

# Chapter 6

## Arrays

- [Chapter Introduction](#)
- 6-1 [Storing Data in Arrays](#)
  - 6-1a [How Arrays Occupy Computer Memory](#)
- 6-2 [How an Array Can Replace Nested Decisions](#)
- 6-3 [Using Constants with Arrays](#)
  - 6-3a [Using a Constant as the Size of an Array](#)
  - 6-3b [Using Constants as Array Element Values](#)
  - 6-3c [Using a Constant as an Array Subscript](#)
- 6-4 [Searching an Array for an Exact Match](#)
- 6-5 [Using Parallel Arrays](#)
  - 6-5a [Improving Search Efficiency](#)
- 6-6 [Searching an Array for a Range Match](#)
- 6-7 [Remaining within Array Bounds](#)
- 6-8 [Using a `for` Loop to Process Arrays](#)
- 6-9 [Chapter Review](#)
  - 6-9a [Chapter Summary](#)
  - 6-9b [Key Terms](#)
  - 6-9c [Review Questions](#)
  - 6-9d [Exercises](#)

## Chapter Introduction

In this chapter, you will learn about:

- Storing data in arrays
- How an array can replace nested decisions
- Using constants with arrays
- Searching an array for an exact match
- Using parallel arrays
- Searching an array for a range match
- Remaining within array bounds
- Using a for loop to process arrays

Chapter 6: Arrays: 6-1 Storing Data in Arrays  
Book Title: Programming Logic and Design  
Printed By: Ronald Suchy (rsuchy@mayfieldschools.org)  
© 2013 ,

## 6-1 Storing Data in Arrays

An **array** (A series or list of variables in computer memory, all of which have the same name but are differentiated with subscripts.) is a series or list of values in computer memory. Usually, all the values in an array have something in common; for example, they might represent a list of employee ID numbers or prices for items sold in a store.

Whenever you require multiple storage locations for objects, you can use a real-life counterpart of a programming array. 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 keep receipts in a stack of shoe boxes and label each box with a month, you are also using the equivalent of an array. Similarly, when you plan courses for the next semester at your school by looking down a list of course offerings, you are using an array.



The arrays discussed in this chapter are single-dimensional arrays, which are similar to lists. Arrays with multiple dimensions are covered in [Chapter 8](#) of the Comprehensive version of this book.

Each of these real-life arrays helps you organize objects or information. You *could* store all

your papers or receipts in one huge cardboard box, or find courses if they 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 will accomplish the same results for your data.

Chapter 6: Arrays: 6-1a How Arrays Occupy Computer Memory  
Book Title: Programming Logic and Design  
Printed By: Ronald Suchy (rsuchy@mayfieldschools.org)  
© 2013 ,

## 6-1a How Arrays Occupy Computer Memory

When you declare an array, you declare a structure that contains multiple data items; each data item is one **element** (A separate array variable.) of the array. Each element has the same data type, and each element occupies an area in memory next to, or contiguous to, the others. You can indicate the number of elements an array will hold—the **size of the array** (The number of elements an array can hold.)—when you declare the array along with your other variables and constants. For example, you might declare an uninitialized, three-element numeric array named `someVals` as follows:

```
num someVals[3]
```

Each array element is differentiated from the others with a unique **subscript** (A number that indicates the position of an element within an array.), also called an **index** (A list of key fields paired with the storage address for the corresponding data record.), which is a number that indicates the position of a particular item within an array. 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. For example, a five-element array uses subscripts 0 through 4, and a ten-element array uses subscripts 0 through 9. In all languages, subscript values must be sequential integers (whole numbers). In most modern languages, such as Visual Basic, Java, C++, and C#, the first array element is accessed using subscript 0, and this book follows that convention.

To use an array element, you place its subscript within parentheses or square brackets (depending on the programming language) after the group name. This book will use square brackets to hold array subscripts so that you don't mistake array names for method names. Many newer programming languages such as C++, Java, and C# also use the bracket notation.

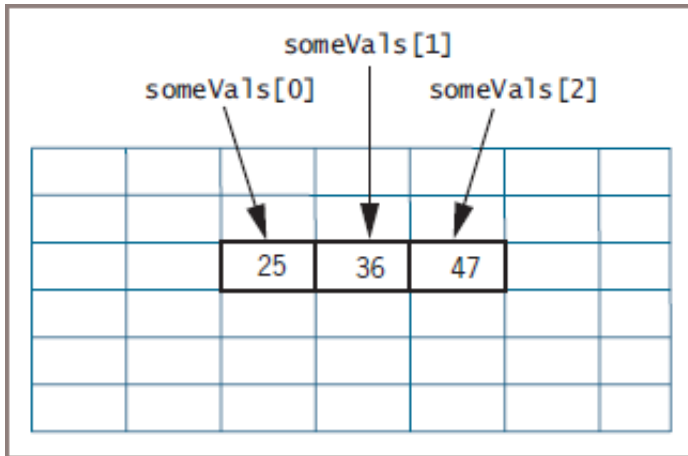
After you declare an array, you can assign values to some or all of the elements individually. Providing array values sometimes is called **populating an array** (populating an array is the act of assigning values to the array elements.). The following code shows a three-element array declaration, followed by three separate statements that populate the array:

```
Declarations
    num someVals[3]
someVals[0] = 25
someVals[1] = 36
someVals[2] = 47
```

Figure 6-1 shows an array named `someVals` that contains three elements, so the elements are `someVals[0]`, `someVals[1]`, and `someVals[2]`. The array elements have been assigned the values 25, 36, and 47, respectively. The element `someVals[0]` is zero numbers away from the beginning of the array. The element `someVals[1]` is one number away from the beginning of the array and `someVals[2]` is two numbers away.

**Figure 6-1**

### Appearance of a Three-Element Array in Computer Memory



If appropriate, you can declare and initialize array elements in one statement. Most programming languages use a statement similar to the following to declare a three-element array and assign values to it:

```
num someVals[3] = 25, 36, 47
```

When you use a list of values to initialize an array, the first value you list is assigned to the first array element (element 0), and the subsequent values are assigned to the remaining elements in order. Many programming languages allow you to initialize an array with fewer starting values than there are array elements declared, but no language allows you to initialize an array using more starting values than positions available. When starting values are supplied for an array in this book, each element will be provided with a value.

After an array has been declared and appropriate values have been assigned to specific elements, you can use an individual element in the same way you would use any other data item of the same type. For example, you can input values to array elements and you can output the values, and if the elements are numeric, you can perform arithmetic with them.



Watch the video *Understanding Arrays*.

Two Truths & A Lie

**Storing Data in Arrays**

1. In an array, each element has the same data type.

T F

2. Each array element is accessed using a subscript, which can be a number or a string.

T F

3. Array elements always occupy adjacent memory locations.

T F

Chapter 6: Arrays: 6-2 How an Array Can Replace Nested Decisions

Book Title: Programming Logic and Design

Printed By: Ronald Suchy (rsuchy@mayfieldschools.org)

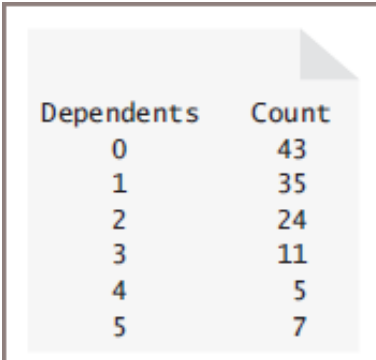
© 2013 ,

## 6-2 How an Array Can Replace Nested Decisions

Consider an application requested by a company's human resources department to produce statistics on employees' claimed dependents. The department wants a report that lists the number of employees who have claimed 0, 1, 2, 3, 4, or 5 dependents. (Assume that you know that no employees have more than five dependents.) For example, [Figure 6-2](#) shows a typical report.

**Figure 6-2**

### Typical Dependents Report

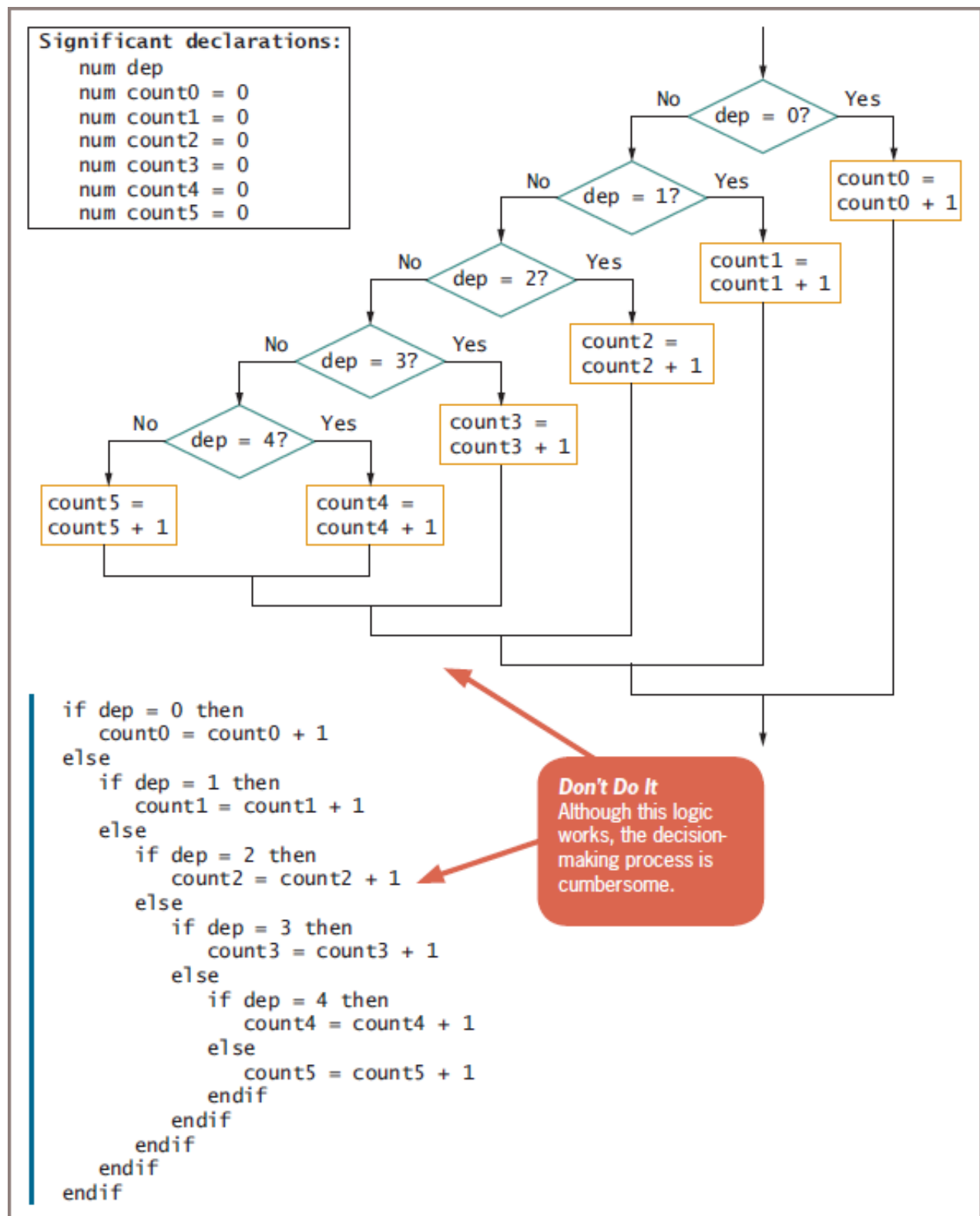


Dependents	Count
0	43
1	35
2	24
3	11
4	5
5	7

Without using an array, you could write the application that produces counts for the six categories of dependents (0 through 5) by using a series of decisions. [Figure 6-3](#) shows the pseudocode and flowchart for the decision-making part of such an application. Although this logic works, its length and complexity are unnecessary once you understand how to use an array.

