Chapter 8

Advanced Data Handling Concepts

Chapter Introduction

In this chapter, you will learn about:

- The need for sorting data
- The bubble sort algorithm
- Sorting multifield records
- The insertion sort algorithm
- Multidimensional arrays
- Indexed files and linked lists

Chapter 8: Advanced Data Handling Concepts: 8-1 Understanding the Need for Sorting Data
Book Title: Programming Logic and Design
Printed By: Ronald Suchy (rsuchy@mayfieldschools.org)

8-1 Understanding the Need for Sorting Data

When you store data records, they exist in some type of order; that is, one record is first, another second, and so on. When records are in **sequential order** (The arrangement of records when they are stored one after another on the basis of the value in a particular field.) , they are arranged one after another on the basis of the value in a particular field. Examples include employee records stored in numeric order by Social Security number or department number, or in alphabetical order by last name or department name. Even if the records are stored in a random order—for example, the order in which a clerk felt like entering them—they still are *in order*, although probably not the order desired for processing or viewing. Such data records need to be sorted, or placed in order, based on the contents of one or more fields. You can sort data either in ascending order, arranging records from lowest to highest value within a field, or in descending order, arranging records from highest to lowest value.

The sorting process usually is reserved for a relatively small number of data items. If thousands of customer records are stored, and they frequently need to be accessed in order based on different fields (alphabetical order by customer name one day, zip code order the next), the records would probably not be sorted at all, but would be indexed or linked. You learn about indexing and linking later in this chapter.

Here are some examples of occasions when you would need to sort records:

- A college stores student records in ascending order by student ID number, but the registrar wants to view the data in descending order by credit hours earned so he can contact students who are close to graduation.
- A department store maintains customer records in ascending order by customer number, but at the end of a billing period, the credit manager wants to contact customers whose balances are 90 or more days overdue. The manager wants to list these overdue customers in descending order by the amount owed, so the customers with the largest debt can be contacted first.
- A sales manager keeps records for her salespeople in alphabetical order by last name, but she needs to list the annual sales figure for each salesperson so she can determine the median annual sale amount.

The **median** (The value in the middle position of a list when the values are sorted.) value in a list is the value of the middle item when the values are listed in order. The median is not the same as the arithmetic average, or **mean** (The arithmetic average.) . The median is often used as a statistic because it represents a more typical case—half the values are below it and half are above it. Unlike the median, the mean is skewed by a few very high or low values.

- A store manager wants to create a control break report in which individual sales are listed in order in groups by their department. As you learned in Chapter 7, when you create a control break report, the records must have been sorted in order by the control break field.

When computers sort data, they always use numeric values to make comparisons between values. This is clear when you sort records by fields such as a numeric customer ID or balance due. However, even alphabetic sorts are numeric, because computer data is stored as a number using a series of 0s and 1s. Ordinary computer users seldom think about the numeric codes behind the letters, numbers, and punctuation marks they enter from their keyboards or see on a monitor. However, they see the consequence of the values behind letters when they see data sorted in alphabetical order. In every popular computer coding scheme, *B* is numerically one greater than *A*, and *y* is numerically one less than *z*. Unfortunately, your system dictates whether *A* is represented by a number that is greater or smaller than the number representing *a*. Therefore, to obtain the most useful and

accurate list of alphabetically sorted records, either a company's data-entry personnel should be consistent in the use of capitalization, or the programmer should convert all the data to use consistent capitalization. Because *A* is always less than *B*, alphabetic sorts are ascending sorts.

The most popular coding schemes include ASCII, Unicode, and EBCDIC. In each code, a number represents a specific computer character. Appendix A contains additional information about these codes.

As a professional programmer, you might never have to write a program that sorts data, because organizations can purchase prewritten, "canned" sorting programs. Additionally, many popular language compilers come with built-in methods that can sort data for you. However, it is beneficial to understand the sorting process so that you can write a special-purpose sort when needed. Understanding the sorting process also improves your array-manipulating skills.

Two Truths & A Lie

Understanding the Need for Sorting Data

1. When you sort data in ascending order, you arrange records from lowest to highest based on the value in a specific field.

   T F

2. Normal alphabetical order, in which *A* precedes *B*, is descending order.

   T F

3. When computers sort data, they use numeric values to make comparisons, even when string values are compared.

   T F

8-2 Using the Bubble Sort Algorithm

One of the simplest sorting techniques to understand is a bubble sort. You can use a bubble sort to arrange data items in either ascending or descending order. In a **bubble sort** (A sorting algorithm in which list elements are arranged in ascending or descending order by comparing items in pairs and swapping them when they are out of order.) , items in a list are compared with each other in pairs. When an item is out of order, it swaps values with the item below it. With an ascending bubble sort, after each adjacent pair of items in a list has been compared once, the largest item in the list will have "sunk" to the bottom. After many passes through the list, the smallest items rise to the top like bubbles in a carbonated drink. A bubble sort is sometimes called a **sinking sort** (A bubble sort) .

When you learn a method like sorting, programmers say you are learning an algorithm. An **algorithm** (The sequence of steps necessary to solve any problem.) is a list of instructions that accomplish a task. In this section, you will learn about the bubble sort algorithm for sorting a list of simple values; later in this chapter you will learn more about how multifield records are sorted. To understand the bubble sort algorithm, you first must learn about swapping values.

Chapter 8: Advanced Data Handling Concepts: 8-2a Understanding Swapping Values
Book Title: Programming Logic and Design
Printed By: Ronald Suchy (rsuchy@mayfieldschools.org)

8-2a Understanding Swapping Values

A central concept to many sorting algorithms, including the bubble sort, is the idea of swapping values. When you **swap values** (To exchange the values of two variables.) stored in two variables, you exchange their values; you set the first variable equal to the value of the second, and the second variable equal to the value of the first. However, there is a trick to swapping any two values. Assume that you have declared two variables as follows:

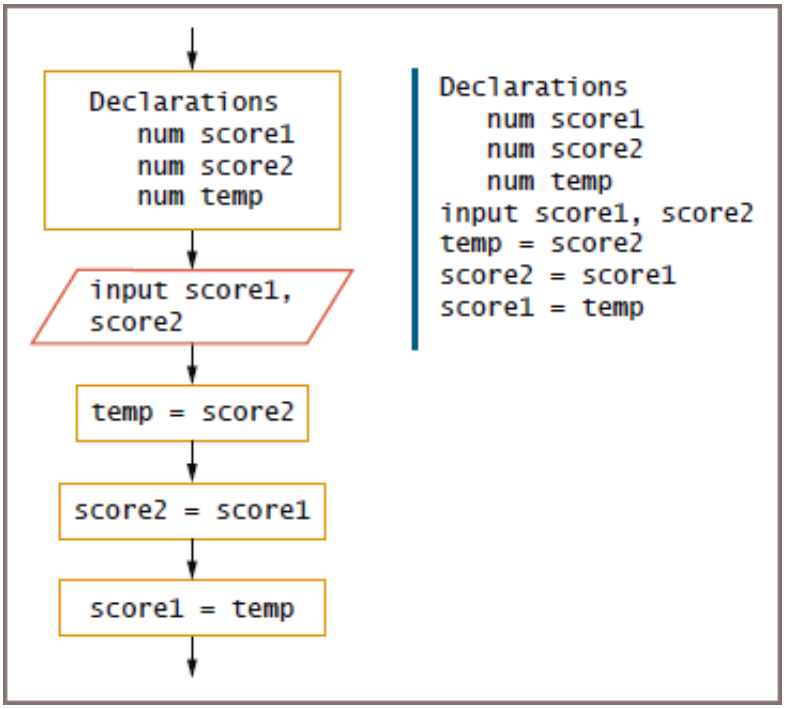
```
num score1 = 90
num score2 = 85
```

You want to swap the values so that `score1` is 85 and `score2` is 90. If you first assign `score1` to `score2` using a statement such as `score2 = score1`, both `score1` and `score2` hold 90 and the value 85 is lost. Similarly, if you first assign `score2` to score1 using a statement such as `score1 = score2`, both variables hold 85 and the value 90 is lost.

To correctly swap two values, you create a temporary variable to hold a copy of one of the scores so it doesn't get lost. Then, you can accomplish the swap as shown in Figure 8-1. First, the value in score2, 85, is assigned to a temporary holding variable named `temp`. Then, the `score1` value, 90, is assigned to `score2`. At this point, both `score1` and `score2` hold 90. Then, the 85 in `temp` is assigned to `score1`. Therefore, after the swap process, `score1` holds 85 and `score2` holds 90.

Figure 8-1

Program Segment That Swaps Two Values



In Figure 8-1, you can accomplish identical results by assigning `score1` to `temp`, assigning `score2` to `score1`, and finally assigning `temp` to `score2`.

Watch the video *Swapping Values*.

