Arrays 7

7.1 INTRODUCTION TO ARRAYS 378
Declaring and Referencing Arrays 378
Programming Tip: Use ***for*** Loops with Arrays 380
Pitfall: Array Indexes Always Start with Zero 380
Programming Tip: Use a Defined *Constant* for the Size of an Array 380
Arrays in Memory 382
Pitfall: Array Index Out of Range 383
Initializing Arrays 386
Programming Tip: C++11 Range-Based for Statement 386

7.2 ARRAYS IN FUNCTIONS 389
Indexed Variables as Function Arguments 389
Entire Arrays as Function Arguments 391
The ***const*** Parameter Modifier 394
Pitfall: Inconsistent Use of ***const*** Parameters 397
Functions That Return an Array 397
Case Study: Production Graph 398

7.3 PROGRAMMING WITH ARRAYS 411
Partially Filled Arrays 411
Programming Tip: Do Not Skimp on Formal Parameters 414
Programming Example: Searching an Array 414
Programming Example: Sorting an Array 417
Programming Example: Bubble Sort 421

7.4 MULTIDIMENSIONAL ARRAYS 424
Multidimensional Array Basics 425
Multidimensional Array Parameters 425
Programming Example: Two-Dimensional Grading Program 427
Pitfall: Using Commas Between Array Indexes 431

Chapter Summary 432
Answers to Self-Test Exercises 433
Practice Programs 437
Programming Projects 439

It is a capital mistake to theorize before one has data.

SIR ARTHUR CONAN DOYLE, *Scandal in Bohemia* (Sherlock Holmes)

INTRODUCTION

An array is used to process a collection of data all of which is of the same type, such as a list of temperatures or a list of names. This chapter introduces the basics of defining and using arrays in C++ and presents many of the basic techniques used when designing algorithms and programs that use arrays.

PREREQUISITES

This chapter uses material from Chapters 2 through 6.

7.1 INTRODUCTION TO ARRAYS

Suppose we wish to write a program that reads in five test scores and performs some manipulations on these scores. For instance, the program might compute the highest test score and then output the amount by which each score falls short of the highest. The highest score is not known until all five scores are read in. Hence, all five scores must be retained in storage so that after the highest score is computed each score can be compared to it.

To retain the five scores, we will need something equivalent to five variables of type `int`. We could use five individual variables of type `int`, but five variables are hard to keep track of, and we may later want to change our program to handle 100 scores; certainly, 100 variables are impractical. An array is the perfect solution. An **array** behaves like a list of variables with a uniform naming mechanism that can be declared in a single line of simple code. For example, the names for the five individual variables we need might be `score[0]`, `score[1]`, `score[2]`, `score[3]`, and `score[4]`. The part that does not change—in this case, `score`—is the name of the array. The part that can change is the integer in the square brackets, `[ ]`.

Declaring and Referencing Arrays

In C++, an array consisting of five variables of type `int` can be declared as follows:

```
    int score[5];
```

This declaration is like declaring the following five variables to all be of type `int`:

```
score[0], score[1], score[2], score[3], score[4]
```

The individual variables that together make up the array are referred to in a variety of different ways. We will call them **indexed variables,** though they are also sometimes called **subscripted variables** or **elements** of the array. The number in square brackets is called an **index** or a **subscript.** In C++, indexes are numbered starting with 0, not with 1 or any other number except 0. The number of indexed variables in an array is called the **declared size** of the array, or sometimes simply the **size** of the array. When an array is declared, the size of the array is given in square brackets after the array name. The indexed variables are then numbered (also using square brackets), starting with 0 and ending with the integer that is one less than the size of the array.

In our example, the indexed variables were of type *int*, but an array can have indexed variables of any type. For example, to declare an array with indexed variables of type *double*, simply use the type name *double* instead of *int* in the declaration of the array. All the indexed variables for one array are, however, of the same type. This type is called the **base type** of the array. Thus, in our example of the array score, the base type is *int*.

You can declare arrays and regular variables together. For example, the following declares the two *int* variables next and max in addition to the array score:

```
int next, score[5], max;
```

An indexed variable like score[3] can be used anyplace that an ordinary variable of type *int* can be used.

Do not confuse the two ways to use the square brackets [ ] with an array name. When used in a declaration, such as

```
int score[5];
```

the number enclosed in the square brackets specifies how many indexed variables the array has. When used anywhere else, the number enclosed in the square brackets tells which indexed variable is meant. For example, score[0] through score[4] are indexed variables.

The index inside the square brackets need not be given as an integer constant. You can use any expression in the square brackets as long as the expression evaluates to one of the integers 0 through the integer that is one less than the size of the array. For example, the following will set the value of score[3] equal to 99:

```
int n = 2;
score[n + 1] = 99;
```

Although they may look different, score[n+1] and score[3] are the same indexed variable in the code above. That is because n + 1 evaluates to 3.

The identity of an indexed variable, such as score[i], is determined by the value of its index, which in this instance is i. Thus, you can write programs that say things such as “do such and such to the ith indexed variable,” where the value of i is computed by the program. For example, the program in Display 7.1 reads in scores and processes them in the way we described at the start of this chapter.

PROGRAMMING TIP Use *for* Loops with Arrays

The second *for* loop in Display 7.1 illustrates a common way to step through an array using a *for* loop:

```
for (i = 0; i < 5; i++)
    cout << score[i] << " off by "
         << (max - score[i]) << endl;
```

The *for* statement is ideally suited to array manipulations.

PITFALL Array Indexes Always Start with Zero

The indexes of an array always start with 0 and end with the integer that is one less than the size of the array.

