Chapter 10

# Some Fundamental Algorithms

## GOALS

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

- Understand and use the simple exchange sort.
- Understand and use the bubble sort.
- Understand and use the comb sort.
- Write procedures using Variants that can be called from many programs.
- Understand and use the linear search algorithm.
- Understand and use the binary search algorithm.
- Open and use random access data files.
- Use various Visual Basic string functions.

## OVERVIEW

*In this chapter you learn common sorting techniques. These procedures put information in order, either alphabetically or numerically, ascending or descending. Once you can write a sorting program, you are no longer tied to the List box to get a sorted list. In the previous chapter's Phone List program, you sorted an array of Person records by converting their contents to strings, then adding those strings to a List box with the Sorted property set to* ***True****. The order in which you concatenated the fields of the records determined the sort order. However, often you will want to rearrange the records in the array in some sorted order. It is unneccessary and slow to use a List box to achieve this.*

*Sorting is often the first step in searching for information you need. For example, if you are looking for a specific file in the File Directory, one way to find it is to sort the files in a directory by extension or by date. If you sort by date and you know the day you created the file, you should be able to find the file quickly.*

*This chapter also presents two searching methods: the linear search and the binary search. Searches allow a program to find data in arrays of information. Array subscripts let you obtain the contents of an element once you know its location.*

*The two topics, sorting and searching, are tied together. Some searches depend on the fact that an array or file is sorted.*


# Sorting

Sorting means putting data in order. For strings, this usually means alphabetical order. For numeric values, you would place them in ascending or descending order.

Student names would be listed in alphabetical order. Your CD collection, if it is cataloged, is probably in alphabetical order by artist or by album title. How else would you find information, if it were not ordered? If you have very little data, you may not need to sort it. However, as soon as you have gathered a lot of information, such as all the book titles in a library, that information must be ordered in some fashion.

To sort data in a program, you use algorithms. An algorithm is a step-by-step solution to a problem such as sorting. There are many sorting algorithms. Some execute very quickly and some execute slowly. Typically, programmers classify sorting algorithms by how fast they execute.

Different situations call for different sorting algorithms. The perfect algorithm for one sorting problem may not be the best for another. As a result, you need to know several of these algorithms. This chapter introduces three different sorting algorithms: simple exchange sort, the bubble sort, and the comb sort.

## The Simple Exchange Sort

If you play cards, you may have used the simple exchange sort. Imagine that you are dealt 13 cards. To keep the example simple, assume all the

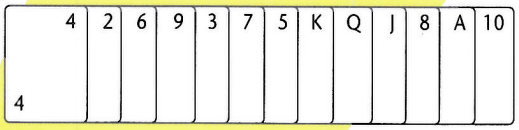



**Figure 10-1**
**Fanning 13 cards**

cards are of the same suit. Pick up all 13 cards and fan them out on the table, as in Figure 10-1.

The goal of the sort is to put all the cards in order from left to right. The sorting algorithm executes the sort by a series of exchanges. It compares two cards to each other, starting from the left. If the card to the right is smaller than the card on the left, they are switched with each other. Look at Figure 10-1. Immediately you notice the 2 is smaller than the 4. The first step of the exchange sort would be to rearrange them, as in Figure 10-2.

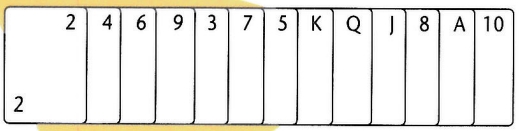



**Figure 10-2**
**The first exchange**

Now, the exchange sort compares each of the remaining cards with the current left-most card (2). None of the remaining cards is smaller than 2, so that card is in the correct position.

The next step is to compare the 4 with each card to the right of it. Compare the 4 with 6, with 9, with 3. The exchange sort would *switch* the 4 and the 3, as shown in Figure 10-3.

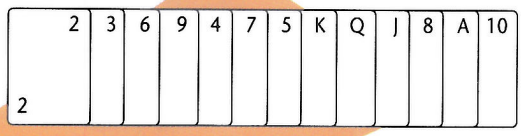



**Figure 10-3**
**Another switch**

As you can see, the switch has moved the 4 further out of order than it was. Still, the first two cards are in the correct order because no other card is smaller than 3. Next, you would start with the 6. As soon as you reach 4, you would switch the two cards, and 4 would now be in the correct position. The first three cards are now in order, as shown in Figure 10-4. You would continue until all the cards were in order from left to right, smallest to largest.

This sort algorithm is very slow. Sorting a large list with this algorithm could take hours. It has an advantage, however. The exchange sort is easy to program. To complete the algorithm, you need two **For-Next** statements, an **If-Then** statement, and some assignment statements. You may want to consider using this sort for small jobs. If your list holds fewer than a hundred items, the speed of your computer will make up for the inefficiency of the algorithm.

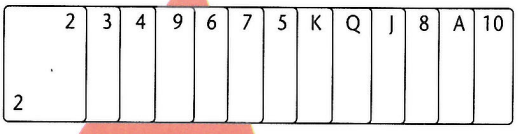



**Figure 10-4**
**The first three cards in position**

## CODING FOR AN EXCHANGE SORT

Try experimenting with an exchange sort, so you can become comfortable with the code. In this sample, you are sorting a list of random numbers from smallest to largest.

To code the sort:

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

2 Change the caption of the default form to **Simple Exchange Sort**.

3 Place two List Boxes on the form. Name the first **lstUnSorted** and the second, **lstSorted**.

4 Place three command buttons on the form. Change their captions to **&Generate**, **&Sort**, and **E&xit**. Change their names to **cmdGenerate**, **cmdSort**, and **cmdExit**. See Figure 10-5 for the completed form.

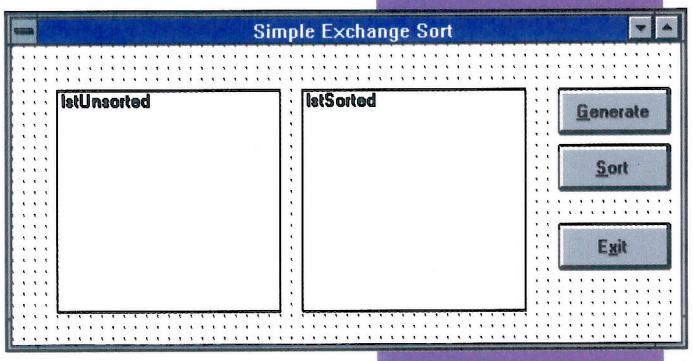



*Figure 10-5*
*Completed simple exchange sort form*

5 In the Code window, enter the **End** statement in the procedure for **cmdExit.**

6 Enter the following lines in the general declarations section of the form:

```
Const MAXITEM = 200
Dim A(MAXITEM) As Single
```

7 Enter these lines in the cmdGenerate procedure:

```
Dim x As Integer
For x = 1 To MAXITEM
   '—Generate random Single and assign to the array
   A(x) = Rnd
   '—Display unsorted numbers in listbox
   lstUnSorted.AddItem Str$(A(x))
Next x
```

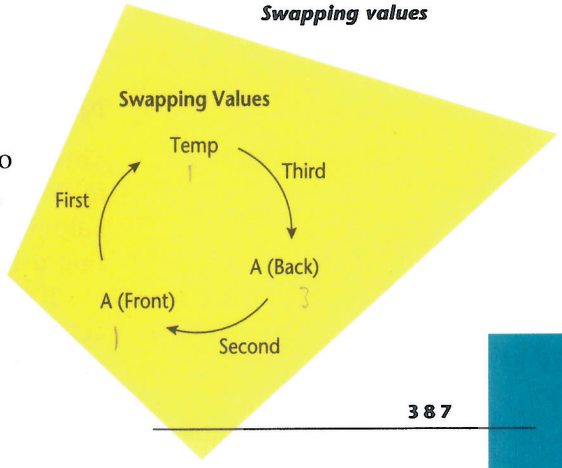
8 Enter these lines, clearing the List Boxes and declaring local variables, in the cmdSort procedure:

```
lstUnSorted.Clear
lstSorted.Clear
'—Declare pointers into the array
Dim Front As Integer
Dim Back As Integer
```

9 Enter the line to declare a variable to hold a swapping value. When two values are out of order, one is put into *Temp* temporarily. The second value is put into the first, and the old first value, stored in *Temp*, is put into the second. See Figure 10-6.

```
'—Declare temporary space to swap values
Dim Temp As Single
```

*Figure 10-6*
*Swapping values*



**10** Enter the line to set up the outer loop:

```
'—Outer loop goes from front to back of list
For Front = 1 To MAXITEM - 1
```

**11** Enter the line to set up the inner loop. The outer loop starts at *Front*. The inner loop starts one position to the right:

```
'—Inner loop starts one after the front and goes to the back
For Back = Front + 1 To MAXITEM
```

**12** Enter the lines to compare the values and swap if necessary:

```
            If A(Front) > A(Back) Then
                '—Values are out of order, swap
                Temp = A(Front)
                A(Front) = A(Back)
                A(Back) = Temp
            End If
        Next Back
    Next Front
```

**13** The array of values is sorted by this point in the code. Enter these lines to display the array in the second List Box:

```
'—Put sorted values into listbox
Dim x As Integer
For x = 1 To MAXITEM
    lstSorted.AddItem Str$(A(x))
Next x
```

**14** Save the form and project files.

**15** Run the program. Click on Generate to generate 200 random numbers and display in the first List Box. Click on Sort to sort the array and display in the second List Box. Give the sorting algorithm a minute to work.

## NOT VERY SMART

This routine is not very smart. Even if there is a very small value in *Front*, when the program finds a smaller value to the right, the values in those places are swapped, leaving the very small value that used to be in *Front* somewhere to the right.