Chapter 8: Advanced Data Handling Concepts: 8-2b Understanding the Bubble Sort
Book Title: Programming Logic and Design
Printed By: Ronald Suchy (rsuchy@mayfieldschools.org)

8-2b Understanding the Bubble Sort

Assume that you want to sort five student test scores in ascending order. Figure 8-2 shows a program in which a constant is declared to hold an array's size, and then the array is declared to hold five scores. (The other variables and constants, which are shaded in the figure, will be discussed in the next paragraphs when they are used.) The program calls three main procedures—one to input the five scores, one to sort them, and the final one to display the sorted result.

Figure 8-2

Mainline Logic for Program That Accepts, Sorts, and Displays Scores

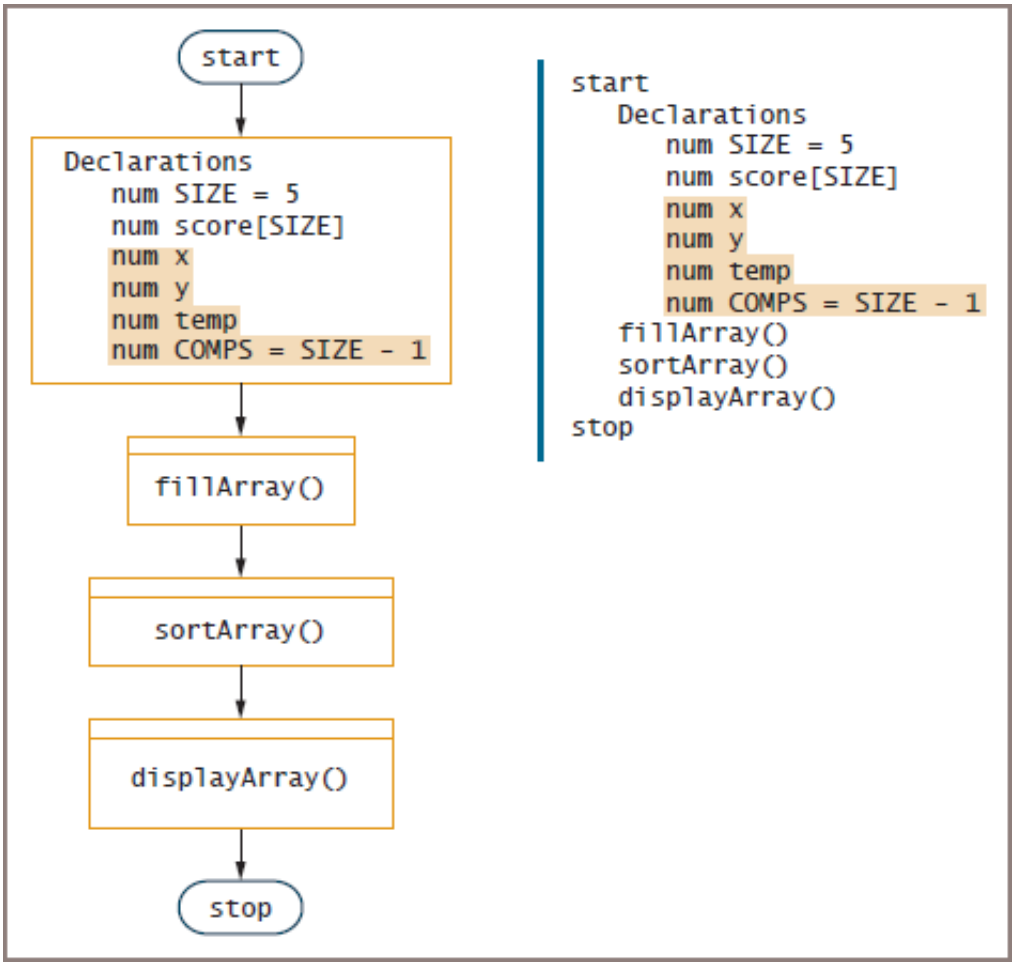
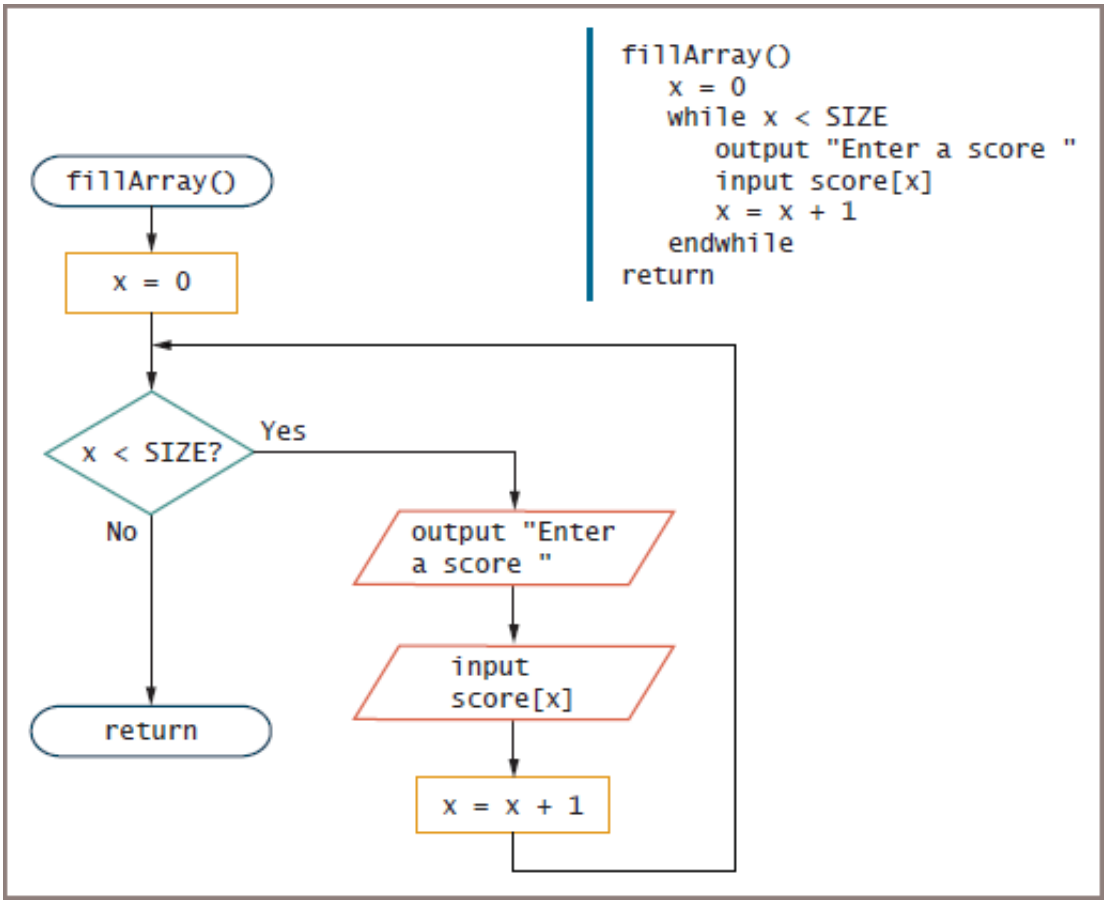


Figure 8-3 shows the `fillArray()` method. Within the method, a subscript, `x`, is initialized to 0 and each array element is filled in turn. After a user enters five scores, control returns to the main program.

