8 ARRAYS

After studying Chapter 8, you should be able to:

- ☐ Understand how arrays are used
- ☐ Understand how arrays occupy computer memory
- ☐ Manipulate an array to replace nested decisions
- ☐ Declare and initialize an array
- ☐ Declare and initialize constant arrays
- ☐ Load array values from a file
- ☐ Search an array for an exact match
- ☐ Use parallel arrays
- ☐ Force subscripts to remain within array bounds
- ☐ Improve search efficiency by using an early exit
- ☐ Search an array for a range match

UNDERSTANDING ARRAYS

An **array** is a series or list of variables in computer memory, all of which have the same name but are differentiated with special numbers called subscripts. A **subscript** is a number that indicates the position of a particular item within an array. Whenever you require multiple storage locations for objects, you are using a real-life counterpart of a programming array. For example, if you store important papers in a series of file folders and label each folder with a consecutive letter of the alphabet, then you are using the equivalent of an array. If you store mementos in a series of stacked shoeboxes, each labeled with a year, or if you sort mail into slots, each labeled with a name, then you are also using a real-life equivalent of a programming array.

TIP Besides the term "subscript," programmers also use the term "**index**" to refer to the number that indicates a position within an array.

When you look down the left side of a tax table to find your income level before looking to the right to find your income tax obligation, you are using an array. Similarly, if you look down the left side of a train schedule to find your station before looking to the right to find the train's arrival time, you also are using an array.

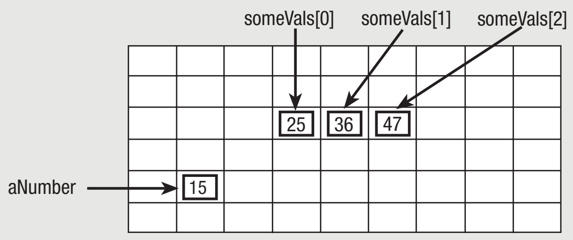
Each of these real-life arrays helps you organize real-life objects. You *could* store all your papers or mementos in one huge cardboard box, or find your tax rate or train's arrival time if both were printed randomly in one large book. However, using an organized storage and display system makes your life easier in each case. Using a programming array accomplishes the same results for your data.

TIP Some programmers refer to an array as a *table* or a *matrix*.

HOW ARRAYS OCCUPY COMPUTER MEMORY

When you declare an array, you declare a programming structure that contains multiple variables. Each variable within an array has the same name and the same data type; each separate array variable is one **element** of the array. Each array element occupies an area in memory next to, or contiguous to, the others, as shown in Figure 8-1. You indicate the number of elements an array will hold—the **size of the array**—when you declare the array along with your other variables.

FIGURE 8-1: APPEARANCE OF A THREE-ELEMENT ARRAY AND A SINGLE VARIABLE IN COMPUTER MEMORY



All array elements have the same group name, but each individual element also has a unique subscript indicating how far away it is from the first element. Therefore, any array's subscripts are always a sequence of integers, such as 0 through 5 or 0 through 10. Depending on the syntax rules of the programming language you use, you place the subscript within either parentheses or square brackets following the group name; when writing pseudocode or drawing a flowchart, you can use either form of notation. This text uses square brackets to hold array element subscripts so that you don't mistake array names for method names. For example, Figure 8-1 shows how a single variable and an array are stored in computer memory. The single variable named `aNumber` holds the value 15. The array named `someVals` contains three elements, so the elements are `someVals[0]`, `someVals[1]`, and `someVals[2]`. The value stored in `someVals[0]` is 25, `someVals[1]` holds 36, and `someVals[2]` holds 47. From the diagram in Figure 8-1, you can see that the memory location `someVals[0]` is zero elements away from the beginning of the array, the location of `someVals[1]` is one memory location away, and the location of `someVals[2]` is two elements away from the start of the array.

TIP In general, older programming languages such as COBOL and RPG use parentheses to hold their array subscripts. Newer languages such as C#, C++, and Java use square brackets.

TIP In many modern languages (for example, Java, Visual Basic .NET, C#, and C++), the first array element's subscript is 0; in others (for example, COBOL and RPG), it is 1. In Pascal, you can identify the starting number as any value you want. In languages in which the first subscript is 0, the subscript alone indicates the distance from the start of the array. In languages that use a starting subscript value other than 0, the compiler does the arithmetic for you to calculate the number of elements past the start of the array that you want to access. In all languages, however, the subscript values must be integers (whole numbers) and sequential.

Because the first element in an array in most programming languages is accessed using a subscript of value 0, the array is called a **zero-based array**. Because the lowest subscript you can use with an array is 0, the highest subscript you are allowed to use with an array is one less than the number of elements in the array. For example, an array with 10 elements uses subscripts 0 through 9, and an array with 200 elements uses subscripts 0 through 199. When you use arrays, you must always keep the limits of subscript values in mind.

TIP If you treat an array as though its lowest legal subscript is 1, when in fact it is 0, you will commit **off-by-one errors**. If you use an invalid subscript—for example, using a 10 in a 10-element array for which the subscripts should be 0 through 9—some language compilers will issue an error message and stop program execution, but others will allow you to make the mistake, resulting in incorrect output.

You are never required to use arrays within your programs, but learning to use arrays correctly can make many programming tasks far more efficient and professional. When you understand how to use arrays, you will be able to provide elegant solutions to problems that otherwise would require tedious programming steps.

TIP When you describe people or events as "elegant," you mean they possess a refined gracefulness. Similarly, programmers use the term "elegant" to describe programs that are well-designed and easy to understand and maintain.

MANIPULATING AN ARRAY TO REPLACE NESTED DECISIONS

Consider a program that keeps statistics for requests about apartments in a large apartment complex. The developer wants to keep track of inquiries so that future building projects are more likely to satisfy customer needs. In particular, the developer wants to keep track of how many requests there are for studio, one-, two-, and three-bedroom apartments. Each time an apartment request is received, a clerk adds a record to a file in the format shown in Figure 8-2.

FIGURE 8-2: FILE DESCRIPTION FOR APARTMENT REQUEST RECORDS

```
APARTMENT INQUIRY FILE DESCRIPTION
File name: APTREQUESTS
FIELD DESCRIPTION      DATA TYPE    COMMENTS
Day of the month       Numeric      1 - 31, day request was made
Bedrooms requested     Numeric      0, 1, 2 or 3 for studio apartment
                                    or number of bedrooms
```

For example, if a call comes in on the third day of the month for a studio apartment, one record is created with a 3 in the date field and a 0 in the number of bedrooms field. If the next call is on the fourth day of the month for a three-bedroom apartment, a record with 4 and 3 is created. The contents of the data file appear as a series of numbers, as follows:

```
3 0
4 3
4 0
```

... and so on.

At the end of the month, after all the records have been collected, the file might contain hundreds of records, each holding a number that represents a date and another number (0, 1, 2, or 3) that represents the number of bedrooms the caller wanted. You want to write a program that summarizes the total number of each type of apartment requested during the month. A typical report appears in Figure 8-3.

FIGURE 8-3: TYPICAL APARTMENT REQUEST REPORT

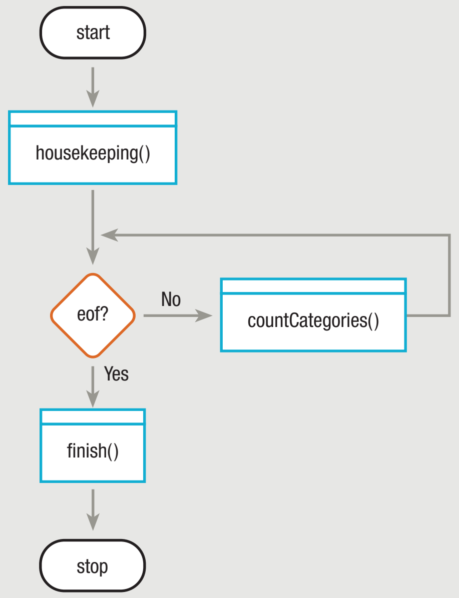
```
Apartment Request Report

Bedrooms   Inquiries

   0          91
   1          44
   2          67
   3         102
```

If all the records were sorted in order by the number of bedrooms requested, this report could be a control break report. You would simply read each record representing an inquiry on a studio apartment (zero bedrooms), counting the number of inquiries. When you read the first record requesting a different number of bedrooms, you would print the count for the previous apartment type, reset the count to zero, and update the control break field before continuing.

TIP You learned about control break logic in Chapter 7.

Assume, however, that the records have not been sorted by apartment type. Without using an array, could you write a program that would accumulate the four apartment-type totals? Of course you could. The program would have the same mainline logic as most of the other programs you have seen, as shown in Figure 8-4.

FIGURE 8-4: FLOWCHART AND PSEUDOCODE FOR MAINLINE LOGIC OF APARTMENT REQUEST REPORT PROGRAM

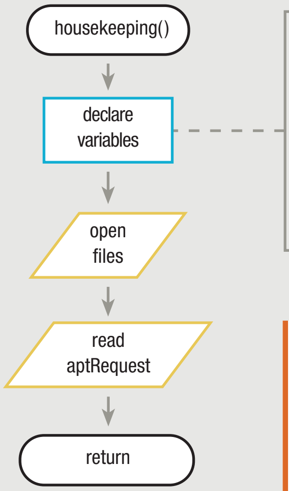


```
start
     perform housekeeping()
     while not eof
          perform countCategories()
     endwhile
     perform finish()
stop
```

TIP The program shown in Figures 8-4 through 8-7 accomplishes its purpose, but is cumbersome. Follow its logic here, so that you understand how the program works. Later in this chapter, you will see how to write the apartment request report program much more efficiently using arrays.

In the `housekeeping()` module of the apartment request report program (Figure 8-5), you declare variables including `day` and `bedrooms`. Then, you open the files and read the first record into memory. The headings *could* print in `housekeeping()` or—because no other printing takes place in this program until the `finish()` module—you can choose to wait and print the headings there.

FIGURE 8-5: THE `housekeeping()` MODULE FOR THE APARTMENT REQUEST PROGRAM WITHOUT AN ARRAY



```
aptRequest
   num day
   num bedrooms
char head1 = "Apartment Request Report"
char head2 = "Bedrooms   Inquiries"
num count0 = 0
num count1 = 0
num count2 = 0
num count3 = 0
```

```
housekeeping()
   declare variables
     aptRequest
       num day
       num bedrooms
     char head1 = "Apartment Request Report"
     char head2 = "Bedrooms   Inquiries"
     num count0 = 0
     num count1 = 0
     num count2 = 0
     num count3 = 0
   open files
   read aptRequest
return
```

TIP In Figure 8-5, the variable list is identical for the flowchart and the pseudocode. It is included twice in this figure and in the next few for clarity because arrays are a new and complicated topic. In later examples in this book, the duplication of the variable list will be eliminated.

Within the `housekeeping()` module, you can declare four variables, `count0`, `count1`, `count2`, and `count3`; the purpose of these variables is to keep running counts of the number of requests for the four apartment types. Each of these four counter variables needs to be initialized to 0. You can tell by looking at the planned output that you need two heading lines, so `head1` is defined as “Apartment Request Report” and `head2` as “Bedrooms Inquiries”.

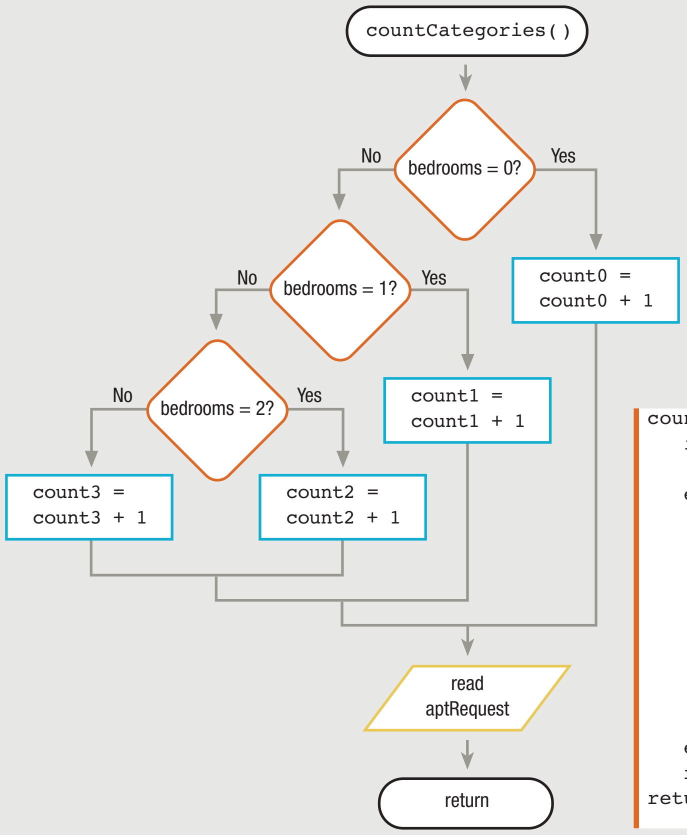
Eventually, four summary lines will be printed in the report, each with a number of bedrooms and a count of inquiries for that apartment type. These lines cannot be printed until the `finish()` module, however, because you won’t have a complete count of each apartment type’s requests until all input records have been read.

