Chapter 12

User-Defined Types and File System Controls

GOALS

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

- Design and implement a plan for a data structure to represent fractions.
- Encode those algorithms in procedures suitable for inclusion in a code module.
- Plan and execute a program that will allow the user to interact with information stored in a file.
- Implement code for maintaining arrays when information is added, deleted, and modified in the array.
- Work with complex user-defined types.
- Integrate the drive, directory, and file boxes into programs that use files.

OVERVIEW

Visual Basic provides a rich variety of data types built into the language. You have already worked with a number of these data types, such as Variant and String. There is a common type of numerical data, however, that is not covered by the built-in data types. Can you think of a data type that is suitable for fractions?

To supply this missing data type, you need to define a user-defined type. Using this data type, you can then create a code module that contains arithmetic functions which operate on fractions represented by the new type. You can then include this code module in any program in which it is more appropriate to work with fractions such as $^{5}/_{12}$ instead of rounded-off floating point numbers such as 0.416666667.

The chapter also looks at the development of a fairly large program. The Multiple Choice Test program is used to create, update, and give multiple choice tests. Along the way, you will see some new statements and use the file management tools included in Visual Basic.

The Arithmetic of Fractions

Dealing with English measurements often requires the use of fractions. Finding the distance between each of four equally spaced shelves in a space 5'8" tall requires fractions. If the answer is provided in decimal form, the tape measure, which divides an inch into fourths, eighths, and sixteenths, will be useless.

In this section, you develop the techniques you need to handle fractions in Visual Basic. You can then use routines built with these expressions in any program in which you need a fraction-handling routine. To reuse the routines, you put them into a code module. Then all you do to use the routines is add the code module to the project.

You will represent fractions with a user-defined type—called Fraction—which stores the numerator and denominator of the fraction separately as integers. This representation makes it simple to write procedures that implement the standard arithmetic operations on fractions.

To work with fractions in a Visual Basic program, you need to be comfortable adding, subtracting, multiplying, and dividing them. You also need to be able to reduce fractions to their lowest terms, finding multiplicative and additive inverses. If you need the review, read this section before you move on to address the issues of coding fraction-handling routines.

Addition

To add fractions, you must first rewrite them so that they have the same denominator. The sum of the fractions is the sum of the resulting numerators, divided by the common denominator. For example, to add ⅓ and ½, rewrite each with the denominator 2 x 3 = 6. You get ⅓ = (2 x 1) / (2 x 3) = ⅔, and ½ = (3 x 1) / (3 x 2) = ⅗. Thus, ⅓ + ½ = ⅔ + ⅗ = (2 + 3) / 6 = ⅚.

In general, if one fractional operand is *a/b* and the second operand is *c/d*, the sum of the fractions is *(ad + cb) / (bd)*. To rewrite the fractions so that they have a common denominator, *bd*, you multiply the first fraction by *d/d*, then multiply the second by *b/b*. Then you add the numerators, and divide by the common denominator *bd*.

The Fraction data type stores the numerator and denominator of the fraction separately as integers. From the result above, the numerator of the sum is *a*d + c*b*, and the denominator is *b*d*. These formulas are used in the addition routine.

Multiplication

You multiply two fractions (such as *a/b* times *c/d*) by multiplying their numerators together, then multiplying the denominators together. The result of multiplying *a/b* and *c/d* is *(ac) / (bd)*. For example, $\frac{2}{5}$ times $\frac{3}{6}$ is (2 x 3)/(5 x 6), or $\frac{6}{30}$.

Inverses

There are two kinds of inverses—the multiplicative inverse, also known as the reciprocal, and the additive inverse, sometimes called "the opposite" of a number. For example, for the number 7, the multiplicative inverse is $\frac{1}{7}$ and the additive inverse is -7.

You calculate the multiplicative inverse of a whole number or fraction by interchanging the numerator and the denominator. Note that you may run into difficulty with a fraction that has a numerator of 0. This type of fraction is perfectly acceptable; but its reciprocal, with 0 as the denominator, is "undefined."

To form the additive inverse of a number, change the sign of the numerator. If the current numerator is positive, make it negative. If the current numerator is negative, make it positive.

Subtraction and Division

After you have defined inverses, you can define subtraction as addition, and division as multiplication. Imagine that you have two fractions, *x* and *y*. To subtract *y* from *x*, you would add the additive inverse of *y* to *x*. In other words:

$x - y$ provides the same result as $x + -y$

Likewise, multiplying *x* by the reciprocal of *y* yields the same result as dividing *x* by *y*. To divide fractions, then, invert the second fraction and multiply:

$x / y = (x)(1/y)$

Why do you need to rewrite subtraction as addition, or division as multiplication? If you do, you will need to create only two routines instead of four. Your addition routine can perform both addition and subtraction, and your multiplication routine can perform both multiplication and division.

Lowest Terms

To build your routines, you also need to know how to reduce fractions to their lowest terms. To do this, find the greatest common factor (gcf)

of the numerator and the denominator. The greatest common factor is the largest number that divides into both numbers evenly. For instance, consider the fraction $^{12}/_{16}$.

A common factor is 2, but the *greatest* common factor is 4. Try dividing the top and bottom of the fraction by 4. Do you get $^{3}/_{4}$? Can the fraction be reduced any further?

Greatest Common Factor

The package of fraction-handling routines you are going to build in this chapter includes a routine to reduce fractions to their lowest terms. This routine uses the Euclidean algorithm to find the greatest common factor. (This is the same Euclid who developed the axioms and theorems that have evolved into the high school geometry course.) This algorithm is based on the definition of division. For nonnegative integers x and y, dividing x by y means finding two numbers, q (the *quotient*) and r (the *remainder*), such that:

$x = q*y + r$, and $0 \leq r < y$

The terminology used to describe the elements of these equations is:
*dividend = quotient * divisor + remainder*

For example, let $x = 10$ and $y = 27$. Because $27 = 2 * 10 + 7$, and $0 \leq 7 < 10$, we see that the quotient is 2, and the remainder is 7.

In this definition, q is the whole number quotient, and r is the remainder. Here's how the Euclidean algorithm can be used to find the greatest common factor of 45 and 105:

45 = 0 * 105 + 45 ($^{45}/_{105}$ = 0 rem. 45)
105 = 2 * 45 + 15 ($^{105}/_{45}$ = 2 rem. 15)
45 = 3 * 15 + 0 ($^{45}/_{15}$ = 3 rem. 0)

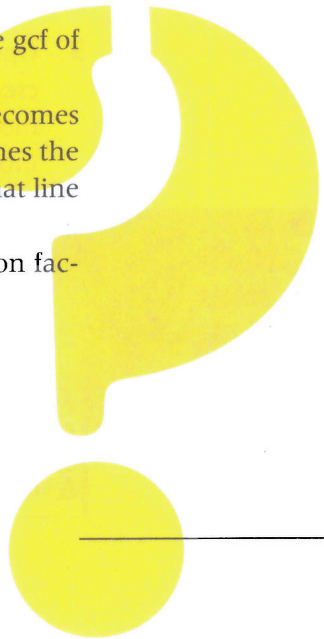
When the remainder is 0, the algorithm is complete: 15 is the gcf of 45 and 105. In the first line, 105 and 45 changed places.

To move from one line to the next, the divisor of one line becomes the dividend of the next line, and the remainder of the line becomes the divisor of the new line. When the remainder is 0, the divisor of that line is the greatest common factor.

This algorithm is efficient because it finds the greatest common factor of pairs of large or small numbers very quickly.

QUESTIONS AND ACTIVITIES

1. List three common factors of 30 and 45. What is the *greatest* common factor of 30 and 45?



2. If we add the fractions *a/b* and *c/d* we get the result *(ad + bc) / (bd)*. What is the result if the fractions are subtracted instead of added?
3. Why is a fraction with a 0 denominator "undefined"? What happens to the value of the fraction $1/x$ when x becomes very small?
4. Use the Euclidean algorithm to find the greatest common factor of the following pairs of numbers:
   a) 17 and 23
   b) 85 and 204
   c) 312 and 910

2 *Section*

Code Modules and User-Defined Types

In this section of the chapter, you create a code module, then within that module, declare a user-defined data type and define a number of routines to handle fractions. Because you will use the data type to declare fractions, you should give it the obvious name: Fraction. You will define the Fraction type as composed of two integers, one for the numerator of the fraction and one for the denominator.

NOTE:

A user-defined type declaration can only be made in a code module.

Creating a Code Module

To create a code module, follow these steps:

1 Start Visual Basic. If Visual Basic is running, select New Project from the File menu.

2 Select New Module from the File menu.

3 Open the Code window for the code module. If you have already created the module, open the window by choosing the name of the module from the Project window. Click on the name of the code module, as shown in Figure 12-1. Double-click to open the general declarations section. You can also open the Code window by clicking on the View Code button in the top-right corner of the Project window.

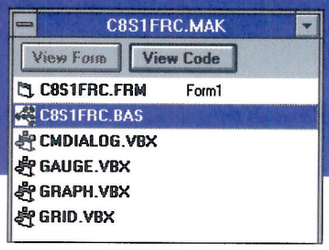


Figure 12-1
Selecting the code module from the Project window

4 Add the **Option Explicit** statement, if it is not already displayed in the general declarations section.

5 Below **Option Explicit**, define the Fraction type by using a **Type** statement:

```
Type Fraction
   Num As Integer
   Den As Integer
End Type
```

Creating a Procedure

Now you are ready to define some procedures for handling fractions; you will include these procedures in the code module. There are two ways to define a new procedure in the code module. The first is to use the Code window:

1 If you have closed the Code window for the code module, reopen it and select the general declarations section.

2 Type **Sub** to indicate a routine, then **NewOne** as the name of the procedure. Press Enter.

3 Visual Basic creates a skeleton for the new procedure. The parentheses after the routine's name are added automatically, along with the last line (**End Sub**). See Figure 12-2. You will enter the names and types of the parameters that will be passed to the procedure within the parentheses.