**Figure 6-3**

## Flowchart and Pseudocode of Decision-making Process Using a Series of Decisions—the Hard Way





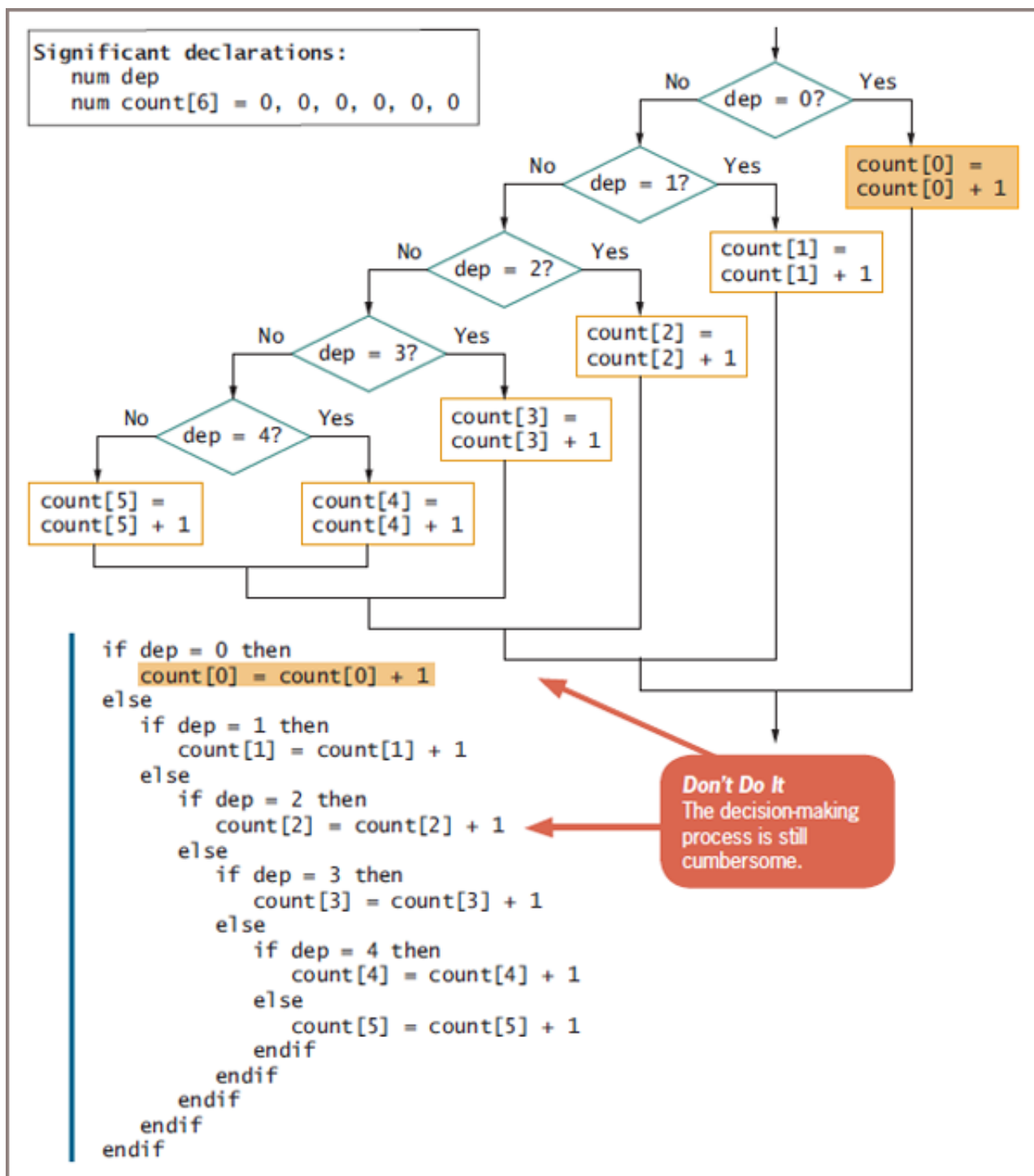
The decision-making process in [Figure 6-3](#) accomplishes its purpose, and the logic is correct, but the process is cumbersome and certainly not recommended. Follow the logic here so that you understand how the application works. In the next pages, you will see how to make the application more elegant.

In [Figure 6-3](#), the variable `dep` is compared to 0. If it is 0, 1 is added to `count0`. If it is not 0, then `dep` is compared to 1. Based on the result, 1 is added to `count1` or `dep` is compared to 2, and so on. Each time the application executes this decision-making process, 1 ultimately is added to one of the six variables that acts as a counter. The dependent-counting logic in [Figure 6-3](#) works, but even with only six categories of dependents, the decision-making process is unwieldy. What if the number of dependents might be any value from 0 to 10, or 0 to 20? With either of these scenarios, the basic logic of the program would remain the same; however, you would need to declare many additional variables to hold the counts, and you would need many additional decisions.

Using an array provides an alternate 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 six dependent count accumulators can be redefined as a single array named `count`. The individual elements become `count[0]`, `count[1]`, `count[2]`, `count[3]`, `count[4]`, and `count[5]`, as shown in the revised decision-making process in [Figure 6-4](#).

#### **Figure 6-4**

### **Flowchart and Pseudocode of Decision-making Process—but Still the Hard Way**



The shaded statement in Figure 6-4 shows that when `dep` is 0, 1 is added to `count[0]`. You can see similar statements for the rest of the `count` elements; when `dep` is 1, 1 is added to `count[1]`, when `dep` is 2, 1 is added to `count[2]`, and so on. When the `dep` value is 5, this means it was not 1, 2, 3, or 4, so 1 is added to `count[5]`. In other words, 1 is added to one of the elements of the `count` array instead of to an individual variable named `count0`, `count1`, `count2`, `count3`, `count4`, or `count5`. Is this version a big improvement over the original in Figure 6-3? Of course it isn't. You still have not taken advantage of the benefits of using the array in this application.

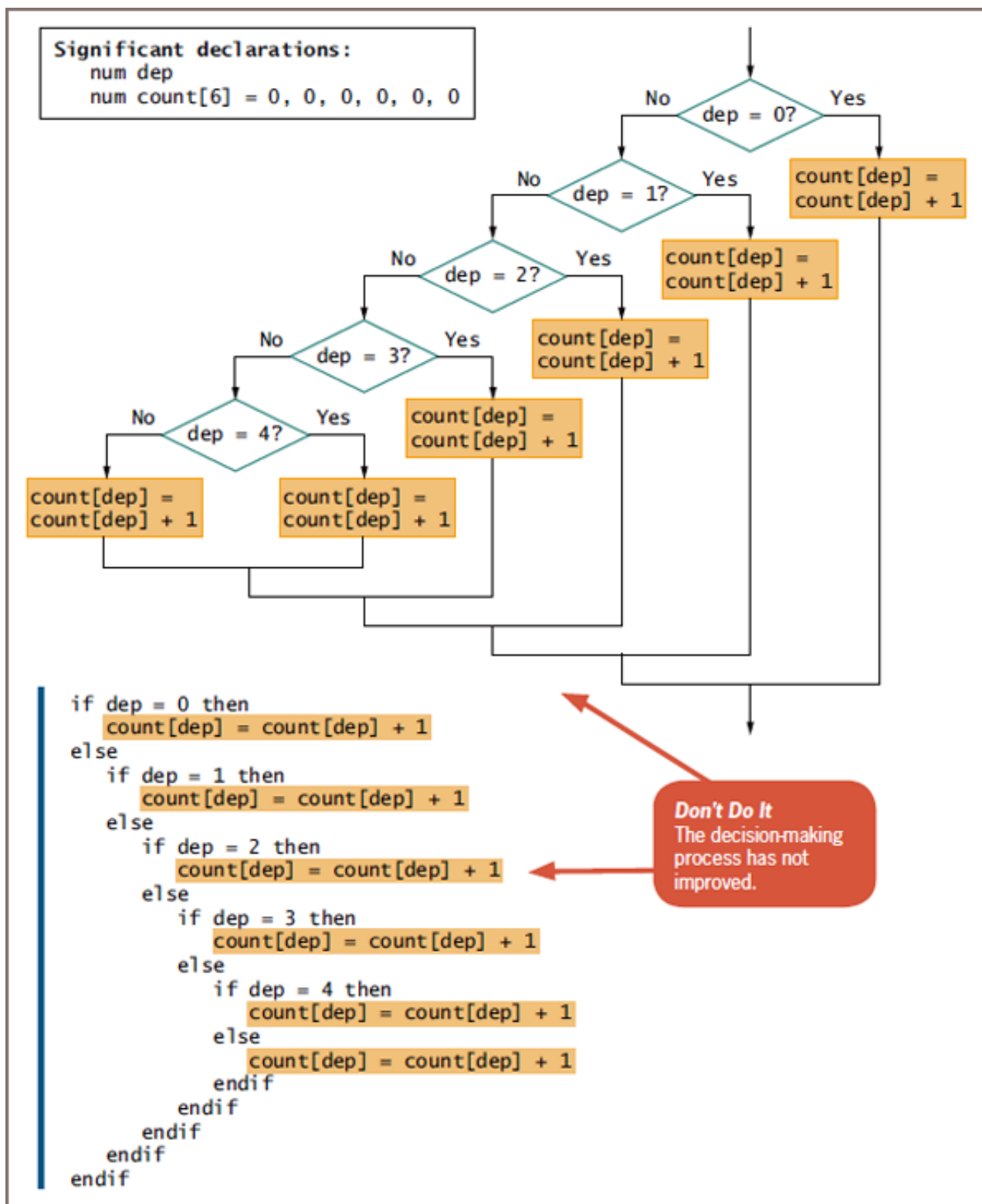
The true benefit of using an array lies in your ability to use a variable as a subscript to the array, instead of using a literal constant such as 0 or 5. Notice in the logic in Figure 6-4 that



within each decision, the value compared to `dep` and the constant that is the subscript in the resulting *Yes* process are always identical. That is, when `dep` is 0, the subscript used to add 1 to the `count` array is 0; when `dep` is 1, the subscript used for the `count` array is 1, and so on. Therefore, you can just use `dep` as a subscript to the array. You can rewrite the decisionmaking process as shown in [Figure 6-5](#).