If the routine is asked to sort an already sorted list, it happily compares each value with every other value, taking a lot of time and accomplishing nothing.

## The Bubble Sort

The bubble sort is a more complicated version of the simple exchange sort. It also finds pairs of values that are out of order and swaps them. The bubble sort, however can stop itself when the list is sorted. It stops by noticing that no swaps have occurred during an entire sweep of the list.

If a list is almost sorted and has just a few items out of order, the bubble sort may only run through the list a couple of times before it stops. The simple exchange sort keeps comparing items until all the items have been compared, whether or not the list is in order.

The bubble sort is really no faster than the simple exchange sort in the worst case. That is, there are unsorted arrays that will make the bubble sort compare every two elements. For example, arrays that are sorted in descending order will cause the bubble sort to make the maximum number of comparisons and swaps. On average, however, the bubble sort is more efficient. The best reason for learning this sort is because you will use it to build a much better sort, the comb sort.

### HOW THE BUBBLE SORT WORKS

The bubble sort compares elements in the list that are next to each other. The sort sweeps through the list over and over again, swapping elements when appropriate, little by little moving each element closer to its final sorted position.

The bubble sort uses a **True/False** variable that keeps track of whether values have been swapped. If the bubble sort sweeps through the entire array, comparing every pair of adjacent values, and there are no swapped values, the list must be in order.

To compare the bubble sort to the exchange sort, look at the card example again. See Figure 10-7.

Starting from the left and moving to the right, the bubble sort would compare two cards at a time. If those two cards are out of order, they are swapped. Here, then, you compare 4 with 2, then swap those cards. The exchange sort would now compare the 2 with the 6. The bubble sort instead compares the 4 with the 6. No swap occurs. Then, 6 is compared with 9, with no change. A swap is made when 9 is compared with 3, and so forth. When the first sweep is through, the cards appear as shown in Figure 10-8.

*Figure 10-7*
*13 unsorted cards*

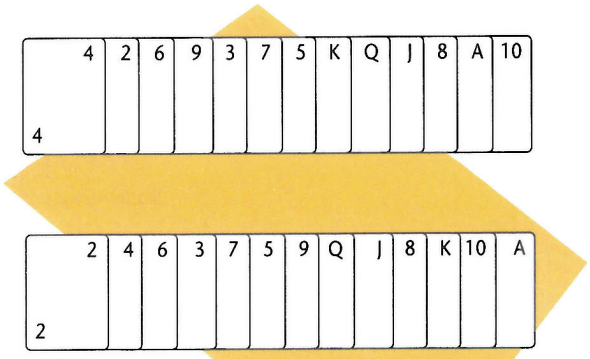



*Figure 10-8*
*After the first bubble sort sweep*

The largest element, the ace, has traveled to the back of the list. On the next sweep, because the ace is already in position, the comparisons

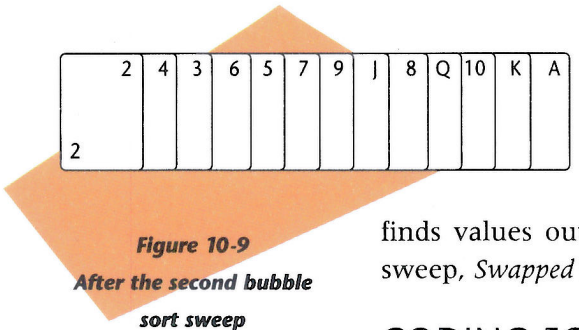



*Figure 10-9*
*After the second bubble sort sweep*

stop one short of the end. When the second sweep is complete, the king will be in position next to the ace. See Figure 10-9.

At the beginning of every sweep, a **True/False** variable called *Swapped* is set to **False**. If the sort finds values out of order, *Swapped* is set to **True**. At the end of the sweep, *Swapped* is tested. If there were no swaps, the sort is finished.

## CODING FOR A BUBBLE SORT

Try coding the bubble sort so that you will remember it. You will use the same form you built for the exchange sort.

To code a bubble sort:

1. Add a command button with the caption **&Bubble** and the name **cmdBubble**.
2. Enter the code for the bubble sort. Enter lines to clear the textboxes:

```
lstUnSorted.Clear
lstSorted.Clear
```

3. The array used to store the unsorted numbers may have already been sorted. Enter the line that calls the routine to generate a new set of random numbers:

```
cmdGenerate_Click
```

4. Enter the lines to declare local variables:

```
'—Declare pointers into the array
Dim j As Integer
Dim i As Integer
'—Declare space for swapping
Dim Temp As Single
'—Variable to keep track if there's been a swap
Dim Swapped As Integer
```

5. A bubble sort works with an indefinite loop. It stops when there are no more swaps. Enter the sort algorithm.

```
'—The indefinite loop stops when list is sorted
i = MAXITEM     ' LARGEST item ends up in spot i
Do
   Swapped = False
   For j = 1 To i - 1
      '—Compare adjacent elements
```

```
            If A(j) > A(j + 1) Then
               '—Swap
               Temp = A(j)
               A(j) = A(j + 1)
               A(j + 1) = Temp
               '—Swap did occur
               Swapped = True
            End If
         Next j
         '—Largest item is already in position
         i = i - 1
      Loop Until Not Swapped
```

6 Once the items in the *A* array are sorted, enter the lines to put them into the List box for display:

```
'—Put sorted values into List box
Dim x As Integer
For x = 1 To MAXITEM
   lstSorted.AddItem Str$(A(x))
Next x
```

7 In step three above, you entered the statement cmdGenerate_Click in the subroutine for cmdBubble. The effect is equivalent to a user clicking on the Generate button on the form. Insert this same statement into cmdSort and delete the Generate button from the form. The code from the button will appear in the general section where it can still be called. In cmdSort, after the lines clearing the listboxes, enter the line:

```
cmdGenerate_Click
```

8 Change the caption of the form to **Exchange and Bubble**.

9 Save the form and project files.

10 Run the program. Click on Sort. Click on Bubble. Time each procedure.

## The Comb Sort

The April 1991 issue of *Byte* magazine introduced the comb sort. This sort is an improvement in design over the bubble sort, executing in far less time. Instead of comparing adjacent elements, the comb sort compares elements that are separated by a *gap*. If the values are out of order, they are swapped. This approach to swapping moves values to their final

positions much more rapidly. Instead of comparing and swapping elements that are next to each other, the comb sort compares and swaps values that are farther apart.

The sort works by reducing the gap between the elements compared each time through the list, until the gap is reduced to 1. From this point on, the sort is the bubble sort. Because most of the elements are very close to their final positions by this point, the bubble sort makes short work of finishing the job.

## HOW THE COMB SORT WORKS

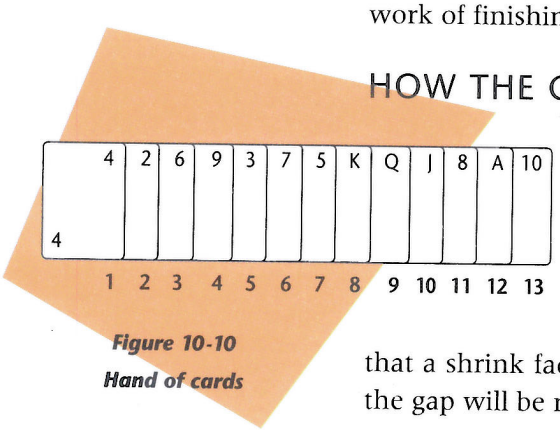



***Figure 10-10***
***Hand of cards***

To learn the comb sort, start with the same hand of cards (see Figure 10-10).

The gap is initialized to *MAXITEM*, the number of elements in the array. Before the first pass, the gap is reduced by a shrink factor. Experience and experimentation with this algorithm has demonstrated that a shrink factor of 1.3 yields the best performance. After each pass, the gap will be reduced by the same shrink factor.

```
'—Shrink factor is 1.3
Gap = Int(Gap / 1.3)
```

For the card example, *MAXITEM* is 13, so *Gap* is Int(13/1.3), or 10. The first time through the loop, the comb sort compares elements 10 spaces apart. The card in position 1 is compared with the card in position 11. The card in position 2 is compared with the card in position 12. The card in position 3 is compared with the card in position 13.

After the first pass, there are no swaps. After each pass, the gap is reduced.

```
Gap = Int(Gap / 1.3)
```

The new value of *Gap* is Int(10/1.3), or 7. During this pass, the card in position 1 is compared with the card in position 8, the card in position 2 with the card in position 9, and so forth. See Figure 10-11.

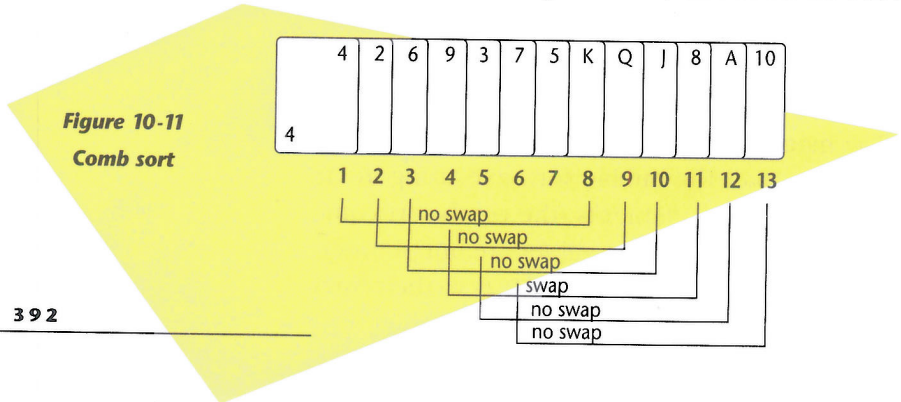



***Figure 10-11***
***Comb sort***

The result is shown in Figure 10-12.