PROGRAMMING TIP Use a Defined *Constant* for the Size of an Array

Look again at the program in Display 7.1. It only works for classes that have exactly five students. Most classes do not have exactly five students. One way to make a program more versatile is to use a defined constant for the size of each array. For example, the program in Display 7.1 could be rewritten to use the following defined constant:

```
const int NUMBER_OF_STUDENTS = 5;
```

The line with the array declaration would then be

```
int i, score[NUMBER_OF_STUDENTS], max;
```

Of course, all places that have a 5 for the size of the array should also be changed to have NUMBER_OF_STUDENTS instead of 5. If these changes are made to the program (or better still, if the program had been written this way in the first place), then the program can be rewritten to work for any number of students by simply changing the one line that defines the constant NUMBER_OF_STUDENTS. Note that on many compilers you cannot use a variable for the array size, such as the following:

```
cout << "Enter number of students:\n";
cin >> number;
int score[number]; //ILLEGAL ON MANY COMPILERS!
```

DISPLAY 7.1 Program Using an Array

```
//Reads in 5 scores and shows how much each
//score differs from the highest score.
#include <iostream>

int main( )
{
    using namespace std;
    int i, score[5], max;

    cout << "Enter 5 scores:\n";
    cin >> score[0];
    max = score[0];
    for (i = 1; i < 5; i++)
    {
        cin >> score[i];
        if (score[i] > max)
            max = score[i];
        //max is the largest of the values score[0],..., score[i].
    }

    cout << "The highest score is " << max << endl
         << "The scores and their\n"
         << "differences from the highest are:\n";
    for (i = 0; i < 5; i++)
        cout << score[i] << " off by "
             << (max - score[i]) << endl;

    return 0;
}
```

Sample Dialogue

```
Enter 5 scores:
5 9 2 10 6
The highest score is 10
The scores and their
differences from the highest are:
5 off by 5
9 off by 1
2 off by 8
10 off by 0
6 off by 4
```

Some but not all compilers will allow you to specify an array size with a variable in this way. However, for the sake of portability you should not do so, even if your compiler permits it. (In Chapter 9 we will discuss a different kind of array whose size can be determined when the program is run. ■

Arrays in Memory

Before discussing how arrays are represented in a computer's memory, let's first see how a simple variable, such as a variable of type `int` or `double`, is represented in the computer's memory. A computer's memory consists of a list of numbered locations called bytes.[1] The number of a byte is known as its address. A simple variable is implemented as a portion of memory consisting of some number of consecutive bytes. The number of bytes is determined by the type of the variable. Thus, a simple variable in memory is described by two pieces of information: an **address** in memory (giving the location of the first byte for that variable) and the type of the variable, which tells how many bytes of memory the variable requires. When we speak of the address of a variable, it is this address we are talking about. When your program stores a value in the variable, what really happens is that the value (coded as 0s and 1s) is placed in those bytes of memory that are assigned to that variable. Similarly, when a variable is given as a (call-by-reference) argument to a function, it is the address of the variable that is actually given to the calling function. Now let's move on to discuss how arrays are stored in memory.

Array indexed variables are represented in memory the same way as ordinary variables, but with arrays there is a little more to the story. The locations of the various array indexed variables are always placed next to one another in memory. For example, consider the following:

```
int a[6];
```

When you declare this array, the computer reserves enough memory to hold six variables of type `int`. Moreover, the computer always places these variables one after the other in memory. The computer then remembers the address of indexed variable `a[0]`, but it does not remember the address of any other indexed variable. When your program needs the address of some other indexed variable, the computer calculates the address for this other indexed variable from the address of `a[0]`. For example, if you start at the address of `a[0]` and count past enough memory for three variables of type `int`, then you will be at the address of `a[3]`. To obtain the address of `a[3]`, the computer starts with the address of `a[0]` (which is a number). The computer then adds the number of bytes needed to hold three variables of type `int` to the number for the address of `a[0]`. The result is the address of `a[3]`. This implementation is diagrammed in Display 7.2.

Many of the peculiarities of arrays in C++ can be understood only in terms of these details about memory. For example, in the next Pitfall section, we use these details to explain what happens when your program uses an illegal array index.

[1] A byte consists of 8 bits, but the exact size of a byte is not important to this discussion.

Array Declaration

SYNTAX

```
Type_Name Array_Name[Declared_Size];
```

EXAMPLES

```
int big_array[100];
double a[3];
double b[5];
char grade[10], one_grade;
```

An array declaration, of the form shown, will define *Declared_Size* indexed variables, namely, the indexed variables *Array_Name*[0] through *Array_Name*[*Declared_Size*-1]. Each indexed variable is a variable of type *Type_Name*.

The array a consists of the indexed variables a[0], a[1], and a[2], all of type *double*. The array b consists of the indexed variables b[0], b[1], b[2], b[3], and b[4], also all of type *double*. You can combine array declarations with the declaration of simple variables such as the variable one_grade shown above.

PITFALL Array Index Out of Range

VideoNote
Array Walkthrough

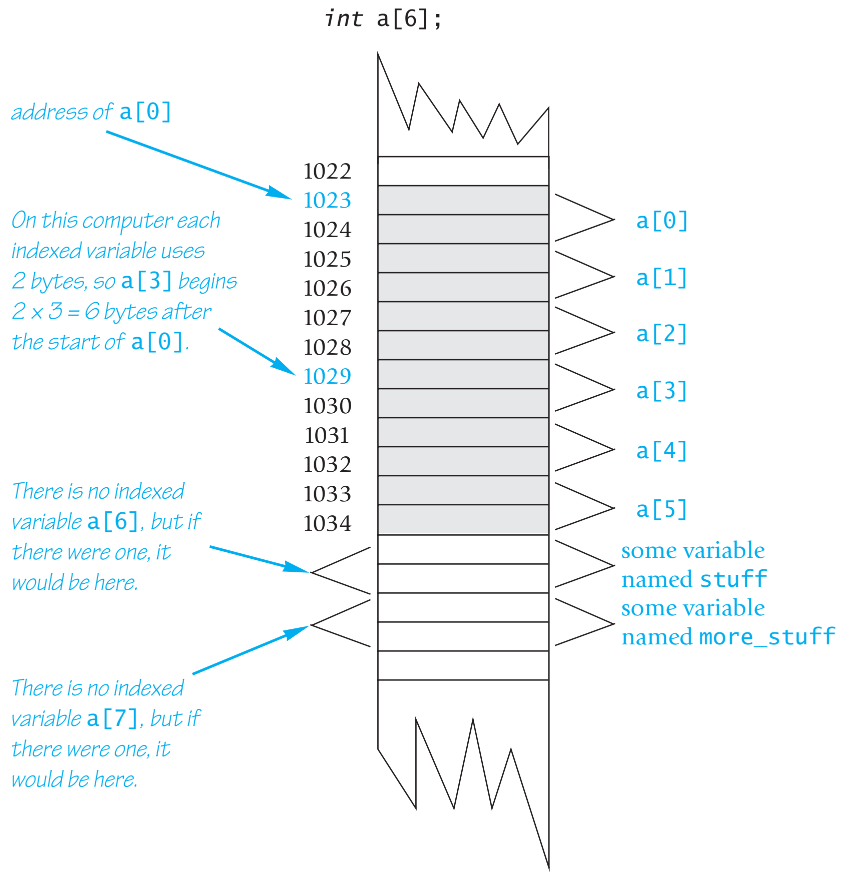
The most common programming error made when using arrays is attempting to reference a nonexistent array index. For example, consider the following array declaration:

```
int a[6];
```