**Figure 6-5**

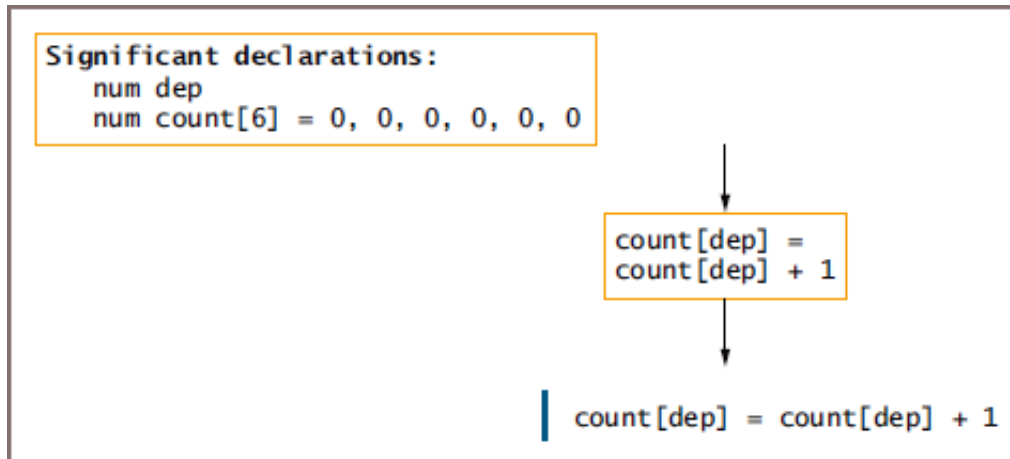
**Flowchart and Pseudocode of Decision-making Process Using an Array—but Still a Hard Way**



The code segment in [Figure 6-5](#) looks no more efficient than the one in [Figure 6-4](#). However, notice the shaded statements in [Figure 6-5](#)—the process that occurs after each decision is exactly the same. In each case, no matter what the value of `dep` is, you always add 1 to `count[dep]`. If you always will take the same action no matter what the answer to a question is, there is no need to ask the question. Instead, you can rewrite the decision-making process as shown in [Figure 6-6](#).

**Figure 6-6**

## Flowchart and Pseudocode of Efficient Decision-making Process Using an Array



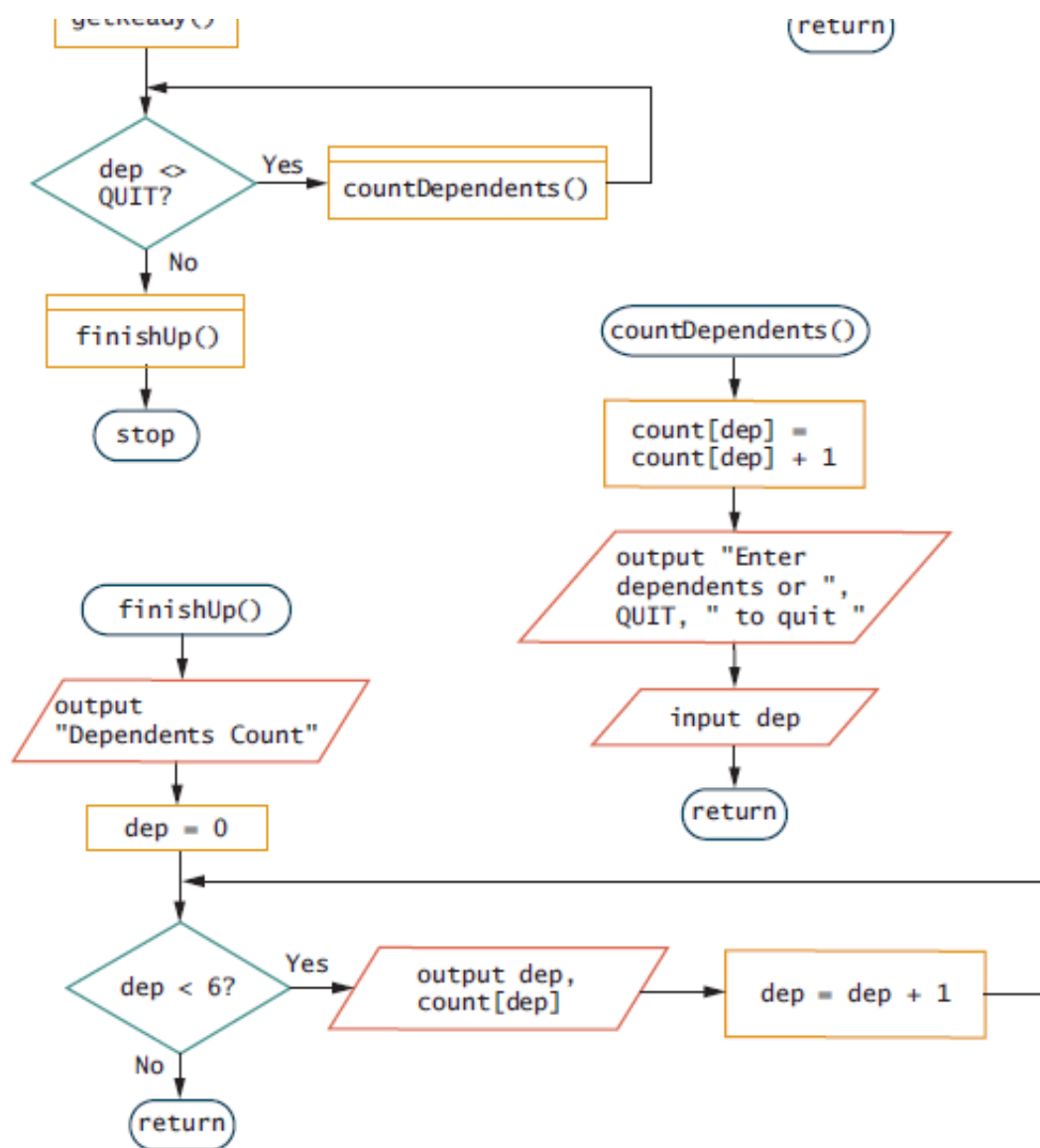
The single statement in Figure 6-6 eliminates the *entire* decision-making process that was the original highlighted section in Figure 6-5! When `dep` is 2, 1 is added to `count[2]`; when `dep` is 4, 1 is added to `count[4]`, and so on. Now you have significantly improved the original logic. What's more, this process does not change whether there are 20, 30, or any other number of possible categories. To use more than five accumulators, you would declare additional `count` elements in the array, but the categorizing logic would remain the same as it is in Figure 6-6.

Figure 6-7 shows an entire program that takes advantage of the array to produce the report that shows counts for dependent categories. Variables and constants are declared and, in the `getReady()` module, a first value for `dep` is entered into the program. In the `countDependents()` module, 1 is added to the appropriate element of the `count` array and the next value is input. The loop in the mainline logic in Figure 6-7 is an indefinite loop; it continues as long as the user does not enter the sentinel value. When data entry is complete, the `finishUp()` module displays the report. First, the heading is output, then `dep` is reset to 0, and then each `dep` and `count[dep]` are output in a loop. The first output statement contains 0 (as the number of dependents) and the value stored in `count[0]`. Then, 1 is added to `dep` and the same set of instructions is used again to display the counts for each number of dependents. The loop in the `finishUp()` module is a definite loop; it executes precisely six times.

**Figure 6-7**

### Flowchart and Pseudocode for Dependents Report Program





```

start
  Declarations
    num dep
    num count[6] = 0, 0, 0, 0, 0, 0
    num QUIT = 999
  getReady()
  while dep <> QUIT
    countDependents()
  endwhile
  finishUp()
stop

getReady()
  output "Enter dependents or ", QUIT, " to quit "
  input dep
return

countDependents()
  count[dep] = count[dep] + 1
  output "Enter dependents or ", QUIT, " to quit "
  input dep
return

finishUp()
  
```

```
output "Dependents Count"
dep = 0
while dep < 6
    output dep, count[dep]
    dep = dep + 1
endwhile
return
```

The dependent-counting program would have *worked* if it contained a long series of decisions and output statements, but the program is easier to write when you use an array and access its values using the number of dependents as a subscript. Additionally, the new 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 your logic easier to understand.

Learning to use arrays properly 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.



Watch the video *Accumulating Values in an Array*.

## Two Truths & A Lie

### How an Array Can Replace Nested Decisions

1. You can use an array to replace a long series of decisions.  
T F
2. You experience a major benefit of arrays when you use an unnamed numeric constant as a subscript as opposed to using a variable.  
T F
3. The process of displaying every element in a 10-element array is basically no different from displaying every element in a 100-element array.  
T F

## 6-3 Using Constants with Arrays

In [Chapter 2](#), you learned that named constants hold values that do not change during a program's execution. When working with arrays, you can use constants in several ways:

- To hold the size of an array
- As the array values
- As a subscript

Chapter 6: Arrays: 6-3a Using a Constant as the Size of an Array  
Book Title: Programming Logic and Design  
Printed By: Ronald Suchy (rsuchy@mayfieldschools.org)  
© 2013 ,

### 6-3a Using a Constant as the Size of an Array

The program in [Figure 6-7](#) still contains one minor flaw. Throughout this book you have learned to avoid *magic numbers*—that is, unnamed constants. As the totals are output in the loop at the end of the program in [Figure 6-7](#), the array subscript is compared to the constant 6. The program can be improved if you use a named constant instead. Using a named constant makes your code easier to modify and understand. In most programming languages you can take one of two approaches:

- You can declare a named numeric constant such as `ARRAY_SIZE = 6`. Then you can use this constant every time you access the array, always making sure any subscript you use remains less than the constant value.
- In many languages, a constant that represents the array size is automatically provided for each array you create. For example, in Java, after you declare an array named `count`, its size is stored in a field named `count.length`. In both C# and Visual Basic, the array size is `count.Length`, with an uppercase *L*.

Chapter 6: Arrays: 6-3b Using Constants as Array Element Values  
Book Title: Programming Logic and Design  
Printed By: Ronald Suchy (rsuchy@mayfieldschools.org)  
© 2013 ,

### 6-3b Using Constants as Array Element Values

Sometimes the values stored in arrays should be constants because they are not changed during program execution. For example, suppose that you create an array that holds names for the months of the year. Don't confuse the array identifier with its contents—the

convention in this book is to use all uppercase letters in constant identifiers, but not necessarily in array values. The array can be declared as follows:

Chapter 6: Arrays: 6-3c Using a Constant as an Array Subscript  
Book Title: Programming Logic and Design  
Printed By: Ronald Suchy (rsuchy@mayfieldschools.org)  
© 2013 ,

## 6-3c Using a Constant as an Array Subscript

Occasionally you will want to use an unnamed numeric constant as a subscript to an array. For example, to display the first value in an array named `salesArray`, you might write a statement that uses an unnamed literal constant as follows:

```
output salesArray[0]
```

You might also have occasion to use a named constant as a subscript. For example, if `salesArray` holds sales values for each of 20 states covered by your company, and Indiana is state 5, you could output the value for Indiana as follows:

```
output salesArray[5]
```

However, if you declare a named constant as `num INDIANA = 5`, then you can display the same value using this statement:

```
output salesArray[INDIANA]
```

An advantage to using a named constant in this case is that the statement becomes self-documenting—anyone who reads your statement more easily understands that your intention is to display the sales value for Indiana.