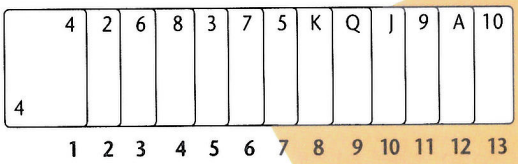
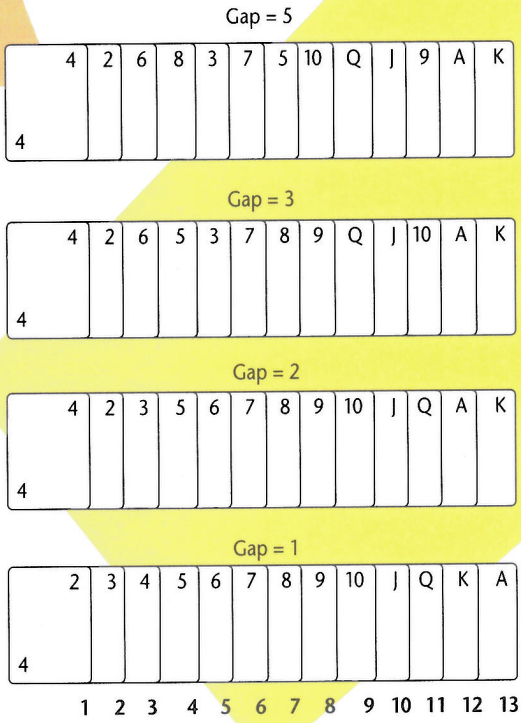



*Figure 10-12*
*Swapped cards*

*Figure 10-13*
*Passes through the cards with a comb sort*




The next value of *Gap* is Int(7/1.3), or 5. Figure 10-13 displays the next four passes, with gaps of 5, 3, 2, and 1.

The code for the comb sort is very similar to that for the bubble sort. The bubble sort compares elements next to each other:

```
If A(j) > A(j + 1) Then
```

The comb sort compares elements that are a gap apart:

```
If A(j) > A(j + Gap) Then
```

The name of the sort refers to the shrinking gap. Each time through the list, the distance between the compared values is reduced, just like using combs with finer and finer teeth on your hair.

## ADDING THE COMB SORT TO THE SORTING EXAMPLE

To add the comb sort to the sorting demo, follow these steps.

1. Add a command button to the form. Change its caption to **&Comb** and its name to **cmdComb**.
2. Change the form's caption to **Three Sorts**. See Figure 10-14.

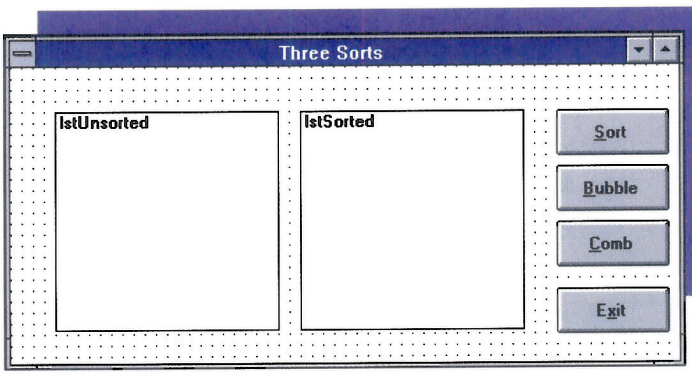



*Figure 10-14*
*Sorting form*

3 Add the comb sort code to the procedure for **cmdComb**. The program is over when there are no swaps and the gap is 1.

```
lstUnSorted.Clear
lstSorted.Clear
cmdGenerate_Click
'—Declare pointer into the array
Dim j As Integer
'—Declare the shrink factor as a constant
Const SHRINK = 1.3
'—Declare Gap
Dim Gap As Single
'—Declare space for swapping
Dim temp As Single
'—Variable to keep track if there's been a swap
Dim Swapped As Integer
'—The indefinite loop stops when list is sorted
Gap = MAXITEM
Do
   Gap = Int(Gap / SHRINK)
   '—The gap must not be less than 1
   If Gap < 1 Then Gap = 1
      Swapped = False
   For j = 1 To MAXITEM - Gap
      'Compare elements, Gap apart
      If A(j) > A(j + Gap) Then
         '—Swap
         temp = A(j)
         A(j) = A(j + Gap)
         A(j + Gap) = temp
         '—Swap did occur
         Swapped = True
      End If
   Next j
Loop Until Not Swapped And Gap = 1
'—Put sorted values into listbox
Dim x As Integer
For x = 1 To MAXITEM
   lstSorted.AddItem Str$(A(x))
Next x
```

4 Run the program. Click on each of the sorts and time each one.

5 Save the form and project files.

## THE COMB SORT IS FAST

The amazing thing about the comb sort is its speed. It doesn't look very different from the bubble sort, but it is much faster. Try increasing the size of the unsorted array. The difference in times to execute between the comb sort and the other sorts becomes more dramatic as the arrays become larger.

## QUESTIONS AND ACTIVITIES

1. Add a fourth command button to the form. Give it the caption **Built&In** and the name **cmdBuiltIn**. Add a third List box to the form. Call the List box **lstBuiltIn**. Set the Sorted property of the box to **True**. The code for the button should clear the List boxes and call cmdGenerate_Click. Copy the entries from lstUnSorted to lstBuiltIn. Run the program and time the results.
2. Modify the comb sort to try different shrink factors. Time the routine before and after each change. Try values of 5, 3, 2, 1.3, 1.1.
3. Write the lines of the simple exchange sort to sort an array, *dognames(30)*, an array of thirty strings.
4. One way to improve the simple exchange sort is to turn it into a selection sort. The selection sort works like the simple exchange sort, but instead of swapping any two values that are out of order, it only swaps at the end of the end of the inner loop. Instead of swapping, the **If** statement sets a pointer to the smallest element of the list. At the end of the inner loop, the smallest element is swapped with the top of the list. The selection sort makes the same number of comparisons, but it makes fewer exchanges. Modify the simple exchange sort to swap just once at the end of the inner loop.
5. Assume information about a list of cars was stored in three arrays, *Make()*, *Model()*, and *Year()*. Write the statements that would exchange the *Front* entry of each array with the *Back* entry.


*Section*

# Creating a General-Purpose Sorting Procedure

The comb sort command handler you created in the previous program is fast, but it is closely tied to that program. The command handler implements the comb sort, but also takes care of tasks that have nothing to do with sorting. The first two lines clear List boxes. The last lines fill a List box with the sorted array. Reusing this code in the future to sort an array would require much editing of the code once you cut and paste it. To save yourself work in the future, you need a general-purpose, or generic, comb sort procedure that you can use in any program.

As you know from Chapter 6, you can create modules of code that you can reuse in any form or any project. In this section, then, you will create a procedure in a code module. The procedure declares a dynamic array to hold the items to be sorted and uses a comb sort to arrange the elements of the array. Whenever you need a sorting mechanism in a project, you can then simply add this code module to the project.

Dynamic arrays are a feature of Visual Basic that you have not yet explored. In the programs you have written so far that used an array, you had to declare the exact amount of memory to be used by the array. **Dim A(20) As Integer**, for example, reserves exactly enough memory for 20 whole numbers.

You declare a dynamic array without specifying a size. This is perfect for a sorting procedure in a code module that you expect to reuse. After all, you are likely to want to sort arrays of many different sizes. The size of a dynamic array is fixed with a **ReDim** statement at the procedure level. By using the **ReDim** statement, you can change the size of the array to be sorted in the middle of the procedure. "Dynamic" refers to the fact that the size of the array changes at run-time. The **ReDim** statement refers to redimensioning, or changing the size of, an array.

You can define dynamic arrays with **Dim** or **Global** statements. In the code module you are about to build, you will declare the array as a global variable in the general declarations section of the module. Clicking on the Numbers command button on the form redimensions the *Numbers()* array to a particular size, fills the array with random numbers, then calls CombSort from the code module. The *Numbers()* array is sent as a parameter to CombSort. The contents of *Numbers()* are transferred to the *A()* array and sorted. See Figure 10-15.

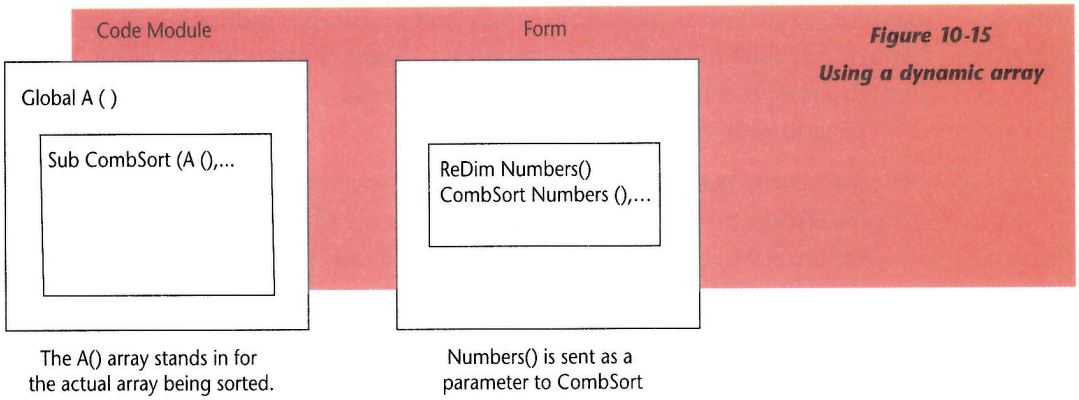



***Figure 10-15***
***Using a dynamic array***

## Setting up the Code Module: Stage 1

The code for the CombSort procedure is borrowed from the last section's project. In these steps you will add a code module to a new project and paste code written for the comb sort. This is the basis for the new code.