When using the array a, every index expression must evaluate to one of the integers 0 through 5. For example, if your program contains the indexed variable a[i], the i must evaluate to one of the six integers 0, 1, 2, 3, 4, or 5. If i evaluates to anything else, that is an error. When an index expression evaluates to some value other than those allowed by the array declaration, the index is said to be out of range or simply **illegal.** On most systems, the result of an illegal array index is that your program will do something wrong, possibly disastrously wrong, and will do so without giving you any warning.

Attackers have also exploited this type of error to break into software. An out-of-range programming error could potentially compromise the entire system, so take great care to avoid this error. In 2011, the Common Weakness Enumeration (CWE)/SANS Institute identified this type of error as the third most dangerous programmer error.

DISPLAY 7.2 An Array in Memory


int a[6];
address of a[0]
On this computer each indexed variable uses 2 bytes, so a[3] begins 2 x 3 = 6 bytes after the start of a[0].
There is no indexed variable a[6], but if there were one, it would be here.
There is no indexed variable a[7], but if there were one, it would be here.
1022
1023
1024
1025
1026
1027
1028
1029
1030
1031
1032
1033
1034
a[0]
a[1]
a[2]
a[3]
a[4]
a[5]
some variable named stuff
some variable named more_stuff

For example, suppose your system is typical, the array a is declared as shown, and your program contains the following:

```
a[i] = 238;
```

Now, suppose the value of i, unfortunately, happens to be 7. The computer proceeds as if a[7] were a legal indexed variable. The computer calculates the address where a[7] would be (if only there were an a[7]), and places the value 238 in that location in memory. However, there is no indexed variable a[7], and the memory that receives this 238 probably belongs to some other variable, maybe a variable named more_stuff. So the value of more_stuff has been unintentionally changed. The situation is illustrated in Display 7.2.

Array indexes get out of range most commonly at the first or last iteration of a loop that processes the array. So, it pays to carefully check all array processing loops to be certain that they begin and end with legal array indexes.

It may sound simple to keep the array indexes within a valid range. In practice it is more difficult, because there are often subtle or unanticipated ways to change an index variable. For example, consider the following code that inputs some numbers into an array:

```
int num;
int a[10];

cout << "How many numbers? (max of 10)" << endl;
cin >> num;
for (int i = 0; i <= num; i++)

{
    cout << "Enter number " << i << endl;
    cin >> a[i];
}
```

This program suffers from two errors. First, the loop has an off-by-one error. By starting at index 0 and continuing up to and including num the loop will input num+1 numbers instead of num numbers. As long as a value less than ten is entered for num then you might not notice the problem. The program won't crash because the numbers will all be entered with the addition of one extra number which still fits in the array. However, if 10 is entered for num then the eleventh number will be stored at index a[10] which is one off the end of the array. To fix this problem the loop should be written as:

```
for (int i = 0; i < num; i++)
```

Another problem is the lack of input validation. A malicious or mischievous user could enter 100 as the number of values to enter; the loop would then simply execute 100 times and input data well past the end of the array (the program may crash before looping 100 times as numbers past the end of the array could cause mischief). To address this problem we can validate that the user’s input is within valid range:

```
cout << "How many numbers? (max of 10)" << endl;
cin >> num;
cout << num << endl;
if (num <= 10)
{
    for (int i = 0; i < num; i++)
    {
        cout << "Enter number " << i << endl;
        cin >> a[i];
    }
}
```

Even this modified version has the potential for error. If a value is entered for num that exceeds its maximum size then there is the possibility for overflow. For example, on most systems a signed short can only store a number up to +32767. Entering a larger value results in overflow which could store 0 or a negative value in `num`. Although the `for` loop will not run if `num` is zero or negative the program would erroneously pass the `if` statement. We explore this type of error again in Chapter 8. ■

Initializing Arrays

An array can be initialized when it is declared. When initializing the array, the values for the various indexed variables are enclosed in braces and separated with commas. For example,

```
int children[3] = {2, 12, 1};
```

This declaration is equivalent to the following code:

```
int children[3];
children[0] = 2;
children[1] = 12;
children[2] = 1;
```

If you list fewer values than there are indexed variables, those values will be used to initialize the first few indexed variables, and the remaining indexed variables will be initialized to a 0 of the array base type. In this situation, indexed variables not provided with initializers are initialized to 0. However, arrays with no initializers and other variables declared within a function definition, including the `main` function of a program, are not initialized. Although array indexed variables (and other variables) may sometimes be automatically initialized to 0, you cannot and should not count on it.