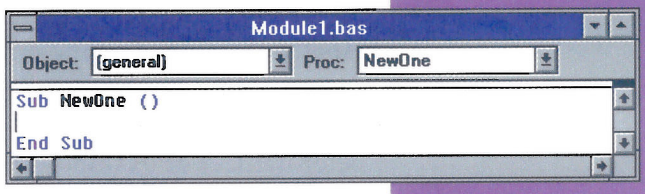


Figure 12-2
A new routine

The other way to create a new procedure in a code or form module is to select the New Procedure command from the View menu. The active window must be a module; otherwise, the New Procedure command is disabled. A dialog box prompts you to enter a procedure name and indicate whether you are creating a function or a subroutine (Figure 12-3). Visual Basic creates a skeleton procedure in the active module. The procedure's parameter list, enclosed within parentheses, is empty; you must enter any parameter declarations yourself.

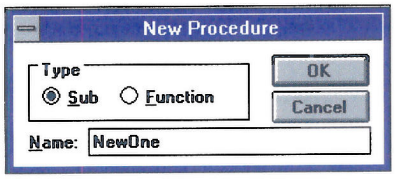


Figure 12-3
Choosing whether to create a function or a subroutine

FUNCTIONS AND SUBROUTINES

A function returns a value, which you assign to the name of the function in the body of the function procedure. A subroutine may perform actions, and/or return values to the calling program via its parameters, but the name of the subroutine is not used to transfer values.

Adding Parameters to the Procedure

After you have created the skeleton of a procedure, you can add parameters inside the parentheses following the name of the procedure. Visual Basic recognizes two kinds of parameters that may be passed to or from procedures and functions: value parameters and reference parameters.

VALUE PARAMETERS

Value parameters are passed as values to a procedure. The value of a variable or expression that you pass to a value parameter is transferred by Visual Basic to temporary storage local to the procedure. The value parameter names, or refers to, that storage. The value parameter may be altered—for instance by assigning it a new value—but the original value of the variable passed to it by the calling program is not altered.

A value parameter is the perfect way to send a value that should be protected from changing. For instance, a procedure might take a person's full name as a parameter and process just the last name. The original value, including the entire name, should not be affected.

To declare a value parameter in the parameter list of a procedure or function use the keyword **ByVal** in the parameter list before the parameter name. For example, after you have the skeleton of the procedure, add the parameter:

```
Sub NewOne(ByVal x As Integer) ' x is passed by value
```

The value of *x* may be altered within the procedure. For example:

```
x = x + 1,  or,  if x were a string,  x = Left$(x, 5)
```

The value of the argument in the calling program corresponding to *x*, however, is not affected.

REFERENCE PARAMETERS

Variables that are allowed or intended to change as a result of the action of the procedure or function are said to be passed by reference. This

means the actual address of the variable is made available to the procedure or function, allowing the procedure to alter the value of the argument in the calling program corresponding to the parameter. In this case, no copy of the argument's value is made; rather, the parameter refers to the storage occupied by the argument to the procedure call.

Variables passed by reference don't require a special keyword. You define them like this:

```
Sub NewOne(x as integer) ' x is passed by reference and can be changed
'_from within the procedure
```

CAUTION

Generally, if you are sending a value of a variable to a subroutine or function, and you don't intend the value of the variable to change, it should be passed by value. If you pass the variable by reference, you run the risk that the procedure will alter the value in a way you don't expect. These unwanted side effects are difficult to track down when the program is not behaving as you intend.

Unfortunately, you cannot send a variable of a user-defined type as a value parameter to a procedure or function. If you try to put the **ByVal** keyword in the parameter list of a subprogram using user-defined types, Visual Basic displays the error message in Figure 12-4. Parameters of user-defined types must be passed by reference.

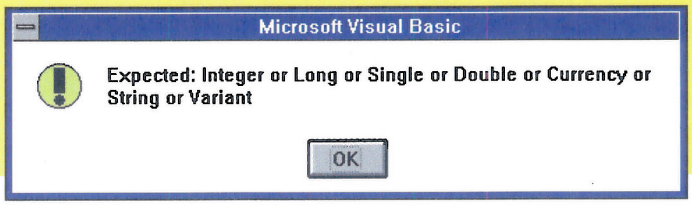


Figure 12-4
Error message displayed when you send a variable of user-defined type as a value parameter

Finishing the Fraction-Handling Routines

Now that you know how to start a routine, you can create as many routines as you need in the code module. You will need routines for adding, negating, subtracting, multiplying, finding reciprocals, dividing, and finding the greatest common factor. Each of these routines is quite short.

THE ADD ROUTINE

To create an add routine in the code module:

1 Open the general declarations section of the code module.

2 Add this code:

```
Sub Add (X As Fraction, Y As Fraction, Rslt As Fraction)
   Rslt.Num = X.Num * Y.Den + Y.Num * X.Den
   Rslt.Den = X.Den * Y.Den
End Sub
```

3 Select Save File from the File menu to save the code module with the name **frac.bas**.

As you can see in this code, you send three parameters to the Add routine. The first two are the fractions to be added. The third is the result of the addition process. The code for this procedure is developed from the equations shown in the last section.

Sending the fraction *X* as a parameter means you are sending both a numerator, *X.Num*, and a denominator, *X.Den*. The result of the procedure is returned in the fraction variable *Rslt*. The numerator is in *Rslt.Num* and the denominator of the answer is in *Rslt.Den*.

THE NEGATE PROCEDURE

This procedure has two parameters. The first is the fraction to be negated and the second is the negated fraction. The procedure works by changing the sign of the numerator of the fraction. This job could be handled directly in the calling program, but it should not be. The calling program should never have to manipulate the values inside a variable of the fraction type.

In the general declarations section of the code module, enter these lines:

```
Sub Negate (X As Fraction, Rslt As Fraction)
   Rslt.Num = -X.Num
   Rslt.Den = X.Den
End Sub
```

THE SUBTRACT ROUTINE

This subroutine works by negating the second operand and then passing the original first operand and the negated second operand to the Add procedure. It implements subtraction in terms of addition and the additive inverse, using the fact that $x - y = x + (-y)$. You declare a local fraction variable *Temp*, to hold the value of the negated second parameter.

Enter the general declarations section of the code module and enter these lines:

```
Sub Subtract (X As Fraction, Y As Fraction, Rslt As Fraction)
Dim Temp As Fraction
   Negate Y, Temp
   Add x, Temp, Rslt
End Sub
```

THE MULTIPLY ROUTINE

This routine sends the two fractions to be multiplied, *X* and *Y*, as parameters and expects the product to be returned in the parameter, *Rslt*.

Enter the general declarations section of the code module and enter these lines:

```
Sub Multiply (X As Fraction, Y As Fraction, Rslt As Fraction)
   Rslt.Num = X.Num * Y.Num
   Rslt.Den = X.Den * Y.Den
End Sub
```

THE RECIPROCAL ROUTINE

This routine returns the multiplicative inverse of a fraction sent as the first parameter. If the numerator of the fraction sent is 0, the reciprocal doesn't exist and the user is sent an appropriate message. An alternative to displaying a message would be to send a **True/False** variable as a parameter. If the reciprocal is successfully calculated, you set the flag to true. If the reciprocal doesn't exist, set the flag to false. Upon return to the calling program, the flag can be checked and appropriate action taken.

Enter the general declarations section of the code module and enter these lines:

```
Sub Recip (X As Fraction, Rslt As Fraction)
   If X.Num = 0 Then
      MsgBox "The program attempted to take the reciprocal of 0."
   Else
      Rslt.Num = X.Den
      Rslt.Den = X.Num
   End If
End Sub
```

THE DIVISION ROUTINE

Now that you have written a multiplicative inverse routine, you can easily write a division routine. To do so, declare a temporary local variable

to hold the reciprocal of the second operand. Then send it along with the first operand to the Multiply routine.

In this routine, you call the necessary subroutines with a second method. In the Subtract routine, you called Negate and Add by stating their names and listing the parameters. In this routine, you call the necessary routines with the **Call** statement. This statement requires that the parameters be enclosed in parentheses. The statement is included here to familiarize you with an older convention that you will see if you look at other Basic programs.

Enter the general declarations section of the code module and enter these lines:

```
Sub Divide (X As Fraction, Y As Fraction, Rslt As Fraction)
   Dim Temp As Fraction
   Call Recip(Y, Temp)
   Call Multiply(X, Temp, Rslt)
End Sub
```

GREATEST COMMON FACTOR ROUTINE

Now you need to write the code to reduce fractions to their lowest terms. To accomplish this, you need the greatest common factor of the numerator and the denominator. The next routine, **Function gcf()**, calculates this value using the Euclidean algorithm.

The parameters are normal integer types, as is the value of the function itself. Therefore, you can use the **ByVal** keyword in the parameter list to send the parameters as value types. The math of the algorithm is described in the previous section.

If you take the variable *a* as the dividend (the number being divided), and *b* as the divisor, the algorithm is finished when the value of *b* is 0. The greatest common factor is the value of *a*. Pairs of numbers that don't have any other divisors have a common divisor of 1, so the function always returns a value. (Two integers whose greatest common divisor is 1 are said to be relatively prime.)

Enter the general declarations section of the code module and enter these lines:

```
Function gcf (ByVal a As Integer, ByVal b As
Integer) As Integer
   Dim Temp As Integer
   Do While b <> 0
      Temp = a Mod b
      a = b
```

```
        b = Abs(Temp)
    Loop
    gcf = a
End Function
```

THE REDUCE ROUTINE

After the greatest common factor of the numerator and divisor is calculated, reducing the fraction requires just a couple of integer division statements.

Enter the general declarations section of the code module and enter these lines:

```
Sub Reduce (Rslt As Fraction)
    Dim Temp As Integer
    Temp = gcf(Rslt.Num, Rslt.Den)
    Rslt.Num = Rslt.Num \ Temp
    Rslt.Den = Rslt.Den \ Temp
End Sub
```

Fraction Handler Program

With the code module built, you are now ready to explore how to use these routines in a program. To do so, you start a new project, then create a form that prompts users to enter values for two fractions. Users can then add, subtract, multiply, or divide the fractions they have entered and display the results (see Figure 12-5). This is the Fraction Handler project.

