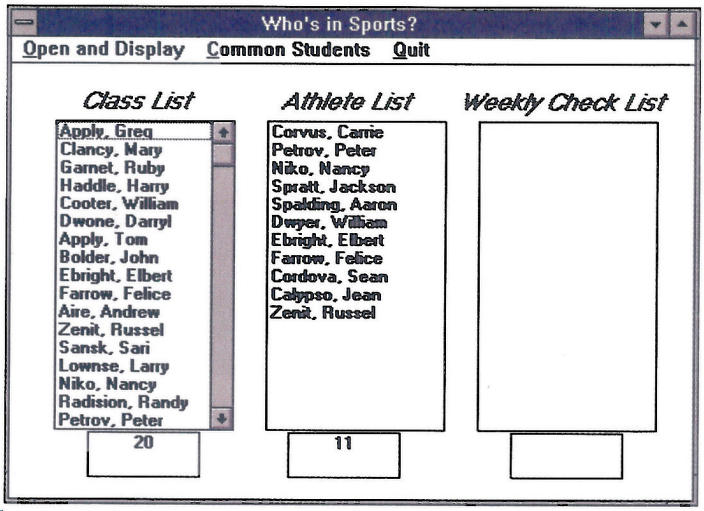
Below the first two listboxes are labels. When you click on the Open and Display command, the program should display the lists, then count the number of entries in each list. These counts of students are shown in the labels. The label under the final listbox displays the percentage of athletes in the class.

To create the menu bar:

1. Select the form by clicking on the body of the form.

2. Click on the Menu Design icon in the toolbar. This opens the Menu Design window.
3. In the Caption textbox, type **&Open and Display**. Then in the Name textbox, type **mnuOpen**.
4. Click on the Next button to clear the entries and repeat step 3 for the other two menu commands.
   - Caption: **&Common Students**; Name: **mnuCommon**
   - Caption: **&Quit**; Name: **mnuQuit**

Figure 5-32
Who's in sports? program shown with data in listboxes



5. Click OK to close the Menu Design window.

Coding the Open and Display Command

You are now ready to write the code for the Open and Display command. You will write the code to display one list, then copy those lines of code for the second list. When you are finished, run the program to test the code (Figure 5-32). Stop the program with the Stop button in the toolbar.

To create the code for the Open and Display command:

1. Click on the command Open and Display to open the Code window.
2. Enter this code for mnuOpen_Click. This opens a file, reads each line from the file, displays each line in a listbox, and closes the file.

```
'–first part handles clsslist.txt
'–Open file
Open "c:\athlete\clsslist.txt" For Input As #1
'–Read lines
Dim i As Integer
Dim strLine As String
For i = 1 To MAXLINES
    If EOF(1) Then Exit For
    Line Input #1, strLine
    strLine = Trim$(strLine)
    If Len(strLine) <> 0 Then ClassList.AddItem strLine
Next i
'–Close file
Close #1
lblCLCount = ClassList.ListCount
```

3. Select all the lines of code and copy them. Use Ctrl+C or the Edit menu.
4. Paste the lines of code below the existing code. In this new section:
   - ◎ Replace references to **clsslist.txt** with **sptslist.txt**.
   - ◎ Replace ClassList with Athletes.
   - ◎ Replace lblCLCount with lblACount.
   - ◎ Delete the duplicate declarations of *i* and *strLine*.

In the code you just entered, the **Open** statement opens the **clsslist.txt** file for input. The file number 1 is assigned to the file. In the **For-Next** loop, the ending value, *MAXLINES*, puts an upper limit on the number of names that may be in a file.

The **EOF(n)** function is true when the end of the file has been reached. It is false otherwise. When the end of file is reached, **Exit For** is executed. This statement halts the **For** loop whether the upper limit has been exceeded or not.

The **Line Input #** statement reads an entire line of text from the text file assigned to the file number. The line is assigned to the string variable *strLine*, then the line is trimmed.

If the length of the line is not 0, it is added to the ClassListbox. The **AddItem** method adds a new entry to a listbox. If the Sorted property is false, a new item is added to the bottom of the list.

The number of items in the list is displayed in the lblCLCount label. The number of items in the list is read from the ListCount property of the listbox.

Coding for the Quit Command

In a number of other projects so far, you have added very simple code for an Exit button or Quit command. This allows the program to end when the command is selected or the button pushed. You will follow the same procedure here:

1. Open the Code window and select the mnuQuit_Click procedure.
2. Enter **End**.

Coding for the Common Students Command

This procedure for this command compares every student name of the class list with every student name of the athlete list. Nested loops control the comparisons. The outer loop reads each name from the class list. The inner loop reads each name from the athlete list. If the names match, the name is put into the common list.