If you initialize an array when it is declared, you can omit the size of the array, and the array will automatically be declared to have the minimum size needed for the initialization values. For example, the following declaration

```
int b[ ] = {5, 12, 11};
```

is equivalent to

```
int b[3] = {5, 12, 11};
```

■ PROGRAMMING TIP C++11 Range-Based `for` Statement

VideoNote
Range-Based For Loop

C++11 includes a new type of `for` loop, the range-based `for` loop, that simplifies iteration over every element in an array. The syntax is shown below:

```
for (datatype varname : array)
{
    // varname is successively set to each element in the array
}
```

For example:

```
int arr[] = {2, 4, 6, 8};
for (int x : arr)
    cout << x;
cout << endl;
```

This will output: 2468

When defining the variable that will iterate through the array we can use the same modifiers that are available when defining a parameter for a function. The example we used above for variable `x` is equivalent to pass-by-value. If we change `x` inside the loop it doesn't change the array. We could define `x` as pass by reference using `&` and then changes to `x` will be made to the array. We could also use `const` to indicate that the variable can't be changed. The example below increments every element in the array and then outputs them. We used the `auto` datatype in the output loop to automatically determine the type of element inside the array.

```
int arr[] = {2, 4, 6, 8};
for (int& x : arr)
    x++;
for (auto x : arr)
    cout << x;
cout << endl;
```

This will output: 3579. The range-based `for` loop is especially convenient when iterating over vectors, which are introduced in Chapter 8, and iterating over containers, which are discussed in Chapter 18.

SELF-TEST EXERCISES

1. Describe the difference in the meaning of `int a[5]` and the meaning of `a[4]`. What is the meaning of the `[5]` and `[4]` in each case?

2. In the array declaration

   ```
   double score[5];
   ```

   state the following:

   a. The array name
   b. The base type
   c. The declared size of the array
   d. The range of values that an index for this array can have
   e. One of the indexed variables (or elements) of this array

3. Identify any errors in the following array declarations.

   a. `int x[4] = { 8, 7, 6, 4, 3 };`

   b. `int x[ ] = { 8, 7, 6, 4 };`

   c. `const int SIZE = 4;`

   d. `int x[SIZE];`

4. What is the output of the following code?

```
char symbol[3] = {'a', 'b', 'c'};

for (int index = 0; index < 3; index++)
    cout << symbol[index];
```

5. What is the output of the following code?

```
double a[3] = {1.1, 2.2, 3.3};
cout << a[0] << " " << a[1] << " " << a[2] << endl;
a[1] = a[2];
cout << a[0] << " " << a[1] << " " << a[2] << endl;
```

6. What is the output of the following code?

```
int i, temp[10];

for (i = 0; i < 10; i++)
    temp[i] = 2 * i;

for (i = 0; i < 10; i++)
    cout << temp[i] << " ";
cout << endl;

for (i = 0; i < 10; i = i + 2)
    cout << temp[i] << " ";
```

7. What is wrong with the following piece of code?

```
int sample_array[10];

for (int index = 1; index <= 10; index++)
    sample_array[index] = 3 * index;
```

8. Suppose we expect the elements of the array `a` to be ordered so that

```
a[0] ≤ a[1] ≤ a[2] ≤ ...
```

   However, to be safe we want our program to test the array and issue a warning in case it turns out that some elements are out of order. The following code is supposed to output such a warning, but it contains a bug. What is it?

```
double a[10];
<Some code to fill the array a goes here.>
```

```
for (int index = 0; index < 10; index++)
    if (a[index] > a[index + 1])
        cout << "Array elements " << index << " and "
             << (index + 1) << " are out of order.";
```

9. Write some C++ code that will fill an array a with 20 values of type *int* read in from the keyboard. You need not write a full program, just the code to do this, but do give the declarations for the array and for all variables.

10. Suppose you have the following array declaration in your program:

    ```
    int your_array[7];
    ```

    Also, suppose that in your implementation of C++, variables of type *int* use 2 bytes of memory. When you run your program, how much memory will this array consume? Suppose that when you run your program, the system assigns the memory address 1000 to the indexed variable `your_array[0]`. What will be the address of the indexed variable `your_array[3]`?

7.2 ARRAYS IN FUNCTIONS

You can use both array indexed variables and entire arrays as arguments to functions. We first discuss array indexed variables as arguments to functions.

Indexed Variables as Function Arguments

An indexed variable can be an argument to a function in exactly the same way that any variable can be an argument. For example, suppose a program contains the following declarations:

```
int i, n, a[10];
```

If `my_function` takes one argument of type *int*, then the following is legal:

```
my_function(n);
```

Since an indexed variable of the array a is also a variable of type *int*, just like n, the following is equally legal:

```
my_function(a[3]);
```

There is one subtlety that does apply to indexed variables used as arguments. For example, consider the following function call:

```
my_function(a[i]);
```

If the value of i is 3, then the argument is a[3]. On the other hand, if the value of i is 0, then this call is equivalent to the following:

```
my_function(a[0]);
```

The indexed expression is evaluated in order to determine exactly which indexed variable is given as the argument.

Display 7.3 contains an example of indexed variables used as function arguments. The program shown gives five additional vacation days to each of three employees in a small business. The program is extremely simple, but it does illustrate how indexed variables are used as arguments to functions. Notice the function `adjust_days`. This function has a formal parameter called `old_days` that is of type *`int`*. In the main body of the program, this function is called with the argument `vacation[number]` for various values of `number`. Notice that there was nothing special about the formal parameter `old_days`. It is just an ordinary formal parameter of type *`int`*, which is the base type of the array `vacation`. In Display 7.3 the indexed variables are call-by-value arguments. The same remarks apply to call-by-reference arguments. An indexed variable can be a call-by-value argument or a call-by-reference argument.