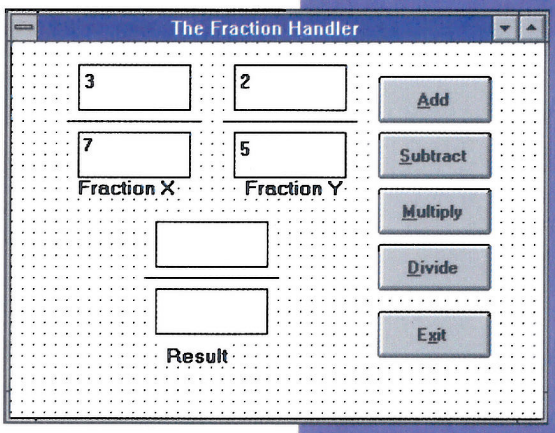


Figure 12-5 Finished form for the Fraction Handler program

GETTING STARTED

As you can see from Figure 12-5, you need four textboxes on the form for the user to enter the numerators and denominators of the fractions. You only need to label two of these, though you should place a line between each pair to indicate that the pair is a fraction. Then, you let users perform actions with the fractions by placing command buttons to the right. You display the result of any of these operations in the two labels at the bottom labeled Result.

As you have done in other programs, you should preload the fraction textboxes with reasonable values instead of just clearing their Text properties. When you do so, the user only needs to click one of the command buttons to see the program work.

BUILDING THE FORM

To build the form for the Fraction Handler project, follow these steps.

1. Select the default form in the Project window.
2. In the Properties window, change the caption of the form to **The Fraction Handler**.
3. Place four textboxes on the form. Change the FontSize property of each to 9.75. Change their names to **txtXNum**, **txtXDen**, **txtYNum**, and **txtYDen**. Arrange the textboxes on the form so that the names of the boxes for the numerators are over the appropriately named boxes for the denominators. Label the fractions.
4. Place two labels on the form to display the result of an operation. One way to save some work is to create a single box, give it all the attributes you want, such as BorderStyle: Single, FontSize: 9.75, Caption: deleted. Make it the size you want for both labels. Select the label with a click of the mouse, copy it with Ctrl+C, and paste it with Ctrl+V. Respond No to the inquiry about creating a control array.
5. Name the labels you just created **lblRsltNum** and **lblRsltDen**.
6. Place lines from the toolbox between the numerator and denominator of each fraction.
7. Place four command buttons on the form. Create a control array by creating a single button. Give the button the name **cmdOp**. Copy the button with Ctrl+C and paste with Ctrl+V. This time, when you are prompted to create a control array, respond Yes. Change the captions of the four buttons to **&Add**, **&Subtract**, **&Multiply**, and **&Divide**. Change their Index properties to 0, 1, 2, and 3, respectively.
8. Create a fifth command button. Change the name of the button to **cmdExit**. Change the caption of the button to **E&xit**.
9. Save the form, project, and code module files.

DECLARING THE VARIABLES

In the general declarations section of the code, you need to declare two variables to represent the fractions. You use the Fraction type you defined in the code module in the declaration (*x* and *y* are of the Fraction type).

To declare the variables:

1. Select the form.

2 In the general declarations section of the form, enter the following code.

```
Option Explicit
Dim x As Fraction, y As Fraction
```

CODING THE COMMAND BUTTONS

Because you gave the command buttons the same name (cmdOp), they all call the same event procedures. To differentiate between one command button and another, you assigned a different index value to each one. The Add button has an index value of 0, Subtract is 1, Multiply is 2, and Divide is 3. The Exit button has a unique name and so calls its own event procedure.

Use a **Select Case** statement in the cmdOp_Click routine to process the statements appropriate for each different command button. This approach works very well in this program. No matter which button is pushed, all the operations start by reading the values of the fractions from the upper textboxes, and finish by displaying the result of the operation in the lower textboxes. Using a control array means you can share the code common to all four operations.

As you write the procedure, you need to consider how the code will handle a negative result. What happens, for example, when you divide a positive fraction by a negative one? Where will the form display the negative sign? That sign should *not* appear in the denominator, as in this example:

$$3/5 \;/\; {}^{-2}/_{7} \;=\; {}^{21}/_{-10}$$

You will need to be sure the program checks the sign of the denominator before displaying any results. If it is negative, the program should change the signs of both the numerator and the denominator of the result (why both?).

Another fraction issue you need to consider is a denominator with the value of 0. This is an undefined fraction and should display no answer at all. In the code for cmdOp, you will need to add a simple **If-Then** statement to check for and handle this possibility.

To code the command buttons:

1 Enter these lines in the cmdOp_Click() event procedure:

```
Dim Rslt As Fraction
X.Num = Val(txtXNum.Text)
X.Den = Val(txtXDen.Text)
Y.Num = Val(txtYNum.Text)
Y.Den = Val(txtYDen.Text)
```

```
Select Case index
   Case 0     ' Add the operands
      Add X, Y, Rslt
   Case 1     ' Subtract
      Subtract X, Y, Rslt
   Case 2     ' Multiply
      Multiply X, Y, Rslt
   Case 3     ' Divide
      Divide X, Y, Rslt
   End Select
'—Check for negative sign in the denominator
If Rslt.Den < 0 Then
   Rslt.Num = -Rslt.Num
   Rslt.Den = -Rslt.Den
End If
'—Check for denominator equal to 0
If Rslt.Den = 0 Then
   MsgBox "The fraction does not exist."
Else
   Reduce Rslt
   lblRsltNum = Str$(Rslt.Num)
   lblRsltDen = Str$(Rslt.Den)
End If
```

2 Add **End** to the procedure for cmdExit.

3 Save the form, project, and code module files.

4 Run the program. Click on each of the command buttons.

5 Enter new values for the numerators and denominators of the fractions. Note the results.

QUESTIONS AND ACTIVITIES

1. What is the significance of code modules? Why are they necessary?
2. Look at the Project window for the Fraction Handler project. What does each line displayed in Figure 12-6 mean?

NOTE:

Although the Fraction Handler program does not offer a command button to reduce fractions, you can still use the program for this purpose. Enter the fraction that you want to reduce as the first fraction, and enter 0/1 as the second. When you add these fractions, the result displayed is the first fraction in lowest terms.

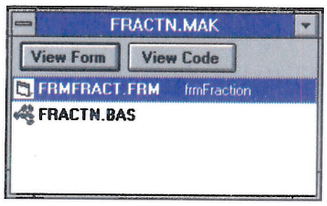


Figure 12-6
Project window for the Fraction Handler project

3. Create a user-defined type named Lumber. Have it include fields for the following information:

tree	a string
dimensions	three numbers such as 5'x 2"x 4"
quantity on hand	an integer
cost	a currency type

4. Write a routine that will take as parameters *a* and *b* of type Lumber and display in a message box the name of the tree for which the cost is lower.
5. Describe two ways to begin a new procedure in a code module.
6. What is the difference between a Sub procedure and a Function procedure?
7. What are value and reference parameters? When is it appropriate to use each? Put the **ByVal** keyword in front of the first parameter in the Add procedure. Run the program and note the results.
8. In the general declarations section of the form, put the following declaration:

```
Dim Z As Integer
```

   In each of the four procedures for adding, subtracting, multiplying and dividing, put the following line:

```
Z = Z + 1
```

   Into the Add procedure, put in a line to display the value of *Z* in a message box. Run the program, click several buttons and note the change in the value of *Z*.
   Although not unwanted, the value of *Z* shows a side effect. Its value changes even though it was not passed as a parameter to the procedures that changed its value. When this happens accidentally, it is difficult to fix.
9. Write a subtract procedure that does not use the Add procedure to do the work.
10. Rewrite the Recip routine with the following first line:

```
Sub Recip (X As Fraction, Rslt As Fraction, Succ As
Integer)
```

    The last parameter, *Succ*, should be set to **True** if the result of taking the reciprocal is valid, and to **False** if the reciprocal does not exist. Don't print a message.

11. In the **gcf()** function, the values of *a* and *b* are completely changed within the function. What keeps these changed values from being sent back to the calling program?

12. Write a procedure that will accept a fraction and a positive integer as parameters and return a fraction equal to the original fraction raised to the power represented by the integer. Use the following first line:

```
Sub Power(X As Fraction, Power As Integer, Rslt As Fraction)
```

13. Add to the code module for the Fraction program a procedure with the following heading:

    ```
    Sub Normalize( fract as Fraction )
    ```

    The routine should "normalize" the fraction sent as a parameter by
    - ◎ Calling Reduce
    - ◎ Changing the signs of the numerator and the denominator if the denominator is negative
    - ◎ Making the numerator and the denominator positive if both are negative

    Add a call to the Normalize procedure to the end of each math procedure defined in the code module.

Section

A Database Program with User-Defined Types

In this program, you'll create a form that lets the user enter, edit, delete, and display a multiple-choice quiz, take the quiz, and display the score. You might want to use the program to create practice quizzes for yourself. Or you and a classmate could create quizzes to exchange.

Building this program covers two sections of this chapter. The first section covers building the forms and defining the user-defined types. The second section covers the code that ties the forms and files together.

Over the course of the program, we will introduce a number of new features of the language and learn something about building a fairly large program. You will declare and use a user-defined type that is more complex than the ones you have used so far; in one of its fields is an array of records of another user-defined type. Nesting user-defined types in this way gives you considerable expressive power. In this program, an entire test is defined by a single user-defined type.

Later in this chapter, you will learn how to use the File System controls, and you will refine the program to use them. These controls let you spare the user from having to type in complete file names whenever your program must interact with the file system. Instead of remembering and typing in long paths and filenames, users of your programs can click with the mouse to choose drives, directories, and files. Your programs become much easier to use.

Starting Out

What data structure do you need to represent a multiple-choice test of 25 questions? A data structure is a framework that stores and organizes data. It is a theoretical structure that is built using user-defined types and arrays. You'll need fields for the body of each question, the five possible answers, and a character for the correct answer. You'll need an array to hold 25 records. The overall data structure should also record information about the title of the test, and how many questions there are.

The algorithms (the step-by-step processes for solving the problems) are mainly bookkeeping procedures: keeping track of where you are in the question array, filling the empty space after a question is deleted, keeping track of correct answers so an overall score can be reported, managing the data files and appearance (and disappearance) of the two forms.

USING MENU COMMANDS AND FORMS AS AN ORGANIZER

You need to determine how many and what kind of forms you will need for the project. In most of the projects to this point, you have used only single forms. This project splits naturally into at least two parts: entering, editing, and deleting questions; and taking the test and keeping score. As a result, you will create one form called the Edit form and another called the Test form.

In your plans for a program like this, you may have thought there should be three forms, or maybe you can envision a single form that will handle the problem. Any of these solutions might work very well. There is no one right answer to any programming problem.

THE EDIT FUNCTIONS

Entering new questions, editing or deleting old ones, and displaying the questions of a test are the kinds of tasks the program will perform. In addition, you will want to save files, open new files, or load existing

files. When you load a file, you'll transfer information from the file to a variable that represents your user-defined test type in the program.