Figure 8-3

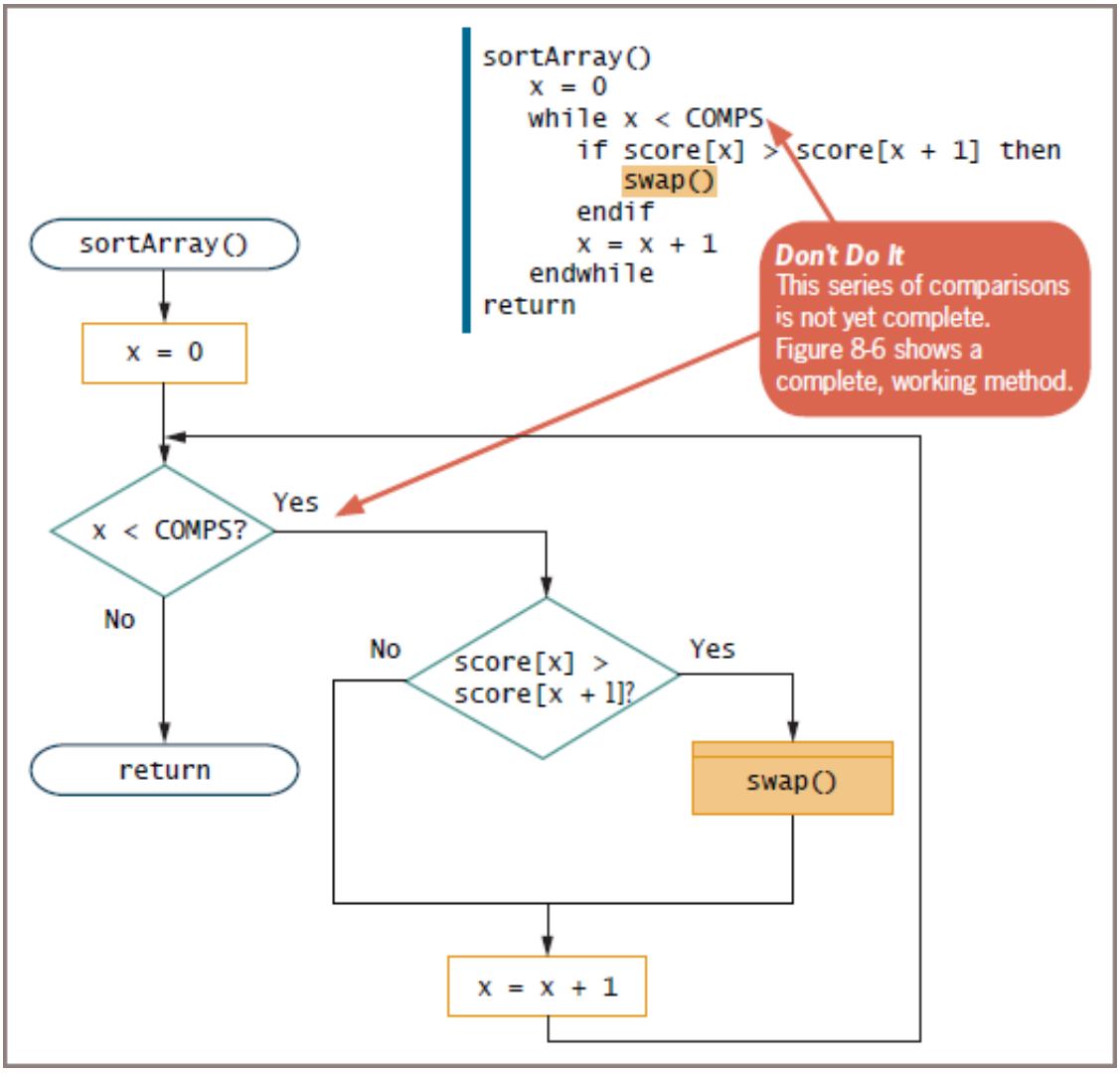
The `fillarray()` Method



The `sortArray()` method in Figure 8-4 sorts the array elements by making a series of comparisons of adjacent element values and swapping them if they are out of order. To begin sorting this list of scores, you compare the first two scores, `score[0]` and `score[1]`. If they are out of order—that is, if `score[0]` is larger than `score[1]`—you want to reverse their positions, or swap their values.

Figure 8-4

The Incomplete `sortarray()` Method



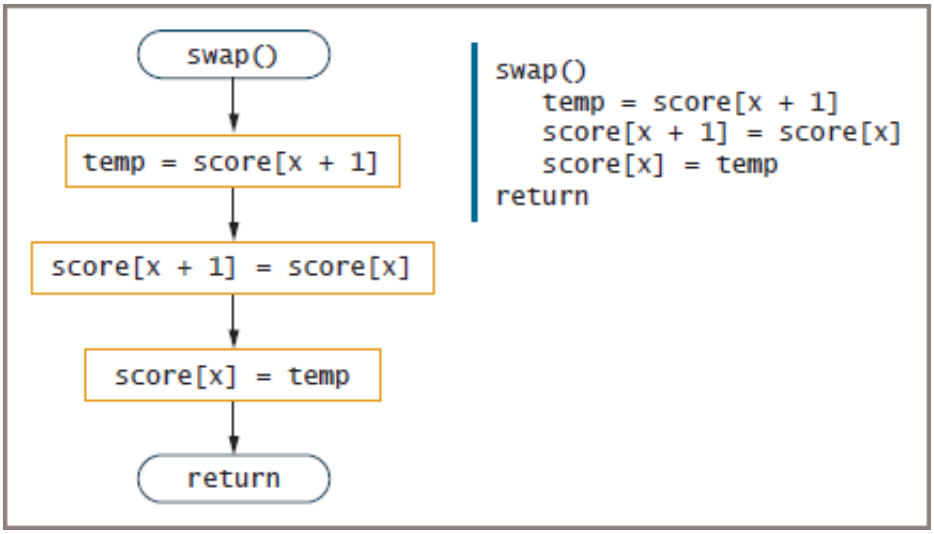
For example, assume that the five entered scores are:

```
score[0] = 90
score[1] = 85
score[2] = 65
score[3] = 95
score[4] = 75
```

In this list, `score[0]` is 90 and `score[1]` is 85; you want to exchange the values of the two elements so that the smaller value ends up earlier in the array. You call the `swap()` method, which places the scores in slightly better order than they were originally. Figure 8-5 shows the `swap()` method. This module switches any two adjacent elements in the `score` array.

Figure 8-5

The `swap()` Method



In Figure 8-4, the number of comparisons made is based on the value of the constant named `COMPS`, which was initialized to the value of `SIZE - 1`. That is, for an array of size 5, the `COMPS` constant will be 4. Therefore, the following comparisons are made:

```
score[0] > score[1]?
score[1] > score[2]?
score[2] > score[3]?
score[3] > score[4]?
```

Each element in the array is compared to the element that follows it. When `x` becomes `COMPS`, the `while` loop ends. If the loop continued when `x` became equal to `COMPS`, then the next comparison would be `score[4] > score[5]?`. This would cause an error because the highest allowed subscript in a five-element array is 4. You must execute the decision `score[x] > score[x + 1]?` four times—when `x` is 0, 1, 2, and 3.

For an ascending sort, you need to perform the `swap()` method whenever any given element of the `score` array has a value greater than the next element. For any `x`, if the xth element is not greater than the element at position `x + 1`, the swap should not take place. For example, when `score[x]` is 90 and `score[x + 1]` is 85, a swap should occur. On the other hand, when `score[x]` is 65 and `score[x + 1]` is 95, then no swap should occur.

For a descending sort in which you want to end up with the highest value first, you would write the decision so that you perform the switch when `score[x]` is *less than* `score[x + 1]`.

As a complete example of how this application works using an ascending sort, suppose that you have these original scores:

```
score[0] = 90
score[1] = 85
score[2] = 65
score[3] = 95
score[4] = 75
```

The logic of the `sortArray()` method proceeds like this:

1. Set `x` to 0.

2. The value of `x` is less than 4 (`COMPS`), so enter the loop.

3. Compare `score[x]`, 90, to `score[x + 1]`, 85. The two scores are out of order, so they are switched.

   The list is now:

   ```
   score[0] = 85
   score[1] = 90
   score[2] = 65
   score[3] = 95
   score[4] = 75
   ```

4. After the swap, add 1 to `x`, so `x` is 1.

5. Return to the top of the loop. The value of `x` is less than 4, so enter the loop a second time.

6. Compare `score[x]`, 90, to `score[x + 1]`, 65. These two values are out of order, so swap them.

   Now the result is:

   ```
   score[0] = 85
   score[1] = 65
   score[2] = 90
   score[3] = 95
   score[4] = 75
   ```

7. Add 1 to `x`, so `x` is now 2.

8. Return to the top of the loop. The value of `x` is less than 4, so enter the loop.

9. Compare `score[x]`, 90, to `score[x + 1]`, 95. These values are in order, so no switch is made.

10. Add 1 to `x`, making it 3.

11. Return to the top of the loop. The value of `x` is less than 4, so enter the loop.

12. Compare `score[x]`, 95, to `score[x + 1]`, 75. These two values are out of order, so switch them.

    Now the list is as follows:

13. Add 1 to `x`, making it 4.

14. Return to the top of the loop. The value of `x` is 4, so do not enter the loop again.

When `x` reaches 4, every element in the list has been compared with the one adjacent to it. The highest score, 95, has "sunk" to the bottom of the list. However, the scores still are not in order. They are in slightly better ascending order than they were when the process began, because the largest value is at the bottom of the list, but they are still out of order. You need to repeat the entire procedure so that 85 and 65 (the current `score[0]` and `score[1]` values) can switch places, and 90 and 75 (the current `score[2]` and `score[3]` values) can switch places. Then, the scores will be 65, 85, 75, 90, and 95. You will have to go through the list yet again to swap 85 and 75.

As a matter of fact, if the scores had started in the worst possible order (95, 90, 85, 75, 65), the comparison process would have to take place four times. In other words, you would have to pass through the list of values four times, making appropriate swaps, before the numbers would appear in perfect ascending order. You need to place the loop in Figure 8-4 within another loop that executes four times.