DISPLAY 7.3 Indexed Variable as an Argument *(part 1 of 2)*

```
//Illustrates the use of an indexed variable as an argument.
//Adds 5 to each employee's allowed number of vacation days.
#include <iostream>
const int NUMBER_OF_EMPLOYEES = 3;

int adjust_days(int old_days);
//Returns old_days plus 5.

int main( )
{
    using namespace std;
    int vacation[NUMBER_OF_EMPLOYEES], number;
    cout << "Enter allowed vacation days for employees 1"
         << " through " << NUMBER_OF_EMPLOYEES << ":\n";
    for (number = 1; number <= NUMBER_OF_EMPLOYEES; number++)
        cin >> vacation[number - 1];
    for (number = 0; number < NUMBER_OF_EMPLOYEES; number++)
        vacation[number] = adjust_days(vacation[number]);
    cout << "The revised number of vacation days are:\n";
    for (number = 1; number <= NUMBER_OF_EMPLOYEES; number++)
        cout << "Employee number " << number
             << " vacation days = " << vacation[number-1] << endl;
    return 0;
}

int adjust_days(int old_days)
{
    return (old_days + 5);
}
```

(continued)

DISPLAY 7.3 Indexed Variable as an Argument *(part 2 of 2)*

Sample Dialogue

```
Enter allowed vacation days for employees 1 through 3:
10 20 5
The revised number of vacation days are:
Employee number 1 vacation days = 15
Employee number 2 vacation days = 25
Employee number 3 vacation days = 10
```

SELF-TEST EXERCISES

11. Consider the following function definition:

```
void tripler(int& n)
{
    n = 3*n;
}
```

Which of the following are acceptable function calls?

```
int a[3] = {4, 5, 6}, number = 2;
tripler(number);
tripler(a[2]);
tripler(a[3]);
tripler(a[number]);
tripler(a);
```

12. What (if anything) is wrong with the following code? The definition of `tripler` is given in Self-Test Exercise 11.

```
int b[5] = {1, 2, 3, 4, 5};
for (int i = 1; i <= 5; i++)
    tripler(b[i]);
```

Entire Arrays as Function Arguments

A function can have a formal parameter for an entire array so that when the function is called, the argument that is plugged in for this formal parameter is an entire array. However, a formal parameter for an entire array is neither a call-by-value parameter nor a call-by-reference parameter; it is a new kind of formal parameter referred to as an **array parameter.** Let's start with an example.

VideoNote
Passing Arrays to Functions

The function defined in Display 7.4 has one array parameter, a, which will be replaced by an entire array when the function is called. It also has one ordinary call-by-value parameter (size) that is assumed to be an integer value equal to the size of the array. This function fills its array argument (that is, fills all the array's indexed variables) with values typed in from the keyboard, and then the function outputs a message to the screen telling the index of the last array index used.

The formal parameter *int* a[ ] is an array parameter. The square brackets, with no index expression inside, are what C++ uses to indicate an array parameter. An array parameter is not quite a call-by-reference parameter, but for most practical purposes it behaves very much like a call-by-reference parameter. Let's go through this example in detail to see how an array argument works in this case. (An **array argument** is, of course, an array that is plugged in for an array parameter, such as a[ ].)

When the function fill_up is called it must have two arguments: The first gives an array of integers, and the second should give the declared size of the array. For example, the following is an acceptable function call:

```
int score[5], number_of_scores = 5;
fill_up(score, number_of_scores);
```

This call to fill_up will fill the array score with five *int*egers typed in at the keyboard. Notice that the formal parameter a[ ] (which is used in the function declaration and the heading of the function definition) is given with square brackets, but no index expression. (You may insert a number inside the square brackets for an array parameter, but the compiler will simply

DISPLAY 7.4 Function with an Array Parameter

Function Declaration

```
void fill_up(int a[], int size);
//Precondition: size is the declared size of the array a.
//The user will type in size integers.
//Postcondition: The array a is filled with size integers
//from the keyboard.
```

Function Definition

```
//Uses iostream:
void fill_up(int a[], int size)
{
    using namespace std;
    cout << "Enter " << size << " numbers:\n";
    for (int i = 0; i < size; i++)
        cin >> a[i];
    size--;
    cout << "The last array index used is " << size << endl;
}
```

ignore the number, so we do not use such numbers in this book.) On the other hand, the argument given in the function call (`score` in this example) is given without any square brackets or any index expression. What happens to the array argument `score` in this function call? Very loosely speaking, the argument `score` is plugged in for the formal array parameter `a` in the body of the function, and then the function body is executed. Thus, the function call

```
fill_up(score, number_of_scores);
```

is equivalent to the following code:

```
{
    using namespace std;                5  is the value of
    size = 5;  <--------------------    number_of_scores
    cout << "Enter " << size << " numbers:\n";
    for (int i = 0; i < size; i++)
        cin >> score[i];
    size--;
    cout << "The last array index used is " << size << endl;
}
```

The formal parameter `a` is a different kind of parameter from the ones we have seen before now. The formal parameter `a` is merely a placeholder for the argument `score`. When the function `fill_up` is called with `score` as the array argument, the computer behaves as if `a` were replaced with the corresponding argument `score`. *When an array is used as an argument in a function call, any action that is performed on the array parameter is performed on the array argument, so the values of the indexed variables of the array argument can be changed by the function.* If the formal parameter in the function body is changed (for example, with a `cin` statement), then the array argument will be changed.

So far it looks like an array parameter is simply a call-by-reference parameter for an array. That is close to being true, but an array parameter is slightly different from a call-by-reference parameter. To help explain the difference, let's review some details about arrays.

Arrays in memory

Recall that an array is stored as a contiguous chunk of memory. For example, consider the following declaration for the array `score`:

```
int score[5];
```

When you declare this array, the computer reserves enough memory to hold five variables of type *`int`*, which are stored one after the other in the computer's memory. The computer does not remember the addresses of each of these five indexed variables; it remembers only the address of indexed variable `score[0]`. For example, when your program needs `score[3]`, the computer calculates the address of `score[3]` from the address of `score[0]`. The computer knows that `score[3]` is located three *`int`* variables past `score[0]`. Thus, to obtain the address of `score[3]`, the computer takes the address of `score[0]` and adds a number that represents the amount of memory used by three *`int`* variables; the result is the address of `score[3]`.