Start the program by looking over the Edit form in Figure 12-7. Do you agree with the design decisions made here? If not, what can you improve, and why? The form contains:

- A label to indicate the test name
- A line for "other" information
- A textbox for the body of the question
- Five textboxes for the test choices
- A box for the correct answer
- Labels on the form indicating the current question number, the total number of questions and the filename of the test file
- Two command buttons: one for moving to the next question in the list and one for moving to the previous question

When a user clicks on one of the two command buttons, a different question is displayed. An edited question is added to the array only if the user selects Add from the Questions submenu.

What you can't see from Figure 12-7 is the structure of the menus on the Edit form. That structure is:

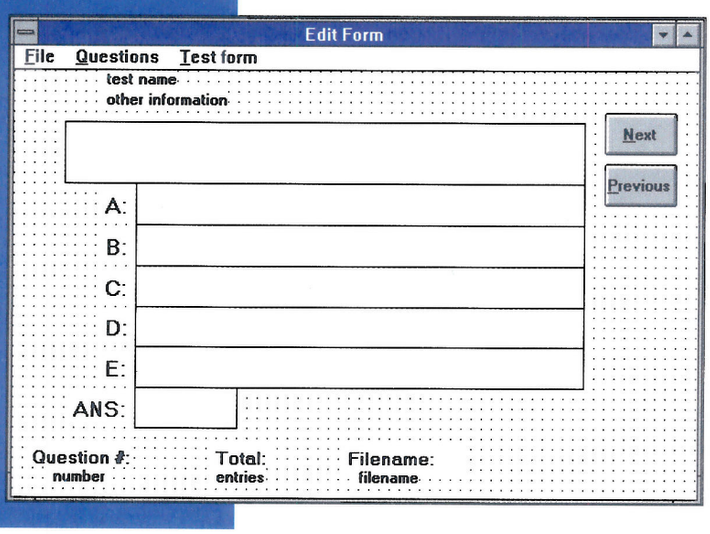


Figure 12-7
Edit form for the quiz program

- File
  - New
  - Open
  - Save
  - Exit
- Questions
  - Enter
  - Add
  - Delete
  - Display
- Test form

THE TEST FORM

The Test form is similar to the Edit form (see Figure 12-8). The user clicks Next for the next question. Labels on the form keep track of:

- Name of the test
- "Other" test information
- Record number of the current question
- How many questions are in the test
- Filename of the test file chosen

When finished with the test, the user chooses the Score menu item to see the score. After viewing the score, the user returns to the Edit form or exits the program.

In this form the menu structure is:

- Tests
  - Take a Test
  - Score
- Edit Form
- Exit

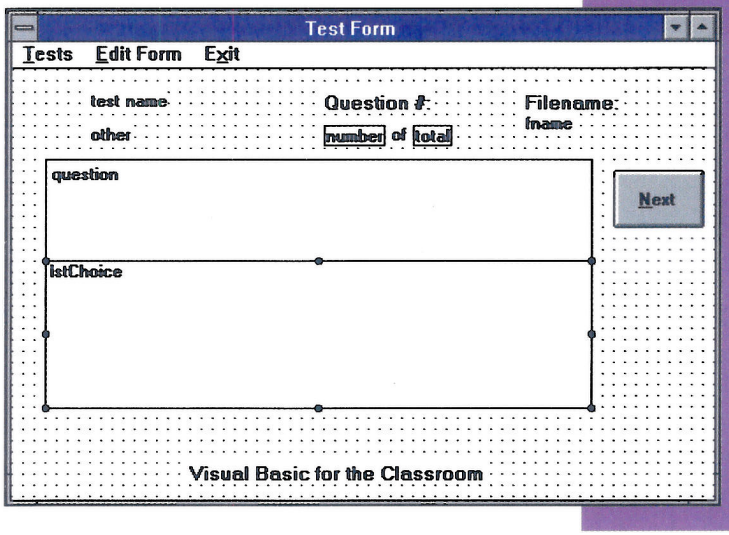


Figure 12-8
Test form for the quiz program

Choosing the Take a Test command lets the user enter the filename of the test to take. The program opens the file, loads the information into the Test data structure, and displays the first question. After an answer is entered, it is compared to the correct answer recorded with the question and a score is kept.

Creating the Edit Form

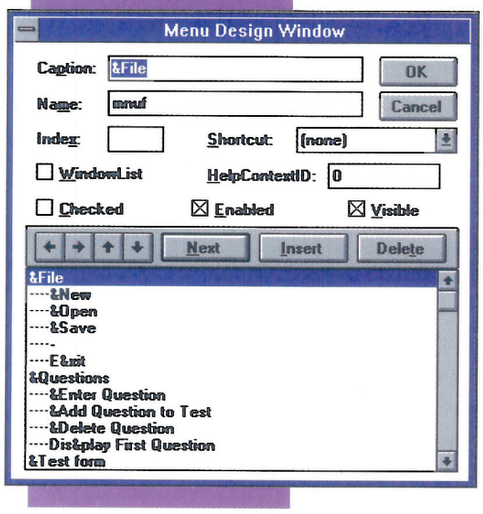
Follow these steps to create the Edit form:

1. Select the default form.
2. Change the name of the form to **frmEdit**. Change the caption of the form to **Edit Form**.
3. Select the form. Open the Menu Design window.

4 Using Figure 12-9 and the information below, create the menu for the Edit form.

Caption	*Name*	*Index*
&File	mnuF	
&New	mnuFile	1
&Open	mnuFile	2
&Save	mnuFile	3
-	mnuFile	4
E&xit	mnuExit	
&Questions	mnuQuestions	
&Enter Question	mnuQuest	
&Add Question to Test	mnuAddTo	
&Delete Question	mnuDelete	
Dis&play First Question	mnuDisplay	
&Test Form	mnuTest	

Figure 12-9
Menu Design window for the Edit form menu



5 At the top-left of the form, place two labels. Change the names to **lblTestName** and **lblOther**. Change the captions to **test name** and **other information**. These labels are intended to display a title for the test, such as "VB File Test," and other information such as "July 4, 1996."

6 Place a textbox across the top of the form to hold the question. Change the name of the box to **txtQuestion**. In the Properties window, change Multi Line to **True**, and TabIndex to 0. Delete the text.

7 Place five textboxes for the responses to the question. Change the names of the boxes to **txtResp1**, **txtResp2**, and so on. Delete the captions. Set MultiLine to **True** for each.

8 Put labels in front of each box representing the letter of the choice. Change the captions to **A:, B:,** and so on, and place them as shown in Figure 12-7. The labels should retain their default names. Change the FontSize to 12 and AutoSize to **True** for all of them.

9 Place a textbox on the form below the five response boxes, for the answer to the problem. Name the box **txtAns**. Delete the text.

10 Place a label on the box just created. Set the FontSize to 12 and the caption to *ANS:*.

11 Below the response boxes and across the bottom of the form, place six labels in pairs, one above the other.

12 Change the captions of the top boxes to **Question #1**, **Total:**, and **Filename**.

13 Change the names of the lower boxes to **lblNum**, **lblTotal**, and **lblFileName**. Change the captions of those boxes to **number**, **entries**, and **filename**.

14 Place two command buttons on the form. Change the captions to **&Next** and **&Previous**. Change both the names to **cmdNextQ**. Set the Index property of the Next button to 0. Set the Index property of the Previous button to 1.

15 Save the form and project files in a directory named mchoice.

Creating the Test Form

Follow these steps to create the Test form:

1 Choose New Form from the File menu.

2 Change the name of the form to **frmTest**. Change the caption of the form to **Test Form**.

3 Select the form. Open the Menu Design window.

4 Using Figure 12-10 and the chart below, create the menu for the Test form.

Caption	*Name*
&Tests	mnuTests
&Take a Test	mnuTake
&Score	mnuScore
&Edit Form	mnuEditForm
E&xit	mnuExit

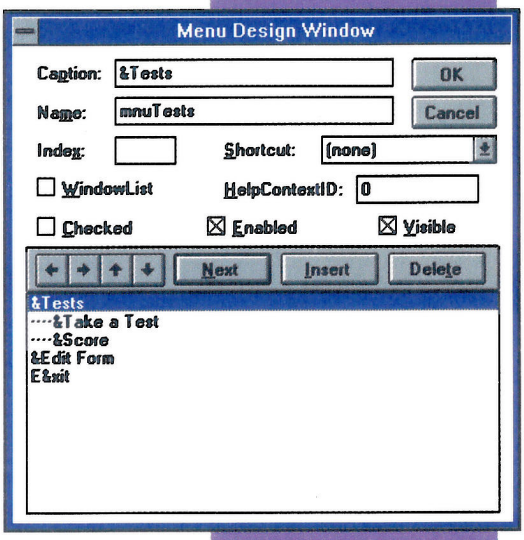


Figure 12-10
Menu Design window for the Test form

5 Create two labels in the upper-left corner of the form. Change the captions to **test name** and **other**. Change the names to **lblTestName** and **lblOther**.

6 Near the top center of the form, create a label with the caption **Question #**. Set the FontSize to 9.75. This label heads three labels, two of which are set to reflect the current question number and the total number of questions in the test.

7 Put three labels in a row below the Question # label. Change the captions to **number**, **of**, and **total**. Change the name of the first to

lblQNum and the last to **lblTotal**. Change the BorderStyle of the first and last to Single.

8. Put two labels near the top right of the form. Change the captions to **Filename:** and **FName**. Change the FontSize of the first to 9.75. Change the name of the second to **lblFileName**.
9. Put a large multiline textbox in the center of the form. Change the name of the box to **txtQuestion**. Change the Text property to **question**.
10. Put a large listbox below. Change the name to **lstChoice**.
11. Put a command button on the right side of the form. Change the caption of the button to **&Next**. Change the name of the button to **cmdNext**.
12. Save the form and project files.