Figure 8-6 shows the complete logic for the `sortArray()` module. The module uses a loop control variable named `y` to cycle through the list of scores four times. (The initialization, comparison, and alteration of this loop control variable are shaded in the figure.) With an array of five elements, it takes four comparisons to work through the array once, comparing each pair, and it takes four sets of those comparisons to ensure that every element in the entire array is in sorted order. In the `sortArray()` method in Figure 8-6, `x` must be reset to 0 for each new value of `y` so that the comparisons always start at the top of the list.

Figure 8-6

The Completed `sortarray()` Method

When you sort the elements in an array this way, you use nested loops—an inner loop that swaps out-of-order pairs, and an outer loop that goes through the list multiple times. The general rules for making comparisons with the bubble sort are:

- The greatest number of pair comparisons you need to make during each loop is *one less* than the number of elements in the array. You use an inner loop to make the pair comparisons.
- The number of times you need to process the list of values is `one less` than the

number of elements in the array. You use an outer loop to control the number of times you walk through the list.

As an example, if you want to sort a 10-element array, you make nine pair comparisons on each of nine rotations through the loop, executing a total of 81 score comparison statements.

The last method called by the score-sorting program in Figure 8-2 is the one that displays the sorted array contents. Figure 8-7 shows this method.

Figure 8-7

The `displayarray()` Method


Watch the video *The Bubble Sort*.

Chapter 8: Advanced Data Handling Concepts: 8-2c Sorting a List of Variable Size
Book Title: Programming Logic and Design
Printed By: Ronald Suchy (rsuchy@mayfieldschools.org)

8-2c Sorting a List of Variable Size

In the score-sorting program in the previous section, a `SIZE` constant was initialized to the number of elements to be sorted at the start of the program. At times, however, you don't want to create such a value because you might not know how many array elements will

hold valid values. For example, on one program run you might want to sort only three or four scores, and on another run you might want to sort 20. In other words, what if the size of the list to be sorted might vary? Rather than sorting a fixed number of array elements, you can count the input scores and then sort just that many.

To keep track of the number of elements stored in an array, you can create the application shown in Figure 8-8. As in the original version of the program, you call the `fillArray()` method, and when you input each score, you increase `x` by 1 to place each new score into a successive element of the `score` array. After you input one `score` value and place it in the first element of the array, `x` is 1. After a second score is input and placed in `score[1]`, `x` is 2, and so on. After you reach the end of input, `x` holds the number of scores that have been placed in the array, so you can store `x` in `numberOfEls`, and compute `comparisons` as `numberOfEls - 1`. With this approach, it doesn't matter if there are not enough `score` values to fill the array. The `sortArray()` and `displayArray()` methods use `comparisons` and `numberOfEls` instead of `COMPS` and `SIZE` to process the array. For example, if 35 scores are input, `numberOfEls` will be set to 35 in the `fillArray()` module, and when the program sorts, it will use 34 as a cutoff point for the number of pair comparisons to make. The sorting program will never make pair comparisons on array elements 36 through 100—those elements will just "sit there," never being involved in a comparison or swap.

Figure 8-8

Score-Sorting Application in Which Number of Elements to Sort Can Vary

In the `fillArray()` method in Figure 8-8, notice that a priming read has

been added to the method. If the user enters the `QUIT` value at the first input, then the number of elements to be sorted will be 0.

When you count the input values and use the `numberOfEls` variable, it does not matter if there are not enough scores to fill the array. However, an error occurs if you attempt to store more values than the array can hold. When you don't know how many elements will be stored in an array, you must overestimate the number of elements you declare.

8-2d Refining the Bubble Sort to Reduce Unnecessary Comparisons

You can make additional improvements to the bubble sort created in the previous sections. As illustrated in Figure 8-8, when you perform the sorting module for a bubble sort, you pass through a list, making comparisons and swapping values if two adjacent values are out of order. If you are performing an ascending sort and you have made one pass through the list, the largest value is guaranteed to be in its correct final position at the bottom of the list. Similarly, the second-largest element is guaranteed to be in its correct second-to-last position after the second pass through the list, and so on. If you continue to compare every element pair on every pass through the list, you are comparing elements that are already guaranteed to be in their final correct position. In other words, after the first pass through the list, you no longer need to check the bottom element; after the second pass, you don't need to check the two bottom elements.

On each pass through the array, you can afford to stop your pair comparisons one element sooner. You can avoid comparing the values that are already in place by creating a new variable, `pairsToCompare`, and setting it equal to the value of `numberOfEls - 1`. On the first pass through the list, every pair of elements is compared, so `pairsToCompare` *should equal* `numberOfEls - 1`. In other words, with five array elements to sort, four pairs are compared, and with 50 elements to sort, 49 pairs are compared. On each subsequent pass through the list, `pairsToCompare` should be reduced by 1; for example, after the first pass is completed, it is not necessary to check the bottom element. See Figure 8-9 to examine the use of the `pairsToCompare` variable.

Figure 8-9

Flowchart and Pseudocode for `sortarray()` Method Using `pairstocompare` Variable

8-2e Refining the Bubble Sort to Eliminate Unnecessary Passes

You could also improve the bubble sort module in Figure 8-9 by reducing the number of passes through the array. If array elements are badly out of order or in reverse order, many passes through the list are required to place it in order; it takes one fewer pass than the value in `numberOfEls` to complete all the comparisons and swaps needed to sort the list. However, when the array elements are in order or nearly in order to start, all the elements might be correctly arranged after only a few passes through the list. All subsequent passes result in no swaps. For example, assume that the original scores are as follows:

The bubble sort module in Figure 8-9 would pass through the array list four times, making four sets of pair comparisons. It would always find that each `score[x]` is *not* greater than the corresponding `score[x + 1]`, so no switches would ever be made. The scores would end up in the proper order, but they *were* in the proper order in the first place; therefore, a lot of time would be wasted.

A possible remedy is to add a flag variable set to a "continue" value on any pass through the list in which any pair of elements is swapped (even if just one pair), and which holds a different "finished" value when no swaps are made—that is, when all elements in the list are already in the correct order. For example, you can create a variable named `didSwap` and set it to `"No"` at the start of each pass through the list. You can change its value to `"Yes"` each time the `swap()` module is performed (that is, each time a switch is necessary).

If you make it through the entire list of pairs without making a switch, the `didSwap` flag will *not* have been set to `"Yes"`, meaning that no swap has occurred and that the array elements must already be in the correct order. This situation might occur on the first or second pass through the array list, or it might not occur until a much later pass. Once the array elements are in the correct order, you can stop making passes through the list.

Figure 8-10 illustrates a module that sorts scores and uses a `didSwap` flag. At the beginning of the `sortArray()` module, initialize `didSwap` to `"Yes"` before entering the comparison loop the first time. Then, immediately set `didSwap` to `"No"`. When a switch occurs—that is, when the `swap()` module executes—set `didSwap` to `"Yes"`.

Figure 8-10

Flowchart and Pseudocode for `sortarray()` Method Using `didswap` Variable

With the addition of the flag variable in Figure 8-10, you no longer need the variable `y`, which was keeping track of the number of passes through the list. Instead, you keep going through the list until you can make a complete pass without any switches.

Two Truths & A Lie

Using the Bubble Sort Algorithm

1. You can use a bubble sort to arrange records in ascending or descending order.

   T F

2. In a bubble sort, items in a list are compared with each other in pairs, and when an item is out of order, it swaps values with the item below it.

   T F

3. With any bubble sort, after each adjacent pair of items in a list has been compared once, the largest item in the list will have "sunk" to the bottom.

   T F

8-3 Sorting Multifield Records

The bubble sort algorithm is useful for sorting a list of values, such as a list of test scores in ascending order or a list of names in alphabetical order. Records, however, are most frequently composed of multiple fields. When you want to sort records, you need to make sure data that belongs together stays together. When you sort records, two approaches you can take are to place related data items in parallel arrays and to sort records as a whole.

8-3a Sorting Data Stored in Parallel Arrays

Suppose that you have parallel arrays containing student names and test scores, like the arrays shown in Figure 8-11. Each student's name appears in the same relative position in the `name` array as his or her test `score` appears in the score array. Further suppose that you want to sort the student names and their scores in alphabetical order. If you use a sort algorithm on the `name` array to place the names in alphabetical order, the name that starts in position 3, *Anna*, should end up in position 0. If you also neglect to rearrange the `score` array, Anna's name will no longer be in the same relative position as her score, which is 85. Notice that you don't want to sort the `score` values. If you did, `score[2]`, 60, would move to position 0, and that is not Anna's score. Instead, when you sort the names, you want to make sure that each corresponding score is moved to the same position as the name to which it belongs.