Array argument

Viewed this way, an array has three parts: the address (location in memory) of the first indexed variable, the base type of the array (which determines how much memory each indexed variable uses), and the size of the array (that is, the number of indexed variables). When an array is used as an array argument to a function, only the first of these three parts is given to the function. When an array argument is plugged in for its corresponding formal parameter, all that is plugged in is the address of the array's first indexed variable. The base type of the array argument must match the base type of the formal parameter, so the function also knows the base type of the array. However, the array argument does not tell the function the size of the array. When the code in the function body is executed, the computer knows where the array starts in memory and how much memory each indexed variable uses, but(unless you make special provisions) it does not know how many indexed variables the array has. That is why it is critical that you always have another `int` argument telling the function the size of the array. That is also why an array parameter is not the same as a call-by-reference parameter. You can think of an array parameter as a weak form of call-by-reference parameter in which everything about the array is told to the function except for the size of the array.[2]

Different size array arguments can be plugged in for the same array parameter

These array parameters may seem a little strange, but they have at least one very nice property as a direct result of their seemingly strange definition. This advantage is best illustrated by again looking at our example of the function `fill_up` given in Display 7.4. *That same function can be used to fill an array of any size*, as long as the base type of the array is `int`. For example, suppose you have the following array declarations:

```
int score[5], time[10];
```

The first of the following calls to `fill_up` fills the array `score` with five values and the second fills the array `time` with ten values:

```
fill_up(score, 5);
fill_up(time, 10);
```

You can use the same function for array arguments of different sizes because the size is a separate argument.

The *const* Parameter Modifier

When you use an array argument in a function call, the function can change the values stored in the array. This is usually fine. However, in a complicated function definition, you might write code that inadvertently changes one or more of the values stored in an array, even though the array should not be changed at all. As a precaution, you can tell the compiler that you do not

[2] If you have heard of pointers, this will sound like pointers, and indeed an array argument is passed by passing a pointer to its first (zeroth) index variable. We will discuss this in Chapter 9. If you have not yet learned about pointers, you can safely ignore this footnote.

intend to change the array argument, and the computer will then check to make sure your code does not inadvertently change any of the values in the array. To tell the compiler that an array argument should not be changed by your function, you insert the modifier *const* before the array parameter for that argument position. An array parameter that is modified with a *const* is called a **constant array parameter.**

For example, the following function outputs the values in an array but does not change the values in the array:

```
void show_the_world(int a[ ], int size_of_a)
//Precondition: size_of_a is the declared size of the array a.
//All indexed variables of a have been given values.
//Postcondition: The values in a have been written
//to the screen.
{
    cout << "The array contains the following values:\n";
    for (int i = 0; i < size_of_a; i++)
        cout << a[i] << " ";
    cout << endl;
}
```

This function will work fine. However, as an added safety measure you can add the modifier *const* to the function heading as follows:

```
void show_the_world(const int a[ ], int size_of_a)
```

With the addition of this modifier *const*, the computer will issue an error message if your function definition contains a mistake that changes any of the

Array Formal Parameters and Arguments

An argument to a function may be an entire array, but an argument for an entire array is neither a call-by-value argument nor a call-by-reference argument. It is a new kind of argument known as an **array argument.** When an array argument is plugged in for an **array parameter,** all that is given to the function is the address in memory of the first indexed variable of the array argument (the one indexed by 0). The array argument does not tell the function the size of the array. Therefore, when you have an array parameter to a function, you normally must also have another formal parameter of type *int* that gives the size of the array (as in the example below).

An array argument is like a call-by-reference argument in the following way: If the function body changes the array parameter, then when the function is called, that change is actually made to the array argument. Thus, a function can change the values of an array argument (that is, can change the values of its indexed variables).

(continued)

The syntax for a function declaration with an array parameter is as follows:

SYNTAX

```
Type_Returned Function_Name(..., Base_Type Array_
Name[],...);
```

EXAMPLE

```
void sum_array(double& sum, double a[ ], int size);
```

values in the array argument. For example, the following is a version of the function show_the_world that contains a mistake that inadvertently changes the value of the array argument. Fortunately, this version of the function definition includes the modifier *const*, so that an error message will tell us that the array a is changed. This error message will help to explain the mistake:

```
void show_the_world(const int a[ ], int size_of_a)
//Precondition: size_of_a is the declared size of the array a.
//All indexed variables of a have been given values.
//Postcondition: The values in a have been written
//to the screen.
{
    cout << "The array contains the following values:\n";
    for (int i = 0; i < size_of_a; a[i]++)
        cout << a[i] << " ";
    cout << endl;
}
```

Mistake, but the compiler will not catch it unless you use the const modifier.

If we had not used the *const* modifier in this function definition and if we made the mistake shown, the function would compile and run with no error messages. However, the code would contain an infinite loop that continually increments a[0] and writes its new value to the screen.

The problem with this incorrect version of show_the_world is that the wrong item is incremented in the *for* loop. The indexed variable a[i] is incremented, but it should be the index i that is incremented. In this incorrect version, the index i starts with the value 0 and that value is never changed. But a[i], which is the same as a[0], is incremented. When the indexed variable a[i] is incremented, that changes a value in the array, and since we included the modifier *const*, the computer will issue a warning message. That error message should serve as a clue to what is wrong.

You normally have a function declaration in your program in addition to the function definition. When you use the *const* modifier in a function definition, you must also use it in the function declaration so that the function heading and the function declaration are consistent.

The modifier *const* can be used with any kind of parameter, but it is normally used only with array parameters and call-by-reference parameters for classes, which are discussed in Chapter 11.