Setting Up the Data Structures

You must place declarations for global data types, which are types available to all forms and modules, in the declarations section of a code module. To do so:

1. Add a code module to the project. Choose New Module from the File menu.
2. Enter the following lines to define the special data types for the project:

```
Option Explicit
Type Quest
   N As Integer
   Body As String * 250
   Res1 As String * 100
   Res2 As String * 100
   Res3 As String * 100
   Res4 As String * 100
   Res5 As String * 100
   Ans As String * 1
End Type
Type Test     '26487 is record length
   Title As String * 80
   Other As String * 80
   Current As Integer
   Question(0 To 25) As Quest
End Type
```

Two new types are defined. The first is used as a component of the second. The Quest type represents a single question. It starts with a 250-character string for the question itself. The fields *Res1* through *Res5* are 100-character strings holding the answer choices for each question. The letter of the answer itself goes into a 1-character string with the variable name *Ans*.

The entire test is stored as a single data item. A single record in Visual Basic has a maximum of 32767 characters, more than enough for a multiple choice test of 25 questions. Because each test is organized as a single data item, it is possible to put more than one test into a file. The program as currently implemented stores only a single test per file.

The Test type starts with two fields to store information about the test: the first field, Title, for the name of the test and the second, called Other, for the teacher's name and the date the test was written. The variable *Current* contains the current number of questions in the test, and the array, *Question(0 to 25)* contains the actual questions, choices, and answers for each question on the test.

QUESTIONS AND ACTIVITIES

1. You have probably heard the advice that you should never be afraid to start a project over from the beginning. What is the reason for such a statement?

2. What is the record length for a record of the following type?

```
Type day
   name As String*15
   date As Integer   ' integers are two bytes
End Type
Type month
   name As String*15
   days(30) As day
End Type
```

3. Write out in words (not code) a plan for a program to store information for a pet store. The program should store information about animals the store sells. The animal information should include the breed, the birth date, the price, the quantity, and the place obtained. What kind of data structure would you use? What kinds of jobs should the program handle? What kinds of displays might be generated?

4. If you were writing a program to store information that would be useful in filling out a questionnaire or a petition (pick one), what kind of information would you store?

Section

Writing Code for the Quiz Program

You have created two forms and the data structure for the quiz program. This section addresses the code that ties them all together. You will add most of the code to the menu commands and command buttons of the Edit form. Code to be shared by both forms is put into a code module (see Figure 12-11). (In contrast, if you want a procedure to be available only to different event procedures of a single form, you define it in the general declarations section of that form.)

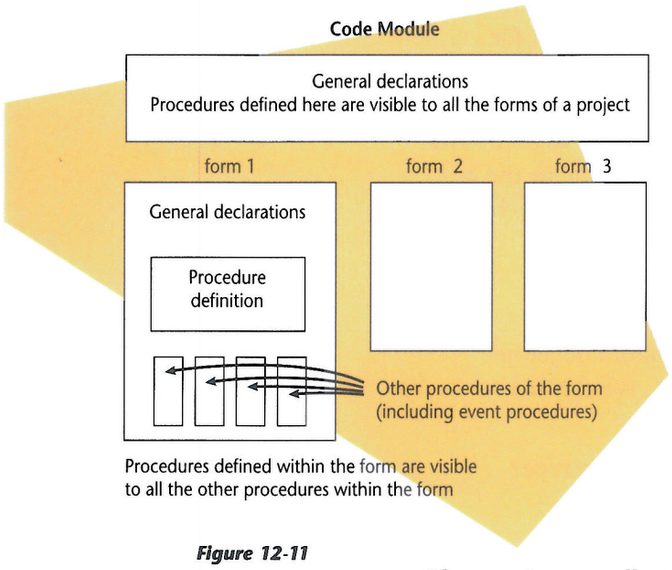


Figure 12-11
Determining where to place procedures

The Edit Form

You now add code to the Edit form that implements its menu commands and command buttons. The menu commands consist of commands in the File and Questions submenus, together with the Test Form command. The commands in the File menu create a new test, load an existing test, and save the current test. The commands in the Questions menu let you or the user add, delete, and edit questions in the current test. The Next and Previous command buttons navigate between the questions.

COLLECTING THE FILENAME

The routine to collect the filename of the test from the user is called in both the Edit form and the Test form. The code for the Filename procedure appears in the code module you have already defined.

This routine, a programmer-defined function, starts with the keyword **Function** in the first line, followed by the name of the function, any parameters sent to the function, and the data type of the information returned by the function to the calling program. In this program, no parameters are sent and the name of the file returned is a string.

In the code, you need to prompt the user to enter the name of the file to be opened. Use string concatenation to build a complete filename that includes the path. Display the filename in lblFileName of both the Edit and Test forms. Assign the string constructed to the name of the function, FileName, so that it can be returned to the calling program.

To enter the code:

1 Select the code module from the Project window.

2 Click on View Code and open the general declarations section.

3 Enter these lines:

```
Function FileName () As String
   Dim Msg As String
   Dim F As String
   Msg = "enter the file name"
   '—InputBox provides default filename
   '—The mchoice directory holds test data
   F = "c:\mchoice\" & InputBox$(Msg, , "test1") & ".dat"
   frmEdit.lblFileName = F
   frmTest.lblFileName = F
   FileName = F
End Function
```

DECLARING VARIABLES

Now you declare the variables that the procedures and event handlers of the Edit form operate on mutually. You need a global variable of Test type to hold the edited test itself, an integer to hold the current question number, and a string to hold the filename of the test. To declare these variables for the Edit form:

1 Open the Code window for the Edit form.

2 Enter the variable declarations you need in the general declarations section:

```
Option Explicit
Dim T As Test
Dim QNum As Integer
Dim FName As String
```

The variable *T* represents the entire Test data structure. *FName* represents the name of the file accessed for loading and storing the test, and *QNum* represents the subscript of the question currently being processed.

DISPLAYING THE FIRST QUESTION OF THE TEST

The routine to display the current question of the test on the Edit form is called from a number of event procedures: it's called from cmdDisplay_Click(), the procedure associated with the Display first question command in the Questions menu. It is also called from cmdDelete_Click(), a procedure that deletes the currently displayed question

and replaces it with a display of the previous question. Finally, it's called from the two command buttons, Next and Previous.

The routine receives from the calling program a single parameter, *QNum*, designating the array subscript of the element to be displayed. It transfers values from the Test data type to the textboxes and label on the form.

To write the routine that displays the current question on the Edit form:

1. If you do not have the Code window for the Edit form open, then open it.
2. Open the general declarations section of the form.
3. Add the following lines to the Display procedure:

```
Sub Display (QNum As Integer)
   '—T.Question(QNum) is one question of the test
   '—Body accesses the question itself
   '—Res x accesses the possible RESponses to the question
   txtQuestion = T.Question(QNum).Body
   txtResp1 = T.Question(QNum).Res1
   txtResp2 = T.Question(QNum).Res2
   txtResp3 = T.Question(QNum).Res3
   txtResp4 = T.Question(QNum).Res4
   txtResp5 = T.Question(QNum).Res5
   '—The real answer is designated by a single character
   txtAns = T.Question(QNum).ANS
   '—Display the current total number of questions.
   lblTotal = T.Current
   '—Display this question's number.
   lblNum = QNum
End Sub
```

ACCESSING ELEMENTS OF THE QUESTION ARRAY

To access the elements of the *Question* array, you start with the name of the Test, *T*. Dots separate each level of access to the fields of the record. The dot is followed by the array name, *Question*. The subscript *QNum* is next, followed by the field in the array to be displayed: *Body*, for the actual question, *Res1..Res5* for the choices, and *Ans* for the proper answer.

T	Represents the entire test.
T.Question	Represents the question array.
T.Question(QNum)	Represents a single question.

T.Question(QNum).Body	Represents the actual question for entry *QNum*.
T. Question(QNum).Res1	Represents the first choice for the answer.

DELETING QUESTIONS

One of the edit functions of the program is to delete a question from the test. A question must be displayed, to be deleted.

To code this procedure:

1. Select the Edit form, frmEdit, from the Project window.
2. Enter these lines of code between the first and last lines in the mnuDelete_Click procedure:

```
'—This routine deletes question QNum from the test.
'—Local declarations.
Dim Msg As String
Dim Ys As String
Dim MaxValue As Integer, x As Integer
'—Prompt for InputBox.
Msg = "Are you sure?<y,n>:"
'—Get confirmation for delete.
Ys = InputBox$(Msg, , "Y")
'—Check for yes and whether the question can be deleted.
If (Ys = "y" Or Ys = "Y") And (QNum >= 1) And (QNum <= T.Current) Then
   '—MaxValue is the number of the last question.
   MaxValue = T.Current
   For x = QNum To MaxValue - 1
      T.Question(x) = T.Question(x + 1)
   Next x
   T.Current = T.Current - 1
   QNum = QNum - 1
   Display (QNum)
End If
```

The trick in this routine is to delete a question by filling its place with the question that follows. Once the question is deleted, you reset *T.Current* by setting it equal to *T.Current - 1*. If you delete the last question of the test, there is no subsequent question to replace it. This is not a problem: once *T.Current* is decremented, the last question is effectively cut out.

Your only real problem is what to do if there is only one question in the test. When it is deleted, what will be displayed? The code above allows the first question to be deleted and displays the question at *QNum - 1*. There is no real question at this spot, but it is a spot in the *Question* array. You will recall the declaration for the *Question* array was from 0 to

25. Visual Basic automatically fills this first space in the array (subscript = 0) with blanks, so a blank question is displayed.

DISPLAYING QUESTIONS

The procedure you create in this section displays the first question of the test, provided the test contains any questions at all. Assuming that *T.Current* is greater than or equal to 1, this procedure sets *QNum* to 1 and calls the **Display***(QNum)* routine, which displays question *QNum*. If *Current* is 0, no action is taken.