### Two Truths & A Lie

#### Using Constants with Arrays

1. If you create a named constant equal to an array size, you can use it as a subscript to the array.

T F

2. If you create a named constant equal to an array size, you can use it as a limit against which to compare subscript values.

T F

3. When you declare an array in Java, C#, and Visual Basic, a constant that

represents the array size is automatically provided.

T F

Chapter 6: Arrays: 6-4 Searching an Array for an Exact Match  
Book Title: Programming Logic and Design  
Printed By: Ronald Suchy (rsuchy@mayfieldschools.org)  
© 2013 ,

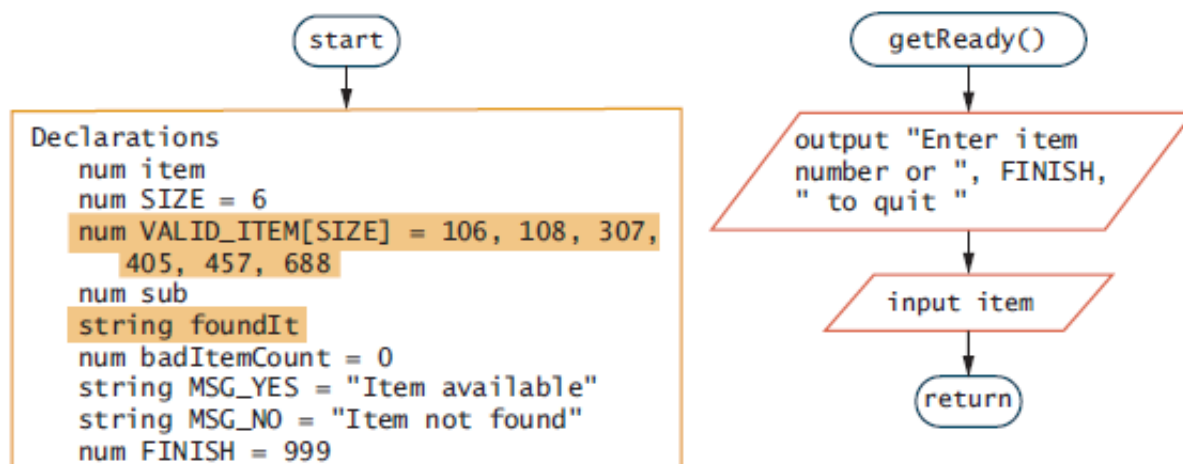
## 6-4 Searching an Array for an Exact Match

In the dependent-counting application in this chapter, the array's subscript variable conveniently held small whole numbers—the number of dependents allowed was 0 through 5—and the `dep` variable directly accessed the array. 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.

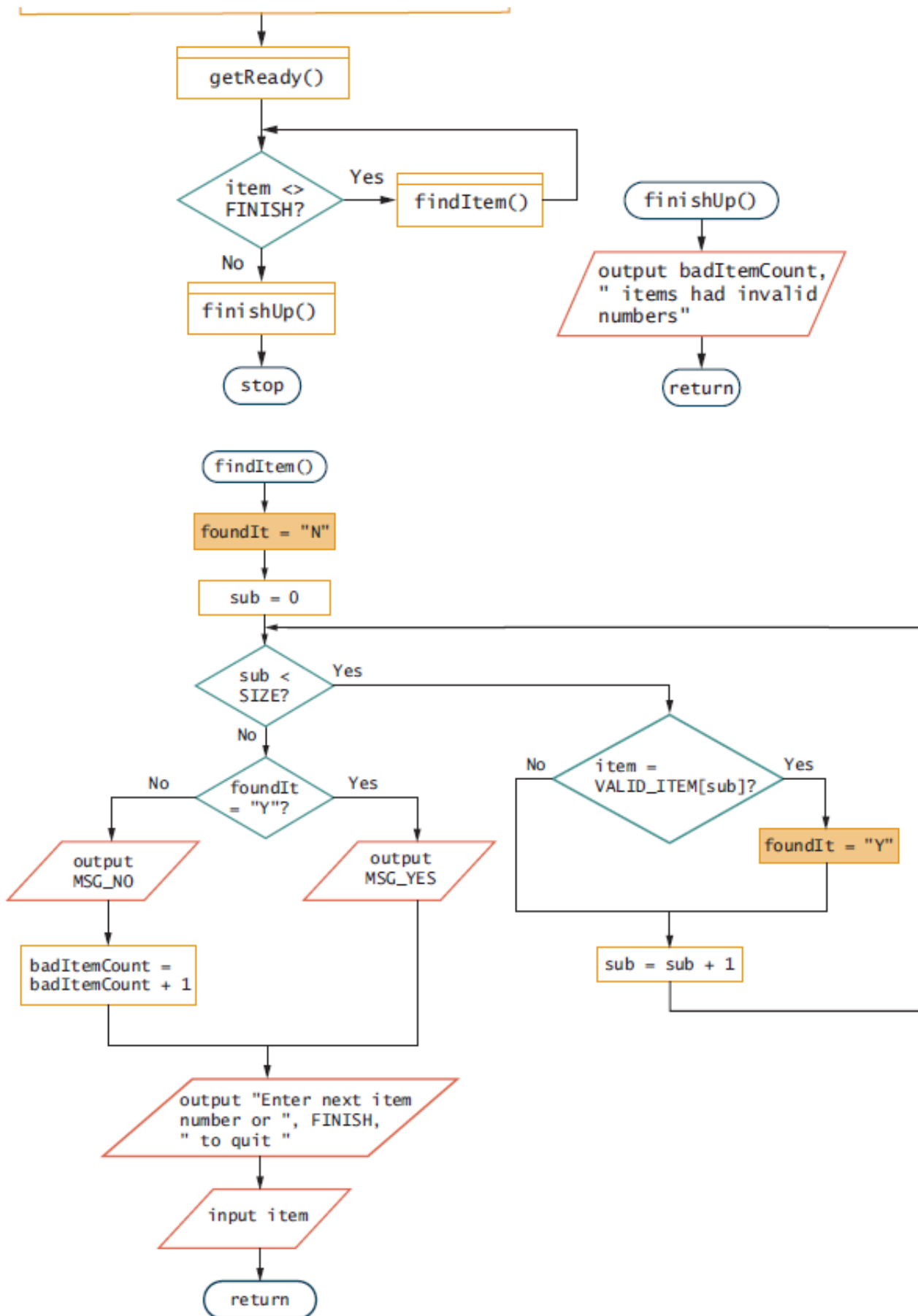
Consider a mail-order business in which customers place orders that contain a name, address, item number, and quantity ordered. Assume that the item numbers from which a customer can choose are three-digit numbers, but perhaps they are not consecutively numbered 001 through 999. For example, let's say that you offer six items: 106, 108, 307, 405, 457, and 688, as shown in the shaded `VALID_ITEM` array declaration in [Figure 6-8](#). (The array is declared as constant because the item numbers do not change during program execution.) When a customer orders an item, a clerical worker can tell whether the order is valid by looking down the list and manually verifying that the ordered item number is on it. In a similar fashion, a computer program can use a loop to test the ordered item number against each `VALID_ITEM`, looking for an exact match. When you search through a list from one end to the other, you are performing a [linear search](#) (A search through a list from one end to the other.) .

**Figure 6-8**

**Flowchart and Pseudocode for Program That Verifies Item Availability**







start

Declarations

num item

num SIZE = 6

```

num SIZE = 6
num VALID_ITEM[SIZE] = 106, 108, 307,
    405, 457, 688
num sub
string foundIt
num badItemCount = 0
string MSG_YES = "Item available"
string MSG_NO = "Item not found"
num FINISH = 999
getReady()
while item <> FINISH
    findItem()
endwhile
finishUp()
stop

getReady()
    output "Enter item number or ", FINISH, " to quit "
    input item
return

findItem()
    foundIt = "N"
    sub = 0
    while sub < SIZE
        if item = VALID_ITEM[sub] then
            foundIt = "Y"
        endif
        sub = sub + 1
    endwhile
    if foundIt = "Y" then
        output MSG_YES
    else
        output MSG_NO
        badItemCount = badItemCount + 1
    endif
    output "Enter next item number or ", FINISH, " to quit "
    input item
return

finishUp()
    output badItemCount, " items had invalid numbers"
return

```

To determine if an ordered item number is valid, you could use a series of six decisions to compare the number to each of the six allowed values. However, the superior approach shown in [Figure 6-8](#) is to create an array that holds the list of valid item numbers and then to 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, it means the ordered item number is not valid.

The `findItem()` module in [Figure 6-8](#) takes the following steps to verify that an item number exists:

- A flag variable named `foundIt` is set to "N". A **flag** (A variable that indicates whether some event has occurred.) is a variable that is set to indicate whether some event has occurred. In this example, N indicates that the item number has not yet been found in the list. (See the first shaded statement in the `findItem()` method in Figure 6-8.)
- A subscript, `sub`, is set to 0. This subscript will be used to access each `VALID_ITEM` element.
- A loop executes, varying `sub` from 0 through one less than the size of the array. Within the loop, the customer's ordered item number is compared to each item number in the array. If the customer-ordered item matches any item in the array, the flag variable is assigned "Y". (See the last shaded statement in the `findItem()` method in Figure 6-8.) After all six valid item numbers have been compared to the ordered item, if the customer item matches none of them, then the flag variable `foundIt` will still hold the value "N".
- If the flag variable's value is "Y" after the entire list has been searched, it means that the item is valid and an appropriate message is displayed, but if the flag has not been assigned "Y", the item was not found in the array of valid items. In this case, an error message is output and 1 is added to a count of bad item numbers.



As an alternative to using the string `foundIt` variable in the method in

Figure 6-8, you might prefer to use a numeric variable that you set to 1 or 0. Most programming languages also support a Boolean data type that you can use for `foundIt`; when you declare a variable to be Boolean, you can set its value to true or false.

## Two Truths & A Lie

### Searching an Array for an Exact Match

1. Only whole numbers can be stored in arrays.

T F

2. Only whole numbers can be used as array subscripts.

T F

3. A flag is a variable that indicates whether some event has occurred.

T F

Chapter 6: Arrays: 6-5 Using Parallel Arrays  
Book Title: Programming Logic and Design  
Printed By: Ronald Suchy (rsuchy@mayfieldschools.org)  
© 2013 ,

## 6-5 Using Parallel Arrays

When you accept an item number into a mail-order company program, you usually want to accomplish more than simply verifying the item's existence. For example, you might want to determine the name, price, or available quantity of the ordered item. Tasks like these can be completed efficiently using parallel arrays. **Parallel arrays** (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.) 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. Although any array can contain just one data type, each array in a set of parallel arrays might be a different type.

Suppose that you have a list of item numbers and their associated prices. One array named `VALID_ITEM` contains six elements; each element is a valid item number. Its parallel array also has six elements. The array is named `VALID_PRICE`; each element is a price of an item. Each price in the `VALID_PRICE` array is conveniently and purposely in the same position as the corresponding item number in the `VALID_ITEM` array. Figure 6-9 shows how the parallel arrays might look in computer memory.