The logic within the `countCategories()` module of the program requires adding a 1 to `count0`, `count1`, `count2`, or `count3`, depending on the `bedrooms` variable. After 1 has been added to one of the four counters, you read the next record, and if it is not `eof`, you repeat the decision-making and counting process. When all records have been read, you proceed to the `finish()` module, where you print the four summary lines with the counts for the four apartment types. See Figures 8-6 and 8-7.

TIP

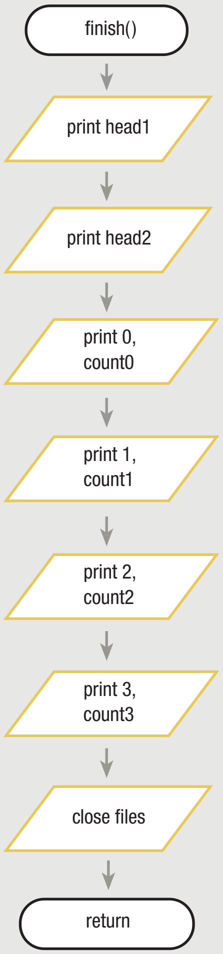
In the apartment request report program, assume that the input data has been previously edited to ensure that all apartment requests are for zero, one, two, or three bedrooms. In other words, there is no bad data. If this were not true, then the program would also need to include a step to check for incorrect data and take some appropriate action—perhaps ignoring it, or counting it in an error category.

FIGURE 8-6: THE `countCategories()` MODULE FOR THE APARTMENT REQUEST PROGRAM WITHOUT AN ARRAY



```
countCategories()
   if bedrooms = 0 then
      count0 = count0 + 1
   else
      if bedrooms = 1 then
         count1 = count1 + 1
      else
         if bedrooms = 2 then
            count2 = count2 + 1
         else
            count3 = count3 + 1
         endif
      endif
   endif
   read aptRequest
return
```

FIGURE 8-7: THE `finish()` MODULE FOR THE APARTMENT REQUEST PROGRAM WITHOUT AN ARRAY

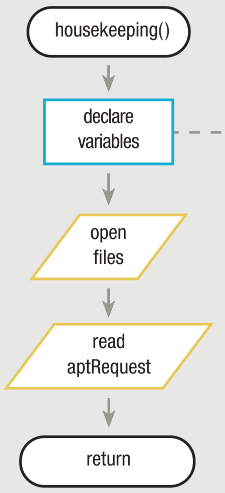


```
finish()
   print head1
   print head2
   print 0, count0
   print 1, count1
   print 2, count2
   print 3, count3
   close files
return
```

The apartment request report program works just fine, and there is absolutely nothing wrong with it logically; a decision is made for each of the first three types of apartments, defaulting to a three-bedroom apartment if the request is not for zero, one, or two bedrooms. But what if there were four types of apartments, or 12, or 30? With any of these scenarios, the basic logic of the program would remain the same; however, you would need to declare many additional counter variables. You also would need many additional decisions within the `countCategories()` module and many additional print statements within the `finish()` module to complete the processing.

Using an array provides an alternative approach to this programming problem, and greatly reduces the number of statements you need. When you declare an array, you provide a group name for a number of associated variables in memory. For example, the four apartment-type counters can be redefined as a single array named `count`. The individual elements become `count[0]`, `count[1]`, `count[2]`, and `count[3]`, as shown in the new `housekeeping()` module in Figure 8-8.

FIGURE 8-8: MODIFIED `housekeeping()` MODULE FOR APARTMENT REQUEST PROGRAM THAT DECLARES AN ARRAY TO COUNT REQUESTS

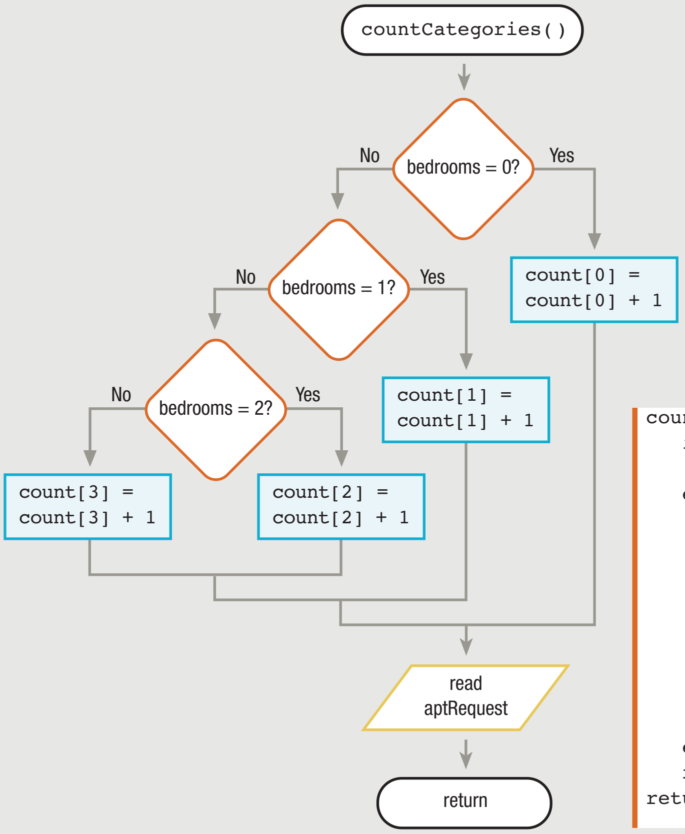


```
aptRequest
   num day
   num bedrooms
char head1 = "Apartment Request Report"
char head2 = "Bedrooms   Inquiries"
num count[0] = 0
num count[1] = 0
num count[2] = 0
num count[3] = 0
```

```
housekeeping()
   declare variables
     aptRequest
       num day
       num bedrooms
     char head1 = "Apartment Request Report"
     char head2 = "Bedrooms   Inquiries"
     num count[0] = 0
     num count[1] = 0
     num count[2] = 0
     num count[3] = 0
   open files
   read aptRequest
return
```

With the change to `housekeeping()` shown in Figure 8-8, the `countCategories()` module changes to the version shown in Figure 8-9.

FIGURE 8-9: MODIFIED `countCategories()` MODULE THAT USES `count` ARRAY

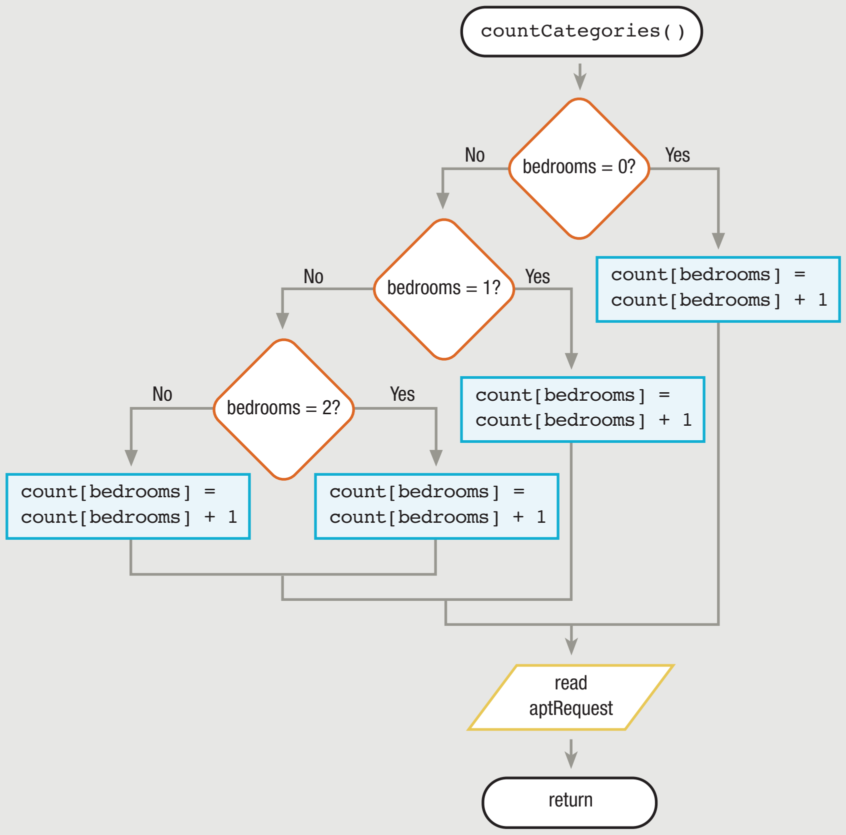


```
countCategories()
   if bedrooms = 0 then
      count[0] = count[0] + 1
   else
      if bedrooms = 1 then
         count[1] = count[1] + 1
      else
         if bedrooms = 2 then
            count[2] = count[2] + 1
         else
            count[3] = count[3] + 1
         endif
      endif
   endif
   read aptRequest
return
```

Figure 8-9 shows that when the **`bedrooms`** variable value is 0, one is added to **`count[0]`**; when the **`bedrooms`** value is 3, one is added to **`count[3]`**. In other words, one is added to one of the elements of the **`count`** array instead of to a single variable named **`count0`**, **`count1`**, **`count2`**, or **`count3`**. Is this a big improvement over the original? Of course it isn't. You still have not taken advantage of the benefits of using the array in this program.

The true benefit of using an array lies in your ability to use a variable as a subscript to the array, instead of using a constant such as 1 or 4. Notice in the **`countCategories()`** module in Figure 8-9 that within each decision, the value you are comparing to **`bedrooms`** and the constant you are using as a subscript in the resulting "Yes" process are always identical. That is, when the **`bedrooms`** value is 0, the subscript used to add 1 to the **`count`** array is 0; when the **`bedrooms`** value is 1, the subscript used for the **`count`** array is 1, and so on. Therefore, why not just use the value of **`bedrooms`** as a subscript? You can rewrite the **`countCategories()`** module as shown in Figure 8-10.

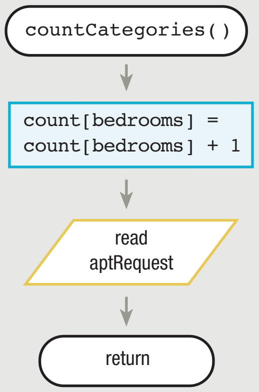
FIGURE 8-10: MODIFIED `countCategories()` MODULE USING THE VARIABLE `bedrooms` AS A SUBSCRIPT TO THE `count` ARRAY


countCategories()
bedrooms = 0?
No
Yes
count[bedrooms] = count[bedrooms] + 1
bedrooms = 1?
No
Yes
count[bedrooms] = count[bedrooms] + 1
bedrooms = 2?
No
Yes
count[bedrooms] = count[bedrooms] + 1
count[bedrooms] = count[bedrooms] + 1
read aptRequest
return

```
countCategories()
   if bedrooms = 0 then
      count[bedrooms] = count[bedrooms] + 1
   else
      if bedrooms = 1 then
         count[bedrooms] = count[bedrooms] + 1
      else
         if bedrooms = 2 then
            count[bedrooms] = count[bedrooms] + 1
         else
            count[bedrooms] = count[bedrooms] + 1
         endif
      endif
   endif
   read aptRequest
return
```

Of course, the code segment in Figure 8-10 looks no more efficient than the one in Figure 8-9. However, notice that in Figure 8-10 the process that occurs after each decision is exactly the same. In each case, no matter what the value of `bedrooms`, you always add one to `count[bedrooms]`. If you are always going to take the same action no matter what the answer to a question is, why ask the question? Instead, you can write the `countCategories()` module as shown in Figure 8-11.