To create this procedure:

1. Click on Common Students to open the Code window.
2. In the mnuCommon_Click procedure, enter this code:

```
'—Compare two lists and write the third
Dim i As Integer, j As Integer
For i = 0 To ClassList.ListCount - 1
    For j = 0 To Athletes.ListCount - 1
    If StrComp(ClassList.List(i), Athletes.List(j)) = 0 Then
       WeeklyCheck.AddItem ClassList.List(i)
    End If
    Next j
Next i
'—Calculate and display Percentage
lblCheckCount = WeeklyCheck.ListCount * 100 / ClassList.ListCount & " %"
```

ADDRESSING THE ITEMS OF A LISTBOX

Each item of a listbox has a number. The first item of the list is number 0. The ListCount property shows the number of items in the list. The last item of the list is number ListCount - 1. The *i*th item of the list can be accessed as ListName.List(*i*). See Figure 5-33. This value is a string and can be used in string comparisons and operations.

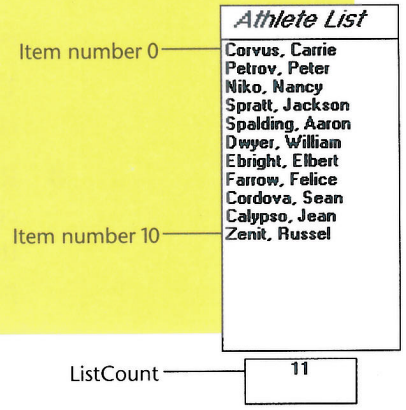


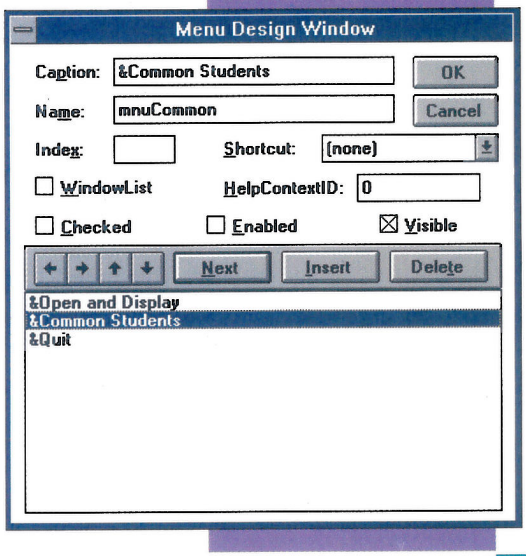
Figure 5-33
ListCount property of items in a listbox

Finishing the Project

Running the program and clicking on the Common Students command causes the two files, the class list and the list of athletes files, to be opened, read, and compared. The names that appear on both lists appear in the third list. After you run the program, you will change the lists and run the program again to see the results. Once the program has been tested and *MAXLINES* has been set to a more reasonable upper bound, the program is ready to use.

1. Run the program. Click on Open and Display. Click on Common Students.
2. Exit the program. Click on one of the listboxes. In the Properties window, change the Sorted property to True. Run the program.
3. Exit the program. Save the project in the Athlete subdirectory.
4. Change the *MAXLINES* constant to allow larger files and run the program again. Use NotePad or MiniEdit to add names to either or both files and test the program again.
5. Edit the program and reenter the Menu Design window. The form must be selected to enter this window.
6. Change the Enabled property of Common Students to false (Figure 5-34).
   When the program is run, the Common Students menu item is dimmed. It doesn't make sense to pick the common students before the files are read and the lists prepared.

Figure 5-34
Changing the Enabled property to false



7 Add this line to the end of the mnuOpen_Click procedure.

```
mnuCommon.Enabled = True
```

This statement removes the dimmed **Common Student** command.

8 Run the program. Click on the dimmed **Common Students** command. Click on the Open and Display command. Now that the Common Students command is active, click on it. Exit the program.

Section

The Two-Point Problem

The Two-Point Problem doesn't introduce any new features of Visual Basic. Instead, it shows how Visual Basic can be used to solve a common high school math problem. Any programming language is a tool to be used to solve problems. Visual Basic is an excellent tool to use in almost any class. This problem, finding the properties of a line that passes through two points, with coordinates supplied by the user, can be solved with simple textboxes, command buttons, and labels. The code is no more complicated than an **If-Then-Else** statement. Despite the simplicity of the program (from a programming point of view), it accomplishes a lot.

Starting Out

Computers can calculate far faster than humans. As a result, one of the common uses for computers is to make mathematical calculations. The Two-Point problem illustrates how you can use a Visual Basic program to solve a fairly simple mathematical problem. The user defines the positions of two points. The program then calculates the distance between these two points. As part of that process, it also displays information about the line drawn between the points, including its slope and midpoint.