**Figure 6-9**  
**Parallel Arrays in Memory**

		VALID_ITEM[1]		VALID_ITEM[3]		VALID_ITEM[5]
VALID_ITEM[0]			VALID_ITEM[2]		VALID_ITEM[4]	
		106	108	307	405	457
						688
		0.59	0.99	4.50	15.99	17.50
						39.00
		VALID_PRICE[0]	VALID_PRICE[1]	VALID_PRICE[2]	VALID_PRICE[3]	VALID_PRICE[4]
						VALID_PRICE[5]

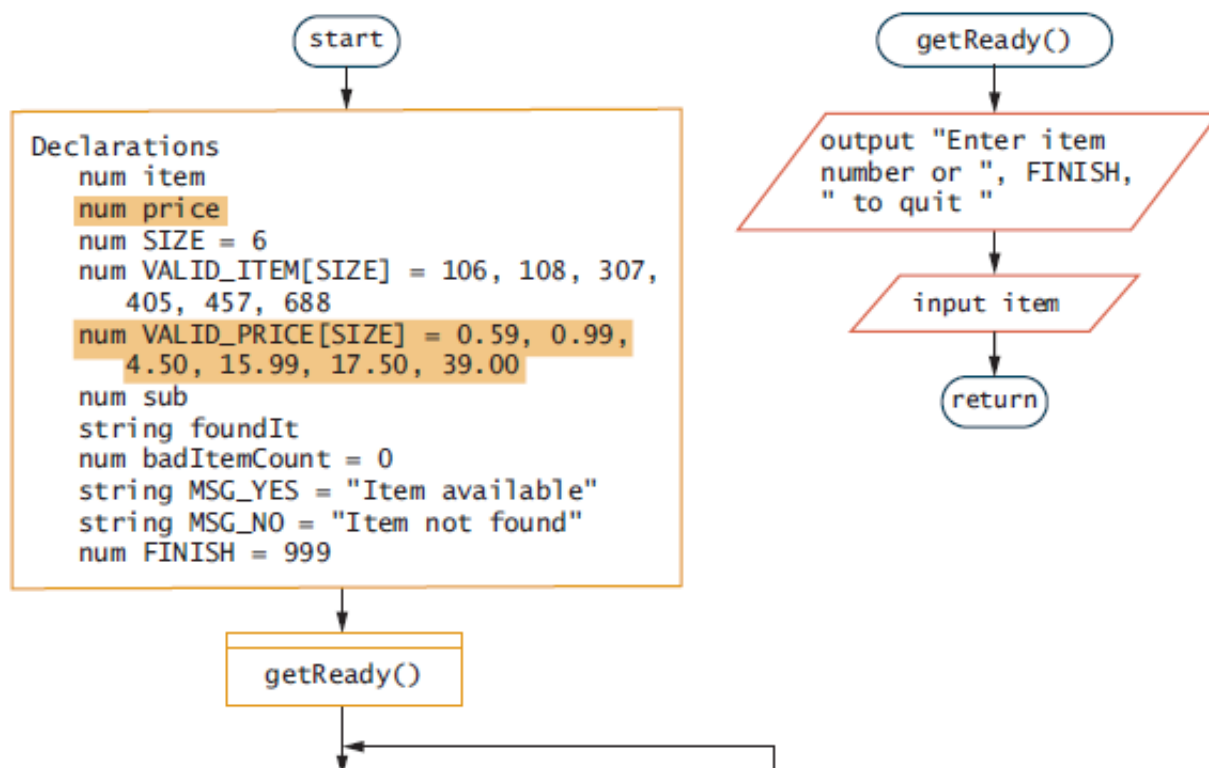
When you use parallel arrays:

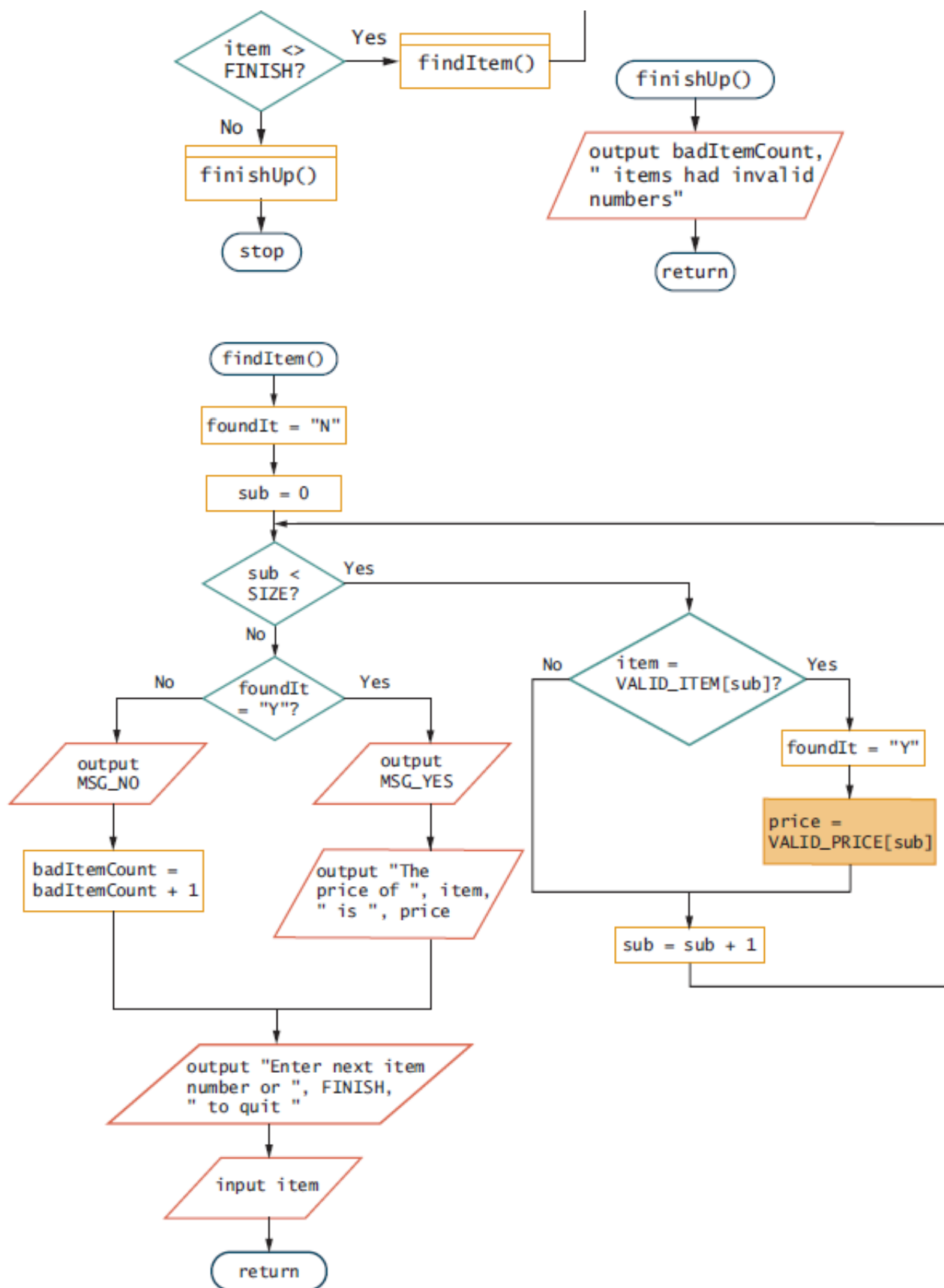
- Two or more arrays contain related data.
- A subscript relates the arrays. That is, elements at the same position in each array are logically related.

Figure 6-10 shows a program that declares parallel arrays. The `VALID_PRICE` array is shaded; each element in it corresponds to a valid item number.

**Figure 6-10**

### Flowchart and Pseudocode of Program That Finds an Item Price Using Parallel Arrays





```

start
  Declarations
    num item
    num price
    num SIZE = 6
    num VALID_ITEM[SIZE] = 106, 108, 307,
  
```

```

    405, 457, 688
    num VALID_PRICE[SIZE] = 0.59, 0.99,
    4.50, 15.99, 17.50, 39.00
    num sub
    string foundIt
    num badItemCount = 0
    string MSG_YES = "Item available"
    string MSG_NO = "Item not found"
    num FINISH = 999
    getReady()
    while item <> FINISH
        findItem()
    endwhile
    finishUp()
stop

getReady()
    output "Enter item number or ", FINISH, " to quit "
    input item
    return

findItem()
    foundIt = "N"
    sub = 0
    while sub < SIZE
        if item = VALID_ITEM[sub] then
            foundIt = "Y"
            price = VALID_PRICE[sub]
        endif
        sub = sub + 1
    endwhile
    if foundIt = "Y" then
        output MSG_YES
        output "The price of ", item, " is ", price
    else
        output MSG_NO
        badItemCount = badItemCount + 1
    endif
    output "Enter next item number or ", FINISH, " to quit "
    input item
    return

finishUp()
    output badItemCount, " items had invalid numbers"
    return

```



Some programmers object to using a cryptic variable name for a subscript,

such as `sub` in [Figure 6-10](#), because such names are not descriptive. These programmers would prefer a name like `priceIndex`. Others approve of short names when the variable is used only in a limited area of a program, as it is used here, to step through an array. Programmers disagree on many style issues like this one. As a programmer, it is your responsibility to find out what conventions are

used among your peers in an organization.

As the program in [Figure 6-10](#) receives a customer's order, it looks through each of the `VALID_ITEM` values separately by varying the subscript `sub` from 0 to the number of items available. When a match for the item number is found, the program pulls the corresponding parallel price out of the list of `VALID_PRICE` values and stores it in the `price` variable. (See shaded statements in [Figure 6-10](#).)

The relationship between an item's number and its price is an **indirect relationship** (Describes the relationship between parallel arrays in which an element in the first array does not directly access its corresponding value in the second array.). That means you don't access a price directly by knowing the item number. Instead, you determine the price by knowing an item number's array position. Once you find a match for the ordered item number in the `VALID_ITEM` array, you know that the price of the item is in the same position in the other array, `VALID_PRICE`. When `VALID_ITEM[sub]` is the correct item, `VALID_PRICE[sub]` must be the correct price, so `sub` links the parallel arrays.

Parallel arrays are most useful when value pairs have an indirect relationship. If values in your program have a direct relationship, you probably don't need parallel arrays. For example, if items were numbered 0, 1, 2, 3, and so on consecutively, you could use the item number as a subscript to the price array instead of using a parallel array to hold item numbers. Even if the items were numbered 200, 201, 202, and so on consecutively, you could subtract a constant value (200) from each and use that as a subscript instead of using a parallel array.

Suppose that a customer orders item 457. Walk through the logic yourself to see if you come up with the correct price per item, \$17.50. Then, suppose that a customer orders item 458. Walk through the logic and see whether the appropriate *Item not found* message is displayed.