FIGURE 8-11: MODIFIED `countCategories()` MODULE, ELIMINATING NESTED DECISIONS

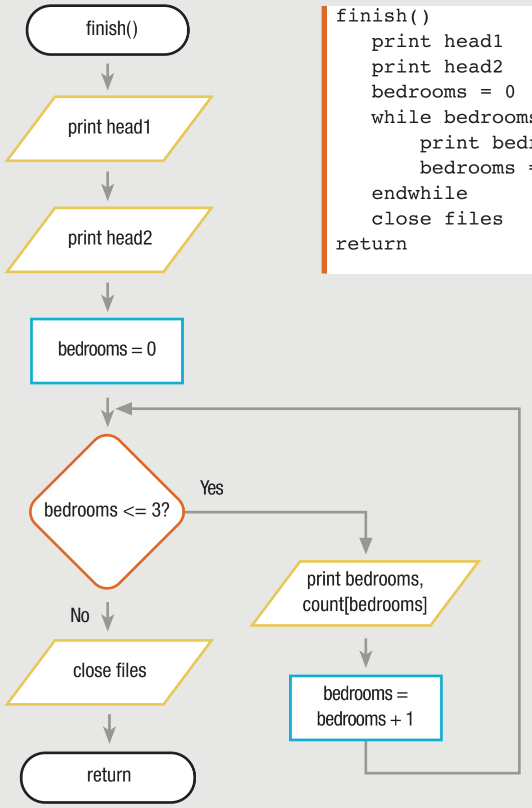


```
countCategories()
   count[bedrooms] = count[bedrooms] + 1
   read aptRequest
return
```

The two steps in Figure 8-11 represent the *entire* `countCategories()` module! When the value of `bedrooms` is 0, one is added to `count[0]`; when the value of `bedrooms` is 1, one is added to `count[1]`, and so on. *Now*, you have a big improvement to the previous `countCategories()` module from Figure 8-9. What's more, this `countCategories()` module does not change whether there are eight, 30, or any other number of types of apartment requests and `count` array elements, as long as the values in the `bedrooms` variable are numbered sequentially. To use more than four counters, you would declare additional `count` elements in the `housekeeping()` module, but the `countCategories()` logic would remain the same as it is in Figure 8-11.

The `finish()` module originally shown in Figure 8-7 can also be improved. Instead of four separate print statements, you can use a variable to control a printing loop, as shown in Figure 8-12. Because the `finish()` module follows the `eof` condition, all input records have been used, and the `bedrooms` variable is not currently holding any needed information. In `finish()`, you can set `bedrooms` to 0, and then print `bedrooms` and `count[bedrooms]`. Then add 1 to `bedrooms` and use the same set of instructions again. You can use `bedrooms` as a loop control variable to print the four individual `count` values. The improvement in this `finish()` module over the one shown in Figure 8-7 is not as dramatic as the improvement in the `countCategories()` module, but in a program with more `count` elements, the only change to the `finish()` module would be in the constant value you use to control the end of the loop. Twelve or 30 `count` values can print as easily as four if they are stored in an array.

FIGURE 8-12: MODIFIED `finish()` MODULE THAT USES AN ARRAY



```
finish()
   print head1
   print head2
   bedrooms = 0
   while bedrooms <= 3
       print bedrooms, count[bedrooms]
       bedrooms = bedrooms + 1
   endwhile
   close files
return
```

TIP

In the `finish()` module in Figure 8-12, instead of reusing the `bedrooms` variable as a subscript, many programmers prefer to declare a separate numeric work variable to initialize to 0, use it as a subscript to the array while printing, and increment it during each cycle through the loop. Their reasoning is that `bedrooms` is part of the input record and should be used only to hold actual data being input—not used as a work variable in the program. Use this approach if it makes more sense to you. You might be required to use this technique if the input data is accessed from databases containing an input field that is no longer available after the input has reached the `eof` condition.

Within the `finish()` module in Figure 8-12, the `bedrooms` variable is handy to use as a subscript, but any variable could have been used as long as it was:

- Numeric with no decimal places
- Initialized to 0
- Incremented by 1 each time the logic passed through the loop

In other words, nothing is linking `bedrooms` to the `count` array per se; within the `finish()` module, you can simply use the `bedrooms` variable as a subscript to indicate each successive element within the `count` array.

The apartment request report program *worked* when the `countCategories()` module contained a long series of decisions and the `finish()` module contained a long series of print statements, but the program is easier to write when you employ arrays. Additionally, the program is more efficient, easier for other programmers to understand, and easier to maintain. Arrays are never mandatory, but often they can drastically cut down on your programming time and make a program easier to understand.

ARRAY DECLARATION AND INITIALIZATION

In the apartment request report program, the four `count` array elements were declared and initialized to 0s in the `housekeeping()` module. The `count` values need to start at 0 so they can be added to during the course of the program. Originally (see Figure 8-8), you provided initialization in the `housekeeping()` module as:

```
num count[0] = 0
num count[1] = 0
num count[2] = 0
num count[3] = 0
```

Separately declaring and initializing each `count` element is acceptable only if there are a small number of `count`s. If the apartment request report program were updated to keep track of 30 types of apartments, you would have to initialize 30 separate `count` fields. It would be tedious to write 30 separate declaration statements.

Programming languages do not require the programmer to name each of the 30 `count`s: `count[0]`, `count[1]`, and so on. Instead, you can make a declaration such as one of those in Figure 8-13.

FIGURE 8-13: DECLARING A 30-ELEMENT ARRAY NAMED `count` IN SEVERAL COMMON LANGUAGES

Declaration	**Programming Language**
`DIM COUNT(30)`	BASIC, Visual Basic
`int count[30];`	C#, C++
`int[] count = new int[30];`	Java
`COUNT OCCURS 30 TIMES PICTURE 9999.`	COBOL
`array count [1..30] of integer;`	Pascal

TIP C, C++, C#, and Java programmers typically use lowercase variable names. COBOL and BASIC programmers often use all uppercase. Visual Basic programmers are likely to begin with an uppercase letter.

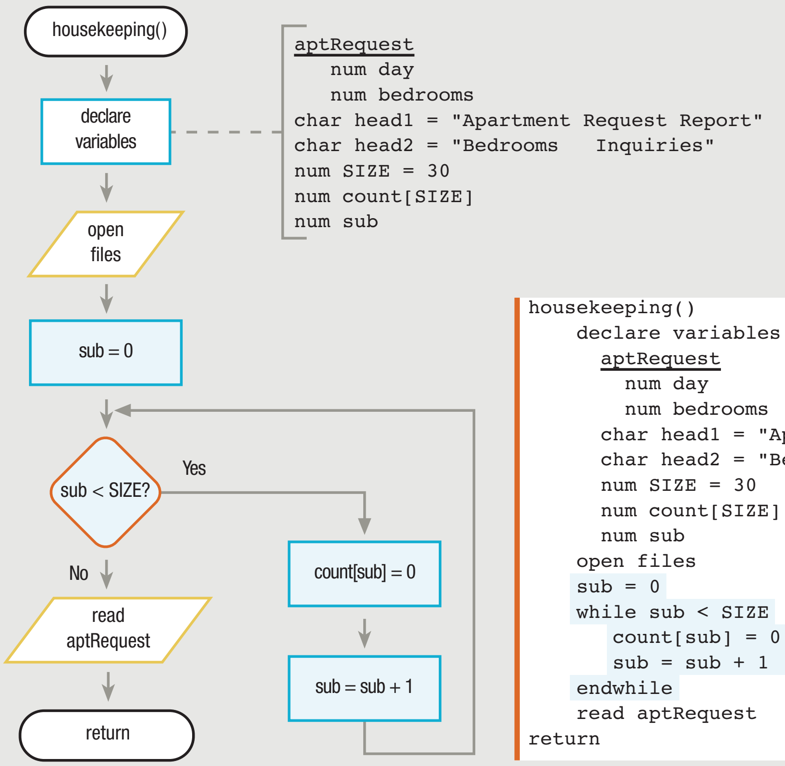
TIP The terms `int` and `integer` in the code samples within Figure 8-13 both indicate that the `count` array will hold whole-number values. The value 9999 in the COBOL example indicates that each `count` will be a four-digit integer. These terms are more specific than the `num` identifier this book uses to declare all numeric variables.

All the declarations in Figure 8-13 have two things in common: They name the `count` array and indicate that there will be 30 separate numeric elements. For flowcharting or pseudocode purposes, a statement such as `num count[30]` indicates the same thing.

Declaring a numeric array does not necessarily set its individual elements to 0 (although it does in some programming languages, such as BASIC, Visual Basic, and Java). Most programming languages allow the equivalent of `num count[30] all set to 0`; you should use a statement like this when you want to initialize an array in your flowcharts or pseudocode. Explicitly initializing all variables is a good programming practice; assuming anything about noninitialized variable values is a dangerous practice. Array elements are no exception to this rule.

Alternatively, to start all array elements with the same initial value, you can use an initialization loop within the `housekeeping()` module. An **initialization loop** is a loop structure that provides initial values for every element in any array. To create an initialization loop, you must use a numeric variable as a subscript. For example, if you declare a field named `sub`, and initialize `sub` to 0, then you can use a loop like the one shown in the `housekeeping()` module in Figure 8-14 to set all the array elements to 0. As the value of `sub` increases from 0 through 29, each corresponding `count` element is assigned 0.

FIGURE 8-14: A `housekeeping()` MODULE DEMONSTRATING ONE METHOD OF INITIALIZING ARRAY ELEMENTS



```
aptRequest
   num day
   num bedrooms
char head1 = "Apartment Request Report"
char head2 = "Bedrooms   Inquiries"
num SIZE = 30
num count[SIZE]
num sub
```

```
housekeeping()
    declare variables
      aptRequest
        num day
        num bedrooms
      char head1 = "Apartment Request Report"
      char head2 = "Bedrooms   Inquiries"
      num SIZE = 30
      num count[SIZE]
      num sub
    open files
    sub = 0
    while sub < SIZE
       count[sub] = 0
       sub = sub + 1
    endwhile
    read aptRequest
return
```

TIP

In Figure 8-14, a named constant SIZE is initialized to 30. This constant is then used in both the array declaration and the loop that controls how many elements are set to 0. Using a constant such as SIZE is a convenient way to make sure you access all the array elements. Additionally, if you want to alter the program to handle some other number of apartment types, the only change you need to make to the program is to provide a different value for the constant. You first learned about named constants in Chapter 4.

DECLARING AND INITIALIZING CONSTANT ARRAYS

The array that you used to accumulate apartment-type requests in the previous section contained four variables whose values were altered during the execution of the program. The values in which you were most interested, the count of the number of requests for each type of apartment, were created during an actual run, or execution, of the program. In other words, if 1,000 prospective tenants are interested in studio apartments, you don't know that fact at the beginning of the program. Instead, that value is accumulated during the execution of the program and not known until the end.

Some arrays are not variable, but are meant to be constant. With some arrays, the final desired values are fixed at the beginning of the program.

For example, let's say you own an apartment building with five floors, including a basement, and you have records for all your tenants with the information shown in Figure 8-15. The combination of each tenant's floor number and apartment letter provides you with a specific apartment—for example, apartment 0D or 3B.

FIGURE 8-15: TENANT FILE DESCRIPTION

```
TENANT FILE DESCRIPTION
File name: TENANTS
FIELD DESCRIPTION      DATA TYPE    COMMENTS
Tenant name            Character    Full name, first and last
Floor number           Numeric      0 through 4 - 0 is basement
Apartment letter       Character    Single letter - A through F
```

Every month, you print a rent bill for each tenant. Your rent charges are based on the floor of the building, as shown in Figure 8-16.

FIGURE 8-16: RENTS BY FLOOR

```
Floor               Rent in $
0 (the basement)       350
1                      400
2                      475
3                      600
4 (the penthouse)     1000
```

To create a computer program that prints each tenant's name and rent due, you could use five decisions concerning the floor number. However, it is more efficient to use an array to hold the five rent figures. The array's values are constant because you set them once at the beginning of the program, and they never change.

TIP Remember that another name for an array is a *table*. If you can use paper and pencil to list items like tenants' rent values in a table format, then using an array is an appropriate programming option.