PITFALL Inconsistent Use of *const* Parameters

The *const* parameter modifier is an all-or-nothing proposition. If you use it for one array parameter of a particular type, then you should use it for every other array parameter that has that type and that is not changed by the function. The reason has to do with function calls within function calls. Consider the definition of the function `show_difference`, which is given below along with the declaration of a function used in the definition:

```
double compute_average(int a[ ], int number_used);
//Returns the average of the elements in the first number_used
//elements of the array a. The array a is unchanged.

void show_difference(const int a[ ], int number_used)
{
    double average = compute_average(a, number_used);
    cout << "Average of the " << number_used
         << " numbers = " << average << endl
         << "The numbers are:\n";
    for (int index = 0; index < number_used; index++)
        cout << a[index] << " differs from average by "
             << (a[index] - average) << endl;
}
```

This code will generate an error message or warning message with most compilers. The function `compute_average` does not change its parameter `a`. However, when the compiler processes the function definition for `show_difference`, it will think that `compute_average` does (or at least might) change the value of its parameter `a`. This is because, when it is translating the function definition for `show_difference`, all the compiler knows about the function `compute_average` is the function declaration for `compute_average`, and the function declaration does not contain a *const* to tell the compiler that the parameter `a` will not be changed. Thus, if you use *const* with the parameter `a` in the function `show_difference`, then you should also use the modifier *const* with the parameter `a` in the function `compute_average`. The function declaration for `compute_average` should be as follows:

```
double compute_average(const int a[ ], int number_used);
```
■

Functions That Return an Array

A function may not return an array in the same way that it returns a value of type *int* or *double*. There is a way to obtain something more or less equivalent to a function that returns an array. The thing to do is to return a pointer to the array. However, we have not yet covered pointers. We will discuss returning a pointer to an array when we discuss the interaction of arrays and pointers in Chapter 9. Until then, you have no way to write a function that returns an array.

CASE STUDY Production Graph

In this case study we use arrays in the top-down design of a program. We use both indexed variables and entire arrays as arguments to the functions for subtasks.

Problem Definition

The Apex Plastic Spoon Manufacturing Company has commissioned us to write a program that will display a bar graph showing the productivity of each of its four manufacturing plants for any given week. Plants keep separate production figures for each department, such as the teaspoon department, soup spoon department, plain cocktail spoon department, colored cocktail spoon department, and so forth. Moreover, each plant has a different number of departments. For example, only one plant manufactures colored cocktail spoons. The input is entered plant-by-plant and consists of a list of numbers giving the production for each department in that plant. The output will consist of a bar graph in the following form:

```
Plant #1 **********
Plant #2 *************
Plant #3 *******************
Plant #4 *****
```

Each asterisk represents 1000 units of output.

We decide to read in the input separately for each department in a plant. Since departments cannot produce a negative number of spoons, we know that the production figure for each department will be nonnegative. Hence, we can use a negative number as a sentinel value to mark the end of the production numbers for each plant.

Since output is in units of 1000, it must be scaled by dividing it by 1000. This presents a problem since the computer must display a whole number of asterisks. It cannot display 1.6 asterisks for 1600 units. We will thus round to the nearest 1000th. Thus, 1600 will be the same as 2000 and will produce two asterisks. A precise statement of the program's input and output is as follows.

Input

There are four manufacturing plants numbered 1 through 4. The following input is given for each of the four plants: a list of numbers giving the production for each department in that plant. The list is terminated with a negative number that serves as a sentinel value.

Output

A bar graph showing the total production for each plant. Each asterisk in the bar graph equals 1000 units. The production of each plant is rounded to the nearest 1000 units.

Analysis of the Problem

We will use an array called `production`, which will hold the total production for each of the four plants. In C++, array indexes always start with `0`. But since the plants are numbered `1` through `4`, rather than `0` through `3`, we will not use the plant number as the array index. Instead, we will place the total production for plant number `n` in the indexed variable `production[n−1]`. The total output for plant number 1 will be held in `production[0]`, the figures for plant 2 will be held in `production[1]`, and so forth.

Since the output is in thousands of units, the program will scale the values of the array elements. If the total output for plant number 3 is 4040 units, then the value of `production[2]` will initially be set to `4040`. This value of `4040` will then be scaled to `4` so that the value of `production[2]` is changed to `4`, and four asterisks will be output in the graph to represent the output for plant number 3.

Subtasks

The task for our program can be divided into the following subtasks:

- `input_data`: Read the input data for each plant and set the value of the indexed variable `production[plant_number-1]` equal to the total production for that plant, where `plant_number` is the number of the plant.
- `scale: For each plant_number`, change the value of the indexed variable `production[plant_number - 1]` to the correct number of asterisks.
- `graph`: Output the bar graph.

The entire array `production` will be an argument for the functions that carry out these subtasks. As is usual with an array parameter, this means we must have an additional formal parameter for the size of the array, which in this case is the same as the number of plants. We will use a defined constant for the number of plants, and this constant will serve as the size of the array `production`. The `main` part of our program, together with the function declarations for the functions that perform the subtasks and the defined constant for the number of plants, is shown in Display 7.5. Notice that, since there is no reason to change the array parameter to the function `graph`, we have made that array parameter a constant parameter by adding the *`const`* parameter modifier. The material in Display 7.5 is the outline for our program, and if it is in a separate file, that file can be compiled so that we can check for any syntax errors in this outline before we go on to define the functions corresponding to the function declarations shown.

Having compiled the file shown in Display 7.5, we are ready to design the implementation of the functions for the three subtasks. For each of these three functions, we will design an algorithm, write the code for the function, and test the function before we go on to design the next function.

Algorithm Design for input_data

The function declaration and descriptive comment for the function input_data is shown in Display 7.5. As indicated in the body of the main part of our program (also shown in Display 7.5), when input_data is called, the formal array parameter a will be replaced with the array production, and since the last plant number is the same as the number of plants, the formal parameter last_plant_number will be replaced by NUMBER_OF_PLANTS. The algorithm for input_data is straightforward:

```
For plant_number equal to each of 1, 2, through last_plant_number do
the following:

Read in all the data for plant whose number is plant_number.

Sum the numbers.

Set production[plant_number -1] equal to that total.
```

Coding for input_data

The algorithm for the function input_data translates to the following code:

```
//Uses iostream:
void input_data(int a[ ], int last_plant_number)
{
    using namespace std;
    for (int plant_number = 1;
        plant_number <= last_plant_number; plant_number++)
    {
        cout << endl
             << "Enter production data for plant number "
             << plant_number << endl;
        get_total(a[plant_number - 1]);
    }
}
```

The code is routine since all the work is done by the function get_total, which we still need to design. But before we move on to discuss the function get_total, let's observe a few things about the function input_data. Notice that we store the figures for plant number plant_number in the indexed variable with index plant_number-1; this is because arrays always start with index 0, while the plant numbers start with 1. Also, notice that we use an indexed variable for the argument to the function get_total. The function get_total really does all the work for the function input_data.