Figure 8-11

Appearance of `name` and `score` Arrays in Memory

Figure 8-12 shows the `swap()` module for a program that sorts `name` array values in alphabetical order and moves `score` array values correspondingly. This version of the `swap()` module uses two temporary variables—a `string` named `tempName` and a numeric variable named `tempScore`. The `swap()` method executes whenever two names in positions `x` and `x + 1` are out of order. Besides swapping the names in positions `x` and `x + 1`, the module also swaps the scores in the same positions. Therefore, each student's score always moves along with its student's name.

Figure 8-12

The `swap()` Method for a Program That Sorts Student Names and Retains their Correct Scores

8-3b Sorting Records as a Whole

In most modern programming languages, you can create group items that can be manipulated more easily than single data items. (You first learned about such group names in Chapter 7.) Creating a group name for a set of related data fields is beneficial when you want to move related data items together, as when you sort records with multiple fields. These group items are sometimes called *structures*, but more frequently are created as *classes*. Chapters 10 and 11 provide much more detail on creating classes, but for now, understand that you can create a group item with syntax similar to the following:

To sort student records using the group name, you could do the following:

- Define a class named `StudentRecord`, as shown in the preceding code.
- Define what *greater than* means for a `StudentRecord`. For example, to sort records by student name, you would define `greater than` to compare `name` values, not `score` values. The process for creating this definition varies among programming

languages.

- Use a sort algorithm that swaps `StudentRecord` items, including both `name`s and `score`s, whenever two `StudentRecord`s are out of order.

Two Truths & A Lie

Sorting Multifield Records

1. To sort related parallel arrays, you must sort each in the same order—either ascending or descending.

   T F

2. When you sort related parallel arrays and swap values in one array, you must make sure that all associated arrays make the same relative swap.

   T F

3. Most modern programming languages allow you to create a group name for associated fields in a record.

   T F

8-4 Using the Insertion Sort Algorithm

The bubble sort works well and is relatively easy to understand and manipulate, but many other sorting algorithms have been developed. For example, when you use an **insertion sort** (A sorting algorithm in which each list element is examined one at a time; if an element is out of order relative to any of the items earlier in the list, each earlier item is moved down one position and then the tested element is inserted.) , you look at each list element one at a time. If an element is out of order relative to any of the items earlier in the list, you move each earlier item down one position and then insert the tested element. The insertion sort is similar to the technique you would most likely use to sort a group of objects manually. For example, if a list contains the values 2, 3, 1, and 4, and you want to place them in ascending order using an insertion sort, you test the values 2 and 3, but you do not move them because they are in order. However, when you test the third value in the list, 1, you move both 2 and 3 to later positions and insert 1 at the first position.

Figure 8-13 shows the logic that performs an ascending insertion sort using a five-element array named `score`. Assume that a constant named `SIZE` has been set to 5, and that the five scores in the array are as follows:

Figure 8-13

Flowchart and Pseudocode for the `insertionsort()` Method

The shaded outer loop varies a loop control variable `x` from 1 through one less than the size of the array. The logic proceeds as follows:

First `x` is set to 1, and then the unshaded section in the center of Figure 8-13 executes.

1. The value of `temp` is set to `score[1]`, which is 85, and `y` is set to 0.
2. Because `y` is greater than or equal to 0 and `score[y]` (90) is greater than `temp`, the inner loop is entered. (If you were performing a descending sort, then you would ask whether `score[y]` was less than `temp`.)
3. The value of `score[1]` becomes 90 and `y` is decremented, making it –1, so `y` is no longer greater than or equal to 0, and the inner loop ends.
4. Then `score[0]` is set to `temp`, which is 85.

After these steps, 90 was moved down one position and 85 was inserted in the first position, so the array values are in slightly better order than they were originally. The values are as follows:

Now, in the outer loop, `x` becomes 2. The logic in the unshaded portion of Figure 8-13 proceeds as follows:

1. The value of `temp` becomes 65, and `y` is set to 1.
2. The value of `y` is greater than or equal to 0, and `score[y]` (90) is greater than temp, so the inner loop is entered.
3. The value of `score[2]` becomes 90 and `y` is decremented, making it 0, so the loop executes again.
4. The value of `score[1]` becomes 85 and `y` is decremented, making it –1, so the loop ends.
5. Then `score[0]` becomes 65.

After these steps, the array values are in better order than they were originally, because 65 and 85 now both come before 90:

Now, `x` becomes 3. The logic in Figure 8-13 proceeds to work on the new list as follows:

1. The value of `temp` becomes 95, and y is set to 2.

2. For the loop to execute, `y` must be greater than or equal to 0, which it is, and `score[y]` (90) must be greater than `temp`, which it is *not*. So, the inner loop does not execute.

3. Therefore, `score[2]` is set to 90, which it already was. In other words, no changes are made.

Now, `x` is increased to 4. The logic in Figure 8-13 proceeds as follows:

1. The value of `temp` becomes 75, and `y` is set to 3.

2. The value of `y` is greater than or equal to 0, and `score[y]` (95) is greater than `temp`, so the inner loop is entered.

3. The value of `score[4]` becomes 95 and `y` is decremented, making it 2, so the loop executes again.

4. The value of `score[3]` becomes 90 and `y` is decremented, making it 1, so the loop executes again.

5. The value of `score[2]` becomes 85 and `y` is decremented, making it 0; `score[y]` (65) is no longer greater than `temp` (75), so the inner loop ends. In other words, the scores 85, 90, and 95 are each moved down one position, but score 65 is left in place.

6. Then `score[1]` becomes 75.

After these steps, all the array values have been rearranged in ascending order as follows:

Watch the video *The Insertion Sort*.

Many sorting algorithms exist in addition to the bubble sort and insertion sort. You might want to investigate the logic used by the *selection sort, cocktail sort, gnome sort,* and *quick sort*.

Two Truths & A Lie

Using the Insertion Sort Algorithm

1. When you use an insertion sort, you look at each list element one at a time and move items down if the tested element should be inserted before them.

   T F

2. You can create an ascending list using an insertion sort, but not a descending one.

   T F

3. The insertion sort is similar to the technique you would most likely use to sort a group of objects manually.

   T F

Chapter 8: Advanced Data Handling Concepts: 8-5 Using Multidimensional Arrays
Book Title: Programming Logic and Design
Printed By: Ronald Suchy (rsuchy@mayfieldschools.org)

8-5 Using Multidimensional Arrays

In Chapter 6, you learned that an array is a series or list of values in computer memory, all of which have the same name and data type but are differentiated with special numbers called subscripts. Usually, all the values in an array have something in common; for example, they might represent a list of employee ID numbers or a list of prices for items sold in a store. A subscript, also called an index, is a number that indicates the position of a particular item within an array.

An array whose elements you can access using a single subscript is a **one-dimensional array** (A list accessed using a single subscript.) or **single-dimensional array** (A list accessed using a single subscript.) . The array has only one dimension because its data can be stored in a table that has just one dimension—height. If you know the vertical position of a one-dimensional array's element, you can find its value.

For example, suppose that you own an apartment building and charge five different rent amounts for apartments on different floors (including floor 0, the basement), as shown in Table 8-1.

Table 8-1

Rent Schedule based on Floor

Floor	Rent ($)
0	350
1	400
2	475
3	600
4	1000

You could declare the following array to hold the rent values:

The location of any rent value in Table 8-1 depends on only a single variable—the floor of the building. So, when you create a single-dimensional array to hold rent values, you need just one subscript to identify the row.

Sometimes, however, locating a value in an array depends on more than one variable. If you must represent values in a table or grid that contains rows and columns instead of a single list, then you might want to use a **two-dimensional arrays** (Arrays that have rows and columns of values accessed using two subscripts.) . A two-dimensional array contains two dimensions: height and width. That is, the location of any element depends on two factors. For example, if an apartment's rent depends on two variables—both the floor of the building and the number of bedrooms—then you want to create a two-dimensional array.

As an example of how useful two-dimensional arrays can be, assume that you own an apartment building with five floors, and that each of the floors has studio apartments (with no bedroom) and one- and two-bedroom apartments. Table 8-2 shows the rental amounts.

Table 8-2

Rent Schedule based on Floor and Number of Bedrooms

Floor	Studio Apartment	1-bedroom Apartment	2-bedroom Apartment

0	350	390	435
1	400	440	480
2	475	530	575
3	600	650	700
4	1000	1075	1150

To determine a tenant's rent, you need to know two pieces of information: the floor where the tenant lives and the number of bedrooms in the apartment. Each element in a two-dimensional array requires two subscripts to reference it—one subscript to determine the row and a second to determine the column. Thus, the 15 rent values for a two-dimensional array based on Table 8-2 would be arranged in five rows and three columns and defined as follows:

Figure 8-14 shows how the one- and two-dimensional rent arrays might appear in computer memory.