You may need a refresher on the algebraic concepts used in this program. This information is provided in the following "Background" section. If you are already familiar with concepts such as coordinate systems and the Pythagorean theorem, you can turn immediately to the section titled "Designing the Form."

Background

Understanding the math discussed in this section is necessary to understanding the program you are about to build. Use this section for review of algebraic concepts, if necessary.

COORDINATE SYSTEMS

The Two-Point program graphs a line segment on a coordinate system. The *x* and *y* axes are the number lines that define the plane's coordinate system. These lines are at right angles to each other. The point where the lines cross is called the origin. With the axes in place, you can define the position of any point in the plane with an *x* and *y* coordinate.

For example, look at Figure 5-35.

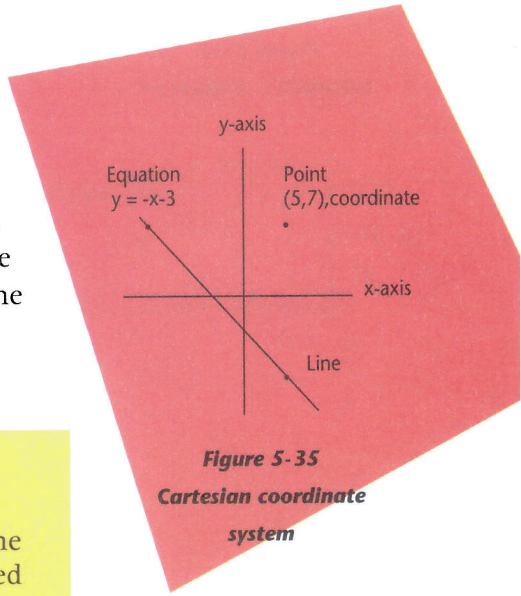


Figure 5-35
Cartesian coordinate system

DEFINITIONS

The point is a fundamental geometric concept. Algebraically, the position of any point on a plane is assigned a unique ordered pair of numbers.

A line is understood to be an infinite collection of points "all in a line."

Two points determine a line. The equation of a line describes the relationship between *x* and *y*, the coordinates of the points of the line.

A line segment is a section of a line with distinct endpoints.

Two intersecting lines determine a plane. A plane is a surface, like a table top.

THE PYTHAGOREAN THEOREM

The Pythagorean theorem is a relationship between the sides of a right triangle. A right triangle contains a right angle, as shown in Figure 5-36.

The Pythagorean theorem states that the length of the hypotenuse—the side opposite the right angle—can be computed from the lengths of the other two sides forming the right angle. The relationship is in fact an identity that is used to compute the length:

$$c = \sqrt{a^2 + b^2}$$

Here, *c* is the length of the side opposite the right angle, and *a* and *b* are the lengths of the sides forming the right angle.

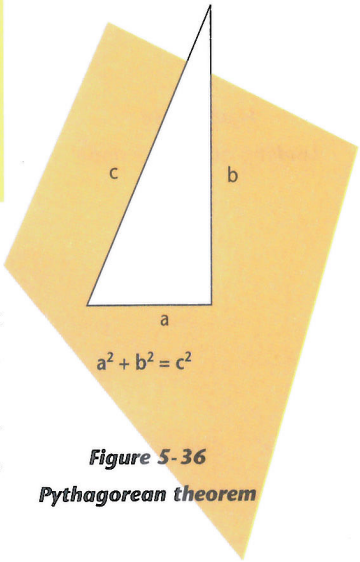


Figure 5-36
Pythagorean theorem

This relationship, known for thousands of years, has numerous proofs. It has been used for building and surveying since the time of the ancient Egyptians.

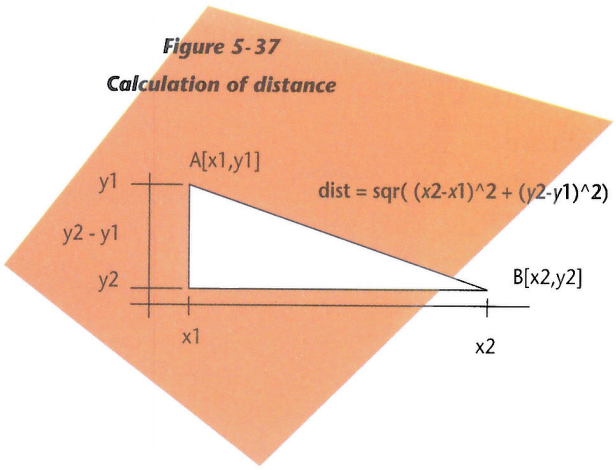


Figure 5-37
Calculation of distance

DISTANCE FORMULA

The distance between two points is calculated using a variation of the Pythagorean theorem, called the Distance formula. If the coordinates of point A are (*x1,y1*) and the coordinates of point B are (*x2,y2*), this distance is calculated as shown in Figure 5-37.