Follow these steps to start creating the code module:

1. Start Visual Basic.
2. Load the Three Sorts project, saved in the last section.
3. Select the form in the Project window and click on View Code.
4. Open the cmdCombSort procedure.
5. Select all the code, including the first and last lines of the procedure.
6. Press Ctrl+C to copy the code to the Clipboard.
7. From the File menu, select New Project.
8. Add a code module to the project. It has the default name **module1.bas**.
9. Select the code module in the Project window and click on View Code.
10. Paste the CombSort code into the general declarations section of the code module.
11. Delete the lines that are specific to the program in the last section. Delete the lines that clear the List boxes. Delete the lines from the end of the procedure that write the contents of the *A* array into a List box.

## Setting up the Code Module: Stage 2

In this second stage, you declare the dynamic array *A* and create a generic version of the comb sort from the code pasted in stage 1. The

generic comb sort is passed the array and an integer indicating the maximum number of items as parameters. When the routine is done, the sorted array is returned to the calling program.

To complete this stage:

1. Add two parameters to the first line of the sort—**A(), MaxItem As Integer**—between the parentheses on the first line. After specific references to the List boxes are deleted, the code should appear as shown here:

```
Sub CombSort (A(), MaxItem As Integer)
'-Entries in the array A() are assumed to start at subscript 1
'-Declare pointer into the array
Dim j As Integer
'-Declare the shrink factor as a constant
Const SHRINK = 1.3
'-Declare Gap
Dim Gap As Single
'-Declare space for swapping
Dim temp As Variant
'-Variable to keep track if there's been a swap
Dim Swapped As Integer
'-The indefinite loop stops when list is sorted
Gap = MaxItem
Do
   Gap = Int(Gap / SHRINK)
   '-The gap must not be less than one
   If Gap < 1 Then Gap = 1
      Swapped = False
   For j = 1 To MaxItem - Gap
      '-Compare elements, Gap apart
      If A(j) > A(j + Gap) Then
         '-Swap
         temp = A(j)
         A(j) = A(j + Gap)
         A(j + Gap) = temp
         '-Swap did occur
         Swapped = True
      End If
   Next j
Loop Until Not Swapped And Gap = 1
End Sub
```

2 Add the declaration of the global dynamic array to the general declarations section of the code module under Option Explicit:

```
Global A()
```

3 Return to the form by selecting the form name from the Project window. Change the caption of the form to **Using Dynamic Arrays**. Change the name of the form to **frmDynamic**.

4 Place a List box on the form. Change the name of the box to lstSorted.

5 Put three command buttons on the form. The caption of the first is **&Numbers**. Change the name to **cmdNumbers**. Change the caption of the second button to **&Strings**. Change the name to **cmdStringSort**. Change the caption of the third button to **E&xit**. Change the name to **cmdExit**. See Figure 10-16.

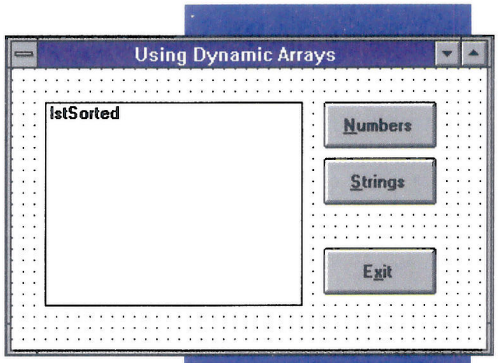



*Figure 10-16*
*Form design in progress*

6 Double-click on the Numbers button to open the Code window. Enter the following code to resize the array to a specific size as determined by *MAXITEMS*. The array is filled with random numbers. CombSort is called to sort the array. Once the array is sorted, it is displayed in the List box.

```
'—Declare number of items as constant
Const MAXITEMS = 200
'—Use ReDim to fix the size of the Numbers array
ReDim Numbers(MAXITEMS)
'—Generate random numbers and fill the array
Dim x As Integer
For x = 1 To MAXITEMS
   Numbers(x) = Rnd
Next x
'—Call CombSort with the array Numbers and MAXITEMS as parameters
CombSort Numbers(), MAXITEMS
'—Display sorted array in a List box.
For x = 1 To MAXITEMS
   lstSorted.AddItem Str$(Numbers(x))
Next x
```

7 Run the program. Click on the Numbers button. Scroll through the results in the List box.

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

## Adding the String Sort Driver

A generic implementation of the comb sort will sort any type of array. The declarations used in the code module never specify a data type for the array. When a data type is not specified, Visual Basic uses the Variant type. This type can represent any of the built-in data types. Given this flexibility, you can use the CombSort procedure for whole numbers, singles, strings, currency, or any of the other built-in types.

Follow the steps below to alter the previous demonstration to sort random alphabetic characters. The same generic comb sort routine used to sort numbers is used to sort characters.

1. Copy the code for the Numbers button and paste it into the procedure for the Strings button. Instead of random numbers, you will be modifying the program to generate random characters.
2. Change every instance of *Numbers* to *Letters*.
3. Remove the **Str$( )** function you used in the **AddItem** method for the List box. Because this part of the program sorts characters, you do not need to convert the data to a string before putting it into the List box.
4. Change the assignment statement that inserts values into the array:

   ```
   Letters(x) = Chr$(Int(26 * Rnd + 65))
   ```

   This statement generates a random integer between 65 and 90. These are the ASCII codes of the capital letters. It then converts the integer to a character and puts it into the array.
5. Run the program and click on Strings. Wait a moment, then scroll through the List box checking to see if the code worked.
6. Save the module, the form, and project files.

The code module you created in this program is ready for new tasks. You can add the module to any project you build so that you can sort lists.

### LIMITATIONS

Limitations of this approach to code reusability are the problems of arrays of user-defined types and information stored in more than one array. The CombSort routine, as saved, will sort one-dimensional arrays. The only flexibility you gain with this

approach is the size of the array is variable and the array may be any built-in data type.

This procedure cannot sort arrays of user-defined types. For example, it cannot be used to sort an array of Person records in alphabetical order (last name, then first name).

It cannot be used to sort multiple arrays in parallel. For example, if you stored people's names in two arrays, one for last names and another for first names, then sorting the collection of names would require that swaps be performed on the elements in the same positions in both arrays.

### EXERCISE

Turn the simple exchange sort and the bubble sort into generic sorts. Put them into the code module with the comb sort.


## Searching

Searching procedures let you find the information you want within a large expanse of data (arrays and files). When you want to look up information in a book, you don't want to have to read that book from the beginning just to find the passage you seek. Instead, you use the book's index if it has one. Fortunately, most books that can be used as references are organized in a way that speeds up searching. The topics in an encyclopedia are sorted alphabetically, so that you can quickly zero in on the volume and page containing the topic you seek. Other less-structured books have indexes, which can be thought of as tables that speed up searching. In the worst case, though, you would have to scan the book from front to back.

Searching for information among the data maintained by programs is no different. The kind of information you want to search for can suggest a way of organizing and sorting the data so that searches can be performed very quickly. Unless you have only very small amounts of data, you (and the users of your programs) will find it very slow to search by examining every record in an array or a file.

Searching procedures have become more important as storage mechanisms have improved. A CD-ROM, for example, holds 550

megabytes of data. That's enough for a multimedia encyclopedia, with images, sounds, and videos. How do you find any information on a CD-ROM that packed with data? The answer is linear and binary searches, which are covered in this section.

## The Linear Search

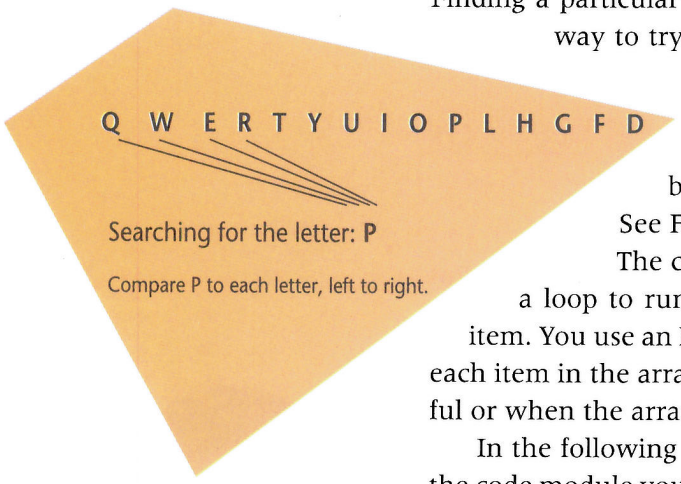



***Figure 10-17***
***A linear search***

Imagine a phone book in which the entries are not in alphabetical order. Finding a particular phone number would be extremely difficult. One way to try to find that number would be to perform a linear search. This type of search compares the search item with every entry in the array. When a match is found, or when every array entry has been examined unsuccessfully, the search is over. See Figure 10-17.

The code you need for a linear search is easy. You set up a loop to run through the array from the first item to the last item. You use an **If-Then** statement to compare the search item with each item in the array. The loop ends either when the search is successful or when the array has been thoroughly checked.

In the following exercise, you will add a linear search procedure to the code module you created for the comb sort. If you did not create that code module, create a new code module for this exercise.

### CREATING A LINEAR SEARCH PROCEDURE

In this section, you create a general-purpose linear search procedure. Like the general-purpose comb sort procedure of the previous sections, the linear search subroutine you create will work only with arrays of built-in types, and not with arrays of user-defined types.