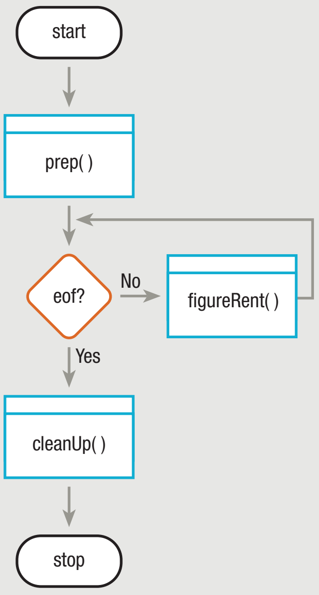
TIP In most programming languages, you would include a modifier such as `const` or `final` in front of the array name to declare it to be truly constant, so that you could not alter any of its elements' values later in the program.

TIP Some programmers use the term "compile-time arrays" to refer to arrays that receive their usable values through initialization at the start of a program, whereas arrays that do not receive their ultimate values until the program is being used are run-time arrays.

The mainline logic for this program is shown in Figure 8-17. The housekeeping module is named `prep()`. When you declare variables within the `prep()` module, you create an array for the five rent figures and set `num rent[0] = 350`, `num rent[1] = 400`, and so on. The rent amounts are **hard coded** into the array; that is, they are explicitly assigned to the array elements. The `prep()` module is shown in Figure 8-18.

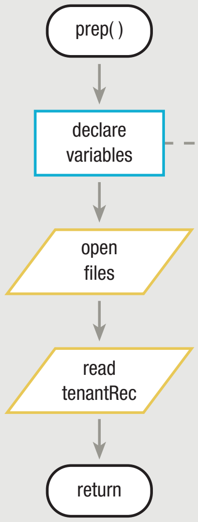
TIP The `prep()` module name was chosen as a change of pace from `housekeeping()`, which has been used in many examples in this book. Some programmers advocate being consistent in naming modules from program to program; others prefer varying names as long as the names are meaningful.

FIGURE 8-17: FLOWCHART AND PSEUDOCODE FOR MAINLINE LOGIC OF RENT PROGRAM



```
start
   perform prep()
   while not eof
      perform figureRent()
   endwhile
   perform cleanUp()
stop
```

FIGURE 8-18: FLOWCHART AND PSEUDOCODE FOR `prep()` MODULE OF RENT PROGRAM



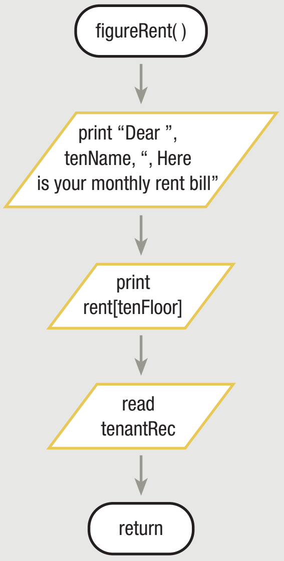
```
tenantRec
   char tenName
   num tenFloor
   char tenAptLetter
num rent[0] = 350
num rent[1] = 400
num rent[2] = 475
num rent[3] = 600
num rent[4] = 1000
```

```
prep()
       declare variables
       open files
       read tenantRec
return
```

TIP As an alternative to defining `rent[0]`, `rent[1]`, and so on, as in Figure 8-18, most programming languages allow a more concise version that takes the general form `num rent[5] = 350, 400, 475, 600, 1000`. When you use this form of array initialization, the first value you list is assigned to the first array element, and the subsequent values are assigned in order. Most programming languages allow you to assign fewer values than there are array elements declared, but none allow you to assign more values.

At the end of the `prep()` module, you read a first record into memory. The record contains a tenant name (`tenName`), floor (`tenFloor`), and apartment letter (`tenAptLetter`). When the logic enters `figureRent()` (the main loop), you can print three items: "Dear ", `tenName`, and ", Here is your monthly rent bill" (the quote begins with a comma that follows the recipient's name). Then, you must print the rent amount. Instead of making a series of selections such as `if tenFloor = 0 then print rent[0]` and `if tenFloor = 1 then print rent[1]`, you want to take advantage of the `rent` array. The solution is to create a `figureRent()` module that looks like Figure 8-19. You use the `tenFloor` variable as a subscript to access the correct `rent` array element. When deciding which variable to use as a subscript with an array, ask yourself, "Of all the values available in the array, what does the correct selection depend on?" When printing a `rent` value, the rent you use depends on the floor on which the tenant lives, so the correct action is `print rent[tenFloor]`.

FIGURE 8-19: FLOWCHART AND PSEUDOCODE FOR THE `figureRent()` MODULE OF THE RENT PROGRAM

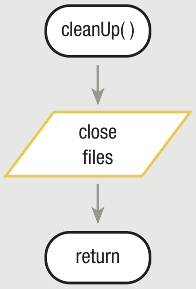


```
figureRent()
     print "Dear ", tenName, ", Here is your monthly rent bill"
     print rent[tenFloor]
     read tenantRec
return
```

TIP Every programming language provides ways to space your output for easy reading. For example, a common technique to separate "Dear" from the tenant's name is to include a space after the *r* in *Dear*, as in `print "Dear ", tenName`.

The `cleanUp()` module for this program is very simple—just close the files. See Figure 8-20.

FIGURE 8-20: THE `cleanUp()` MODULE FOR THE RENT PROGRAM



```
cleanUp()
     close files
return
```

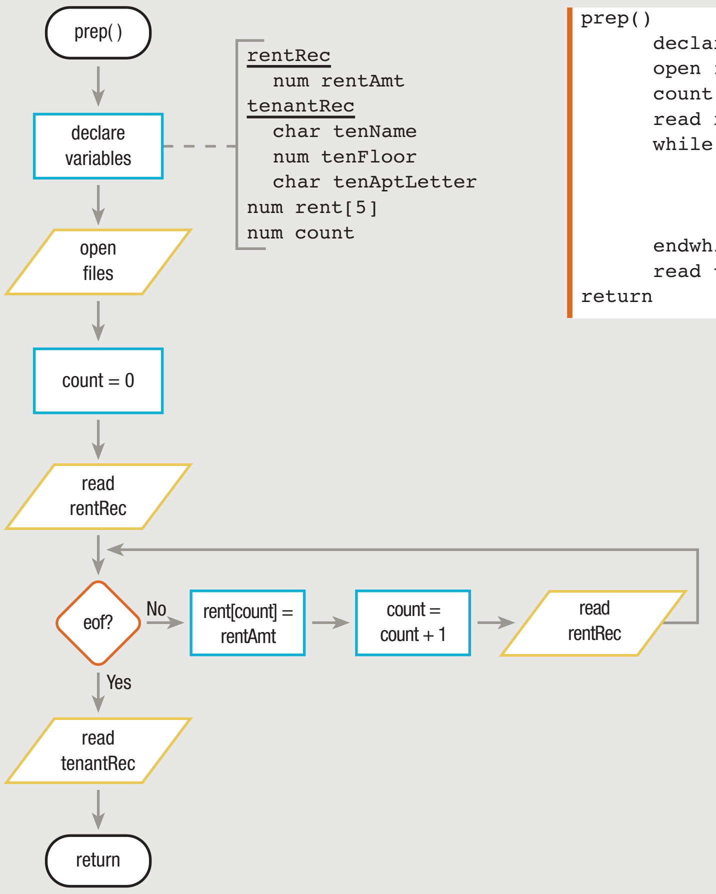
Without a `rent` array, the `figureRent()` module would have to contain four decisions and five different resulting actions. With the `rent` array, there are no decisions. Each tenant's rent is simply based on the `rent` element that corresponds to the `tenFloor` variable because the floor number indicates the positional value of the corresponding rent. Arrays can really lighten the workload required to write a program.

LOADING AN ARRAY FROM A FILE

Writing the rent program from the previous section requires you to set values for five `rent` array elements within the `prep()` module. If you write the rent program for a skyscraper, you may have to initialize 100 array elements. Additionally, when the building management changes the rent amounts, you must alter the array element values within the program to reflect the new rent charges. If the rent values change frequently, it is inconvenient to have hard-coded values in your program. Instead, you can write your program so that it loads the array rent amounts from a file. The array of rent values is an example of an array that gets its values during the execution of the program.

A file that contains all the rent amounts can be updated by apartment building management as frequently as needed. Suppose you periodically receive a file named RENTFILE that is created by the building management and always contains the current rent values. You can write the rent program so that it accepts all records from this input file within the `prep()` module. Figure 8-21 shows how this is accomplished.

FIGURE 8-21: FLOWCHART AND PSEUDOCODE FOR `prep()` MODULE THAT READS RENT VALUES FROM AN INPUT FILE



```
rentRec
   num rentAmt
tenantRec
   char tenName
   num tenFloor
   char tenAptLetter
num rent[5]
num count
```

```
prep()
        declare variables
        open files
        count = 0
        read rentRec
        while not eof
                  rent[count] = rentAmt
                  count = count + 1
                  read rentRec
        endwhile
        read tenantRec
return
```

In the `prep()` module in Figure 8-21, you set the variable `count` to 0 and read a `rentRec` record from RENTFILE. Each record in RENTFILE contains just one field—a numeric `rentAmt` value. For this program, assume that the rent records in RENTFILE are stored in order by floor. When you read the first `rentAmt`, you store it in the first element of the `rent` array. You increase the `count` to 1, read the second record, and, assuming it's not `eof`, you store the second rent in the second element of the `rent` array. After RENTFILE is exhausted and the `rent` array is filled with appropriate rent amounts for each floor, you begin to read the file containing the `tenantRec` records, and then the program proceeds as usual.

TIP You could choose to close RENTFILE at the end of the `prep()` module. Unlike the tenant file and the printer, it will not be used again in the program. Alternatively, you can wait and close all the files at the end of the program.

When you use this method—reading the rents from an input file instead of hard coding them into the program—clerical employees can update the `rentRec` values in RENTFILE. Your program takes care of loading the rents into the program array from the most recent copy of RENTFILE, ensuring that each rent is always accurate and up to date. Using this technique, you avoid the necessity of changing code within the program with each rent update.

TIP Another way to organize RENTFILE is to include two fields within each record—for example, `rentFloor` and `rentAmt`. Then, the records would not have to be read into your program in floor-number order. Instead, you could use the `rentFloor` variable as a subscript to indicate which position in the array to use to store the `rentAmt`.

TIP You might question how the program knows which file's `eof` condition is tested when a program uses two or more input files. In some programming languages, the `eof` condition is tested on the file most recently read. In many programming languages, you have to provide more specific information along with the `eof` question, perhaps `rentRec eof?` or `tenantRec eof?`

TIP The RENTFILE example assumes that management provides you with a file that contains no more records than the number of rents your program is prepared to hold. A more elegant program would check to make sure there are not too many rents. You will learn how to perform such checks later in this chapter.

SEARCHING FOR AN EXACT MATCH IN AN ARRAY

In both the apartment request program and the rent program that you've seen in this chapter, the fields that the arrays depend on conveniently hold small whole numbers. The number of bedrooms available in apartments are zero through three, and the floors of the building are zero through four. Unfortunately, real life doesn't always happen in small integers. Sometimes, you don't have a variable that conveniently holds an array position; sometimes, you have to search through an array to find a value you need.

Consider a mail-order business in which orders come in with a customer name, address, item number ordered, and quantity ordered, as shown in Figure 8-22.

FIGURE 8-22: MAIL-ORDER CUSTOMER FILE DESCRIPTION

```
MAIL-ORDER CUSTOMER FILE DESCRIPTION
File name: CUSTREC
FIELD DESCRIPTION      DATA TYPE    COMMENTS
Customer name          Character
Address                Character
Item number            Numeric      A 3-digit number
Quantity               Numeric      A value from 1 through 99
```

The item numbers are three-digit numbers, but perhaps they are not consecutive 000 through 999. Instead, over the years, items have been deleted and new items have been added. For example, there might no longer be an item with number 005 or 129. Sometimes, there might be a hundred-number gap or more between items.

For example, let's say that this season you are down to the items shown in Figure 8-23. When a customer orders an item, you want to determine whether the order is for a valid item number. You could use a series of six decisions to determine whether the ordered item is valid; in turn, you would compare whether each customer's item number is equal to one of the six allowed values. However, a superior approach is to create an array that holds the list of valid item numbers. Then, you can search through the array for an exact match to the ordered item. If you search through the entire array without finding a match for the item the customer ordered, you can print an error message, such as "No such item."