Chapter 6: Arrays: 6-5a Improving Search Efficiency  
Book Title: Programming Logic and Design  
Printed By: Ronald Suchy (rsuchy@mayfieldschools.org)  
© 2013,

## 6-5a Improving Search Efficiency

The mail-order program in [figure 6-10](#) is still a little inefficient. When a customer orders item 106 or 108, a match is found on the first or second pass through the loop, and continuing to search provides no further benefit. However, even after a match is made, the program in [Figure 6-10](#) continues searching through the item array until `sub` reaches the value `SIZE`. One way to stop the search when the item has been found and `foundIt` is set to "Y" is to change the loop-controlling question. Instead of simply continuing the loop while the number of comparisons does not exceed the highest allowed array subscript, you should continue the loop while the searched item is not found *and* the number of comparisons has not exceeded the maximum. Leaving the loop as soon as a match is found improves the program's efficiency. The larger the array, the more beneficial it becomes to

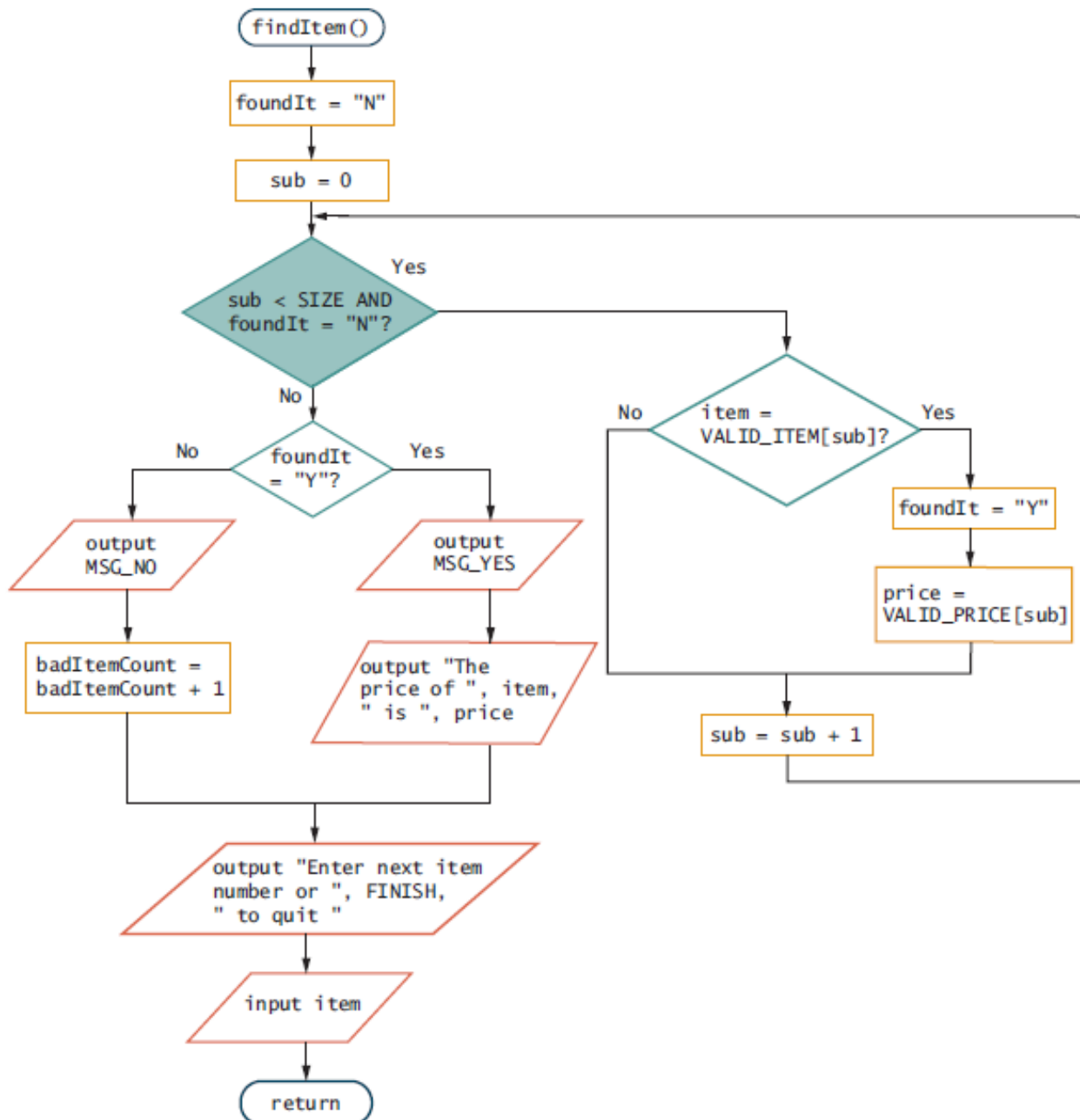


exit the searching loop as soon as you find the desired value.

Figure 6-11 shows the improved version of the `findItem()` module with the altered loopcontrolling question shaded.

**Figure 6-11**

**Flowchart and Pseudocode of the Module That Finds an Item Price and Exits the Loop As Soon As It is Found**



```
findItem()
foundIt = "N"
sub = 0
while sub < SIZE AND foundIt = "N"
    if item = VALID_ITEM[sub] then
        foundIt = "Y"
        price = VALID_PRICE[sub]
    endif
    sub = sub + 1
endwhile
```

```

        sum = sum + 1
    endwhile
    if foundIt = "Y" then
        output MSG_YES
        output "The price of ", item, " is ", price
    else
        output MSG_NO
        badItemCount = badItemCount + 1
    endif
    output "Enter next item number or ", FINISH, " to quit "
    input item
return

```

Notice that the price-finding program offers the greatest efficiency when the most frequently ordered items are stored at the beginning of the array, so that only the seldom-ordered items require many loops before finding a match. Often, you can improve search efficiency by rearranging array elements.



As you study programming, you will learn other search techniques. For

example, a **binary search** (A search that starts in the middle of a sorted list, and then determines whether it should continue higher or lower to find a target value.) starts looking in the middle of a sorted list, and then determines whether it should continue higher or lower.



Watch the video *Using Parallel Arrays*.

## Two Truths & A Lie

### Using Parallel Arrays

1. Parallel arrays must be the same data type.

T F

2. Parallel arrays usually contain the same number of elements.

T F

3. You can improve the efficiency of searching through parallel arrays by using an early exit.

## 6-6 Searching an Array for a Range Match

Customer order item numbers need to match available item numbers exactly to determine the correct price of an item. Sometimes, however, programmers want to work with ranges of values in arrays. In [Chapter 4](#), you learned that a range of values is any series of values—for example, 1 through 5 or 20 through 30.

Suppose that a company decides to offer quantity discounts when a customer orders multiple items, as shown in [Figure 6-12](#).

**Figure 6-12**

### Discounts on Orders By Quantity

Quantity	Discount %
0–8	0
9–12	10
13–25	15
26 or more	20

You want to be able to read in customer order data and determine a discount percentage based on the quantity ordered. For example, if a customer has ordered 20 items, you want to be able to output *Your discount is 15 percent*. 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 6-13](#). This array is set up to contain the discount for 0 items, 1 item, 2 items, and so on. This approach has at least three drawbacks:

**Figure 6-13**

### Usable—but Inefficient—Discount Array

```
numeric DISCOUNT[76]
= 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0.10, 0.10, 0.10, 0.10,
  0.15, 0.15, 0.15, 0.15, 0.15,
  0.15, 0.15, 0.15, 0.15, 0.15,
  0.15, 0.15, 0.15,
  0.20, 0.20, 0.20, 0.20, 0.20,
  0.20, 0.20, 0.20, 0.20, 0.20,
  0.20, 0.20, 0.20, 0.20, 0.20,
  0.20, 0.20, 0.20, 0.20, 0.20,
  0.20, 0.20, 0.20, 0.20, 0.20,
  0.20, 0.20, 0.20, 0.20, 0.20,
  0.20, 0.20, 0.20, 0.20, 0.20,
  0.20, 0.20, 0.20, 0.20, 0.20,
  0.20, 0.20, 0.20, 0.20, 0.20,
  0.20, 0.20, 0.20, 0.20, 0.20
```

**Don't Do It**  
Although this array is usable, it is repetitious, prone to error, and difficult to use.

- 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 nine elements receives the same value, 0, and each of the next four elements receives the same value, 10.
- How do you know you have enough array elements? Is a customer order quantity of 75 items enough? What if a customer orders 100 or 1000 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 two parallel arrays, each with four elements, as shown in [Figure 6-14](#). Each discount rate is listed once in the `DISCOUNT` array, and the low end of each quantity range is listed in the `QUAN_LIMIT` array.

**Figure 6-14**

### Parallel Arrays to Use for Determining Discount

```
num DISCOUNT[4] = 0, 0.10, 0.15, 0.20
num QUAN_LIMIT[4] = 0, 9, 13, 26
```

To find the correct discount for any customer's ordered quantity, you can start with the *last* quantity range limit (`QUAN_LIMIT[3]`). If the quantity ordered is at least that value, 26, the loop is never entered and the customer gets the highest discount rate (`DISCOUNT[3]`, or 20 percent). If the quantity ordered is not at least `QUAN_LIMIT[3]`—that is, if it is less than 26—then you reduce the subscript and check to see if the quantity is at least `QUAN_LIMIT[2]`, or 13. If so, the customer receives `DISCOUNT[2]`, or 15 percent, and so on. [Figure 6-15](#) shows a program that accepts a customer's quantity ordered and determines the appropriate discount rate.

**Figure 6-15**

## Program That Determines Discount Rate

An alternate approach to the one taken in [Figure 6-15](#) 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.

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. The program that determines customer discount rates in [Figure 6-15](#) requires fewer instructions than one that does not use an array, and modifications to the program will be easier to make in the future.

## Two Truths & A Lie

### Searching an Array for a Range Match

1. To help locate a range within which a value falls, you can store the highest value in each range in an array.

T F

2. To help locate a range within which a value falls, you can store the lowest value in each range in an array.

T F

3. When using an array to store range limits, you use a series of comparisons that would otherwise require many separate loop structures.

T F

Chapter 6: Arrays: 6-7 Remaining within Array Bounds  
Book Title: Programming Logic and Design  
Printed By: Ronald Suchy (rsuchy@mayfieldschools.org)  
© 2013 ,

## 6-7 Remaining within Array Bounds

Every array has a finite size. You can think of an array's size in one of two ways—either by the number of elements in the array or by the number of bytes in the array. Arrays are always composed of elements of the same data type, and elements of the same data type always occupy the same number of bytes of memory, so the number of bytes in an array is always a multiple of the number of elements in an array. For example, in Java, integers occupy 4 bytes of memory, so an array of 10 integers occupies exactly 40 bytes.



For a complete discussion of bytes and how they measure computer memory, read [Appendix A](#).

In every programming language, when you access data stored in an array you must use a subscript containing a value that accesses memory occupied by the array. The subscript is actually multiplied by the size of the data type in bytes, and that value is added to the address of the array to find the address of the appropriate element. So, if the subscript is too large or too small, the program will attempt to access an address that is not part of the array's space.