Figure 8-14

One- and Two-Dimensional Arrays in Memory

When you declare a one-dimensional array, you use a set of square brackets after the array type and name. To declare a two-dimensional array, many languages require you to use two sets of brackets after the array type and name. For each element in the array, the first square bracket holds the number of rows and the second one holds the number of columns.

In other words, the two dimensions represent the array's height and its width.

Instead of two sets of brackets to indicate a position in a two-dimensional array, some languages use a single set of brackets but separate the subscripts with commas. Therefore, the elements in row 1, column 2 would be `RENT_BY_FLOOR_AND_BDRMS[1, 2]`.

In the `RENT_BY_FLOOR_AND_BDRMS` array declaration, the values that are assigned to each row are enclosed in braces to help you picture the placement of each number in the array. The first row of the array holds the three rent values 350, 390, and 435 for floor 0; the second row holds 400, 440, and 480 for floor 1; and so on.

You access a two-dimensional array value using two subscripts, in which the first subscript represents the row and the second one represents the column. For example, some of the values in the array are as follows:

- `RENT_BY_FLOOR_AND_BDRMS[0][0]` is 350
- `RENT_BY_FLOOR_AND_BDRMS[0][1]` is 390
- `RENT_BY_FLOOR_AND_BDRMS[0][2]` is 435
- `RENT_BY_FLOOR_AND_BDRMS[4][0]` is 1000
- `RENT_BY_FLOOR_AND_BDRMS[4][1]` is 1075
- `RENT_BY_FLOOR_AND_BDRMS[4][2]` is 1150

If you declare two variables to hold the floor number and bedroom count as `num floor` and `num bedrooms`, any tenant's rent is `RENT_BY_FLOOR_AND_BDRMS[floor][bedrooms]`.

When mathematicians use a two-dimensional array, they often call it a **matrix** (A term sometimes used by mathematicians to describe a two-dimensional array.) or a **table** (A database file that contains data in rows and columns; also, a term sometimes used by mathematicians to describe a two-dimensional array.) . You may have used a spreadsheet, which is a two-dimensional array in which you need to know a row number and a column letter to access a specific cell.

Figure 8-15 shows a program that continuously displays rents for apartments based on

renter requests for floor location and number of bedrooms. Notice that although significant setup is required to provide all the values for the rents, the basic program is extremely brief and easy to follow. (You could improve the program in Figure 8-15 by making sure the values for `floor` and `bedrooms` are within range before using them as array subscripts.)

Figure 8-15

A Program That Determines Rents

 Watch the video *Two-Dimensional Arrays*.

Two-dimensional arrays are never actually *required* in order to achieve a useful program. The same 15 categories of rent information could be stored in three separate single-dimensional arrays of five elements each, and you could use a decision to determine which array to access. Of course, don't forget that even one-dimensional arrays are never required to solve a problem. You could also declare 15 separate rent variables and make 15 separate decisions to determine the rent.

Besides one- and two-dimensional arrays, many programming languages also support **three-dimensional arrays** (Arrays in which each element is accessed using three subscripts.) . For example, if you own a multistory apartment building with different numbers of bedrooms available in apartments on each floor, you can use a two-dimensional array to store the rental fees, but if you own several apartment buildings, you might want to employ a third dimension to store the building number. For example, if a three-dimensional array is stored on paper, you might need to know an element's row, column, and page to access it, as shown in Figure 8-16.

Figure 8-16

Picturing a Three-Dimensional Array

If you declare a three-dimensional array named `RENT_BY_3_FACTORS`, then you can use an expression such as `RENT_BY_3_FACTORS[floor][bedrooms][building]`, which refers to a specific rent figure for an apartment whose floor and bedroom numbers are stored in the `floor` and `bedrooms` variables, and whose building number is stored in the `building` variable. Specifically, `RENT_BY_3_FACTORS[0][1][2]` refers to a one-bedroom apartment on floor 0 of building 2.

Both two- and three-dimensional arrays are examples of **multidimensional arrays** (Lists with more than one dimension.) , which are arrays that have more than one dimension. Some languages allow many dimensions. For example, in C# and Visual Basic, an array can have 32 dimensions. However, it's usually hard for people to keep track of more than three dimensions.

Two Truths & A Lie

Using Multidimensional Arrays

1. In every multidimensional array, the location of any element depends on two factors.

T F

2. For each element in a two-dimensional array, the first subscript represents the row number and the second one represents the column.

   T F

3. Multidimensional arrays are never actually *required* in order to achieve a useful program.

   T F

8-6 Using Indexed Files and Linked Lists

Sorting a list of five or even 100 scores does not require significant computer resources. However, many data files contain thousands of records, and each record might contain dozens of data fields. Sorting large numbers of data records requires considerable time and computer memory. When a large data file needs to be processed in ascending or descending order based on a particular field, the most efficient approach is usually to store and access records based on their logical order rather than sorting and accessing them in their physical order. **Physical order** (The order in which a list is actually stored even though it might be accessed in a different logical order.) refers to a "real" order for storage; an example would be writing the names of 10 friends, each one on a separate index card. You can arrange the cards alphabetically by the friends' last names, chronologically by age of the friendship, or randomly by throwing the cards in the air and picking them up as you find them. Whichever way you do it, the records still follow each other in *some* order. In addition to their current physical order, you can think of the cards as having a **logical order** (The order in which a list is used, even though it is not necessarily stored in that physical order.) ; that is, a virtual order, based on any criterion you choose—from the tallest friend to the shortest, from the one who lives farthest away to the closest, and so on. Sorting the cards in a new physical order can take a lot of time; using the cards in their logical order without physically rearranging them is often more efficient.

8-6a Using Indexed Files

A common method of accessing records in logical order requires using an index. Using an index involves identifying a key field for each record. A record's **key field** (The field whose contents make a record unique among all records in a file.) is the field whose contents make the record unique among all records in a file. For example, multiple employees can have the same last name, first name, salary, or street address, but each employee possesses a unique employee identification number, so an ID number field might make a good key field for a personnel file. Similarly, a product number makes a good key field in an inventory file.

As pages in a book have numbers, computer memory and storage locations have **addresses** (Numbers that identify computer memory and storage locations.) . In Chapter 1, you learned that every variable has a numeric address in computer memory; likewise, every data record on a disk has a numeric address where it is stored. You can store records in any physical order on the disk, but when you **index** (A list of key fields paired with the storage address for the corresponding data record.) records, you store a list of key fields paired with the storage address for the corresponding data record. Then you can use the index to find the records in order based on their addresses.

When you use an index, you can store records on a **random-access storage device** (A storage device, such as a disk, from which records can be accessed in any order.) , such as a disk, from which records can be accessed in any order. Each record can be placed in any physical location on the disk, and you can use the index as you would use an index in the back of a book. If you pick up a 600-page American history book because you need some facts about Betsy Ross, you do not want to start on page one and work your way through the book. Instead, you turn to the index, discover that Betsy Ross is mentioned on , and go directly to that page. As a programmer, you do not need to determine a record's exact physical address in order to use it. A computer's operating system takes care of locating available storage for your records.

Chapter 7 contains a discussion of random access files.

You can picture an index based on ID numbers by looking at the index in Figure 8-17. The index is stored on a portion of the disk. The address in the index refers to other scattered locations on the disk.

Figure 8-17

An Index on a Disk That Associates Id Numbers with Disk Addresses

When you want to access the data for employee 333, you tell your computer to look through the ID numbers in the index, find a match, and then proceed to the memory location specified. Similarly, when you want to process records in order based on ID number, you tell your system to retrieve records at the locations in the index in sequence. Thus, employee 111 may have been hired last and the record may be stored at the highest physical address on the disk, but if the employee record has the lowest ID number, it will be accessed first during any ordered processing.

When a record is removed from an indexed file, it does not have to be physically removed. Its reference can simply be deleted from the index, and then it will not be part of any further processing.


Watch the video *Using an Indexed File*.

8-6b Using Linked Lists

Another way to access records in a desired order, even though they might not be physically stored in that order, is to create a linked list. In its simplest form, creating a **linked list** (A list that contains an extra field in every record that holds the physical address of the next logical record.) involves creating one extra field in every record of stored data. This extra

field holds the physical address of the next logical record. For example, a record that holds a customer's ID, name, and phone number might contain the following fields:

Every time you use a record, you access the next record based on the address held in the `nextCustAddress` field.

Every time you add a new record to a linked list, you search through the list for the correct logical location of the new record. For example, assume that customer records are stored at the addresses shown in Table 8-3 and that they are linked in customer ID order. Notice that the addresses of the records are not shown in sequential order. The records are shown in their logical order by `idNum`.

Table 8-3

Sample Linked Customer List

Address	idNum	name	phoneNum	nextCustAddress of Record
0000	111	Baker	234-5676	7200
7200	222	Vincent	456-2345	4400
4400	333	Silvers	543-0912	6000
6000	444	Donovan	328-8744	`eof`

You can see from Table 8-3 that each customer record contains a `nextCustAddress` field that stores the address of the next customer who follows in customer ID number order (and not necessarily in address order). For any individual customer, the next logical customer's address might be physically distant.