You will be sending the following parameters to LinearSearch:

- Dynamic array, *A()*
- Search item, *SearchItem*
- Maximum number of items in the array, *MaxItems*
- Position found—a whole number, *Position*. The linear search procedure passes the result of the search back to its caller by setting the value of this parameter. If *SearchItem* is found, then *Position* is set to its array subscript. If it is not found, *Position* is set to 0. (Entries in the array are assumed to start at subscript 1.)

Follow these steps to add the LinearSearch procedure to the code module.

1. Open the code module.
2. Select New Procedure from the View menu.
3. Enter the name **LinearSearch** and click on Sub.
4. Enter these lines:

```
Sub LinearSearch (A(), SearchItem, MaxItems As Integer, Position As Integer)
'-Parameters:
'-A() - Dynamic array of the variant type
'-SearchItem - Item to be found - variant type
'-MaxItems - Number of items in the array
'-Position - 0 if item not found,
'equal to subscript of item; otherwise
'assume item will not be found
Position = 0
Dim x As Integer
'-Loop traverses entire list
For x = 1 To MaxItems
   If SearchItem = A(x) Then
      ' -If the item is found, record the position in the
      'array and leave the For loop.
      Position = x
      Exit For
   End If
Next x
End Sub
```

## ADDING A DRIVER TO THE FORM

Once the routine is in the code module, you can call it from a program. A routine that calls the LinearSearch procedure for testing purposes is called a driver. Follow these steps to add a driver to the form:

1. Open the Using Dynamic Arrays project, if it is not already open.
2. Add a command button to the form. Change the caption to **S&tring Search**. Change the name to **cmdStringSearch**. Change the Enabled property to **False**. The button should not be clicked until the Strings button has been clicked.
3. Open the Code window for the form.
4. In the general declarations section of the form, enter:

```
Dim Letters()
```

   This turns the *Letters* array into an array available to all the event procedures of the form, not just cmdStringSort_Click.

5 Open the cmdStringSort_Click procedure. Add this line as the last line of the procedure:

```
cmdStringSearch.Enabled = True
```

6 Open the cmdStringSearch_Click procedure and add the following code:

```
'—Char represents character to search for
Dim Char As String
Const MAXITEMS = 200
'—Position is the subscript at which the character is found
Dim Position As Integer
'—Collect search item from user
Char = InputBox$("Enter character for which to search ")
'—Call the LinearSearch procedure
LinearSearch Letters(), Char, MAXITEMS, Position
If Position = 0 Then
   MsgBox "Character not found."
Else
   MsgBox "Character found at position: " & Str$(Position)
End If
```

7 Save the form, module, and project files.

8 Run the program. Note the String Search button is dimmed. Click on Strings. Click on String Search.

9 Enter a character that occurs in the list (a capital letter).

10 Enter a character that doesn't occur in the list (a lowercase letter).

## SEARCHING AN UNORDERED LIST

The only way to search an unordered list is to examine every element of the list. The linear search examines every element.

If the list is ordered, a search that examines every element is wasteful. It doesn't take into account the order of the list.

There are many applications in which lists grow to extreme lengths. When the list is very long, as in the telephone directory of a large metropolitan area, searching an ordered list in a way that takes advantage of its order is far more efficient than searching an unordered list.

## The Binary Search

When searching a long ordered list, people take advantage of the fact that the list is sorted. For example, when you look up a person's name in the phonebook, you don't start reading the phone book from the beginning; instead, you first get close to the name you seek by repeatedly subdividing the pages of the book into two ranges: one which the name cannot fall within, and another which does contain the name.

Similarly, computer programs can implement this same idea when searching a long ordered list. The search algorithm that embodies this strategy of repeated subdivision is called the binary search. To get an idea of how efficient the binary search is, find a partner and play the dictionary game.

### THE DICTIONARY GAME

To play this game:

1. Get a fairly large dictionary.
2. Determine the number of entries in the dictionary. A typical desk-size volume will have fewer than 60,000 entries.
3. Ask your partner to select a word from the dictionary. Have your partner write the word on a piece of paper, but don't look at the word.
4. Tell your partner you will guess the word in 16 guesses, so long as you can have some help.
5. Make a note of the number of pages in the dictionary. Turn to the middle page. Pick a word off one of the pages, say the word to your partner, then ask whether the word written on the paper comes before or after your word.
6. If your partner says "before," discard the last half of the dictionary and repeat step 5. Use page one as the beginning and the middle page as the end.
7. If your partner says "after," the middle page becomes your starting point and the last page your ending point. Repeat step 5.

Each time you make a guess, you eliminate half the remaining words in the dictionary. See Figure 10-18.

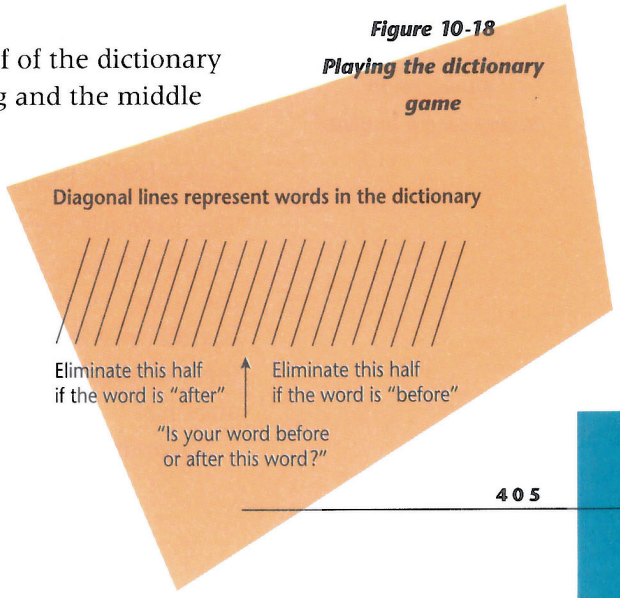



***Figure 10-18***
***Playing the dictionary game***

If you start with 60,000 words, the number of words remaining after each guess is shown in the following table. In practice, you can often spot the word before you get to the sixteenth guess.

| *Guess Number* | *Words Remaining* |
|---|---|
| 1 | 30,000 |
| 2 | 15,000 |
| 3 | 7,500 |
| 4 | 3,750 |
| 5 | 1,875 |
| 6 | 938 |
| 7 | 469 |
| 8 | 235 |
| 9 | 118 |
| 10 | 59 |
| 11 | 30 |
| 12 | 15 |
| 13 | 8 |
| 14 | 4 |
| 15 | 2 |
| 16 | 1 |

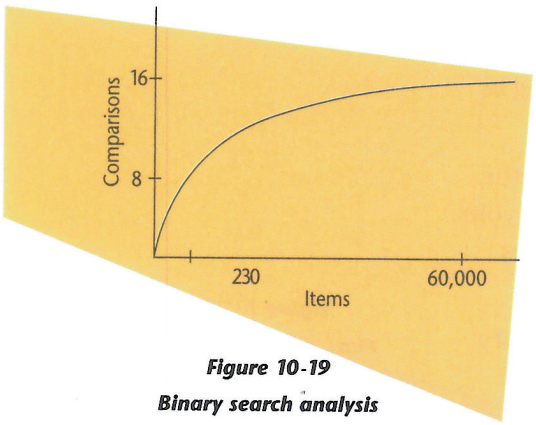



***Figure 10-19***
***Binary search analysis***

You have just used the binary search algorithm to find the word. If you look at a graph of this function you'll see it flattens out quite rapidly. Even a large change in the number of elements to search adds very few comparisons. See Figure 10-19.

In contrast, because the linear search examines each element of the array, on the average, it must examine half the elements to find the search item. In an array of 40,000 elements, instead of 16 comparisons—the maximum used by the binary search—an average of 20,000 comparisons are made by the linear search.

## CODING FOR THE BINARY SEARCH

The code for the binary search routine is not difficult, but often subtle errors creep in, preventing the search from being successful. First of all, the array of information to be searched must be sorted. The binary search only works when you can ask the question, "If the item searched for is not here in the middle, does it come before the middle element or after it?"

The idea behind the code is easy:

- Find the middle element of the array being searched.
- Compare the item you are trying to find with the middle element. If they match, the search is over.
- If the item you are trying to find is smaller than the middle element, restrict your search to the first half of the list.
- If the item you are trying to find is larger than the middle element, restrict your search to the last half of the list.

The dictionary game is an illustration of the algorithm described here.

As before, you add the code to the code module developed in the last section and added to in this section. The procedure for adding a binary search module to the Dynamic Arrays program is the same as for the linear search. The parameters sent to the binary search are identical to the ones used in the linear search.

Follow these steps to add a binary search procedure to the code module and a driver to the Dynamic Arrays form:

1. Enter the code module.
2. Select New Procedure from the View menu.
3. Enter the name **BinarySearch**, and click on Sub.
4. Enter these lines:

```
Sub BinarySearch (A(), SearchItem, MaxItems As Integer, position As Integer)
    '—Parameters:
    '—A() — Dynamic array of the variant type
    '—SearchItem — Item to be found - variant type
    '—MaxItems — Number of items in the array
    '—Position — 0 if item not found,
    'equal to subscript of item, otherwise
    'assume item will not be found
    position = 0
    Dim x As Integer
    '—Low and High point to the first and last
    'elements of a range of elements
    Dim Low As Integer, High As Integer, Md As Integer
    Dim Rslt As Integer     ' result of string compare
    '—Start with entire range of array values
    Low = 1
    High = MaxItems
    Do While Low <= High
       Md = (Low + High) \ 2     ' an integer division
       Rslt = StrComp(SearchItem, A(Md))
```

*continued*

```
        Select Case Rslt
           Case 0
              position = Md
              Exit Do    ' search is over
           Case -1    ' Search item is in first half of list
              High = Md - 1
           Case 1     ' Search item is in last half of list
              Low = Md + 1
        End Select
     Loop
End Sub
```

## ADDING A DRIVER FOR THE BINARY SEARCH MODULE

The steps for adding a driver to call the Binary Search module is identical to steps followed on pages 403 and 404 to add a driver for the Linear Search module. Use those steps as a model to add a command button and the code to call the Binary Search routine.