The distance between the points *A* and *B* is given by the equation (written in BASIC notation):

```
Dist = Sqr ((x2 - x1)^2  +  (y2 - y1)^2)
```

where the caret symbol, ^, indicates "raising to a power," and **Sqr(n)** is the square root function. **Sqr(n)** takes a positive number and returns the square root of the number.

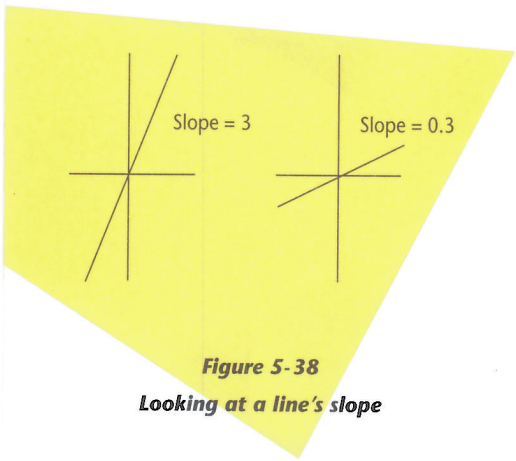


Figure 5-38
Looking at a line's slope

SLOPE

The slope of a line is its slant. If the slope is greater than 1, the slant is steep. If the slope is less than 1, but greater than 0, the slant of the line is "flatter." See Figure 5-38.

The slope is calculated by taking a ratio:

1. Mark two points somewhere along a line.
2. The difference in the heights of the points marked is called the "change in *y*."
3. The distance between the points in the right to left direction is called the "change in *x*."
4. The slope of the line is the ratio of the "change in *y*" to the "change in *x*." See Figure 5-39.

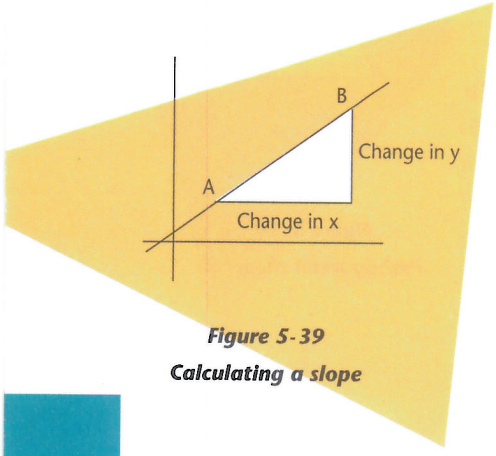


Figure 5-39
Calculating a slope

The slope of a line between the points A(*x1,y1*) and B(*x2,y2*) is:

```
slope =  ( y2 - y1 ) / ( x2 - x1 )
```

MIDPOINT

A line segment not only has a length, defined by the distance between the two endpoints; it also has a midpoint. The midpoint of a line is the point half way between the endpoints of the line segment. See Figure 5-40.

The coordinates of the midpoint of a line segment with endpoints A(*x1*,*y1*) and B(*x2*,*y2*) are given by:

((*x1*+*x2*)/2 , (*y1*+*y2*)/2)

The coordinates of the midpoint are the averages of the coordinates of the two endpoints.

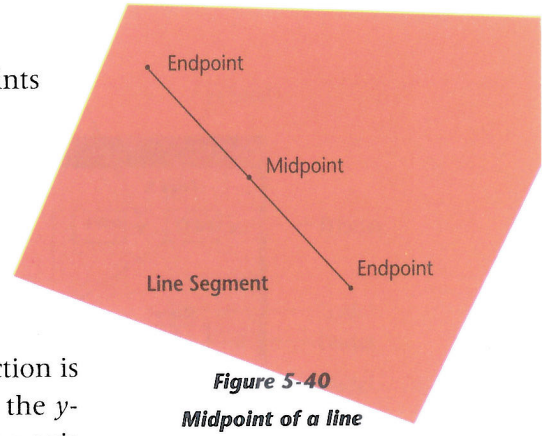


Figure 5-40
Midpoint of a line

INTERSECTING LINES

Any line that is not parallel to the *y*-axis intersects it. An intersection is a point where two lines cross. The point at which a line crosses the *y*-axis is called the *y*-intercept. The point at which a line crosses the *x*-axis is the *x*-intercept.

THE Y-INTERCEPT

The slope-intercept form of a line's equation:

```
y = mx + b
```

shows both the value of the slope of the line and the coordinate of the *y*-intercept. The *m* variable represents the slope and *b* is the *y*-intercept of the line. You can rearrange the equation to give an expression for the *y*-intercept:

```
b = y - mx
```

Once the user has provided the coordinates for two points, the program you are building will calculate and display the equation of the line in slope-intercept form ($y = mx+b$).