To display the first question:

1. Select frmEdit from the Project window.
2. Click on View Code.
3. Open the code procedure for mnuDisplay_Click. Enter this code between the first and last lines of the procedure:

```
If T.Current <> 0 Then
   QNum = 1
   Display (QNum)
End If
```

4. Save the form, project, and code module files.

CODING THE FILE ITEMS

Now you need to write the procedure that implements the commands on the Edit form's File menu. This procedure creates new tests, opens and loads existing tests, and saves tests to disk. Enter the code for the file-managing routines by following these steps:

1. Select the Edit form.
2. Select New from the File menu of the Edit form to open the Code window.
3. Enter the following lines of code for mnuFile.

```
Dim Msg As String
'—Read the filename from the form.
FName = lblFileName
'—If file name is blank or New is clicked, collect filename from user
If (FName = "") Or Index = 1 Then
   FName = FileName()
   lblFileName = FName
End If
```

```
'-Process menu items with Select Case statement
Select Case Index
'-New test chosen
   Case 1
      Msg = "Enter the NAME of test:"   ' not the filename
      lblTestName = InputBox$(Msg)      ' get the name
      T.Title = lblTestName.Caption     ' display the name
      '-Collect and display the "other" information.
      Msg = "Enter other information (date or description):"
      lblOther = InputBox$(Msg)
      T.Other = lblOther.Caption
      '-Set current number of questions to 0
      T.Current = 0
      '-Display blank question to clear display
      Display (0)
      '-Set record number to 1
      QNum = 1
      '-Prepare to enter text in question
      txtQuestion.SetFocus
      '-Open saved test from disk
   Case 2
      Open FName For Random Access Read As #1 Len = 30000
      Get #1, , T
      Close
      '-Load test information onto form
      lblTestName = T.Title
      lblOther = T.Other
      lblTotal = T.Current
      lblFileName = FName
      '-Save the current test
   Case 3
      Open FName For Random Access Write As #1 Len = 30000
      Put #1, , T
      Close
End Select
```

The **Open** and **Save** statements open and save a test file. The test file itself is 30,000 bytes long. Although using a single **Get** or **Put** statement to read and write the file is convenient, it wastes disk space.

CODING THE NEXT AND PREVIOUS COMMAND BUTTONS

These two buttons are a control array, meaning they share the same name. The index value for the first button is 0 and for the second, 1. The code controls the display of the next or previous question. If neither of these questions exists, the display is left unchanged.

Follow these steps to code the buttons:

1. If necessary, select the Edit form.
2. Double-click on either the Next or the Previous button to open the Code window.
3. Enter the following lines of code:

```
Sub cmdNextQ_Click (Index As Integer)
   '-Index = 0 is the Next button
   '-Index = 1 is the Previous button
   If Index = 0 Then
      '-QNum is followed by a question it is OK to
      'display the next question.
      If QNum < T.Current And QNum < 25 Then
         QNum = QNum + 1
         '-The Display routine is called to update the display
         Display (QNum)
      End If
   Else
      If QNum > 1 Then
         QNum = QNum - 1
         Display (QNum)
      End If
   End If
End Sub
```

ENTERING THE QUESTION HANDLERS

The Enter Question command in the Questions menu clears the edit space to prepare for a new question. Follow these steps to write the code for this command:

1. If necessary, select the Edit form.
2. Select Enter Question from the Questions menu. In the cmdQuest_Click routine, enter the following code:

```
'—Clear the display.
txtQuestion = ""
txtResp1 = ""
txtResp2 = ""
txtResp3 = ""
txtResp4 = ""
txtResp5 = ""
txtAns = ""
txtQuestion.SetFocus
```

ADDING A QUESTION

A user selects the Add Question to Test menu command when the question displayed on the screen is ready to be added to the test. In the code, you first need to make sure that there is a file currently being edited. The value of *Current* is incremented, showing another question has been added to the test. The information about the question is then read from the textboxes on the form and transferred to the array.

To write this code:

1. If necessary, select the Edit form.
2. Select Add Question to Test from the Questions menu.
3. Enter the following code into the mnuAddTo_Click () subroutine:

```
If txtQuestion.Text <> "" Then
   QNum = T.Current + 1
   T.Question(QNum).Body = txtQuestion
   T.Question(QNum).Res1 = "A) " & txtResp1
   T.Question(QNum).Res2 = "B) " & txtResp2
   T.Question(QNum).Res3 = "C) " & txtResp3
   T.Question(QNum).Res4 = "D) " & txtResp4
   T.Question(QNum).Res5 = "E) " & txtResp5
   T.Question(QNum).Ans = txtAns
   T.Current = QNum
   lblNum = QNum
   lblTotal = T.Current
End If
```

4. Save the form and project files.

SHIFTING FOCUS

The Test Form command shifts the focus of the program to the test form. To write the code that shifts focus:

1. Click on the Test Form menu item in the Edit form menu bar.
2. In the Code window, enter the following code for mnuTest_Click():

```
'—Unload Me unloads the Edit form from memory
Unload Me
'—Show loads a form into memory and shifts the
'focus to that form
frmTest.Show
```

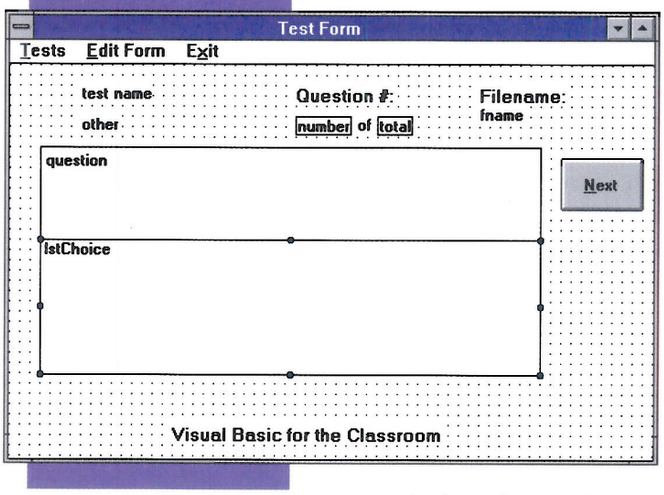


Figure 12-12
Test form for the quiz program

The Test Form

Compared to the Edit form, the Test form is quite straightforward to code. Its purpose is to display the questions one by one for the user to answer. It keeps score and displays the score when the user clicks on the Score command. See Figure 12-12.

DECLARING THE VARIABLES

The Test form requires a few global variables. *T* represents the test. *Scr* is the current score, a whole number. *QNum* is the subscript for the question, just as in the Edit form.

To declare the variables:

1. Select the Test form.
2. Double-click on the form to open the Code window.
3. In the general declarations section of the form, insert the following lines.

```
Option Explicit
Dim T As Test
Dim Scr As Integer
Dim QNum As Integer
```

WRITING THE DISPLAY2 PROCEDURE

The arrangement of the question and answers in this form is different from the arrangement in the Edit form. Part of the difference is in the method of displaying the choices for the answer. The box in the middle of the form is a listbox, not a textbox. You use the **AddItem** method to display the answer choices.

Using the listbox for the answer choices allows the user to select the correct answer by clicking on it in the box. When an answer is clicked, it is highlighted. When the user clicks Next, the answer is processed by reading the index of the item clicked in the listbox, and the score is updated. If the user chooses the wrong answer, a message box displays the correct choice.

To code the procedure:

1. Select the Test form.
2. Open the general declarations section of the form.
3. Enter the following code.

```
Sub Display2 (QNum As Integer)
   If QNum <= T.Current Then
      lblQNum = QNum
      txtQuestion = T.Question(QNum).Body
      lstChoice.Clear
      lstChoice.AddItem T.Question(QNum).Res1
      lstChoice.AddItem T.Question(QNum).Res2
      lstChoice.AddItem T.Question(QNum).Res3
      lstChoice.AddItem T.Question(QNum).Res4
      lstChoice.AddItem T.Question(QNum).Res5
      QNum = QNum + 1
   End If
End Sub
```

The **Display2** routine checks to see that the parameter passed, *QNum*, represents a valid question in the test. If it does, the label on the form indicating the question number is updated, the *Body* of the question is displayed in the upper textbox, and the answer choices are added to the listbox. The procedure ends by incrementing *QNum* to point at the next question in the test (if it exists).

CODING THE TAKE A TEST COMMAND

The cmdTake_Click() procedure, called by the Take a Test command, opens up the test file, transfers data from the file into the variable *T*, initializes the score to 0, sets *QNum* to 1, sets various labels, and displays the first question. The code that actually keeps the score is in the Next button.

To write the code for the command:

1. Select the Test form.
2. Click on Tests, then click on Take a Test.

3 Enter the following lines in the subroutine for this command:

```
Sub mnuTake_Click ()
    '—Local variable for filename
    Dim FName As String
    '—Read filename from form
    FName = lblFileName.Caption
    '—If there is no filename, go back to the Edit form
    If FName = "" Then
        MsgBox ("Choose a filename")
        Unload Me
        frmEdit.Show
        Exit Sub
    Else
        FName = lblFileName
        Open FName For Random Access Read As #1 Len = 30000
        Get #1, , T
        Close
        '—If Current is 0, there are no questions
        If T.Current = 0 Then
            MsgBox "The test has no questions."
        Else
            '—Initialize score to 0
            Scr = 0     ' current score on test
            lblTotal = T.Current
            lblTestName = T.Title
            lblOther = T.Other
            QNum = 1
            Display2 (QNum)
        End If
    End If
End Sub
```

CODING THE NEXT COMMAND BUTTON

In this procedure, the program reads a choice made by the user from the listbox, using the ListIndex property. This property contains a number corresponding to the choice highlighted in the box. The first choice in the box corresponds with ListIndex = 0.

After this index is read, it is converted to an uppercase character using the **Chr$()** function. The ASCII code of "A" is calculated using the **Asc()** function, the value of the ListIndex is added to that code, and the result is converted back into a character.

```
choice = Chr$(Asc("A") + txtChoice.ListIndex)
```

For instance, if choice *b)* is chosen as the correct answer from the listbox, the ListIndex will be set to 1. The ASCII code for "A" is 65. Then 1 is added to 65, giving 66. This value is converted back into a character by the **Chr$()** function. This character is the answer entered by the user, which must be compared with the real answer recorded along with the question as a part of the test.

The value of *choice*, which is the answer entered by the user, and the real answer from the test are compared. The **UCase$()** function is used so that case does not determine whether an answer is correct; either upper- or lowercase answers will result in the same value.

If the answer is correct, the score, *Scr*, is incremented. If not, the correct answer is displayed. *QNum* is incremented to point at the next question of the test. For a correct answer, 1 is added to the total of correct answers.

If *QNum* points to a valid question of the test, that question is displayed. When all the questions have been answered, the user receives the message: "the test is complete".

To enter this code:

1. Select the Test form.
2. Double-click the Next button to open the Code window. Enter the following code in the cmdNext_Click() subroutine:.

```
Dim Choice As String
'—Choice is built from the ASCII code for "A" and
'the ListIndex value of the answer chosen
Choice = Chr$(Asc("A") + lstChoice.ListIndex)
If Choice = UCase$(T.Question(QNum).Ans) Then
   Scr = Scr + 1
Else
   MsgBox "The answer was: " & T.Question(QNum).Ans
End If
'—Set QNum to the next question
QNum = QNum + 1
If QNum <= T.Current Then
   '—Display the next question
   Display2 (QNum)
Else
   MsgBox "the test is complete"
End If
```

CONCLUSION

A balance must be maintained in presenting a project such as this one. Should the program be presented as a "perfect" piece, with all the little problems fixed up, all glitches anticipated and trapped? When a complete work is presented, it is easy to get lost in the details. The details obscure the main ideas of the program and the logic that knits it together.