## HOW A BINARY SEARCH WORKS

To see how the binary search works, start with a sorted list of numbers, such as in Figure 10-20. Assume the search item is 560.

At the start, *Low* is 1 and *High* is 12.

$$Md = \frac{(Low + High)}{2} = \frac{(1 + 12)}{2} = \frac{13}{2} = 6$$

If *Md* is 6, then *A(6)* is 499. If 560 is greater than 499, then bring the lower bound up to the middle plus 1. *Low* becomes 7. *High* is still 12. *Md* becomes 9. See Figure 10-21.

Is 560 equal to 509? If 560 is greater than 509, bring *Low* up to *Md* + 1. See Figure 10-22.

*Md* points to the search item. The program ends when *Md* is assigned to *Position* and the **Exit Do** leaves the loop.

***Figure 10-20***
***Sorted list of numbers***

| 234 | 333 | 412 | 444 | 490 | 499 | 501 | 505 | 509 | 550 | 560 | 566 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
| Low | | | | | Mid | | | | | | High |

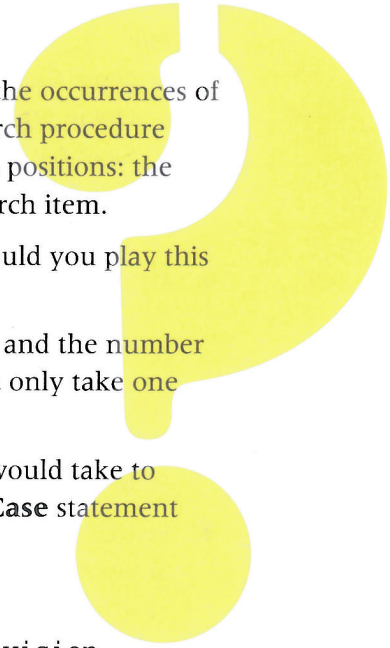
| 234 | 333 | 412 | 444 | 490 | 499 | 501 | 505 | 509 | 550 | 560 | 566 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
| | | | | | | Low | | Mid | | | High |

***Figure 10-21***
***Working with the binary search***

| 234 | 333 | 412 | 444 | 490 | 499 | 501 | 505 | 509 | 550 | 560 | 566 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
| | | | | | | | | | Low | Mid | High |

***Figure 10-22***
***Continuing to work with the binary search***

## QUESTIONS AND ACTIVITIES



1. There are times when a search should display all the occurrences of a value. The **Exit For** statement of the LinearSearch procedure prevents this. Rewrite LinearSearch to return two positions: the first occurrence and the last occurrence of the search item.
2. Play the dictionary game. With what other lists could you play this game?
3. Imagine that you are playing the dictionary game and the number of words in the dictionary is doubled. Why does it only take one additional question to find the chosen word?
4. In the binary search algorithm, write the lines it would take to replace the **StrComp()** function and the **Select Case** statement with a series of **If-Then** statements.
5. In the binary search algorithm, replace:

```
Md = (low + high) \ 2    ' an integer division
```

with:

```
Md = (low + high) \ 3
```

Describe the effect of this change.


# Searching Files and Arrays

In this section you develop a program to gather, display, search, and store information. The CD Database program stores the name of a CD, the artist, and the shelf location.

File management, arrays, user-defined types, sorting, and searching are all a part of this project. The methods used here can be applied to any collection or database of information.

## Starting Out

The start-up form displays a menu and a List box. The List box displays the CD list. The List box displays the results of searches. A dialog box is used to gather data from the user.

The menu offers the user a number of choices. The menu structure with explanation follows:

- File
  - New: Clears the List box and sets current number of entries to 0.
  - Open: Opens an existing file of CDs.
  - Save As: Saves a file of CDs using a path and file name entered through an InputBox.
  - Exit: Halts the program.
- Data
  - Enter: Calls the dialog box to enter new data.
  - Delete: Deletes entry whose index number is entered.
- Listing
  - By Artist: Displays entries with artist name on the first line.
  - By Name: Displays entries with name of CD on the first line.
- Search
  - By Artist: Collects artist name from user and lists CDs by that artist in the List box.
  - By Name: Collects name from user and lists CDs by that name in the List box.
- Sort: Sorts items by CD name.

## Creating the Menu

Follow these steps to set up the project and create the menu.

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

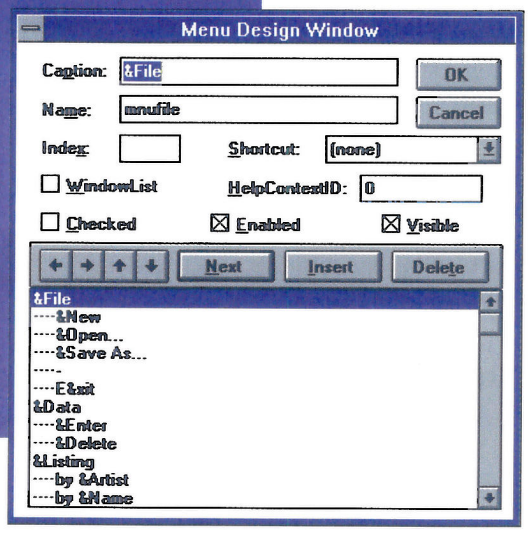



*Figure 10-23*
*Menu Design window with menu completed*

2. Add the file with the CombSort routine to the project. Choose Add File and add the code module with the CombSort procedure. The code will be borrowed later in the project.
3. Select the default form. Change the name of the form to **frmMain**. Change the caption to **CD Database**.
4. Select the form and open the Menu Design window.
5. Enter the menu items in Table 10-1, using the arrows to make submenus. When you are done, your Menu Design window should look like that shown in Figure 10-23.

**Table 10-1 Menu Items**

| Caption | Name | Index | Enabled |
|---|---|---|---|
| &File | mnuFile | | |
| &New | mnuNew | | |
| &Open | mnuOpen | | |
| &Save As | mnuSave | | No |
| E&xit | mnuExit | | |
| &Data | mnuData | | |
| &Enter | mnuEnter | | |
| &Delete | mnuDelete | | |
| &Listing | mnuList | | No |
| By &Artist | mnuListHandler | 1 | |
| By &Name | mnuListHandler | 2 | |
| &Search | mnuSearch | | No |
| By &Artist | mnuSearchBy | 1 | |
| By &Name | mnuSearchBy | 2 | |
| So&rt | mnuSort | | No |

## Creating the Code Module

The code module contains the definition of CDType. CDType is a data type with three fields:

- Name, a 30-character field for the name of the CD
- Artist, a 20-character field for the artist
- Location, a 15-character field used to specify the location of the CD in the collection

A global array is declared to hold the entries. The values of the array are of the CDType defined above. A global integer variable, *Current*, is used to keep track of the current number of entries in the array.

Follow these steps to set up the code module:

1. From the File menu, choose New Module.
2. In the general declarations section, enter the following code:

```
Option Explicit
Type CDType
'—Record length is 30+20+15 = 65
     Name As String * 30
     Artist As String * 20
```

*continued*

```
        Location As String * 15
End Type
Global CDList(1 To 100) As CDType
Global Current As Integer
```

3 Save the file.

Declarations made in the code module with **Global** are available to all procedures in all forms throughout the project.

## Creating the Dialog Box

The dialog box is a form used to collect the name, artist, and location of the CD. The dialog form contains three textboxes, three labels, and a command button. The command button reads the information from the textboxes, and transfers the information into the array.

Follow these instructions to set up the dialog box.

1 Choose New Form from the File menu.

2 Select the form. Change the name of the form to **frmData**. Change the caption of the form to **Enter CD Data**.

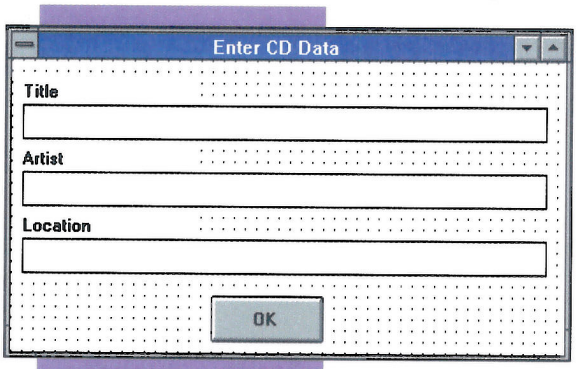



*Figure 10-24*
*Enter CD Data form*

3 Figure 10-24 above shows the layout of the form. Put three textboxes on the form. Name them **txtName**, **txtArtist**, and **txtLocation**. Delete the text from all three.

4 Put three labels on the form, one above each textbox. Change the captions to **Title**, **Artist**, and **Location**.

5 Place a command button on the form. Change the name of the button to **cmdOK**. Change the caption of the button to **OK**.

6 Double-click on the command button to open the Code window.

7 Enter the following code for cmdOK:

```
'—Declare temporary storage place for form's data.
Dim Transfer As CDType
'—Check to see if the name of the CD has been
'entered. If it has, collect the rest of the
'information from the form.
If Trim$(txtName) <> "" Then
   Transfer.Name = txtName
   Transfer.Artist = txtArtist
   Transfer.Location = txtLocation
```

```
    '—Update Current and load information into the array.
    Current = Current + 1
    CDList(Current) = Transfer
Else
    '—If the name of the CD has not been entered,
    'display an error message and return to the
    'calling program.
    MsgBox "Error: Enter complete information!"
End If
Unload Me
```

**Unload Me** not only hides the form from view, it also unloads the form from memory. When the form is displayed again in response to a **Show** statement, the form is loaded in memory. This ensures the Form_Load procedure is executed.