Examine the file shown in Table 8-3, and suppose that a new customer is acquired with number 245 and the name *Newberg*. Also suppose that the computer operating system finds an available storage location for Newberg's data at address 8400. In this case, the procedure to add Newberg to the list is:

1. Create a variable named `currentAddress` to hold the address of the record in the list you are examining. Store the address of the first record in the list, 0000, in this variable.

2. Compare the new customer Newberg's ID, 245, with the current (first) record's ID, 111 (in other words, the ID at address 0000). The value 245 is higher than 111, so you save the first customer's address—0000, the address you are currently examining—in a variable you can name `saveAddress`. The `saveAddress` variable always holds the address you just finished examining. The first customer record contains a link to the address of the next logical customer—7200. Store 7200 in the `currentAddress` variable.

3. Examine the second customer record, the one that physically exists at the address 7200, which is currently held in the `currentAddress` variable.

4. Compare Newberg's ID, 245, with the ID stored in the record at `currentAddress`, 222. The value 245 is higher, so save the current address, 7200, in `saveAddress` and store its `nextCustAddress` address field, 4400, in the `currentAddress` variable.

5. Compare Newberg's ID, 245, with 333, which is the ID at `currentAddress` (4400). Up to this point, 245 had been higher than each ID tested, but this time the value 245 is lower, so customer 245 should logically precede customer 333. Set the `nextCustAddress` field in Newberg's record (customer 245) to 4400, which is the address of customer 333 and the address stored in `currentAddress`. This means that in any future processing, Newberg's record will logically be followed by the record containing 333. Also set the `nextCustAddress` field of the record located at `saveAddress` (7200, customer 222, Vincent, who logically preceded Newberg) to the new customer Newberg's address, 8400. The updated list appears in Table 8-4.

Table 8-4

Updated Customer List

Address	idNum	name	phoneNum	nextCustAddress of Record
0000	111	Baker	234-5676	7200
7200	222	Vincent	456-2345	8400
8400	245	Newberg	222-9876	4400
4400	333	Silvers	543-0912	6000
6000	444	Donovan	328-8744	`eof`

As with indexing, when removing records from a linked list, the records do not need to be physically deleted from the medium on which they are stored. If you need to remove customer 333 from the preceding list, all you need to do is change Newberg's `nextCustAddress` field to the value in Silvers' `nextCustAddress` field, which is Donovan's address: 6000. In other words, the value of 6000 is obtained not by knowing to which record Newberg should point, but by knowing to which record Silvers previously pointed. When Newberg's record points to Donovan, Silvers' record is then bypassed during any further processing that uses the links to travel from one record to the next.

More sophisticated linked lists store *two* additional fields with each record. One field stores the address of the next record, and the other field stores the address of the *previous* record so that the list can be accessed either forward or backward.

 Watch the video *Using a Linked List*.

Two Truths & A Lie

Using Indexed Files and Linked Lists

1. When a large data file needs to be processed in order based on a particular field, the most efficient approach is usually to sort the records.

   T F

2. A record's key field contains a value that makes the record unique among all records in a file.

   T F

3. Creating a linked list requires you to create one extra field for every record; this extra field holds the physical address of the next logical record.

   T F

8-7 Chapter Review

8-7a Chapter Summary

- Frequently, data items need to be sorted. When you sort data, you can sort either in ascending order, arranging records from lowest to highest value, or in descending order, arranging records from highest to lowest value.
- In a bubble sort, items in a list are compared with each other in pairs. When an item is out of order, it swaps values with the item below it. With an ascending bubble sort, after each adjacent pair of items in a list has been compared once, the largest item in the list will have "sunk" to the bottom; after many passes through the list, the smallest items rise to the top. The bubble sort algorithm can be improved to sort varying numbers of values and to eliminate unnecessary comparisons.
- When you sort records, two possible approaches are to place related data items in parallel arrays and to sort records as a whole.
- When you use an insertion sort, you look at each list element one at a time. If an element is out of order relative to any of the items earlier in the list, you move each earlier item down one position and then insert the tested element.
- Two-dimensional arrays have both rows and columns of values. You must use two subscripts when you access an element in a two-dimensional array. Many languages support arrays with even more dimensions.
- You can use an index or linked list to access data records in a logical order that differs from their physical order. Using an index involves identifying a physical address and key field for each record. Creating a linked list involves creating an extra field within every record to hold the physical address of the next logical record.

Chapter Review

8-7b Key Terms

sequential order (The arrangement of records when they are stored one after another on the basis of the value in a particular field.)

median (The value in the middle position of a list when the values are sorted.)

mean (The arithmetic average.)

bubble sort (A sorting algorithm in which list elements are arranged in ascending or descending order by comparing items in pairs and swapping them when they are out of order.)

sinking sort (A bubble sort)

algorithm (The sequence of steps necessary to solve any problem.)

swap values (To exchange the values of two variables.)

insertion sort (A sorting algorithm in which each list element is examined one at a time; if an element is out of order relative to any of the items earlier in the list, each earlier item is moved down one position and then the tested element is inserted.)

one-dimensional array (A list accessed using a single subscript.)

single-dimensional array (A list accessed using a single subscript.)

two-dimensional arrays (Arrays that have rows and columns of values accessed using two subscripts.)

matrix (A term sometimes used by mathematicians to describe a two-dimensional array.)

table (A database file that contains data in rows and columns; also, a term sometimes used by mathematicians to describe a two-dimensional array.)

three-dimensional arrays (Arrays in which each element is accessed using three subscripts.)

multidimensional arrays (Lists with more than one dimension.)

Physical order (The order in which a list is actually stored even though it might be accessed in a different logical order.)

logical order (The order in which a list is used, even though it is not necessarily stored in that physical order.)

key field (The field whose contents make a record unique among all records in a file.)

addresses (Numbers that identify computer memory and storage locations.)

index (A list of key fields paired with the storage address for the corresponding data record.)

random-access storage device (A storage device, such as a disk, from which records can be accessed in any order.)

linked list (A list that contains an extra field in every record that holds the physical address of the next logical record.)

Chapter Review

8-7c Review Questions

1. Employee records stored in order from highest-paid to lowest-paid have been sorted in ____ order.
    a. ascending
    b. descending
    c. staggered
    d. recursive
2. Student records stored in alphabetical order by last name have been sorted in ____ order.
    a. ascending
    b. descending
    c. staggered
    d. recursive
3. When computers sort data, they always ____ .
    a. place items in ascending order
    b. use a bubble sort
    c. use numeric values when making comparisons
    d. begin the process by locating the position of the lowest value
4. Which of the following code segments correctly swaps the values of variables named `x` and `y`?
    a. x = y

```
y = temp

x = temp
```

b.
```
temp = x

x = y

y = temp
```

c.
```
x = y

temp = x

y = temp
```

d.
```
temp = x

y = x

x = temp
```

5. Which type of sort compares list items in pairs, swapping any two adjacent values that are out of order?

   a. bubble sort

   b. indexed sort

   c. insertion sort

   d. selection sort

6. To sort a list of eight values using a bubble sort, the greatest number of times you would have to pass through the list making comparisons is ____ .

   a. six

   b. seven

   c. eight

   d. nine

7. To completely sort a list of eight values using a bubble sort, the greatest possible number of required pair comparisons is ____ .

   a. seven

b. eight

c. 49

d. 64

8. When you do not know how many items need to be sorted in a program, you can create an array that has ____ .

   a. variable-sized elements

   b. at least as many elements as the number you predict you will need

   c. at least one element less than the number you predict you will need

   d. You cannot sort items if you do not know the number of items when you write the program.