The approach taken here is that an unpolished program that works and is short enough to be understood is better than a program that is incomprehensible, but perfect.

You'll know you are really starting to become a programmer when you can see places where the code could be a little better, or you figure out how to write code to cover a possibility the program does not handle.

QUESTIONS AND ACTIVITIES

1. Assume the string variable *Name* has the value of "John", and *crlf* has the value *Chr$(13) & Chr$(10)*. Write the statements you would need to produce the following :
   "Dear John,
   It's over between us.
   Sincerely, Mary"

2. In an earlier program, you saw how to create a control array of textboxes on a form. If a control array were used for the textboxes txtresp1..txtresp5 representing the answers to a question, how could the code be modified in the **Display***(QNum As Integer)* procedure in the Edit form? Assume the textboxes in the control array have the name txtResp(). Write the code. (Hint: If you are not using a **For-Next** statement, you should rethink your solution.)

3. In the code for cmdDelete_Click (), what would happen if the line:

   ```
   For x = QNum To MaxValue - 1
   ```

   was replaced with:

   ```
   For x = MaxValue - 1 to QNum step -1
   ```

4. Explain each part of the condition that is tested in the **If-Then** statement in the cmdDelete_Click () routine:

```
If (Ys = "y" Or Ys = "Y") And (QNum >= 1) And (QNum <= T.Current) Then
```

5. What do you think would be the effect of executing **Display(0)**? Would this be a way to simplify the code used to blank the textboxes on the screen?

6. When the last question of a test is deleted, no questions are actually moved. *Current* is decremented. Why doesn't it matter that the question deleted is still in its spot of the array?
7. How is it possible to store a 25-question test using a single **Put** command?
8. Suppose an incorrect question is present in the test. How would you make sure the corrected question got into the test?
9. Rewrite the cmdNextQ_Click (index As Integer) routine using a **Select Case** statement.
10. As currently coded, the cmdAddTo_Click routine appends a letter to the beginning of each answer as it writes it to the array: *A), B), ..., E)*. If you are trying to add a corrected version of a question to the test, you wind up with two layers of letters: *A) A)*. Fix the code to take care of the problem.

Using File System Controls

Until now, pathnames have been entered by the user in bits and pieces from the keyboard and joined into a single string. In this section you'll see how to use the controls provided by Visual Basic to accomplish these same actions with the mouse instead of with the keyboard. Using the file system controls in your programs lets the user navigate around the file system to choose files and directories. The user does not have to remember and type in long path and filenames. Because nobody likes to type complete filenames, your programs will be perceived as much more friendly and easy to use.

Types of File System Controls

There are three file system controls: the Drive Box, the Directory Listbox, and the File Listbox. The Drive Box displays a combobox of existing drives from which the user can choose. The Directory Listbox contains a list displaying directories in a tree-like form. This list has the same appearance as the Directories list in the standard Windows Open and Save As dialog boxes. The File Listbox displays all files in a directory.

THE DRIVE BOX

If you add a Drive Box control to a form, the user of your program will be able to choose a disk drive from a combobox of available drives from

which the user can choose a drive. The name of the selected drive becomes the value of the Drive property of the control. The value of the Drive property can be changed at run-time. Therefore, you will not see this property listed in the Properties window of the control.

Double-clicking on the drive box icon in the Toolbox places a drive box on the active form. You can change the length, but not the height, of this control. When the program is run, the value of the Drive property is set to the default drive. See Figure 12-13.

Figure 12-13
Value of the property set to the default drive

When a user runs one of your programs that contains a drive box, a click on the arrow in the right-hand box displays a list of available drives. When the user selects a drive from that list, the control's Change event is executed. At that point, your Change event handler should set the path of the directory listbox to the name of the newly selected drive. (See the following section for more on the directory listbox control.) The directory listbox will respond by listing the root directory of the chosen drive.

Assuming the drive box has its default name, drive1, and the directory box has its default name, dir1, the code to write the drive name is:

```
Sub Drive1_Change ()
   Dir1.Path = Drive1.Drive
   Dir1.SetFocus
End Sub
```

THE DIRECTORY LISTBOX

Double-clicking on the directory icon in the Toolbox places a resizable directory listbox on the active form. This control lists the directory of the drive written into its Path property. Like the Drive property of the drive box, the Path property of the directory box is available only at run-time. A change to the Path property results in a change in the directory listing. To select a directory from those listed, double-click on the directory name.

When a user clicks on a directory name, it is selected. When a user double clicks, the Path property is updated to include the selected directory. The display in the directory listing window is updated accordingly. See Figure 12-14.

The new path information should be written to the Path property of the file box. The box updates its file listing to reflect the contents of the new directory.

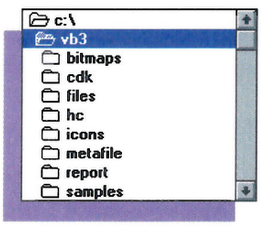


Figure 12-14
Display in the directory listing window

THE FILE LISTBOX

The file box lists the files contained in the directory written into its Path property, a property available only at run-time. Double-clicking on the file box icon in the Toolbox places a resizable file box on the active form. See Figure 12-15.

This box displays a scrollable listing of the files contained in the path written into the Path property. You set the Path property in the Change routine of the directory listbox. If an item in the file list is double-clicked, that filename is appended to the path information already contained in the Path property of the file box. See Figure 12-16.

Figure 12-15
Placing a resizable file box on the active form

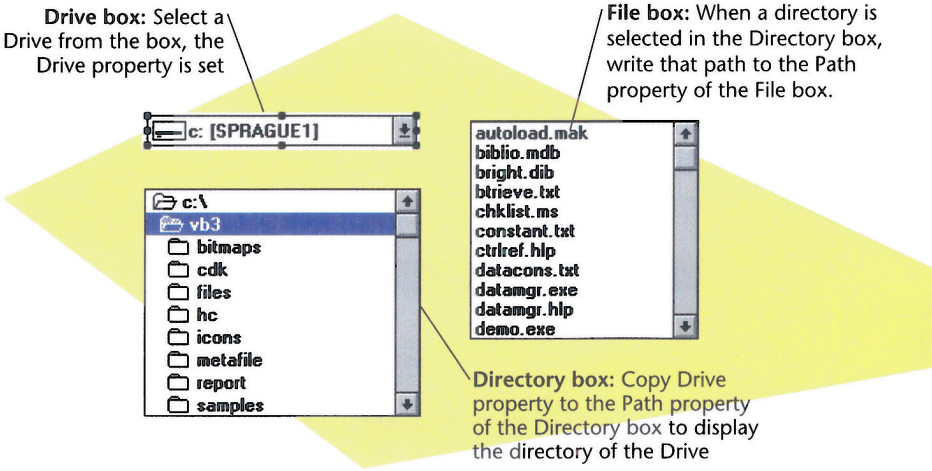


Figure 12-16
Working with the File listbox

Adding a Form to the Quiz Program

The Quiz program used an InputBox to prompt the user for the filename of the test file. This information was joined with other path information to build a complete pathname. The pathname was used to save and open files containing test data.

Adding the controls described above to a form allows the user to select a file by pointing and clicking existing paths and files. You should now enhance the Quiz program by adding a form to handle file and path names.

STARTING OUT

On this third form, you will be adding drive, directory, and file boxes, as shown in Figure 12-17. The form needs some additional objects as well:

- A number of labels
- A textbox
- A command button

Think about what the user needs to use this form effectively. You should place a label above the file listing containing an instruction for the user, such as "Choose file from list". If you do so, the user will understand the contents of that box. Above that label, you should place a second label with the caption Pathname. When a user selects a directory from the directory box, the Change event will update that label to show what path has been transferred to the file box. Of course, the file box updates its display and reflects the contents of the new directory.

At times, users will want to enter filenames from the keyboard rather than clicking on a selection from a list. To cover this requirement, you need to place a textbox below the file listing. Then place two more labels below that textbox. Use one of the labels to provide an explanation of the information provided in the other label. The program will display the complete pathname of the chosen or selected file in the lower of these two labels. Then, when the user clicks on the Return button, the program writes this pathname into the appropriate places in the Edit and Test forms of the Quiz project.

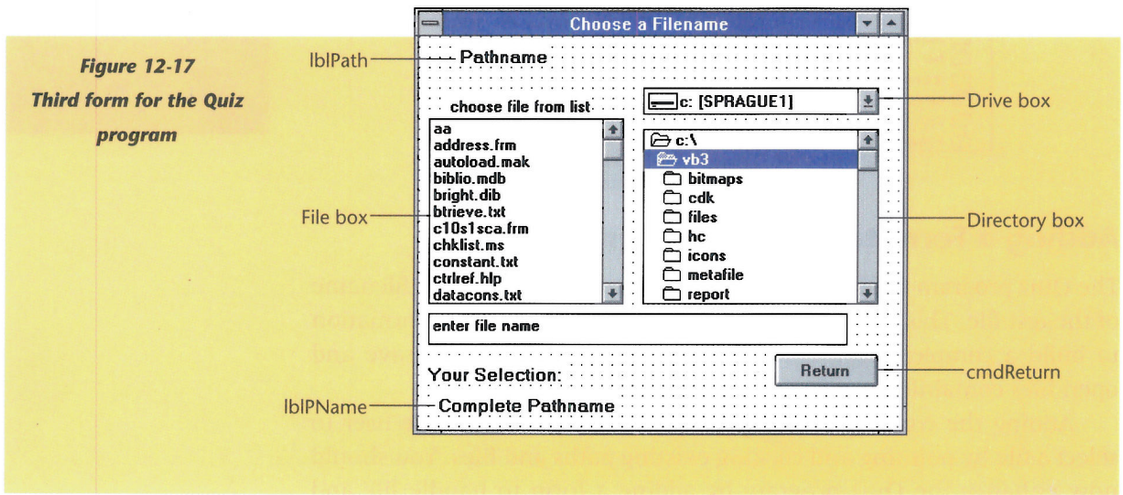


Figure 12-17
Third form for the Quiz program

BUILDING THE FORM

Follow these steps to build the third form for the Quiz project:

1. Open the Quiz project.
2. Choose New Form from the File menu.
3. Change the name of the form to frmFile. Change the caption of the form to **Choose a Filename**.