8 Enter the following code into the Form_Load procedure of frmData:

```
txtName = ""
txtArtist = ""
txtLocation = ""
```

9 Save the form.

## Adding Code: Stage 1

In this first stage, after the main form is set up, you enter the code to turn on the dialog box. The dialog box prompts the user to enter information about the CD.

The main form displays entries in a List box. The user interface has already been created. Follow these steps to add a List box to the start-up form and enter the code. Refer to Figure 10-25.

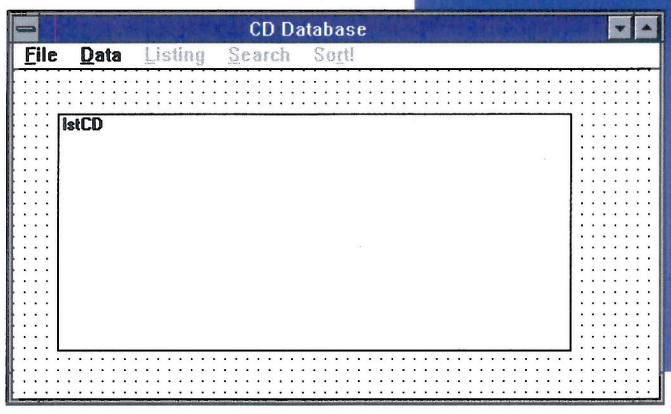



*Figure 10-25*
*Form for the project*

1 Select frmMain from the Project window.

2 Put a List box on the form. Change the name to **lstCD**.

3 Select the form in the Project window and click on View Code. In the general declarations section make sure the following line appears.

```
Option Explicit
```

4 Enter the **End** command in the procedure for mnuExit.

5 Enter this code in the mnuEnter_Click() procedure.

```
'—Load and show the data collection dialog box.
frmData.Show
'—Switch on the disabled menu items.
mnuSave.Enabled = True
mnuList.Enabled = True
mnuSearch.Enabled = True
mnuSort.Enabled = True
```

6 Run the program. Click on Data. Click on Enter. The dialog box should appear.

7 Stop the program by selecting File, then Exit.

8 Enter the following lines in the mnuListHandler_Click routine.

```
'—Clear the display ListBox.
lstCD.Clear
'—List all entries
'—List each field on a separate line. If index = 1
'list the artist first. If index = 2, list the name
'of the CD first. Prepare each line and add to the
'listbox.
Dim x As Integer
For x = 1 To Current
   If index = 1 Then    ' by Artist
      lstCD.AddItem Str$(x) & ": " & CDList(x).Artist
      lstCD.AddItem "    " & CDList(x).Name
      lstCD.AddItem "    " & CDList(x).Location
   Else
      lstCD.AddItem Str$(x) & ": " & CDList(x).Name
      lstCD.AddItem "    " & CDList(x).Artist
      lstCD.AddItem "    " & CDList(x).Location
   End If
Next x
```

9 Run the program. Click on Data. Click on Enter. Enter data into the dialog box and click OK.

10 Repeat step 9.

11 Click on Listing. Click on By Artist. The List box should display items just entered. Click on Listing. Click on By Name.

12 Halt the program.

## Adding Code: Stage 2

The code you add in this second stage enables the Delete command in the menu. This command deletes the current entry in the CD list. At this point, even though a number of commands, particularly the File commands, have not been entered, it is possible to run the program and test it.

This stage also covers the addition of code to create a new file, open an existing file, and save the current file.

To enter the stage 2 code:

1 Enter the following lines in mnuDelete_Click.

```
'—Variable to hold index number of item to delete
Dim ItemNumber As Integer
Dim x As Integer
'—If Current isn't greater than 0, there is no list
If Current > 0 Then
   '—Set up error checking.
   On Error GoTo DelError
   '—Collect item number from the user
   ItemNumber = Val(InputBox("Enter number of CD to delete:"))
   '—The ItemNumber could be too big, go to the error handler.
   If ItemNumber > Current Then GoTo DelError
   '—Delete ItemNumber by moving all the items below it up
   For x = ItemNumber To Current
      CDList(x) = CDList(x + 1)
   Next x
   '—Reset Current to reflect the loss of an item
   Current = Current - 1
End If
'—Exit procedure before blundering into error handler code
Exit Sub
'—Error handler
DelError:
   MsgBox "A bad value has been entered"
Exit Sub
```

2 Run the program. Add a couple of entries. Choose Data, then Delete. Enter the number of the first item.

3 Choose Listing to see if item has been removed.

4 Halt the program.

5 Enter the following lines in mnuNew_Click.

```
'—Choosing New clears the ListBox and resets
'—Current to 0. This effectively clears the array.
Current = 0
mnuSave.Enabled = True
lstCD.Clear
```

6 Enter the following lines in the mnuOpen_Click routine:

```
Dim FName As String
'—Get file name from user
On Error GoTo ErrorHandler2
FName = InputBox("Enter path and file name:")
'—Random access file is used.
Open FName For Random Access Read As #1 Len = 65
Dim x As Integer
'—Opening the file, Current starts at 0
Current = 0
Do
   '—Current is incremented.
   Current = Current + 1
   '—Data is collected from the file.
   Get #1, , CDList(Current)
   '—Loop executes until the file is empty.
Loop Until EOF(1)
'—EOF lets code go until Get fails to collect data
'—At that point, Current is one too many
Current = Current - 1
Close #1
'—Turn on menu items. Now that a file is open, the
'data can be saved, listed, searched, and sorted.
mnuSave.Enabled = True
mnuList.Enabled = True
mnuSearch.Enabled = True
mnuSort.Enabled = True
Exit Sub
'—Error handler
'—Label names like ErrorHandler cannot recur in the form.
ErrorHandler2:
   MsgBox "Illegal File or Path Name. Error #: " & Str$(Err)
   Exit Sub
```

7 Enter the following lines in mnuSave_Click:

```
Dim FName As String
'-Get file name from user
'-Set up On Error statement
On Error GoTo ErrorHandler
FName = InputBox("Enter path and file name:")
'-Open file for random access.
Open FName For Random Access Write As #1 Len = 65
Dim x As Integer
'-Write (Put) each record into the file.
For x = 1 To Current
   Put #1, , CDList(x)
Next x
Close #1
Exit Sub
'-Error Handler code
ErrorHandler:
   MsgBox "Illegal File or Path Name. Error #: " & Str$(Err)
   Exit Sub
```

## Adding Code: Stage 3

The third stage adds code to sort and search the list of CDs. The search routine is the linear search, and the sorting routine is the comb sort from the beginning of the chapter. You can copy the code from earlier programs and paste it into the routines here, or enter the code directly from the listing.

A linear search is used to find items by particular artists or by names. The search is used to display each of the items that match the search string. A match of the first five characters is sufficient to display the item. Actual lines to display items in the List box can be copied and pasted from the Listing routine.

1 Enter these lines in mnuSearchBy_Click.

```
'-Can't use the generic search because of the
'array of user-defined type
'-Declare variable for search item
Dim SearchItem As String
Dim x As Integer
'-Clear the List box
lstCD.Clear
```

*continued*

```
'–The value of index reflects which menu item has been
'chosen, by artist or by name
If index = 1 Then    ' search by artist
   '–Enter the name of the artist, the loop will print
   'each entire entry whose artist name matches SearchItem
   SearchItem = InputBox("Enter the name of the Artist:")
   For x = 1 To Current
      '–Compare just the first 5 characters of each string
      '–Display each entry where the first 5 character match
      If Left$(SearchItem, 5) = Left$(CDList(x).Artist, 5) Then
         '–Add items to listbox
         lstCD.AddItem Str$(x) & ":" & CDList(x).Artist
         lstCD.AddItem "    " & CDList(x).Name
         lstCD.AddItem "    " & CDList(x).Location
      End If
   Next x
End If
If index = 2 Then    ' search by name
   '–Repeat the whole procedure with search by name
   SearchItem = InputBox("Enter the name of the CD:")
   For x = 1 To Current
      If Left$(SearchItem, 5) = Left$(CDList(x).Name, 5) Then
         lstCD.AddItem Str$(x) & ":" & CDList(x).Name
         lstCD.AddItem "    " & CDList(x).Artist
         lstCD.AddItem "    " & CDList(x).Location
      End If
   Next x
End If
```

**2** Copy the code from the CombSort routine. Paste the code into mnuSort_Click. Change the references as follows:
MaxItems → Current
A() → CDList()

*Temp* is declared as type CDType, instead of Variant type. The **If** statement that performs the comparison is altered to compare the *Name* field of the array.

```
Sub mnuSort_Click ()
   '–Declare pointer into the array
   Dim j As Integer
   '–Declare the shrink factor as a constant
   Const SHRINK = 1.3
```

```
    '—Declare Gap
    Dim Gap As Single
    '—Declare space for swapping
    Dim temp As CDType
    '—Variable to keep track if there's been a swap
    Dim Swapped As Integer
    '—The indefinite loop stops when list is sorted
    Gap = Current
    Do
       Gap = Int(Gap / SHRINK)
       '—The gap must not be  less than one
       If Gap < 1 Then Gap = 1
          Swapped = False
       For j = 1 To Current - Gap
          '—Compare elements, Gap apart
          If CDList(j).Name > CDList(j + Gap).Name Then
                ' —Swap
                temp = CDList(j)
                CDList(j) = CDList(j + Gap)
                CDList(j + Gap) = temp
                '—Swap did occur
                Swapped = True
          End If
       Next j
    Loop Until Not Swapped And Gap = 1
End Sub
```

3 Run the program. Enter several items. Save the file. Provide an entire pathname, for example, **c:\temp\cdlist.cdl**.

4 List by artist and by name.

5 Sort the list.

6 Search for an entry by name and by artist.

7 Save the file.

8 Halt the program. Start the program again and open the data file.

9 List the file.

10 Halt the program.

11 Remove the CombSort file from the project. Save the project. Save two forms, one code module, and the project files.

## QUESTIONS AND ACTIVITIES

1. After a file has been opened, the file name provided should be the default name for subsequent file activities. Modify the CD Database project to save the name of the file from the beginning to the end of the program.