Suppose you create an array with the six elements shown in Figure 8-24. If a customer orders item 307, a clerical worker can tell whether it is valid by looking down the list and verifying that 307 is a member of the list. In a similar fashion, you can use a loop to test each `validItem` against the ordered item number.

FIGURE 8-23: AVAILABLE ITEMS IN MAIL-ORDER COMPANY

```
ITEM NUMBER
106
108
307
405
457
688
```

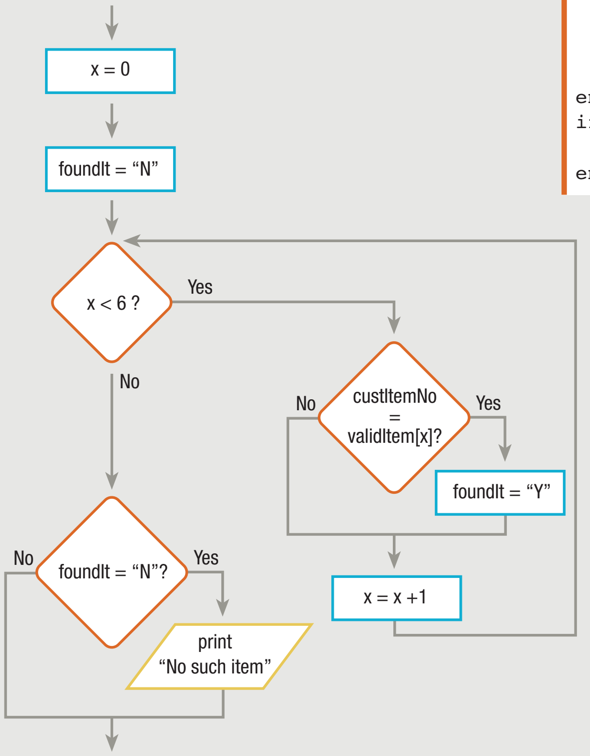
FIGURE 8-24: ARRAY OF VALID ITEM NUMBERS

```
num validItem[0] = 106
num validItem[1] = 108
num validItem[2] = 307
num validItem[3] = 405
num validItem[4] = 457
num validItem[5] = 688
```

The technique for verifying that an item number exists involves setting a subscript to 0 so that you can start searching from the first array element, and initializing a flag variable to indicate that you have not yet determined whether the customer's order is valid. A **flag** is a variable that you set to indicate whether some event has occurred; frequently, it holds a True or False value. For example, you can set a character variable named `foundIt` to "N", indicating "No". Then you compare the customer's ordered item number to the first item in the array. If the customer-ordered item matches the first item in the array, you can set the flag variable to "Y", or any other value that is not "N". If the items do not match, you increase the subscript and continue to look down the list of numbers stored in the array. If you check all six valid item numbers and the customer item matches none of them, then the flag variable `foundIt` still holds the value "N". If the flag variable is "N" after you have looked through the entire list, you can issue an error message indicating that no match was ever found. Assuming you declare the customer item as `custItemNo` and the subscript as `x`, then Figure 8-25 shows a flowchart segment and the pseudocode that accomplishes the item verification.

FIGURE 8-25: FLOWCHART AND PSEUDOCODE SEGMENTS FOR FINDING AN EXACT MATCH TO A CUSTOMER ITEM NUMBER

```
x = 0
foundIt = "N"
while x < 6
      if custItemNo = validItem[x] then
            foundIt = "Y"
      endif
      x = x + 1
endwhile
if foundIt = "N" then
     print "No such item"
endif
```



USING PARALLEL ARRAYS

In a mail-order company, when you read a customer’s order, you usually want to accomplish more than simply verifying that the item exists. You want to determine the price of the ordered item, multiply that price by the quantity ordered, and print a bill. Suppose you have prices for six available items, as shown in Figure 8-26.

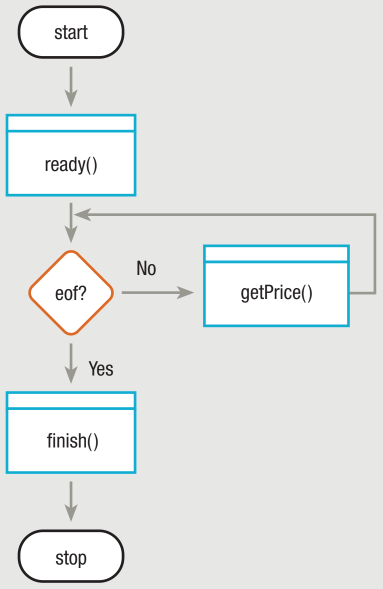
FIGURE 8-26: AVAILABLE ITEMS WITH PRICES FOR MAIL-ORDER COMPANY

ITEM NUMBER	ITEM PRICE
106	0.59
108	0.99
307	4.50
405	15.99
457	17.50
688	39.00

You *could* write a program in which you read a customer order record and then use the customer's item number as a subscript to pull a price from an array. To use this method, you need an array with at least 689 elements. If a customer orders item 405, the price is found at `validItem[custItemNo]`, which is `validItem[405]`, or the 406th element of the array (because the 0th element is the first element of the array). Such an array would need 689 elements (because the highest item number is 688), but because you sell only six items, you would waste 683 of the memory positions. Instead of reserving a large quantity of memory that remains unused, you can set up this program to use two arrays.

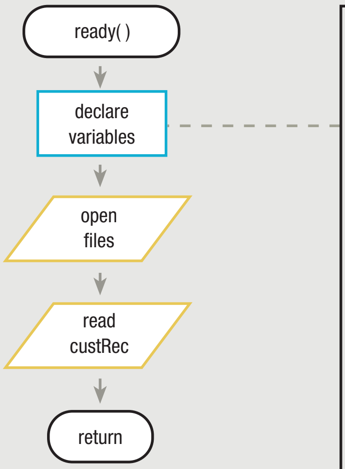
Consider the mainline logic in Figure 8-27 and the `ready()` module in Figure 8-28. Two arrays are set up within the `ready()` module. One contains six elements named `validItem`; all six elements are valid item numbers. The other array also has six elements. These are named `validItemPrice`; all six elements are prices. Each price in this `validItemPrice` array is conveniently and purposely in the same position as the corresponding item number in the other `validItem` array. Two corresponding arrays such as these are **parallel arrays** because each element in one array is associated with the element in the same relative position in the other array.

FIGURE 8-27: MAINLINE LOGIC FOR THE PRICE PROGRAM



```
start
     perform ready()
     while not eof
          perform getPrice()
     endwhile
     perform finish()
stop
```

FIGURE 8-28: THE `ready()` MODULE FOR THE PRICE PROGRAM

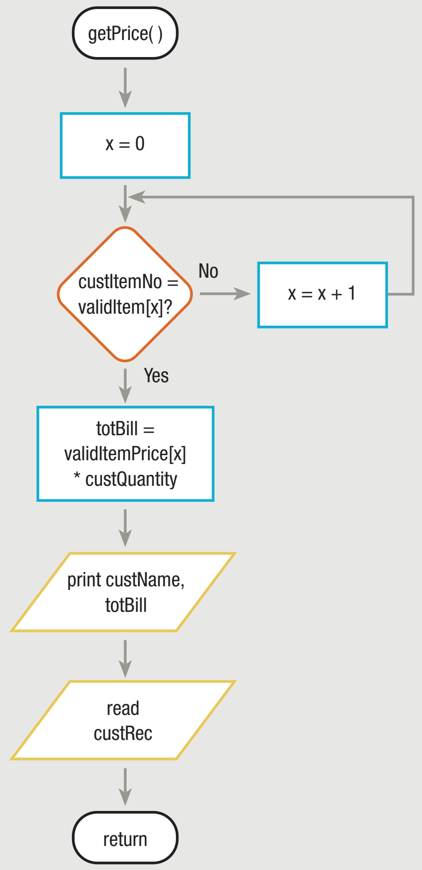


```
custRec
   char custName
   char custAddress
   num custItemNo
   num custQuantity

num validItem[0] = 106
num validItem[1] = 108
num validItem[2] = 307
num validItem[3] = 405
num validItem[4] = 457
num validItem[5] = 688

num validItemPrice[0] = 0.59
num validItemPrice[1] = 0.99
num validItemPrice[2] = 4.50
num validItemPrice[3] = 15.99
num validItemPrice[4] = 17.50
num validItemPrice[5] = 39.00

num totBill
num x
char foundIt
```

```
ready()
     declare variables
     open files
     read custRec
return
```

You can write the `getPrice()` module as shown in Figure 8-29. The general procedure is to read each item number, look through each of the `validItem` values separately, and when a match for the `custItemNo` variable on the input record is found, pull the corresponding parallel price out of the list of `validItemPrice` values.

FIGURE 8-29: THE `getPrice()` MODULE FOR THE PRICE PROGRAM



```
getPrice()
     x = 0
     while custItemNo not equal to validItem[x]
          x = x + 1
     endwhile
     totBill = validItemPrice[x] * custQuantity
     print custName, totBill
     read custRec
return
```

TIP

In this book, you have repeatedly seen the flowchart decision that asks the `eof` question phrased as a positive question ("`eof?`") so the program continues while the answer is No. You also have seen the pseudocode decision that asks the `eof` question in a negative form ("`while not eof`") so that the program continues while the condition is true. Figure 8-29 follows the same convention—the flowchart compares the customer item number to a valid item using a positive question so the loop continues while the answer is No, whereas the pseudocode asks if the customer item number is *not* equal to a valid item number, continuing while the answer is Yes. The logic is the same either way.

You must create a variable to use as a subscript for the arrays. If you name the subscript `x` (see the declaration of `x` in the variable list in Figure 8-28), then you can start by setting `x` equal to 0. Then, if `custItemNo` is the same as `validItem[x]`, you can use the corresponding price from the other table, `validItemPrice[x]`, to calculate the customer's bill.

TIP

Some programmers object to using a cryptic variable name such as x because it is not descriptive. These programmers would prefer a name such as priceIndex. Others approve of short names like x when the variable is used only in a limited area of a program, as it is used here, to step through an array. There are many style issues on which programmers disagree. As a programmer, it is your responsibility to find out what conventions are used among your peers in your organization.

Within the getPrice() module, the variable used as a subscript, x, is set to 0. If custItemNo is *not* the same as validItem[x], then add 1 to x. Because x now holds the value 1, you next compare the customer's requested item number to validItem[1]. The value of x keeps increasing, and eventually a match between custItemNo and some validItem[x] should be found.

After you find a match for the custItemNo variable in the validItem array, you know that the price of that item is in the same position in the other array, validItemPrice. When validItem[x] is the correct item, validItemPrice[x] must be the correct price.