A common error by beginning programmers is to forget that array subscripts start with 0. If you assume that an array's first subscript is 1, you will always be “off by one” in your array manipulation. For example, if you try to manipulate a 10-element array using subscripts 1 through 10, you will commit two errors: You will fail to access the first element that uses subscript 0, and you will attempt to access an extra element at position 10 when the highest usable subscript is 9.

For example, examine the program in [Figure 6-16](#). The method accepts a numeric value for `monthNum` and displays the name associated with that month. The logic in [Figure 6-16](#) makes a questionable assumption: that every number entered by the user is a valid month

number.

### Figure 6-16

#### Determining the Month String from a User's Numeric Entry



In the program in [Figure 6-16](#), notice that 1 is subtracted from `monthNum`

before it is used as a subscript. Although January is the first month in the year, its name occupies the location in the array with the 0 subscript. With values that seem naturally to start with 1, like month numbers, some programmers would prefer to create a 13-element array and simply never use the zero-position element. That way, each “natural” month number would be the correct value to access its data without subtracting. Other programmers dislike wasting memory by creating an extra, unused array element. Although workable programs can be created with or without the extra array element, professional programmers should follow the conventions and preferences of their colleagues and managers.

In [Figure 6-16](#), if the user enters a number that is too small or too large, one of two things will happen depending on the programming language you use. When you use a subscript value that is negative or higher than the highest allowed subscript:

- Some programming languages will stop execution of the program and issue an error



message.

- Other programming languages will not issue an error message but will access a value in a memory location that is outside the area occupied by the array. That area might contain garbage, or worse, it accidentally might contain the name of an incorrect month.

Either way, a logical error occurs. When you use a subscript that is not within the range of acceptable subscripts, it is said to be **out of bounds** ([Describes an array subscript that is not within the range of acceptable subscripts.](#)). Users enter incorrect data frequently; a good program should be able to handle the mistake and not allow the subscript to be out of bounds.



A user might enter an invalid number or might not enter a number at all.

In [Chapter 5](#), you learned that many languages have a built-in method with a name like `isNumeric()` that can test for such mistakes.

You can improve the program in [Figure 6-16](#) by adding a test that ensures the subscript used to access the array is within the array bounds. If you find that the input value is not between 1 and 12 inclusive, you might take one of the following approaches:

- Display an error message and end the program.
- Use a default value for the month. For example, when an entered month is invalid, you might want to assume that it is December.
- Continuously reprompt the user for a new value until it is valid.

The way you handle an invalid month depends on the requirements of your program as spelled out by your user, supervisor, or company policy.

## Two Truths & A Lie

### Remaining within Array Bounds

1. Elements in an array frequently are different data types, so calculating the amount of memory the array occupies is difficult.

T F

2. If you attempt to access an array with a subscript that is too small, some

programming languages will stop execution of the program and issue an error message.

T F

3. If you attempt to access an array with a subscript that is too large, some programming languages access an incorrect memory location outside the array bounds.

T F

Chapter 6: Arrays: 6-8 Using a `for` Loop to Process Arrays  
Book Title: Programming Logic and Design  
Printed By: Ronald Suchy (rsuchy@mayfieldschools.org)  
© 2013 ,

## 6-8 Using a `for` Loop to Process Arrays

In [Chapter 5](#), you learned about the `for` loop—a loop that, in a single statement, initializes a loop control variable, compares it to a limit, and alters it. The `for` loop is a particularly convenient tool when working with arrays because you frequently need to process every element of an array from beginning to end. As with a `while` loop, when you use a `for` loop, you must be careful to stay within array bounds, remembering that the highest usable array subscript is one less than the size of the array. [Figure 6-17](#) shows a `for` loop that correctly displays all of a company's department names that are stored in an array declared as `DEPTS`. Notice that `dep` is incremented through one less than the number of departments because with a five-item array, the subscripts you can use are 0 through 4.

### Figure 6-17

#### Pseudocode That Uses a `for` Loop to Display an Array of Department Names

The loop in [Figure 6-17](#) is slightly inefficient because, as it executes five times, the subtraction operation that deducts 1 from `SIZE` occurs each time. Five subtraction operations do not consume much computer power or time, but in a loop that processes thousands or millions of array elements, the program's efficiency would be compromised.

Figure 6-18 shows a superior solution. A new constant called `ARRAY_LIMIT` is calculated once, then used repeatedly in the comparison operation to determine when to stop cycling through the array.

### Figure 6-18

#### Pseudocode That Uses a More Efficient `for` Loop to Output Department Names

Two Truths & A Lie

##### Using a `for` Loop to Process Arrays

1. The `for` loop is a particularly convenient tool when working with arrays.

T F

2. You frequently need to process every element of an array from beginning to end.

T F

3. Not being concerned with array bounds is one advantage to using a `for` loop to process array elements.

T F

## 6-9a Chapter Summary

- An array is a named series or list of values in computer memory, all of which have the same data type but are differentiated with subscripts. Each array element occupies an area in memory next to, or contiguous to, the others.
- You often can use a variable as a subscript to an array, which allows you to replace multiple nested decisions with many fewer statements.
- Constants can be used to hold an array's size or to represent its values. Using a named constant for an array's size makes the code easier to understand and less likely to contain an error. Array values are declared as constant when they should not change during program execution.
- 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.
- With parallel arrays, each element in one array is associated with the element in the same relative position in the other array.
- 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.
- When you access data stored in an array, it is important to use a subscript containing a value that accesses memory occupied by the array. When you use a subscript that is not within the defined range of acceptable subscripts, your subscript is said to be out of bounds.
- The `for` loop is a particularly convenient tool when working with arrays because you frequently need to process every element of an array from beginning to end.

Chapter 6: Arrays: 6-9b Key Terms  
Book Title: Programming Logic and Design  
Printed By: Ronald Suchy (rsuchy@mayfieldschools.org)  
© 2013 ,

## Chapter Review

### 6-9b Key Terms

**array** (A series or list of variables in computer memory, all of which have the same name but are differentiated with subscripts.)

**element** (A separate array variable.)

**size of the array** (The number of elements an array can hold.)

**subscript** (A number that indicates the position of an element within an array.)

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

**populating an array** (populating an array is the act of assigning values to the array elements.)

**linear search** (A search through a list from one end to the other.)

**flag** (A variable that indicates whether some event has occurred.)

**Parallel arrays** (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.)

**indirect relationship** (Describes the relationship between parallel arrays in which an element in the first array does not directly access its corresponding value in the second array.)

**binary search** (A search that starts in the middle of a sorted list, and then determines whether it should continue higher or lower to find a target value.)

**out of bounds** (Describes an array subscript that is not within the range of acceptable subscripts.)

Chapter 6: Arrays: 6-9c Review Questions  
Book Title: Programming Logic and Design  
Printed By: Ronald Suchy (rsuchy@mayfieldschools.org)  
© 2013 ,

## Chapter Review

### 6-9c Review Questions

1. A subscript is a(n) \_\_\_\_ .
  - a. element in an array
  - b. alternate name for an array
  - c. number that represents the highest value stored within an array
  - d. number that indicates the position of an array element
2. Each variable in an array must have the same \_\_\_\_ as the others.
  - a. data type

- b. subscript
  - c. value
  - d. memory location
3. Each data item in an array is called a(n) \_\_\_\_ .
- a. data type
  - b. subscript
  - c. component
  - d. element
4. The subscripts of any array are always \_\_\_\_ .
- a. integers
  - b. fractions
  - c. characters
  - d. strings of characters
5. Suppose that 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 array holds exactly four elements
  - c. there are exactly two elements between those two elements
  - d. the two elements are at the same memory location
6. Suppose that you want to write a program that inputs customer data and displays a summary of the number of customers who owe more than \$1000 each, in each of 12 sales regions. Customer data variables 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. The most useful type of subscript for manipulating arrays is a \_\_\_\_ .
- a. numeric constant
  - b. variable
  - c. character
  - d. filename
8. A program contains a seven-element array that holds the names of the days of the week. At the start of the program, you display the day names using a subscript named `dayNum`. You display the same array values again at the end of the program, where 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 instead of a variable
9. Suppose that you have declared an array as follows: `num values[4] = 0, 0, 0, 0`. Which of the following is an allowed operation?
- a. `values[2] = 17`
  - b. `input values[0]`
  - c. `values[3] = values[0] + 10`
  - d. all of the above
10. Filling an array with values during a program's execution is known as \_\_\_\_ the array.
- a. executing
  - b. colonizing
  - c. populating
  - d. declaring

11. Using an array can make a program \_\_\_\_ .
- a. easier to understand
  - b. illegal in some modern languages
  - c. harder to maintain
  - d. all of the above
12. A \_\_\_\_ is a variable that can be set to indicate whether some event has occurred.
- a. subscript
  - b. banner
  - c. counter
  - d. flag
13. What do you call two arrays in which each element in one array is associated with the element in the same relative position in the other array?
- a. cohesive arrays
  - b. parallel arrays
  - c. hidden arrays
  - d. perpendicular arrays
14. In most modern programming languages, the highest subscript you should use with a 12-element array is \_\_\_\_ .
- a. 10
  - b. 11
  - c. 12
  - d. 13
15. Parallel arrays \_\_\_\_ .
- a. frequently have an indirect relationship
  - b. never have an indirect relationship



- c. must be the same data type
  - d. must not be the same data type
16. Each element in a seven-element array can hold \_\_\_\_ value(s).
- a. one
  - b. seven
  - c. at least seven
  - d. an unlimited number of
17. 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 criteria in [Table 6-1](#).

Table 6-1

**Barkley Dog Training Academy Achievement Levels**

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, so that each dog can receive 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 the above
18. When you use a subscript value that is negative or higher than the number of elements in an array, \_\_\_\_ .
- a. execution of the program stops and an error message is issued
  - b. a value in a memory location that is outside the area occupied by the array will be accessed
  - c. a value in a memory location that is outside the area occupied by the array will be accessed, but only if the value is the correct data type
  - d. the resulting action depends on the programming language used
19. In every array, a subscript is out of bounds when it is \_\_\_\_ .
- a. negative
  - b. 0
  - c. 1
  - d. 999
20. You can access every element of an array using a \_\_\_\_ .
- a. while loop
  - b. for loop
  - c. both of the above
  - d. none of the above

## Chapter Review

### 6-9d Exercises

1.
  - a. Design the logic for a program that allows a user to enter 15 numbers, then displays them in the reverse order of entry.
  - b. Modify the reverse-display program so that the user can enter any amount of numbers up to 15 until a sentinel value is entered.
2.
  - a. Design the logic for a program that allows a user to enter 15 numbers, then displays each number and its difference from the numeric average of the numbers entered.
  - b. Modify the program in Exercise 2a so that the user can enter any amount of numbers up to 15 until a sentinel value is entered.
3.
  - a. Registration workers at a conference for authors of children's books have collected data about conference participants, including the number of books each author has written and the target age of their readers. The participants have written from 1 to 40 books each, and target readers' ages range from 0 through 16. Design a program that continuously accepts the number of books written until a sentinel value is entered, and then displays a list of how many participants have written each number of books (1 through 40).
  - b. Modify the author registration program so that a target age for each author's audience is input until a sentinel value is entered. The output is a count of the number of books written for each of the following age groups: under 3, 3 through 7, 8 through 10, 11 through 13, and 14 and older.
4.
  - a. The Downdog Yoga Studio offers five types of classes, as shown in [Table 6-2](#). Design a program that accepts a number representing a class and then displays the name of the class.