Designing the Form

On this form, you will need:

- Four textboxes for the user to enter *x* and *y* coordinates of two points
- Twenty labels for the slope, y-intercept, distance, midpoint and equation of the line containing the two points
- Three command buttons to make calculations, clear the textboxes and labels, and stop the program

You should set up the form similarly to that shown in Figure 5-41. To create the form:

1. Open Visual Basic or, if the application is open, select New Project from the File menu.

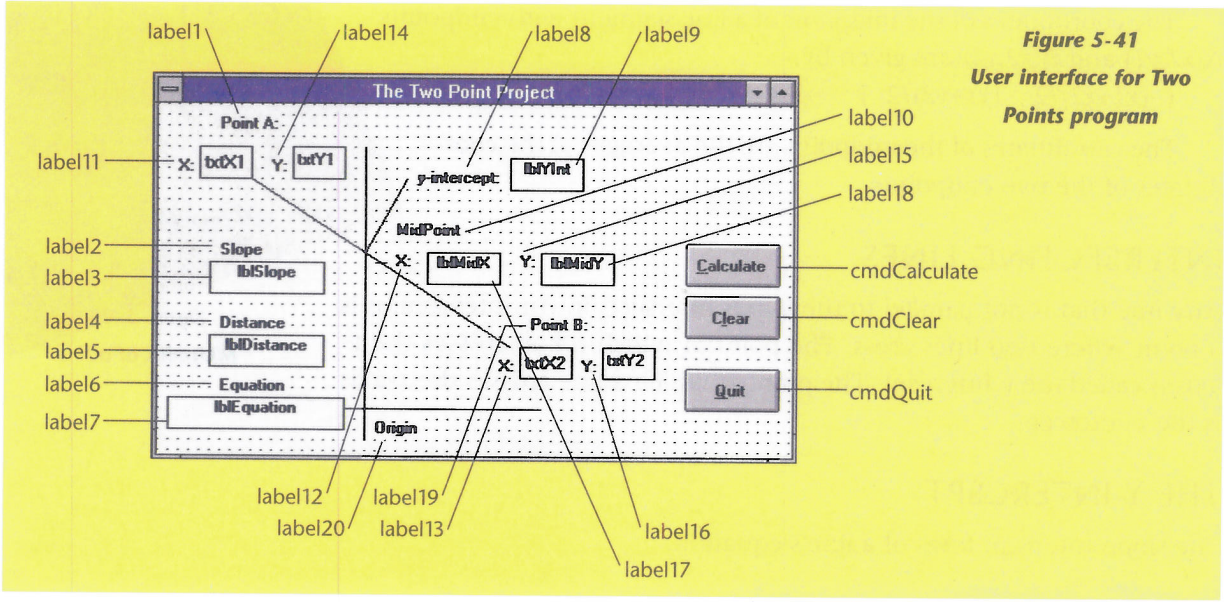


Figure 5-41
User interface for Two Points program

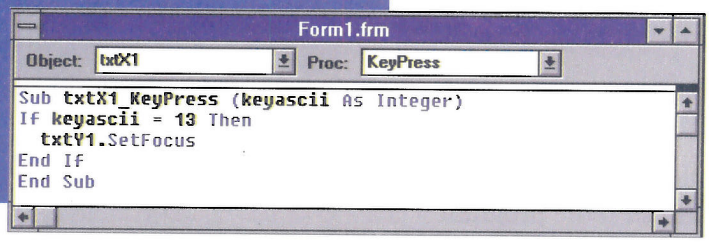
2 Place the objects on the form, as shown in the previous figure.

3 In the Properties window, make the changes shown in Table 5-1.

Moving Focus Among the Textboxes

Users enter four values in this form: the *x* and *y* coordinates for both endpoints of the line. To enter these values, they need to be able to move from one textbox to the next. For each textbox, then, you need to insert code into the KeyPress event procedure. This code instructs the program to move the focus to the next textbox when the user presses Enter. For an example of the code, see Figure 5-42. How does this code need to change for the other textboxes?

Figure 5-42
Moving focus among the textboxes



Writing Code for the Calculate Button

To write the code for this button:

1 With the Code window open, select cmdCalculate.

Table 5-1 Properties for Objects

Object	Properties to Change
Text1	Text: delete text
	Name: txtX1
	FontSize: 9.75
Text2	Text: delete text
	Name: txtY1
	FontSize: 9.75
Text3	Text: delete text
	Name: txtX2
	FontSize: 9.75
Text4	Text: delete text
	Name: txtY2
	FontSize: 9.75
Label1	Caption: Point A:
	AutoSize:True
Label2	Caption: Slope
	AutoSize:True
Label3	Name: lblSlope
	Caption: delete caption
	BorderStyle=1
	Alignment:Center
Label4	Caption: Distance
	AutoSize:True
Label5	Name: lblDistance
	Caption: delete caption
	BorderStyle=1
	Alignment: Center
Label6	Caption: Equation
	AutoSize:True
Label7	Name: lblEquation
	Caption: delete caption
	BorderStyle=1
	Alignment: Center
Label8	Caption: y-intercept:
	AutoSize: True
Label9	Name: lblYInt
	Caption: delete caption
	BorderStyle=1
	Alignment: Center
Label10	Caption: Midpoint
	AutoSize: True
Label11	Caption: X:
	AutoSize: True
Label12	Caption: X:
	AutoSize: True
Label13	Caption: X:
	AutoSize: True
Label14	Caption: Y:
	AutoSize: True
Label15	Caption: Y:
	AutoSize: True
Label16	Caption: Y:
	AutoSize: True
Label17	Name: lblMidX
	Caption: delete caption
	BorderStyle=1
	Alignment: Center
Label18	Name: lblMidY
	Caption: delete caption
	BorderStyle=1
	Alignment: Center
Label19	Caption: Point B:
	AutoSize: True
Label20	Caption: Origin
Command1	Name: cmdCalculate
	Caption: &Calculate
Command2	Name: cmdClear
	Caption: &Clear
Command3	Name: cmdQuit
	Caption: &Quit

2 Insert the following code in the Calculate_Click procedure:

```
Dim x1, x2, y1, y2
Dim Slope, YIntercept, Distance, MidX, MidY
Dim Equation As String
'—transfer text and convert to value
x1 = Val(txtX1)
y1 = Val(txtY1)
x2 = Val(txtX2)
y2 = Val(txtY2)
'—check to see if slope exists
If x2 - x1 <> 0 Then  ' slope exists
    Slope = (y2 - y1) / (x2 - x1)
    lblSlope = Format$(Slope, "fixed")
'—y-intercept
    YIntercept = y1 - Slope * x1
    lblYInt = Format$(YIntercept, "fixed")
'—equation
    Equation = "y = " & Format$(Slope, "fixed") & "x + " & Format$(YIntercept,
"fixed")
    lblEquation = Equation
Else
    lblSlope = "None"
    lblYInt = "None"
    lblEquation = " x = " & Str$(x1)
End If
'—distance
Distance = Sqr((x2 - x1) ^ 2 + (y2 - y1) ^ 2)
lblDistance = Format$(Distance, "fixed")
'—midpoint
MidX = (x1 + x2) / 2
MidY = (y1 + y2) / 2
lblMidX = Str$(MidX)
lblMidY = Str$(MidY)
'—move focus to Clear button
cmdClear.SetFocus
```

Now let's look back at the code we just created. The four lines of code after the variable declarations read entries from textboxes and convert those entries to the values they represent. *x1* and *y1* represent the coordinates of the first point. *x2* and *y2* represent the coordinates of the second point. The two points define a line segment.

Then comes the **If-Then-Else** statement. If the change in the *x*

coordinates, from one point to the next, is any number other than 0, the True branch of the statement is executed.

The line:

```
lblSlope = Format$(Slope, "fixed")
```

displays the slope value in the appropriate label object. Values are assigned to labels to display them on the form. The **Str$(n)** function has been used to convert values to strings. In this program, the **Format$(Value, FormatString)** function is used. Values sent to this function are formatted according to the format string. Predefined format strings let you use a variety of formats. **Fixed** converts a number into a string with two decimal places.

The False branch of the **If-Then-Else** statement displays "None" in the appropriate label objects to indicate that the line has no slope and no y-intercept. It also displays the special equation for the line, given that its slope is 0.

The rest of the code calculates the distance between the points as well as the x and y coordinates of the midpoint. These values are displayed, then focus is shifted to the Clear button.

Coding the Clear Button

You need code that clears all the textboxes and labels of their contents if a user clicks on the Clear button. Clicking on this button should also move the focus back to the X1 textbox. A user can then enter another set of coordinates for the two endpoints and recalculate the midpoint, y-intercept, and so forth.

The empty string, "", is used to blank textboxes and labels. Because the Text property is the default property for a textbox, and Caption is the default for the label, just the names of the controls are used in expressions. For example, we use:

```
txtX1 = ""
...
x1 = Val(txtX1)
```

instead of:

```
txtX1.Text = ""
...
x1 = Val(txtX1.Text)
```

Likewise, we use:

```
lblSlope = "None"
```

instead of:

```
lblSlope.Caption = "None"
```

Running the Program

Run the program. Use the sample data provided in the following table:

Trial #	*X1*	*Y1*	*X2*	*Y2*
1	3	4	2	5
2	3	-4	2	5
3	3	-2	3	5
4	-2	-6	-2	8
5	-2	4	5	4
6	1.2	3.2	-4.1	2.3

Designing good test data is a very important part of program debugging. Your test data should systematically test each part of the program. Every possible outcome of every comparison should be executed by at least one set of input values.