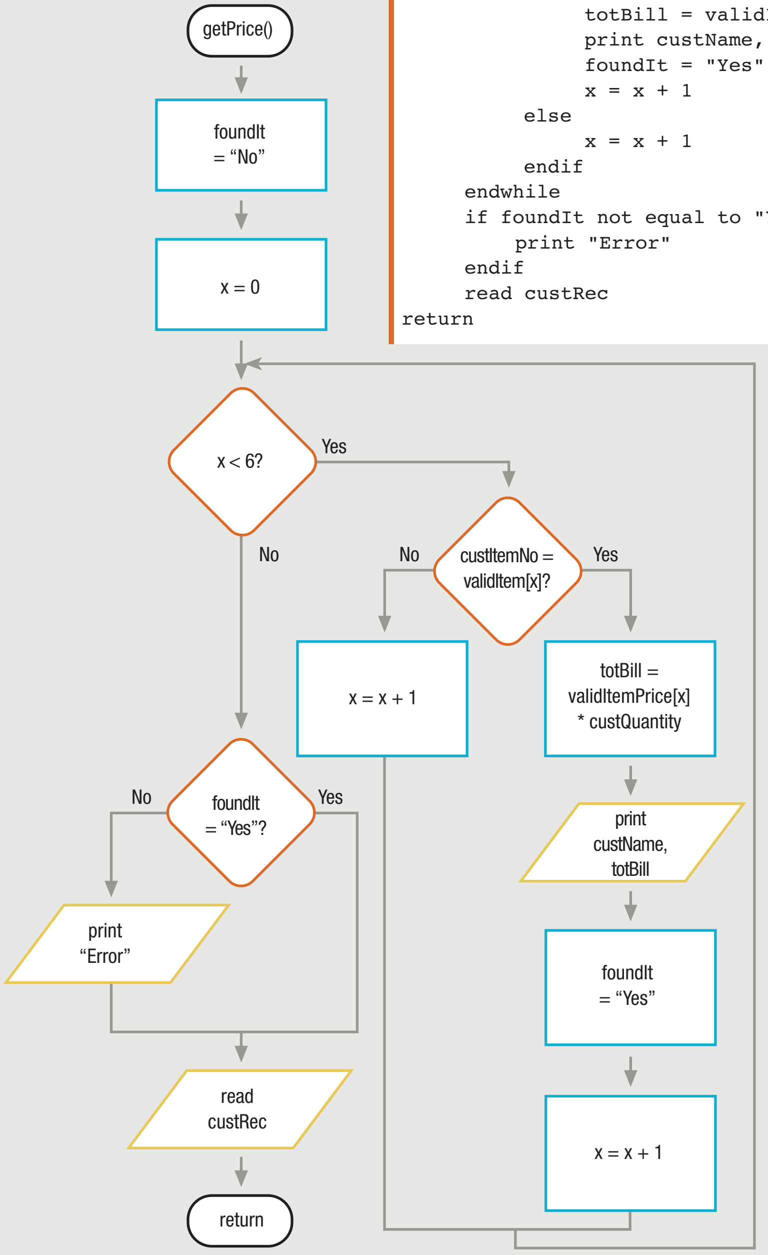
Suppose that a customer orders item 457, and walk through the flowchart yourself to see if you come up with the correct price.

REMAINING WITHIN ARRAY BOUNDS

The getPrice() module in Figure 8-29 is not perfect. The logic makes one dangerous assumption: that every customer will order a valid item number. If a customer is looking at an old catalog and orders item 107, the program will never find a match. The value of x will just continue to increase until it reaches a value higher than the number of elements in the array. At that point, one of two things happens. When you use a subscript value that is higher than the number of elements in an array, some programming languages stop execution of the program and issue an error message. Other programming languages do not issue an error message but continue to search through computer memory beyond the end of the array. Either way, the program doesn't end elegantly. When you use a subscript that is not within the range of acceptable subscripts, your subscript is said to be **out of bounds**. Ordering a wrong item number is a frequent customer error; a good program should be able to handle the mistake and not allow the subscript to go out of bounds.

You can improve the price-finding program by adding a flag variable and a test to the getPrice() module. You can set the flag when you find a valid item in the validItem array, and after searching the array, check whether the flag has been altered. See Figure 8-30.

FIGURE 8-30: THE `getPrice()` MODULE USING THE `foundIt` FLAG



```
getPrice()
     foundIt = "No"
     x = 0
     while x < 6
          if custItemNo = validItem[x] then
               totBill = validItemPrice[x] * custQuantity
               print custName, totBill
               foundIt = "Yes"
               x = x + 1
          else
               x = x + 1
          endif
     endwhile
     if foundIt not equal to "Yes" then
         print "Error"
     endif
     read custRec
return
```

In the `ready()` module, you can declare a variable named `foundIt` that acts as a flag. When you enter the `getPrice()` module, you can set `foundIt` equal to "No". Then, after setting `x` to 0, check to see if `x` is still less than 6. If it is, compare `custItemNo` to `validItem[x]`. If they are equal, you know the position of the item's price, and you can use the price to print the customer's bill and set the `foundIt` flag to "Yes". If `custItemNo` is not equal to `validItem[x]`, you increase `x` by 1 and continue to search through the array. When `x` is 6, you shouldn't look through the array anymore; you've gone through all six legitimate items, and you've reached the end. The legitimate subscripts for a six-element array are 0 through 5; your subscript variable should not be used with the array when it reaches 6. If `foundIt` doesn't have a "Yes" in it at this point, it means you never found a match for the ordered item number; you never took the Yes path leading from the `custItemNo = validItem[x]?` question. If `foundIt` does not have "Yes" stored in it, you should print an error message; the customer has ordered a nonexistent item.

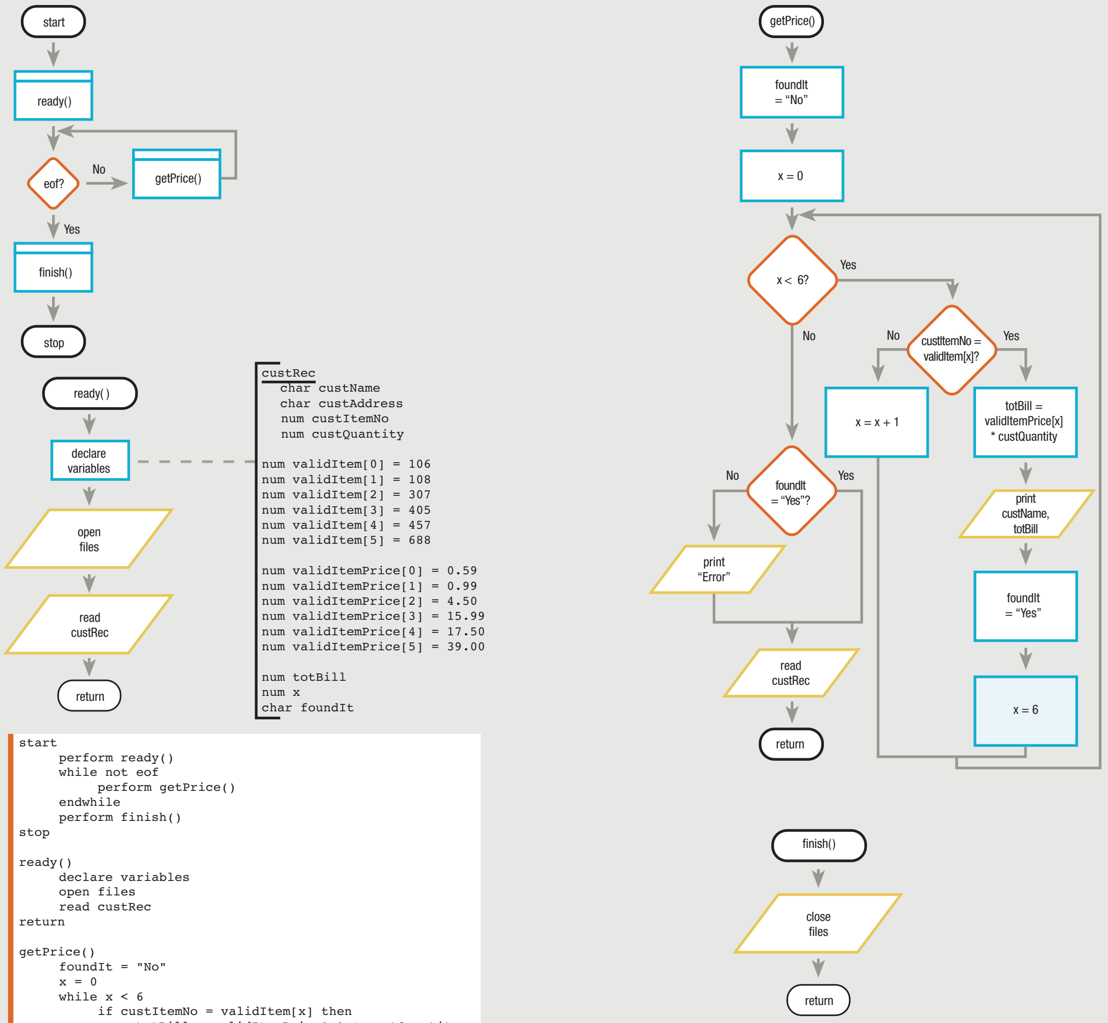
IMPROVING SEARCH EFFICIENCY USING AN EARLY EXIT

The mail-order program is still a little inefficient. The problem is that if lots of customers order item 106 or 108, their price is found on the first or second pass through the loop. The program continues searching through the item array, however, until `x` reaches the value 6. One way to stop the search once the item has been found, and `foundIt` is set to "Yes", is to set `x` to 6 immediately. (Setting a variable to a specific value, particularly when the new value is an abrupt change, is also called **forcing** the variable to that value.) Then, when the program loops back to check whether `x` is still less than 6, the loop will be exited and the program won't bother checking any of the higher item numbers. Leaving a loop as soon as a match is found is called an **early exit**; it improves the program's efficiency. The larger the array, the more beneficial it becomes to exit the searching loop as soon as you find what you're looking for.

TIP Some programmers prefer to use a flag variable for early exits; others think it is fine to force a loop control variable to a value that stops loop execution if that is more convenient.

Figure 8-31 shows the final version of the price program. Notice the improvement to the `getPrice()` module. You search the `validItem` array, element by element. If an item number is not matched in a given location, the subscript is increased and the next location is checked. As soon as an item number is located in the array, you print a line, turn on the flag, and force the subscript to a high number (6) so the program will not check the item number array any further.

FIGURE 8-31: THE FINAL VERSION OF THE PRICE PROGRAM THAT EFFICIENTLY SEARCHES FOR PRICES BASED ON THE ITEM A CUSTOMER ORDERS



```
custRec
   char custName
   char custAddress
   num custItemNo
   num custQuantity

num validItem[0] = 106
num validItem[1] = 108
num validItem[2] = 307
num validItem[3] = 405
num validItem[4] = 457
num validItem[5] = 688

num validItemPrice[0] = 0.59
num validItemPrice[1] = 0.99
num validItemPrice[2] = 4.50
num validItemPrice[3] = 15.99
num validItemPrice[4] = 17.50
num validItemPrice[5] = 39.00

num totBill
num x
char foundIt
```

```
start
     perform ready()
     while not eof
          perform getPrice()
     endwhile
     perform finish()
stop

ready()
     declare variables
     open files
     read custRec
return

getPrice()
     foundIt = "No"
     x = 0
     while x < 6
          if custItemNo = validItem[x] then
               totBill = validItemPrice[x] * custQuantity
               print custName, totBill
               foundIt = "Yes"
               x = 6
          else
               x = x + 1
          endif
     endwhile
     if foundIt not equal to "Yes" then
          print "Error"
     endif
     read custRec
return

finish()
     close files
return
```

TIP

Notice that the price program is most efficient when the most frequently ordered items are stored at the beginning of the array. When you use this technique, only the seldom-ordered items require many cycles through the searching loop before finding a match.

TIP

Remember that you can make programs that contain arrays more flexible by declaring a constant to hold the size of the array. Then, whenever you need to refer to the size of the array within the program—for example, when you loop through the array during a search operation—you can use the variable name instead of a hard-coded value like 6. If the program must be altered later to accommodate more or fewer array elements, you need to make only one change—you change the value of the array-size variable where it is declared.

SEARCHING AN ARRAY FOR A RANGE MATCH

In the previous example, customer item numbers needed to exactly match item numbers stored in a table to determine the correct price of an item. Sometimes, however, instead of finding exact matches, programmers want to work with ranges of values in arrays. A **range of values** is any set of contiguous values, such as 1 through 5.

Recall the customer file description from earlier in this chapter, shown again in Figure 8-32. Suppose the company decides to offer quantity discounts, as shown in Figure 8-33.

FIGURE 8-32: MAIL-ORDER CUSTOMER FILE DESCRIPTION

```
MAIL-ORDER CUSTOMER FILE DESCRIPTION
File name: CUSTREC
FIELD DESCRIPTION       DATA TYPE     COMMENTS
Customer name           Character
Address                 Character
Item number             Numeric       A 3-digit number
Quantity                Numeric       A value from 1 through 99
```

FIGURE 8-33: DISCOUNTS ON ORDERS BY QUANTITY

Number of items ordered	Discount %
1-9	0
10-24	10
25-48	15
49 or more	25

You want to be able to read a record and determine a discount percentage based on the value in the quantity field. One ill-advised approach might be to set up an array with as many elements as any customer might ever order, and store the appropriate discount for each possible number, as shown in Figure 8-34.