Table 6-2

**Downdog Yoga Studio classes**

<b>Class Number</b>	<b>Class Name</b>
-------------------------	-------------------

1	Yoga 1
---	--------

2	Yoga 2
3	Children's Yoga
4	Prenatal Yoga
5	Senior Yoga

- b. Modify the Downdog Yoga Studio program so that numeric class requests can be entered continuously until a sentinel value is entered. Then display each class number, name, and a count of the number of requests for each class.
5.
  - a. Watson 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. Write a program that accepts data for each student in the school—student ID, classroom number, and score on the achievement test. Design a program that lists the total points scored for each of the 30 classrooms.
  - b. Modify the Watson Elementary School program so that the average of the test scores is output for each classroom, rather than total scores for each classroom.
6. The Jumpin' Jive coffee shop charges \$2.00 for a cup of coffee and offers the add-ins shown in [Table 6-3](#).

Table 6-3

**Add-in List for Jumpin' Jive Coffee Shop**

Product	Price (\$)
---------	------------

Whipped cream	0. 89
---------------	-------

Cinnamon	0. 25
----------	-------

---

Chocolate	0. 59
sauce	

---

Amaretto	1. 50
----------	-------

---

Irish	1. 75
whiskey	

---

Design the logic for an application that allows a user to enter ordered add-ins continuously until a sentinel value is entered. After each item, display its price or the message *Sorry, we do not carry that* as output. After all items have been entered, display the total price for the order.

7. Design the application logic for a company that wants a report containing a breakdown of payroll by department. Input includes each employee's department number, hourly salary, and number of hours worked. The output is a list of the seven departments in the company and the total gross payroll (rate times hours) for each department. The department names are shown in [Table 6-4](#).

Table 6-4

**Department Numbers and Names**

<b>Department Number</b>	<b>Department Name</b>
------------------------------	----------------------------

---

1	Personnel
---	-----------

---

2	Marketing
---	-----------

---

3	Manufacturing
---	---------------

---

4	Computer Services
---	----------------------

---

5	Sales
---	-------

---

6                      Accounting

7                      Shipping

8. Design a program that computes pay for employees. Allow a user to continuously input employees' names until an appropriate sentinel value is entered. Also input each employee's hourly wage and hours worked. Compute each employee's gross pay (hours times rate), withholding tax percentage (based on [Table 6-5](#)), withholding tax amount, and net pay (gross pay minus withholding tax). Display all the results for each employee. After the last employee has been entered, display the sum of all the hours worked, the total gross payroll, the total withholding for all employees, and the total net payroll.

Table 6-5

**Withholding Percentage based on Gross Pay**

<b>Weekly Gross Pay(\$)</b>	<b>Withholding Percentage (%)</b>
-------------------------------------	---

0.00- 300.00	10
-----------------	----

300. 01- 550.00	13
--------------------	----

550. 01- 800.00	16
--------------------	----

800. 01 and up	20
-------------------	----

9. Countrywide Tours conducts sightseeing trips for groups from its home base in Iowa. Create an application that continuously accepts tour data, including a three-digit tour number; the numeric month, day, and year values representing the tour start date; the number of travelers taking the tour; and

a numeric code that represents the destination. As data is entered for each tour, verify that the month, day, year, and destination code are valid; if any of these is not valid, continue to prompt the user until valid data is entered. The valid destination codes are shown in [Table 6-6](#).

Table 6-6

**Countrywide Tours Codes and Prices**

<b>Code</b>	<b>Destination</b>	<b>Price per Person (\$)</b>
-------------	--------------------	--

1	Chicago	300.00
---	---------	--------

2	Boston	480.00
---	--------	--------

3	Miami	1050.00
---	-------	---------

4	San Francisco	1300.00
---	------------------	---------

Design the logic for an application that outputs each tour number, validated start date, destination code, destination name, number of travelers, gross total price for the tour, and price for the tour after discount. The gross total price is the tour price per guest times the number of travelers. The final price includes a discount for each person in larger tour groups, based on [Table 6-7](#).

Table 6-7

**Countrywide Tours Discounts**

<b>Number of Tourists</b>	<b>Discount per Tourist (\$)</b>
-----------------------------------	--------------------------------------

1-5	0
-----	---

6-12	75
13-20	125
21-50	200
51 and over	300

10.
  - a. *Daily Life Magazine* wants an analysis of the demographic characteristics of its readers. The marketing department has collected reader survey records containing the age, gender, marital status, and annual income of readers. Design an application that allows a user to enter reader data and, when data entry is complete, produces a count of readers by age groups as follows: under 20, 20–29, 30–39, 40–49, and 50 and older.
  - b. Modify the *Daily Life Magazine* program so that it produces a count of readers by gender within age group—that is, under-20 females, under-20 males, and so on.
  - c. Modify the *Daily Life Magazine* program so that it produces a count of readers by income groups as follows: under \$30,000, \$30,000–\$49,999, \$50,000–\$69,999, and \$70,000 and up.
11. Glen Ross Vacation Property Sales employs seven salespeople, as shown in [Table 6-8](#).

Table 6-8

**Glen Ross Salespeople**

ID Number	Salesperson 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. The time is expressed in hours and minutes, based on a 24-hour clock. The sale amount is expressed in whole dollars. Salespeople earn a commission that differs for each sale, based on the rate schedule in [Table 6-9](#).

Table 6-9

**Glen Ross Commission Schedule**

<b>Sale Amount (\$)</b>	<b>Commission Rate (%)</b>
-----------------------------	--------------------------------

---

0-50, 999	4
-----------	---

---

51, 000- 125, 999	5
----------------------	---

---

126, 000- 200, 999	6
-----------------------	---

---

201, 000 and up	7
--------------------	---

---

Design an application that produces each of the following:

- A list of each salesperson number, name, total sales, and total commissions

- b. A list of 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 list of 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
12. Design an application in which the number of days for each month in the year is stored in an array. (For example, January has 31 days, February has 28, and so on. Assume that the year is not a leap year.) Prompt a user to enter a birth month and day, and continue to prompt until the day entered is in range for the month. Compute the day's numeric position in the year. (For example, February 2 is day 33.) Then, using parallel arrays, find and display the traditional Zodiac sign for the date. For example, the sign for February 2 is Aquarius.

## Find the Bugs

13. Your downloadable files for [Chapter 6](#) include DEBUG06-01.txt, DEBUG06-02.txt, and DEBUG06-03.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. Create the logic for a Magic 8 Ball game in which the user enters a question such as *What does my future hold?* The computer randomly selects one of eight possible vague answers, such as *It remains to be seen*.
15. Create the logic for an application that contains an array of 10 multiple-choice questions related to your favorite hobby. Each question contains three answer choices. Also create a parallel array that holds the correct answer to each question—A, B, or C. Display each question and verify that the user enters only A, B, or C as the answer—if not, keep prompting the user until a valid response is entered. If the user responds to a question correctly, display *Correct!*; otherwise, display *The correct answer is* and the letter of the correct

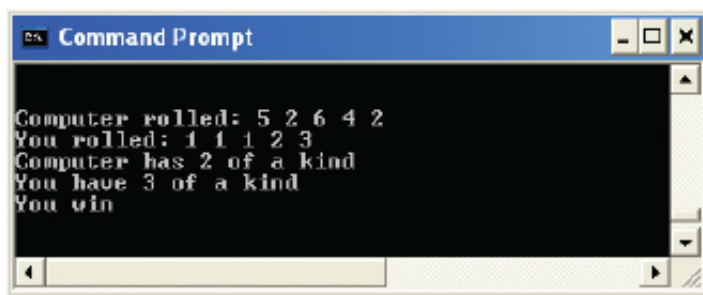
answer. After the user answers all the questions, display the number of correct and incorrect answers.

16. Create the logic for a dice game. The application randomly “throws” five dice for the computer and five dice for the player. After each random throw, store the results in an array. The application displays all the values, which can be from 1 to 6 inclusive for each die. Decide the winner based on the following hierarchy of die values. Any higher combination beats a lower one; for example, five of a kind beats four of a kind.
  - Five of a kind
  - Four of a kind
  - Three of a kind
  - A pair

For this game, the numeric dice values do not count. For example, if both players have three of a kind, it’s a tie, no matter what the values of the three dice are. Additionally, the game does not recognize a full house (three of a kind plus two of a kind). [Figure 6-19](#) shows how the game might be played in a command-line environment.

**Figure 6-19**

### Typical Execution of the Dice Game



Improve the dice game so that when both players have the same number of matching dice, the higher value wins. For example, two 6s beats two 5s.

17. Design the logic for the game Hangman, in which the user guesses letters in a hidden word. Store the letters of a word in an array of characters. Display a dash for each missing letter. Allow the user to continuously guess a letter until all the letters in the word are guessed correctly. As the user enters each guess, display the word again, filling in the guessed letter if it was correct. For example, if the hidden word is *computer*, first display a series of eight dashes: \_\_\_\_\_. After the user guesses *p*, the display becomes ---p---. Make

sure that when a user makes a correct guess, all the matching letters are filled in. For example, if the word is *banana* and the user guesses *a*, all three *a* characters should be filled in.

18. Create two parallel arrays that represent a standard deck of 52 playing cards. One array is numeric and holds the values 1 through 13 (representing Ace, 2 through 10, Jack, Queen, and King). The other array is a string array that holds suits (Clubs, Diamonds, Hearts, and Spades). Create the arrays so that all 52 cards are represented. Then, create a War card game that randomly selects two cards (one for the player and one for the computer) and declares a winner or a tie based on the numeric value of the two cards. The game should last for 26 rounds and use a full deck with no repeated cards. For this game, assume that the lowest card is the Ace. Display the values of the player's and computer's cards, compare their values, and determine the winner. When all the cards in the deck are exhausted, display a count of the number of times the player wins, the number of times the computer wins, and the number of ties.

Here are some hints:

- Start by creating an array of all 52 playing cards.
- Select a random number for the deck position of the player's first card and assign the card at that array position to the player.
- Move every higher-positioned card in the deck "down" one to fill in the gap. In other words, if the player's first random number is 49, select the card at position 49 (both the numeric value and the string), move the card that was in position 50 to position 49, and move the card that was in position 51 to position 50. Only 51 cards remain in the deck after the player's first card is dealt, so the available-card array is smaller by one.
- In the same way, randomly select a card for the computer and "remove" the card from the deck.

## Up for Discussion

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

Chapter 6: Arrays: 6-9d Exercises

Book Title: Programming Logic and Design

Printed By: Ronald Suchy (rsuchy@mayfieldschools.org)

© 2013 ,

© 2015 Cengage Learning Inc. All rights reserved. No part of this work may be reproduced or used in any form or by any means - graphic, electronic, or mechanical, or in any other manner - without the written permission of the copyright holder.