QUESTIONS AND ACTIVITIES

1. Why is it important to use the **Val(str)** function to convert strings to values when assigning those values to a variant-type variable?
2. Why is it a problem to calculate the slope of a line when the "change in y" is 0?
3. Assume a student's average is 72 in the third week of class and 84 in the fifth week. Use the Two-Point program to help you predict the student's average in the seventh week. (Hint: use (3,72) and (5,84) as your points.)
4. In the equation generated from the information in the problem above, what is the significance of the y-intercept?
5. In the third week of track practice, a shot putter tossed the shot put 42 feet. In the fourth week, the shot putter threw 44 feet. Use the Two-Point program to find the equation that describes this situation.
6. How are the **Str$(n)** function and the **Format$(n)** function alike? How are they different?

Strings are series of characters. The empty string is shown as ""; this represents a string with no characters. Strings can be compared using the following operators: =, <, >, and <>. When you use the less than and greater than operators, the program compares the strings by comparing the ASCII codes of the characters. "A" is less than "Z", but it is also less than "a". Concatenation is the joining together of strings. The concatenation operator is the ampersand (&).

The **UCase$(str)** and **LCase$(str)** functions convert a string to all uppercase or all lowercase characters. Characters that do not represent letters are not affected.

The **StrComp(a$, b$, 1)** function compares the two strings *a$* and *b$* and returns:

a) -1 if *a$* is less than *b$*
b) 0 if the strings are equal
c) 1 if *a$* is greater than *b$*

The **Len(str)** function takes a string as a parameter and returns a whole number representing the number of characters in the string.

The **Format$(n,format string)** function determines the appearance of values displayed. It displays values with a particular number of decimal places, or a percentage sign or commas are included in the display.

Format$(x, "Fixed") displays the value of *x* with at least one digit to the left of the decimal point and two digits to the right.

The Scrollbars property of a textbox adds a horizontal or a vertical scrollbar to a multiline textbox. You can also choose to add both of these scrollbars. The textbox control itself handles the behavior of the scrollbars; unlike scrollbar controls, you do not have to write code for the textbox scrollbars to work.

The syntax for a **For-Next** statement is:

For *control variable* = *expression* **To** *expression* **Step** *expression*

...

Next *control variable*

where *expression* represents a constant, a variable, or an expression (such as T+1).

Loops peform sections of code repeatedly, using a different value of some variable on each repetition. Loops can be nested inside one another.

A scrollbar can be used for output only, to display the value of a variable within the program; or it can be used to let the user input values to

the program by moving the graphic scroll box. The Value property provides the connection between a numeric value and the position of the scroll box. The Min and Max properties control the minimum and maximum value the scrollbar can assume.

The SmallChange and LargeChange properties of the scrollbar control what happens when a mouse is clicked on various parts of the scrollbar.

The dimensions of a textbox are determined by the settings in the Height and Width properties of the box. The dimensions of a form are determined by its ScaleHeight and ScaleWidth properties.

When the dimensions of a form are changed, a Resize event occurs.

The syntax of the **InputBox** function is:

variable name = InputBox (prompt string, box title string, default value)

Use the **Open** statement to open a file for input or output.

LOF(n) stands for *l*ength *o*f *f*ile. The function returns the length in bytes of the file at that file number, *n*.

The **Input$(FileSize, 1)** function returns a string of *FileSize* length, from the file with file number *1*.

The **Print# n** statement writes a string to the file with file number *n*.

The **ChDrive** statement changes the current drive to the drive indicated in the string that follows.

The **ChDir** statement changes the current directory to the directory indicated in the string that follows.

The **App** object is the currently running application.

The listbox is a control that contains a list, to which items are added using the **AddItem** method. The list can be scrolled, the number of display columns changed, using the control's Columns property. You can maintain the list in sorted order by setting the Sorted property.

1. Menu Calculator

Write a program with the following menu structure:

Calculation
.....Add
.....Subtract
.....Multiply
.....Divide
Exit

When an operation is chosen, use the **InputBox** function to collect

each of the two numbers from the user, then perform the operation. Display the result of the operation in a textbox on the form.

2. Order Please

Write a program using three textboxes to enter three names of animals. Put the names in alphabetical order using **StrComp** functions in **If-Then** statements. Display ordered names in a textbox.

3. Random Numbers

Use a **For-Next** statement to generate 100 random numbers between 0 and 1. The **Rnd** function generates a random number between 0 and 1. Insert the values in a listbox. Run the program. Change the Sorted property of the listbox to True. Run the program again.

4. Random Number File

Use a **For-Next** statement to generate 100 random numbers between 0 and 1. Write the numbers, converted to strings, into a file named **c:\temp\random.dat**.