4 Double-click on the drive box icon. Position the drive box as shown in Figure 12-17.

5 Double-click on the directory box. Position the object.

6 Double-click on the file box. Position the object.

7 Create a long textbox for the file and pathname. Change the name of the box to **txtFName**. Change the text to **Enter the filename**.

8 Create a label for the top of the form. Change the name of the label to **lblPath**. Change the FontSize to 9.75. Change the AutoSize property to True.

9 Create a label for the bottom of the form. Change the name of the label to **lblPName**. Change the FontSize to 9.75. Change the AutoSize property to True.

10 Create a label for the bottom label. Change the caption to **Your selection:**

11 Create a label for above the file box. Change the caption to **choose file from list**.

12 Add a command button to the form. Change the caption to **&Return**. Change the name to **cmdReturn**.

CODING THE FILE FORM

Follow these steps to add code to the file form.

1 Select the File form from the Project window.

2 Click on View Code to open the Code window.

3 In the general declarations section of the form, enter the following lines.

```
Option Explicit
Dim FName As String
```

4 The path label at the top of the form is initialized to the default drive when the form is activated. Enter these lines in the Form_Load procedure:

```
Sub Form_Load ()
   lblPath = Dir1.Path
End Sub
```

5 When a new drive is selected, a Change event occurs. The Change event procedure writes the new drive name into the Path property of the directory box. Enter these lines in the Drive1_Change event procedure:

```
Sub Drive1_Change ()
   Dir1.Path = Drive1.Drive
   Dir1.SetFocus
End Sub
```

6 When a new directory is selected from the directory box, a Change event occurs. Code in this event writes the new pathname to the file box. Enter these lines in Dir1_Change:

```
Sub Dir1_Change ()
   File1.Path = Dir1.Path
   lblPath = "Path: " & Dir1.Path
   File1.SetFocus
End Sub
```

7 When a file in the file box is clicked, it is selected, then the path and file name are joined and displayed in the txtFName box on the form. The selected filename is available through the file box's FileName property. Enter these lines in the File1_Click procedure:

```
Sub File1_Click ()
   txtFName.Text = Dir1.Path & "\" & UCase(File1.FileName)
End Sub
```

8 When an entry in the file box is double-clicked, the complete pathname is joined with the filename and displayed in labels. Enter these lines in the File1_DblClick procedure:

```
Sub File1_DblClick ()
   '—Display the selected file name when DblClicked.
   lblPName.Caption = Dir1.Path & "\" & UCase(File1.FileName)
   txtFName.Text = Dir1.Path & "\" & UCase(File1.FileName)
   cmdReturn.SetFocus
End Sub
```

9 When the pathname changes, lblPath at the top of the form is updated to reflect the change. Enter these lines in File1_PathChange:

```
Sub File1_PathChange ()
   lblPath.Caption = "Path: " & Dir1.Path
End Sub
```

10 The textbox is used to enter the filename from the keyboard. When a name is entered and the Enter key is pressed, the path and file names are joined and displayed in the upper label. Focus is shifted to the Return button. Enter these lines in txtFName_KeyPress:

```
Sub txtFName_KeyPress (keyascii As Integer)
   If KeyAscii = 13 Then
      lblPName = Dir1.Path & "\" & txtFName.Text
      cmdReturn.SetFocus
   End If
End Sub
```

11 The Return command button writes the filename to the two other forms, the Edit and the Test forms. Enter these lines in the cmdReturn_Click procedure:

```
Sub cmdReturn_Click ()
   frmEdit.lblFileName = lblPName.Caption
   frmTest.lblFileName = lblPName.Caption
   frmFile.Hide
   frmEdit.Show
End Sub
```

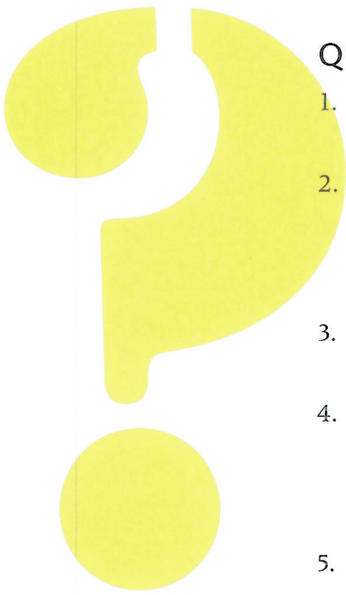
12 Save the form and project files.

13 Integrate the new form into the Quiz Project by inserting two lines of code in the mnuFile_Click procedure. Below are a few lines from that procedure. Two old lines are commented out and two new lines are added. Make the changes:

```
Sub mnuFile_Click (Index As Integer)
   Dim Msg As String
   '—Read the filename from the form
   FName = lblFileName
   '—If file name is blank, collect from user
   If (FName = "") Or Index = 1 Then
      '—FName = FileName()
      '—lblFileName = FName
      frmFile.Show    ' New line.
      Exit Sub        ' New line.
   End If
```

14 Run the program. Test the code by selecting New or Open.

15 Save the form, project, and code module files.



QUESTIONS AND ACTIVITIES

1. For a drive box, why does it make sense that the Drive property is only available at run-time?
2. The Path property of a directory box can be assigned string values and string values can be read from the Path property. What directory box event causes the contents of the Path property to change?
3. What happens when a new path is assigned to the Path property of a file box?
4. Add some file error handling to the mnuFile_Click event of the Edit form. If a bad file name is entered, the program crashes. Trap this error with an **On Error Go To** statement. The handler should call frmFile to get a filename from the user.
5. Add a Save As command to the File menu in the Edit form. This choice should always call frmFile to get a file and pathname from the user.
6. Add a separate menu item to the File menu in the Edit form just to collect a file and pathname.
7. The Test form opens files when the user wants to take a test. Add a file menu command to the menu in the Test form. Provide a single command to collect the file and pathname from the user.

The greatest common factor (gcf) is the largest number that divides evenly into two given numbers. The gcf of 12 and 16 is 4. You can find the gcf of very large numbers quickly with the Euclidean algorithm.

The code module is a collection of type definitions, variable declarations, and procedure definitions (both sub and function procedures) that are available to every form in a project.

Value parameters are values sent to subprograms without reference to the actual address of the value being sent. The values can be changed within the subprogram without affecting the value of the variable in the calling program.

When parameters are sent by reference to a subprogram, the subprogram is given the address of the value being sent. Because the subprogram knows the real location of the variable in memory, it can make changes to the value of the variable.

Parameters sent by value to a Visual Basic subprogram must be of the built-in Visual Basic types. A user-defined type may not be sent as a value parameter.

There are two ways to call a procedure:

- Subtract *a, b, Rslt*
- Call subtract(*a, b, Rslt*)

When command buttons are given the same name, they form a control array. The first button is assigned an index value of 0 and the second a value of 1. Subsequent buttons added to the array receive successive index values.

The **Hide** and **Show** methods deactivate and activate forms in your project.

A listbox uses ListIndex property indicating a selected entry in a listbox.

The **Asc()** function and the **Chr$()** functions are inverse functions. The first converts a character to its ASCII code. The second converts an ASCII code to its corresponding character.

The drive box allows the user to choose from a listing of active drives. The drive name is available at run-time in the Drive property.

The Change event is often used to transfer drive information from the Drive property of the drive box to the Path property of the directory box.

The directory box lists the directories of the drive specified in its run-time only Path property. The value of the Path property is changed by double-clicking an entry in the window. This path information is often written using a Change event procedure to the Path property of a file box.

The file box displays the files listed in the directory written into the Path property of the box.

1. The Mixed Numerals Problem

Rewrite the routines to display and read the fractions from the Fraction Handler program to accept and display numbers expressed as mixed numerals. Instead of entering 5/3rds, allow the user to enter 1 2/3rds. Instead of displaying the answer 31/30ths, display the mixed numeral 1 1/30th. You'll need extra textboxes, but you should be able to use the same fraction arithmetic routines.

2. Plotting 1/x Problem

Write a program that will plot the function **f(x) = 1/x** for values of *x* between 0 and 10. Use an **On Error Resume Next** statement to avoid any run time errors. Use a user-defined coordinate system in a picture box. Pick a large maximum value for the y axis.

3. The Two Correct Answers Program

Make a plan to modify the code of the Quiz program to allow two correct answers for each question. Your plan should mention all the areas of code and forms that change, giving specific examples of how you would change the code.

4. The Creating a True/False Test Program

Write a program to create and store a 20-question true/false test. Don't worry about any editing functions (such as deleting or updating); just accept questions and answers from a form, store in an array, and transfer to a file.

5. The Reading a Test Program

Write a program to read the file created in problem 4 and give the test to the user. At the end of the test, print the raw score and the percent correct.

6. The Tutorial Program

Pick one of your favorite activities and write a program that will display a tutorial for that activity. The program should allow the user to create a list of instructions guiding the user through the various parts of the activity. For each instruction, store an additional line with further explanation.

The program should also handle displaying the tutorial. The user should be prompted for a filename. The activities should appear one by one on the screen advancing one frame each time a button is clicked. An additional button should display the extra explanation.

Write the program so that, by assigning different file names, you can record instructions for any number of activities.

7. The Store Inventory Program

Write a program to maintain a store's inventory file. Each item in the file should have a name, a quantity, and a cost. The edit functions of the program should include the ability to add items, delete items, and update the quantity of items. The display portion of the program should display a listing of all the items in inventory along with all the information recorded about each item.

8. The Substitution Code Program

Using the **Asc()** and the **Chr$()** functions create a simple substitution code. Enter a string in a textbox, use **Mid$()** to peel it apart, and store the individual characters in an array (the array is really unnecessary). Convert each character to its ASCII code, add two (or three or whatever you like) and, using **Chr$()**, convert the code back into a character. Be sure the last letters of the alphabet are converted to the first letters (hint: you might want to use the Mod operator). Replace the original message entered in a textbox with the modified message.

```
For x = 0 To
ScaleWidth Step 500
every 500 twips
For y = 0 To
ScaleHeight Step
500
PSet (x, y), 0
Next y
Next x
```