9. In a bubble sort, on each pass through the list that must be sorted, you can stop making pair comparisons ____ .

   a. one comparison sooner

   b. two comparisons sooner

   c. one comparison later

   d. two comparisons later

10. When performing a bubble sort on a list of 10 values, you can stop making passes through the list of values as soon as ____ on a single pass through the list.

    a. no swaps are made

    b. exactly one swap is made

    c. no more than nine swaps are made

    d. no more than 10 swaps are made

11. The bubble sort is ____ .

    a. the most efficient sort

    b. a relatively fast sort compared to others

    c. a relatively easy sort to understand

d. all of the above

12. Data stored in a table that can be accessed using row and column numbers is stored as a ____ array.

    a. single-dimensional

    b. two-dimensional

    c. three-dimensional

    d. nondimensional

13. A two-dimensional array declared as `num myArray[6][7]` has ____ columns.

    a. 5

    b. 6

    c. 7

    d. 8

14. In a two-dimensional array declared as `num myArray[6][7]`, the highest row number is ____ .

    a. 5

    b. 6

    c. 7

    d. 8

15. If you access a two-dimensional array with the expression output `myArray[2][5]`, the output value will be ____ .

    a. 0

    b. 2

    c. 5

    d. impossible to tell from the information given

16. Three-dimensional arrays ____ .

a. are supported in many modern programming languages

b. always contain at least nine elements

c. are used only in object-oriented languages

d. all of the above

17. Student records are stored in ID number order, but accessed by grade-point average for a report. Grade-point average order is a(n) ____ order.

a. imaginary

b. physical

c. logical

d. illogical

18. When you store a list of key fields paired with the storage address for the corresponding data record, you are creating ____ .

a. a directory

b. a three-dimensional array

c. a linked list

d. an index

19. When a record in an indexed file is not needed for further processing, ____ .

a. its first character must be replaced with a special character, indicating it is a deleted record

b. its position must be retained, but its fields must be replaced with blanks

c. it must be physically removed from the file

d. the record can stay in place physically, but its reference is removed from the index

20. With a linked list, every record ____ .

a. is stored in sequential order

b. contains a field that holds the address of another record

c. contains a code that indicates the record's position in an imaginary list

d. is stored in a physical location that corresponds to a key field

Chapter Review

8-7d Exercises

1. Design an application that accepts 10 numbers and displays them in descending order.

2. Design an application that accepts eight friends' first names and displays them in alphabetical order.

3. a. Professor Zak allows students to drop the two lowest scores on the 10 100-point quizzes she gives during the semester. Design an application that accepts a student name and 10 quiz scores. Output the student's name and total points for the student's eight highest-scoring quizzes.

   b. Modify the application in Exercise 3a so that the student's mean and median scores on the eight best quizzes are displayed.

4. The Keen Knife Company has 12 salespeople. Write a program into which a clerk can enter each salesperson's monthly sales goal and actual monthly sales in dollars, and determine the mean and median values of each of the two monthly amounts.

5. The village of Marengo conducted a census and collected records that contain household data, including the number of occupants in each household. The exact number of household records has not yet been determined, but you know that Marengo has fewer than 300 households. Develop the logic for a program that allows a user to enter each household size and determine the mean and median household size in Marengo.

6. a. The Palmertown Elementary School has 30 classrooms. The children in the school donate used books to sell at an annual fundraising book fair. Write a program that accepts each teacher's name and the number of

books donated by that teacher's classroom. Display the names of the four teachers whose classrooms donated the most books.

b. Modify the book donation program so that, besides the teacher's name and number of books donated, the program also accepts the number of students in each classroom. Display the names of the teachers whose classrooms had the four highest ratios of book donations per pupil.

7. *The Daily Trumpet* newspaper accepts classified advertisements in 15 categories such as *Apartments for Rent* and *Pets for Sale*. Develop the logic for a program that accepts classified advertising data, including category code (an integer 1 through 15) and number of words in the ad. Store these values in parallel arrays. Then sort the arrays so that records are in ascending order by category. The output lists each category, the number of ads in each category, and the total number of words in the ads in each category.

8. The MidAmerica Bus Company charges fares to passengers based on the number of travel zones they cross. Additionally, discounts are provided for multiple passengers traveling together. Ticket fares are shown in Table 8-5.

Table 8-5

Bus Fares

	Zones Crossed			
Passengers	**0**	**1**	**2**	**3**
1	7.50	10.00	12.00	12.75
2	14.00	18.50	22.00	23.00
3	20.00	21.00	32.00	33.00
4	25.00	27.50	36.00	37.00

Develop the logic for a program that accepts the number of passengers and zones crossed as input. The output is the ticket charge.

9. In golf, par represents a standard number of strokes a player needs to

complete a hole. Instead of using an absolute score, players can compare their scores on a hole to the par figure. Families can play nine holes of miniature golf at the Family Fun Miniature Golf Park. So that family members can compete fairly, the course provides a different par for each hole based on the player's age. The par figures are shown in Table 8-6.

Table 8-6

Golf Par Values

	Holes								
Age	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
4 and under	8	8	9	7	5	7	8	5	8
5-7	7	7	8	6	5	6	7	5	6
8-11	6	5	6	5	4	5	5	4	5
12-15	5	4	4	4	3	4	3	3	4
16 and over	4	3	3	3	2	3	2	3	3

a. Develop the logic for a program that accepts a player's name, age, and nine-hole score as input. Display the player's name and score on each of the nine holes, with one of the phrases *Over par, Par,* or *Under par* next to each score.

b. Modify the program in Exercise 9a so that, at the end of the golfer's report, the total score is displayed. Include the player's total score in relation to par for the entire course.

10. Building Block Day Care Center charges varying weekly rates depending on the age of the child and the number of days per week the child attends, as shown in Table 8-7. Develop the logic for a program that continuously accepts child care data and displays the appropriate weekly rate.

Table 8-7

Day Care Rates

Days Per Week

Age in Years	1	2	3	4	5
0	30.00	60.00	88.00	115.00	140.00
1	26.00	52.00	70.00	96.00	120.00
2	24.00	46.00	67.00	89.00	110.00
3	22.00	40.00	60.00	75.00	88.00
4 or more	20.00	35.00	50.00	66.00	84.00

11. Executive Training School offers typing classes. Each final exam evaluates a student's typing speed and the number of typing errors made. Develop the logic for a program that produces a summary table of each examination's results. Each row represents the number of students whose typing speed falls within the following ranges of words per minute: 0–19, 20–39, 40–69, and 70 or more. Each column represents the number of students who made different numbers of typing errors—0 through 6 or more.

12. HappyTunes is an application for downloading music files. Each time a file is purchased, a transaction record is created that includes the music genre and price paid. The available genres are *Classical, Easy Listening, Jazz, Pop, Rock,* and *Other*. Develop an application that accepts input data for each transaction and displays a report that lists each of the music genres, along with a count of the number of downloads in each of the following price categories:

    - Over $10.00
    - $6.00 through $9.99

- $3.00 through $5.99
- Under $3.00

Find the Bugs

13. Your downloadable files for Chapter 8 include DEBUG08-01.txt, DEBUG08-02.txt, and DEBUG08-03.txt. Each file starts with some comments that describe the problem. Comments are lines that begin with two slashes (//). Following the comments, each file contains pseudocode that has one or more bugs you must find and correct.

Game Zone

14. In the Game Zone section of Chapter 6, you designed the logic for a quiz that contains multiple-choice questions about a topic of your choice. (Each question had three answer options.) Now, modify the program so it allows the user to retake the quiz up to four additional times or until the user achieves a perfect score, whichever comes first. At the end of all the quiz attempts, display a recap of the user's scores.

15. In the Game Zone section of Chapter 5, you designed a guessing game in which the application generates a random number and the player tries to guess it. After each guess, you displayed a message indicating whether the player's guess was correct, too high, or too low. When the player eventually guessed the correct number, you displayed a score that represented a count of the number of required guesses. Now, modify that program to allow a player to replay the game as many times as he likes, up to 20 times. When the player is done, display the scores from highest to lowest, and display the mean and median scores.

    a. Create a TicTacToe game. In this game, two players alternate placing Xs and Os into a grid until one player has three matching symbols in a row, either horizontally, vertically, or diagonally. Create a game that displays a three-by-three grid containing the digits 1 through 9, similar to the first window shown in Figure 8-18. When the user chooses a position by typing a number, place an X in the appropriate spot. For example, after the user chooses 3, the screen looks like the second window in Figure 8-18.

Generate a random number for the position where the computer will place an O. Do not allow the player or the computer to place a symbol where one has already been placed. When either the player or computer has three symbols in a row, declare a winner. If all positions have been used and no one has three symbols in a row, declare a tie.

Figure 8-18

A Tictactoe Game

b. In the TicTacToe game in Exercise 16a, the computer's selection is chosen randomly. Improve the game so that when the computer has two Os in any row, column, or diagonal, it selects the winning position for its next move rather than selecting a position randomly.

Up for Discussion

17. Now that you are becoming comfortable with arrays, you can see that programming is a complex subject. Should all literate people understand how to program? If so, how much programming should they understand?

18. What are language standards? At this point in your study of programming, what do they mean to you?

19. This chapter discusses sorting data. Suppose that a large hospital hires you to write a program that displays lists of potential organ recipients. The hospital's doctors will consult this list if they have an organ that can be transplanted. The hospital administrators instruct you to sort potential recipients by last name and display them sequentially in alphabetical order. If more than 10 patients are waiting for a particular organ, the first 10 patients are displayed; a doctor can either select one or move on to view the next set of 10 patients. You worry that this system gives an unfair advantage to patients with last names that start with A, B, C, and D. Should you write and install the program? If you do not, many transplant opportunities will be missed while the hospital searches for another programmer to write the

program. Are there different criteria you would want to use to sort the patients?