5. Pythagorean Triples

Write a program to find and display a table of Pythagorean triples. A Pythagorean triple is a set of three integers that satisfy the Pythagorean theorem. The Pythagorean theorem, which applies only to right triangles, is "The length of the hypotenuse squared is equal to the sum of the squares of the lengths of the other two sides."

Use three nested loops, one for *a*, one for *b*, and one for *c*, where $a^2 + b^2 = c^2$.

Let the value of *a* range from 1 to 30. Start the *b* loop at *a+1* and end it at 30. Start the *c* loop at *b+1* and end it at 60.

6. Pythagorean Add-On

Write an addition to the program above to verify that the product of the three numbers forming a Pythagorean triple are divisible by 60. Use the **Mod** operator to test divisibility.

7. The Universal Gravitation Problem

Every mass exerts a gravitational attraction for every other mass. The force of this attraction is given by the formula:

$$F = \frac{Gm_1, m_2}{r^2}$$

where F represents the attractive force, G is the Universal Gravitation Constant, m1 and m2 are the two masses, and r is the distance between the masses.

As the masses increase, the force increases. As the distance increases, the force decreases.

The acceleration of gravity at any given elevation can be calculated from this formula. By Newton's law, $F=ma$, "force" equals mass times acceleration. If this expression is substituted for F in the formula above, it becomes:

$$a = \frac{GMe}{r^2}$$

where G is the Universal Gravitation literal, Me is the mass of the earth, and r is the distance from the center of the earth.

Write a program to display the acceleration of gravity for various elevations. Use a **For-Next** statement to generate values for the elevation. Start at 0 and step by 1000-foot increments to a height of 20,000 feet. The value of r is the radius of the earth, plus the elevation.

Display the elevation in feet and the acceleration of gravity in feet per second squared. You may need the following conversion factors:

$G = 1.07 \times 10^{-9}\ ft^3/(lb\ sec^2)$ Gravitation constant

$Me = 1.315 \times 10^{25}\ lb$ Weight of earth in pounds

Radius of the earth = 3955 miles

1 mile = 5280 feet

8. The Factorial Problem

Write a program to calculate and display the factorials of the first 20 numbers in a listbox. N factorial, symbolized as $N!$, is calculated by the following formula:

$N!=N*(N-1)*(N-2)*...*2*1$

9. Drag Race Problem A

The final velocity of an automobile undergoing a literal acceleration of a, is given by the formula:

$v = a\ t$

where t is the time. One of the statistics calculated for a quarter-mile run down the drag strip is the final speed of the vehicle as it passes through the "traps" at the end of the run.

Write a program that prompts the user to enter a value for the desired final velocity expressed in miles per hour. Display a table of accelerations calculated using values of t from 6 seconds to 20 seconds.

$a = v / t$, a is the acceleration, v is velocity, and t is time.

Column one of the table shows the time elapsed, ranging from 6 to 20 seconds. The second column shows the acceleration in feet per second squared. Convert the final velocity from miles per hour to feet per second.

1 mile = 5280 feet

1 hour = 3600 seconds

Use a listbox to show the table.

10. Solving a Pair of Linear Equations

A linear equation in x and y is of the form:

$ax + by = c$

where a,b, and c are real numbers.

Use textboxes to enter the coefficients of two equations:

$ax + by = c$

$dx + ey = f$

The solution to this system of equations includes the values for x and y that satisfy both of the equations.

One way to solve the system of equations mathematically is to use Cramer's rule. Cramer's rule uses discriminants to solve the system.

First calculate the discriminant of the denominator of the x and y values:

$DEN = a*e - b*d$

where a,b,c,d,e, and f are the coefficients of the two linear equations.

If this value is 0, there is no need to proceed. If the value is non-zero, continue with the following calculations:

$x = (c*e-b*f)/DEN$

$y = (a*f-d*c)/DEN$

Display the solution as an ordered pair (x,y)

11. The Distance Between Cities Problem

The distance formula calculates the distance between two points on the x,y plane. If the two points are $(x1,y1)$ and $(x2,y2)$, the distance between the points is given by:

$d = Sqr((x1-x2)^2 + (y1-y2)^2)$

This formula can be applied to find the distance between cities. If the coordinates of the cities are known, the distance between them can be found. The coordinates for the cities are the latitude and longitude for the cities. Find the coordinates for two U.S. cities, then write a program using textboxes to enter the latitude and longitude of the two cities and calculate the distance between the two. To convert the distance from degrees to miles, use the conversion: 1 degree = 69 miles.

```
DispLine =
DispLine & TbCh &
Format$(TotalInt,
"Currency"
AmortTable(Payment
Number) =
DispLine
txtAmortTable(Index)
AmortTable
```