2. The CD Database project sorts only by the name of the CD. Modify the program to give the user the option to sort by artist.

3. In the search routine, what would be the effect of changing:

```
If Left$(SearchItem, 5) = Left$(CDList(x).Artist, 5) Then
```

to:

```
If SearchItem = Artist Then
```

Make the change and test the program. Change the program back to its original form.

4. Disabling menu items is a kind of error handling. If an item is not appropriate, it cannot be chosen. On the other hand, menu items that are disabled cannot be viewed by someone browsing the program. Remove the statements that enable and disable the menu items.

5. In the procedure that deletes an entry the following code is used:

```
For x = ItemNumber To Current
   CDList(x) = CDList(x + 1)
Next x
```

What would be the effect of substituting the following code:

```
For x = Current To ItemNumber Step -1
   CDList(x - 1) = CDList(x)
Next x
```

Make the replacement and check the results. Restore the original code.


The simple exchange sort is about three lines long: if the array to sort is *A()*, containing *MaxItems* items, the following lines will sort the array:

```
For Front = 1 To MaxItems - 1
   For Back = Front + 1 To MaxItems
   If a(Front) > a(Back) Then
         ' swap the items
   Next Back
Next Front
```

The bubble sort works by comparing adjacent items in the list and swapping out-of-order entries. An advantage of the bubble sort is that it will stop when the items are in order.

The comb sort is a modification of the bubble sort. Instead of comparing adjacent elements, it compares elements that are a certain gap apart. The gap between the compared elements is reduced until finally, the comb sort becomes the bubble sort. It is a very fast sort.

This **Open** statement:

```
Open "c:\temp\cdfile.dat" For Random Access Read As #1
```

opens the file **c:\temp\cdfile.dat** as a random access file for read-only access as file number 1.

The **Put** statement writes information to a random access file:
**Put** #<*file number*>, <*record number*>, <*variable name*>

The **Get** statement reads information from a random access file:
**Get** #<*file number*>, <*record number*>, <*variable name*>

The **EOF()** function is **True** when the program tries to get from a file that is empty (because all the data has been read).

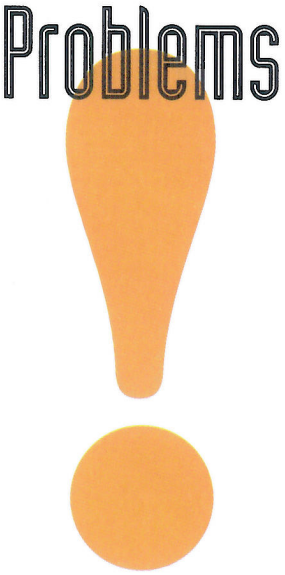
The **On Error Go To "label"** statement shifts program flow to an error-handling routine when a run-time error occurs.

A linear search checks every item of an array looking for the occurrence of a string or value. A simple loop is used to cycle through the items of the array.

A binary search tests the middle element of a sorted array. If the search item is the middle element, the search is over. If the item sought is less than the middle element, search the first half of the list. If the item is greater than the middle element, search the last half of the list.

The **Left$()** function:
**Left$**( *string, number of characters)*

takes the first number of characters from the string and returns the result as a string.




1. Searching for a Random Number Problem
Write a program to generate 100 random integers between 100 and 1000. Store the numbers in an array. Search the array for an occurrence of 750 and display its position (if it occurs) in the array.

2. Searching Through an Ordered List Problem
Write a program to generate 100 random integers between 100 and 1000. Store the numbers in an array. Use the comb sort to put the numbers in ascending order. Use a binary search to look for a number entered from the keyboard using an InputBox and display its position (if it occurs) in the array. Stop the program when the number 0 is entered from the keyboard.

3. Reversing the Characters Problem
Write a program to enter the user's name from the keyboard using an **InputBox$**. With a **For-Next** loop and a **Mid$** function, display the letters of the name in reverse order.

4. Alphabetizing the Characters Problem
Write a program to enter a line from the keyboard using an **InputBox$**. Use a sorting routine (your choice) along with the **Mid$** function to put the characters in order and display the results. The **Mid$** function will return a character of a string as its value. Look up its syntax in Help. Enter the line: "The quick brown fox jumps over the lazy dog." to test your program.

5. The Car Database
Using the following user-defined type:

```
Type Car
   make as string*20
   model as string*20
   year as integer
   cost as currency
End Type
```

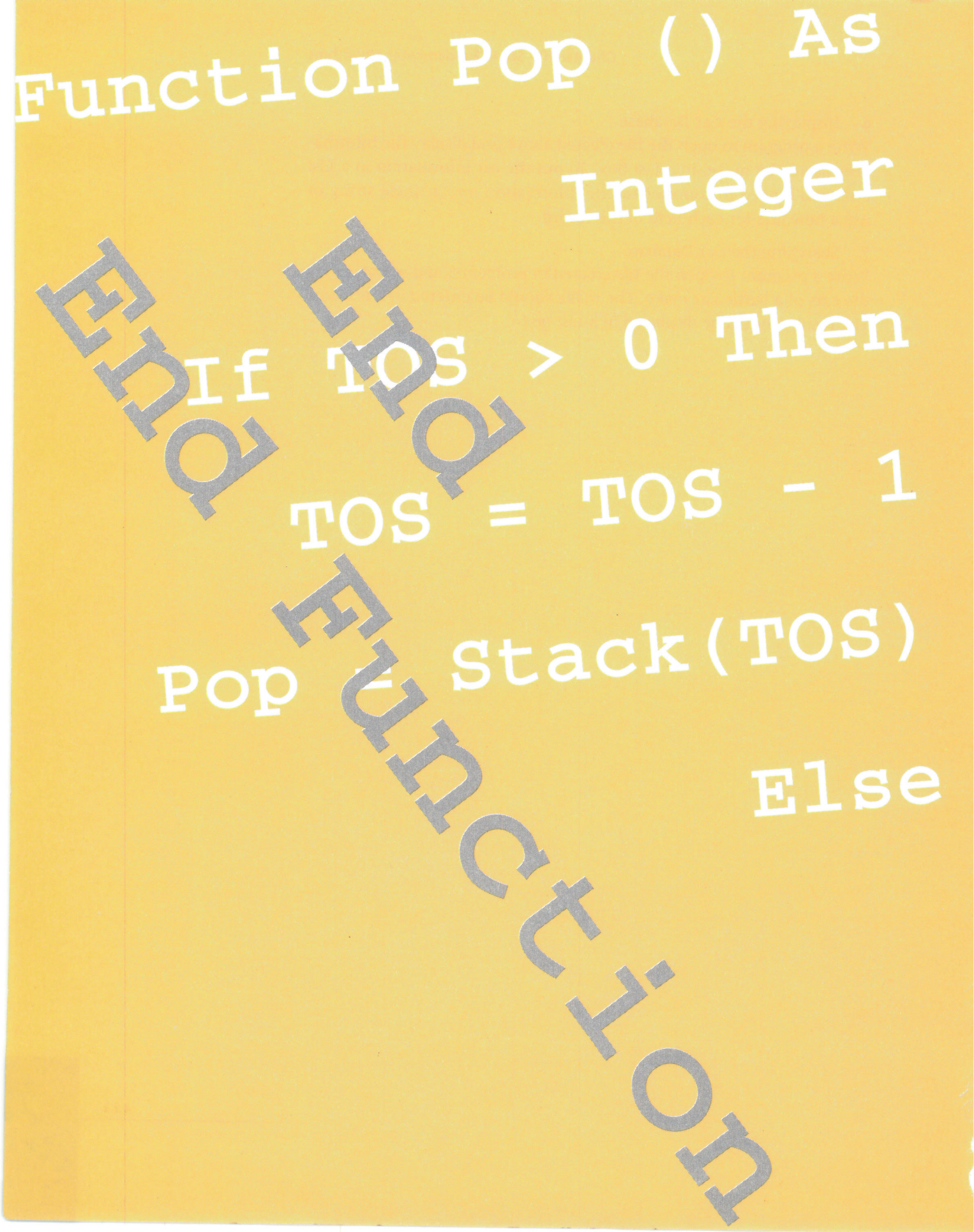
Write a program to set up an array with a capacity of 20 of the Car type. Allow the user to enter data into the array (keeping track of how many elements of the array are full), and then save the array in a file named **c:\...\car.dat**.

6. Displaying the Car Database

Write a program to open the file created above and display the information, sorted by make, in a List box. To put the car information in a List box, you'll need to concatenate the information into a single string or use a new line for each field of the record.

7. Searching the Car Database

Write a program to open the file created in problem 5 and search for all the cars of a particular make. The make should be entered in a textbox. The results should be displayed in a List box.


Function Pop () As
Integer
If TOS > 0 Then
TOS = TOS - 1
Pop = Stack(TOS)
Else
End Function
End
End