FIGURE 8-34: USABLE—BUT INEFFICIENT—DISCOUNT ARRAY

```
num discount[0] = 0
num discount[1] = 0
num discount[2] = 0
.
.
num discount[9] = 0
num discount[10] = 10
.
.
num discount[48] = 15
num discount[49] = 25
num discount[50] = 25
.
.
```

This approach has three drawbacks:

- It requires a very large array that uses a lot of memory.
- You must store the same value repeatedly. For example, each of the first 10 elements receives the same value, 0, because if a customer orders from zero through nine items, there is no discount. Similarly, each of the next 15 elements receives the same value, 10.
- Where do you stop adding array elements? Is a customer order quantity of 75 items enough? What if a customer orders 100 or 1,000 items? No matter how many elements you place in the array, there's always a chance that a customer will order more.

A better approach is to create just four discount array elements, one for each of the possible discount rates, as shown in Figure 8-35.

FIGURE 8-35: SUPERIOR DISCOUNT ARRAY

```
num discount[0] = 0
num discount[1] = 10
num discount[2] = 15
num discount[3] = 25
```

With the new four-element `discount` array, you need a parallel array to search through, to find the appropriate level for the discount. At first, beginning programmers might consider creating an array named `discountRange` and testing whether the quantity ordered equals one of the four stored values. For example:

```
num discountRange[0] = 0 through 9
num discountRange[1] = 10 through 24
num discountRange[2] = 25 through 48
num discountRange[3] = 49 and higher
```

However, you cannot create an array like the previous one. Each element in any array is simply a single variable. Any variable can hold a value such as 6 or 12, but it can't hold every value 6 *through* 12. Similarly, the `discountRange[0]` variable can hold a 1, 2, 9, or any other single value, but it can't hold 0 *through* 9; there is no such numeric value.

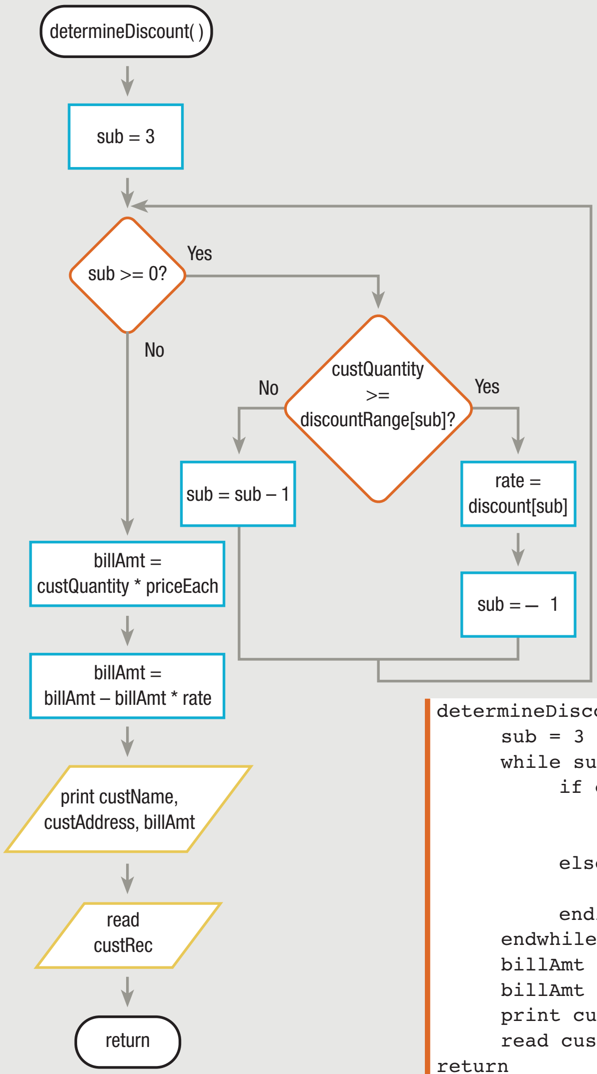
One solution is to create an array that holds only the low-end value of each range, as Figure 8-36 shows.

FIGURE 8-36: THE `discountRange` ARRAY USING LOW END OF EACH DISCOUNT RANGE

```
num discountRange[0] = 0
num discountRange[1] = 10
num discountRange[2] = 25
num discountRange[3] = 49
```

Using such an array, you can compare each `custQuantity` value with each `discountRange` value in turn. You can start with the *last* range limit (`discountRange[3]`). If `custQuantity` is at least that value, 49, the customer gets the highest discount rate (`discount[3]`). If `custQuantity` is not at least `discountRange[3]`, then you check to see if it is at least `discountRange[2]`, or 25. If so, the customer receives `discount[2]`, and so on. If you declare a variable named `rate` to hold the correct discount rate, and another variable named `sub` to use as a subscript, then you can use the `determineDiscount()` module shown in Figure 8-37. This module uses a loop to find the appropriate discount rate for an order, then calculates and prints a customer bill.

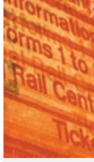
FIGURE 8-37: FLOWCHART AND PSEUDOCODE FOR DISCOUNT DETERMINATION


determineDiscount( )
sub = 3
sub >= 0?
Yes
No
custQuantity >= discountRange[sub]?
No
Yes
sub = sub – 1
rate = discount[sub]
sub = – 1
billAmt = custQuantity * priceEach
billAmt = billAmt – billAmt * rate
print custName, custAddress, billAmt
read custRec
return

```
determineDiscount()
      sub = 3
      while sub >= 0
           if custQuantity >= discountRange[sub]
                rate = discount[sub]
                sub = – 1
           else
                sub = sub – 1
           endif
      endwhile
      billAmt = custQuantity * priceEach
      billAmt = billAmt – billAmt * rate
      print custName, custAddress, billAmt
      read custRec
return
```

TIP An alternative approach is to store the high end of every range in an array. Then, you start with the *lowest* element and check for values *less than or equal to* each array element value before using the appropriate discount in the parallel array.

When using an array to store range limits, you use a loop to make a series of comparisons that would otherwise require many separate decisions. Your program is written using fewer instructions than would be required if you did not use an array, and modifications to your program will be easier to make in the future.



CHAPTER SUMMARY

- An array is a series or list of variables in computer memory, all of which have the same name but are differentiated with special numbers called subscripts.
- When you declare an array, you declare a programming structure that contains multiple elements, each of which has the same name and the same data type. Each array element has a unique integer subscript indicating how far away the individual element is from the first element.
- You often can use a variable as a subscript to an array, replacing multiple nested decisions.
- You can declare and initialize all of the elements in an array using a single statement that provides a type, a name, and a quantity of elements for the array. You also can initialize array values within an initialization loop.
- You can use a constant array when the final desired values are fixed at the beginning of the program.
- You can load an array from a file. This step is often performed in a program's housekeeping module.
- Searching through an array to find a value you need involves initializing a subscript, using a loop to test each array element, and setting a flag when a match is found.
- In parallel arrays, each element in one array is associated with the element in the same relative position in the other array.
- Your programs should ensure that subscript values do not go out of bounds—that is, take on a value out of the range of legal subscripts.
- When you need to compare a value to a range of values in an array, you can store either the low- or high-end value of each range for comparison.

KEY TERMS

An **array** is a series or list of variables in computer memory, all of which have the same name but are differentiated with special numbers called subscripts.

A **subscript** is a number that indicates the position of a particular item within an array.

An **index** is a subscript.

Each separate array variable is one **element** of the array.

The **size of an array** is the number of elements it can hold.

In a **zero-based array**, the first element is accessed using a subscript of 0.

Off-by-one errors usually occur when you assume an array's first subscript is 1 but it actually is 0.

An **initialization loop** is a loop structure that provides initial values for every element in any array.

Hard-coded values are explicitly assigned.

A **flag** is a variable that you set to indicate whether some event has occurred.

Parallel arrays are two or more arrays in which each element in one array is associated with the element in the same relative position in the other array or arrays.

When you use a subscript that is not within the range of acceptable subscripts, your subscript is said to be **out of bounds**.

Forcing a variable to a value is assigning a specific value to it, particularly when the assignment causes a sudden change in value.

Leaving a loop as soon as a match is found is called an **early exit**.

A **range of values** is any set of contiguous values.

REVIEW QUESTIONS

1. **A subscript is a(n) ________.**
   a. element in an array
   b. alternate name for an array
   c. number that indicates the position of a particular item within an array
   d. number that represents the highest value stored within an array

2. **Each variable in an array must have the same ________ as the others.**
   a. subscript
   b. data type
   c. value
   d. memory location

3. **Each variable in an array is called a(n) ________.**
   a. element
   b. subscript
   c. component
   d. data type

4. **The subscripts of any array are always ________.**
   a. characters
   b. fractions
   c. integers
   d. strings of characters

5. **Suppose you have an array named `number`, and two of its elements are `number[1]` and `number[4]`. You know that ________.**
   a. the two elements hold the same value
   b. the two elements are at the same memory location
   c. the array holds exactly four elements
   d. there are exactly two elements between those two elements

6. **Suppose you want to write a program that reads customer records and prints a summary of the number of customers who owe more than $1,000 each, in each of 12 sales regions. Customer fields include `name`, `zipCode`, `balanceDue`, and `regionNumber`. At some point during record processing, you would add 1 to an array element whose subscript would be represented by __________.**
   a. `name`
   b. `zipCode`
   c. `balanceDue`
   d. `regionNumber`

7. **Arrays are most useful when you use a __________ as a subscript.**
   a. numeric constant
   b. character
   c. variable
   d. file name

8. **Suppose you create a program with a seven-element array that contains the names of the days of the week. In the `housekeeping()` module, you display the day names using a subscript named `dayNum`. In the same program, you display the same array values again in the `finish()` module. In the `finish()` module, you __________ as a subscript to the array.**
   a. must use `dayNum`
   b. can use `dayNum`, but can also use another variable
   c. must not use `dayNum`
   d. must use a numeric constant

9. **Declaring a numeric array sets its individual elements' values to __________.**
   a. zero in every programming language
   b. zero in some programming languages
   c. consecutive digits in every programming language
   d. consecutive digits in some programming languages

10. **A __________ array is one in which the stored values are fixed permanently at the start of the program.**
    a. constant
    b. variable
    c. persistent
    d. continual

11. **When you create an array of values that you explicitly set upon creation, using numeric constants, the values are said to be __________.**
    a. postcoded
    b. precoded
    c. soft coded
    d. hard coded

12. **Many arrays contain values that change periodically. For example, a bank program that uses an array containing mortgage rates for various terms might change several times a day. The newest values are most likely __________.**

    a. typed into the program by a programmer who then recompiles the program before it is used
    b. calculated by the program, based on historical trends
    c. read into the program from a file that contains the current rates
    d. typed in by a clerk each time the program is executed for a customer

13. **A __________ is a variable that you set to indicate a True or False state.**

    a. subscript
    b. flag
    c. counter
    d. banner

14. **Two arrays in which each element in one array is associated with the element in the same relative position in the other array are __________ arrays.**

    a. cohesive
    b. perpendicular
    c. hidden
    d. parallel

15. **In most programming languages, the subscript used to access the last element in an array declared as `num values[12]` is __________.**

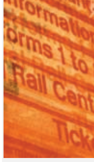
    a. 0
    b. 11
    c. 12
    d. 13

16. **In most programming languages, a subscript for a 10-element array is out of bounds when it __________.**

    a. is lower than 0
    b. is higher than 9
    c. both of these
    d. neither a nor b

17. **If you perform an early exit from a loop while searching through an array for a match, you __________.**

    a. quit searching as soon as you find a match
    b. quit searching before you find a match
    c. set a flag as soon as you find a match, but keep searching for additional matches
    d. repeat a search only if the first search was unsuccessful



18. In programming terminology, the values 4 through 20 represent a(n) __________ of values.

a. assortment
b. range
c. diversity
d. collection

19. Each element in a five-element array can hold __________ value(s).

a. one
b. five
c. at least five
d. an unlimited number of

20. After the annual dog show in which the Barkley Dog Training Academy awards points to each participant, the Academy assigns a status to each dog based on the following criteria:

Points Earned	Level of Achievement
0–5	Good
6–7	Excellent
8–9	Superior
10	Unbelievable

The Academy needs a program that compares a dog's points earned with the grading scale, in order to award a certificate acknowledging the appropriate level of achievement. Of the following, which set of values would be most useful for the contents of an array used in the program?

a. 0, 6, 9, 10
b. 5, 7, 8, 10
c. 5, 7, 9, 10
d. any of these

FIND THE BUGS

Each of the following pseudocode segments contains one or more bugs that you must find and correct.

1. This application prints a summary report for an aluminum can recycling drive at a high school. When a student brings in cans, a record is created that contains two fields—the student's year in school (1, 2, 3, or 4) and the number of cans submitted. Student records have not been sorted. The report lists each of the four classes and the total number of cans recycled for each class.

```
start
     perform housekeeping()
     while not eof
          perform accumulateCans()
     endwhile
     perform finish()
stop

housekepping()
     declare variables
         studentRec
             num year
             num cans
         char heading1 = "Can Recycling Report"
         char heading2 = "Year         Cans"
         const num SIZE = 4
         num collected[SIZE] all set to 0
     open files
     read studentRec
return

accumulateCans()
     if year < 1 OR year >= SIZE then
           year = 0
     endif
     collected[SIZE] = collected[SIZE] + cans
     read studentRec
return


finish()
     print heading1
     print heading2
     year = 1
     while year < SIZE
          print year, collected[SIZE]
          year = year + 1
     endwhile
     close files
return
```

2. This application prints a report card for each student at Pedagogic College. A record has been created for each student containing the student's name, address, and zip code, as well as a numeric average (from 0 through 100) for all the student's work for the semester. A report card is printed for each student containing the student's name, address, city, state, and zip code, as well as a letter grade based on the following scale:

   90–100 A
   80–89 B
   70–79 C
   60–69 D
   59 and below F

   The student's city and state are determined from the student's zip code. A file is read containing three fields—zip code, city, and state—for each of the 100 zip codes the college serves. For this program, assume that all the student averages have been verified to be between 0 and 100 inclusive and that all the zip codes have been verified as valid and stored in the zip code file.

```
start
     perform housekeeping()
     while not eof
          perform produceGradeReport()
     endwhile
     perform finish()
stop

housekepping()
     declare variables
         studentRec
             char name
             char address
             num zipCode
             num average
         zipRec
             num zip
             char city
             char state
          const num ZIPSIZE = 100
          num storedZip[ZIPSIZE]
          char storedCity[ZIPSIZE]
          char storedState[SIZE]

          const num GRADESIZE = 5
          const num gradeLevel[1] = 80
          const num gradeLevel[2] = 70
          const num gradeLevel[3] = 60
          const num gradeLevel[4] = 0

          const char grade[0] = 'A'
          const char grade[1] = 'B'
          const char grade[2] = 'C'
          const char grade[3] = 'S'
          const char grade[4] = 'F'

         num zipCodeCount
         char zipFound
         num sub
     open files
     zipCodeCount = 0
     read zipRec
     while not eof
          zip = storedZip[x]
          storedCity[x] = city
          storedState[x] = state
          zipCodeCount = zipCodeCount + 1
         read zipRec
     endwhile
     read studentRec
return
```

```
produceGradeReport()
     print "Grade Report"
     print name
     print address
     zipFound = "N"
     sub = 0
     while zipFound = "N"
          if zipCode = storedZip[ZIPCODESIZE]
               print storedCity[ZIPCODESIZE]
               print storedState[ZIPCODESIZE]
               print zipCode
               zipFound = "Y"
          endif
          sub = sub + 1
      endwhile
      sub = 0
      while sub < GRADESIZE
          if average >= gradeLevel[sub] then
              print grade[sub]
              sub = 0
          endif
      endwhile
      read studentRec
return

finish()
     close files
return
```

EXERCISES

1. **The city of Cary is holding a special census. The census takers collect one record for each citizen, as follows:**

```
CENSUS FILE DESCRIPTION
File name: CENSUS
Not sorted
FIELD DESCRIPTION         DATA TYPE       EXAMPLE
Age                       Numeric         42
Gender                    Character       F
Marital Status            Character       M
Voting District           Numeric         18
```

The voting district field contains a number from 1 through 22.

Design the logic of a program that would produce a count of the number of citizens residing in each of the 22 voting districts.

a. Design the output for this program; create either sample output or a print chart.
b. Create the hierarchy chart.
c. Draw the flowchart.
d. Write the pseudocode.

2. **The Midville Park District maintains records containing information about players on its soccer teams. Each record contains a player's first name, last name, and team number. The teams are:**

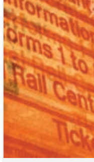
```
Soccer Teams
TEAM NUMBER          TEAM NAME
1                    Goal Getters
2                    The Force
3                    Top Guns
4                    Shooting Stars
5                    Midfield Monsters
```

Design the logic for a report that lists all players along with their team numbers and team names.

a. Design the output for this program; create either sample output or a print chart.
b. Create the hierarchy chart.
c. Draw the flowchart.
d. Write the pseudocode.

3. **Create the logic for a program that produces a count of the number of players registered for each team listed in Exercise 2.**

a. Design the output for this program; create either sample output or a print chart.
b. Create the hierarchy chart.
c. Draw the flowchart.
d. Write the pseudocode.



4. **An elementary school contains 30 classrooms numbered 1 through 30. Each classroom can contain any number of students up to 35. Each student takes an achievement test at the end of the school year and receives a score from 0 through 100. One record is created for each student in the school; each record contains a student ID, classroom number, and score on the achievement test. Design the logic for a program that lists the total points scored for each of the 30 classroom groups.**

a. Design the output for this program; create either sample output or a print chart.
b. Create the hierarchy chart.
c. Draw the flowchart.
d. Write the pseudocode.

5. **Modify Exercise 4 so that each classroom's average of the test scores prints, rather than each classroom's total.**

6. **The school in Exercises 4 and 5 maintains a file containing the teacher's name for each classroom. Each record in this file contains a room number from 1 through 30, and the last name of the teacher. Modify Exercise 5 so that the correct teacher's name appears on the list with his or her class's average.**

7. **A fast-food restaurant sells the following products:**

```
Fast-Food Items
PRODUCT             PRICE
Cheeseburger        2.49
Pepsi               1.00
Chips                .59
```

Design the logic for a program that reads a record containing a customer number and item name, and then prints either the correct price or the message "Sorry, we do not carry that" as output.

a. Draw the flowchart.
b. Write the pseudocode.

8. **Each week, the home office for a fast-food restaurant franchise distributes a file containing new prices for the items it carries. The file contains the item name and current price. Design the logic for a program that loads the current values into arrays. Then, the program reads a record containing a customer number and item name, and prints either the correct price or the message "Sorry, we do not carry that" as output.**

a. Draw the flowchart.
b. Write the pseudocode.

9. **The city of Redgranite is holding a special census. The census takers collect one record for each citizen as follows:**

CENSUS FILE DESCRIPTION
File name: CENSUS
Not sorted

FIELD DESCRIPTION	DATA TYPE	EXAMPLE
Age	Numeric	42
Gender	Character	F
Marital Status	Character	M
Voting District	Numeric	18

Design the logic of a program that produces a count of the number of citizens in each of the following age groups: under 18, 18 through 30, 31 through 45, 46 through 64, and 65 and older.

a. Design the output for this program; create either sample output or a print chart.
b. Create the hierarchy chart.
c. Draw the flowchart.
d. Write the pseudocode.

10. **A company desires a breakdown of payroll by department. Input records are as follows:**

PAYROLL FILE DESCRIPTION
File name: PAY

FIELD DESCRIPTION	DATA TYPE	EXAMPLE
Employee Last Name	Character	Dykeman
Employee First Name	Character	Ellen
Department	Numeric	3
Hourly Salary	Numeric	18.50
Hours Worked	Numeric	40

Input records are organized in alphabetical order by employee, *not* in department number order.

The output is a list of the seven departments in the company (numbered 1 through 7) and the total gross payroll (rate times hours) for each department.

a. Design the output for this program; create either sample output or a print chart.
b. Create the hierarchy chart.
c. Draw the flowchart.
d. Write the pseudocode.

11. **Modify Exercise 10 so that the report lists department names as well as numbers. The department names are:**

Department Names and Numbers

DEPARTMENT NUMBER	DEPARTMENT NAME
1	Personnel
2	Marketing
3	Manufacturing
4	Computer Services
5	Sales
6	Accounting
7	Shipping

12. **Modify the report created in Exercise 11 so that it prints a line of information for each employee before printing the department summary at the end of the report. Each detail line must contain the employee's name, department number, department name, hourly wage, hours worked, gross pay, and withholding tax.**

Withholding taxes are based on the following percentages of gross pay:

Withholding Taxes

WEEKLY SALARY	WITHHOLDING %
0.00–200.00	10
200.01–350.00	14
350.01–500.00	18
500.01–up	22

13. **The Perfect Party Catering Company keeps records concerning the events it caters as follows:**

EVENT FILE DESCRIPTION

File name: CATER

FIELD DESCRIPTION	DATA TYPE	EXAMPLE
Event Number	Numeric	15621
Host Name	Character	Profeta
Month	Numeric	10
Day	Numeric	15
Year	Numeric	2007
Meal Selection	Numeric	4
Number of Guests	Numeric	150

Additionally, a meal file contains the meal selection codes (such as 4), name of entree (such as "Roast beef"), and current price per guest (such as 19.50). Assume there are eight numbered meal records in the file.

Design the logic for a program that produces a report that lists each event number, host name, date, meal, guests, gross total price for the party, and price for the party after discount. Print the month *name*—for example, "October"—rather than "10". Print the meal selection—for example, "Roast beef"—rather than "4". The gross total price for the party is the price per guest for the meal times the number of guests. The final price includes a discount based on the following table:

Discounts for Large Parties

NUMBER OF GUESTS	DISCOUNT
1–25	$0
26–50	$75
51–100	$125
101–250	$200
251 and over	$300

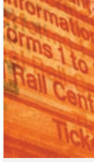
a. Design the output for this program; create either sample output or a print chart.
b. Create the hierarchy chart.
c. Draw the flowchart.
d. Write the pseudocode.

14. ***Daily Life Magazine* wants an analysis of the demographic characteristics of its readers. The Marketing Department has collected reader survey records in the following format:**

Magazine Reader FILE DESCRIPTION
File name: MAGREADERS
Not sorted

FIELD DESCRIPTION	DATA TYPE	EXAMPLE
Age	Numeric	31
Gender	Character	M
Marital Status	Character	S
Annual Income	Numeric	45000

a. Create the logic for a program that would produce a count of readers by age groups as follows: under 20, 20–29, 30–39, 40–49, and 50 and older.
b. Create the logic for a program that would produce a count of readers by gender within age group—that is, under 20 females, under 20 males, under 30 females, under 30 males, and so on.
c. Create the logic for a program that would produce a count of readers by income groups as follows: under $20,000, $20,000–$24,999, $25,000–$34,999, $35,000–$49,999, and $50,000 and up.



15. Glen Ross Vacation Property Sales employs seven salespeople as follows:

Salespeople

ID NUMBER	NAME
103	Darwin
104	Kratz
201	Shulstad
319	Fortune
367	Wickert
388	Miller
435	Vick

When a salesperson makes a sale, a record is created including the date, time, and dollar amount of the sale, as follows: The time is expressed in hours and minutes, based on a 24-hour clock. The sale amount is expressed in whole dollars.

SALE FIELD DESCRIPTION
File name: SALES

FIELD DESCRIPTION	DATA TYPE	EXAMPLE
Salesperson	Numeric	319
Month	Numeric	02
Day	Numeric	21
Year	Numeric	2008
Time	Numeric	1315
Sale Amount	Numeric	95900

Salespeople earn a commission that differs for each sale, based on the following rate schedule:

Commission Rates

SALE AMOUNT	RATE
$0–$50,000	.04
$50,001–$125,000	.05
$125,001–$200,000	.06
$200,001 and up	.07

Design the output and either the flowchart or pseudocode that produces each of the following reports:

a. A report listing each salesperson number, name, total sales, and total commissions
b. A report listing each month of the year as both a number and a word (for example, "01 January"), and the total sales for the month for all salespeople
c. A report listing total sales as well as total commissions earned by all salespeople for each of the following time frames, based on hour of the day: 00–05, 06–12, 13–18, and 19–23

DETECTIVE WORK

1. **Find at least five definitions of an array.**
2. **Using Help in Microsoft Excel or another spreadsheet program, discover how to use the `vlookup()` function. How is this function used?**
3. **What is a Fibonacci sequence? How do Fibonacci sequences apply to natural phenomena? Why do programmers use an array when working with this mathematical concept?**

UP FOR DISCUSSION

1. **A train schedule is an everyday, real-life example of an array. Think of at least four more.**
2. **Every element in an array always has the same data type. Why is this necessary?**