The function get_total does all the input work for one plant. It reads the production figures for that plant, sums the figures, and stores the total in the indexed variable for that plant. But get_total does not need to know that its argument is an indexed variable. To a function such as get_total, an indexed variable is just like any other variable of type *int*. Thus, get_total will have an ordinary call-by-reference parameter of type *int*. That means that

DISPLAY 7.5 Outline of the Graph Program

```
//Reads data and displays a bar graph showing productivity for each plant.
#include <iostream>
const int NUMBER_OF_PLANTS = 4;


void input_data(int a[], int last_plant_number);
//Precondition: last_plant_number is the declared size of the array a.
//Postcondition: For plant_number = 1 through last_plant_number:
//a[plant_number - 1] equals the total production for plant number plant_number.

void scale(int a[], int size);
//Precondition: a[0] through a[size - 1] each has a nonnegative value.
//Postcondition: a[i] has been changed to the number of 1000s (rounded to
//an integer) that were originally in a[i], for all i such that 0 <= i <= size - 1.

void graph(const int asterisk_count[], int last_plant_number);
//Precondition: asterisk_count[0] through asterisk_count[last_plant_number - 1]
//have nonnegative values.
//Postcondition: A bar graph has been displayed saying that plant
//number N has produced asterisk_count[N - 1] 1000s of units, for each N such that
//1 <= N <= last_plant_number

int main( )
{
    using namespace std;
    int production[NUMBER_OF_PLANTS];

    cout << "This program displays a graph showing\n"
         << "production for each plant in the company.\n";

    input_data(production, NUMBER_OF_PLANTS);
    scale(production, NUMBER_OF_PLANTS);
    graph(production, NUMBER_OF_PLANTS);

    return 0;
}

```

`get_total` is just an ordinary input function like others that we have seen before we discussed arrays. The function `get_total` reads in a list of numbers ended with a sentinel value, sums the numbers as it reads them in, and sets the value of its argument, which is a variable of type *int*, equal to this sum. There is nothing new to us in the function `get_total`. Display 7.6 shows the function definitions for both `get_total` and `input_data`. The functions are embedded in a simple test program.

Testing `input_data`

Every function should be tested in a program in which it is the only untested function. The function `input_data` includes a call to the function `get_total`. Therefore, we should test `get_total` in a driver program of its own. Once `get_total` has been completely tested, we can use it in a program, like the one in Display 7.6, to test the function `input_data`.

When testing the function `input_data`, we should include tests with all possible kinds of production figures for a plant. We should include a plant that has no production figures (as we did for plant 4 in Display 7.6); we should include a test for a plant with only one production figure (as we did for plant 3 in Display 7.6); and we should include a test for a plant with more than one production figure (as we did for plants 1 and 2 in Display 7.6). We should test for both nonzero and zero production figures, which is why we included a `0` in the input list for plant 2 in Display 7.6.

Algorithm Design for `scale`

The function `scale` changes the value of each indexed variable in the array `production` so that it shows the number of asterisks to print out. Since there should be one asterisk for every 1000 units of production, the value of each indexed variable must be divided by `1000.0`. Then to get a whole number of asterisks, this number is rounded to the nearest integer. This method can be used to scale the values in any array `a` of any size, so the function declaration for `scale`, shown in Display 7.5 and repeated here, is stated in terms of an arbitrary array `a` of some arbitrary size:

```
void scale(int a[ ], int size);
//Precondition: a[0] through a[size - 1] each has a
//nonnegative value.
//Postcondition: a[i] has been changed to the number of 1000s
//(rounded to an integer) that were originally in a[i], for
//all i such that 0 <= i <= size - 1.
```

When the function `scale` is called, the array parameter `a` will be replaced by the array `production`, and the formal parameter `size` will be replaced by `NUMBER_OF_PLANTS` so that the function call looks like the following:

```
scale(production, NUMBER_OF_PLANTS);
```

The algorithm for the function `scale` is as follows:

```
for (int index = 0; index < size; index++)
```

Divide the value of `a[index]` by 1000 and round the result to the nearest whole number; the result is the new value of `a[index]`.

Coding for `scale`

The algorithm for `scale` translates into the C++ code given next, where `round` is a function we still need to define. The function `round` takes one argument of type *`double`* and returns a type *`int`* value that is the integer nearest to its argument; that is, the function `round` will round its argument to the nearest whole number.

DISPLAY 7.6 Test of Function input_data *(part 1 of 3)*

```
//Tests the function input_data.
#include <iostream>
const int NUMBER_OF_PLANTS = 4;

void input_data(int a[], int last_plant_number);
//Precondition: last_plant_number is the declared size of the array a.
//Postcondition: For plant_number = 1 through last_plant_number:
//a[plant_number-1] equals the total production for plant number plant_number.

void get_total(int& sum);
//Reads nonnegative integers from the keyboard and
//places their total in sum.

int main( )
{
    using namespace std;
    int production[NUMBER_OF_PLANTS];
    char ans;

    do
    {
        input_data(production, NUMBER_OF_PLANTS);
        cout << endl
             << "Total production for each"
             << " of plants 1 through 4:\n";
        for (int number = 1; number <= NUMBER_OF_PLANTS; number++)
            cout << production[number - 1] << " ";

        cout << endl
             << "Test Again?(Type y or n and Return): ";
        cin >> ans;
    } while ( (ans != 'N') && (ans != 'n') );

    cout << endl;

    return 0;
}
//Uses iostream:
void input_data(int a[], int last_plant_number)
{
    using namespace std;
    for (int plant_number = 1;
         plant_number <= last_plant_number; plant_number++)
    {
        cout << endl
             << "Enter production data for plant number "
```

(continued)

DISPLAY 7.6 Test of Function input_data *(part 2 of 3)*

```
                  << plant_number << endl;
            get_total(a[plant_number - 1]);
        }
    }


    //Uses iostream:
    void get_total(int& sum)
    {
        using namespace std;
        cout << "Enter number of units produced by each department.\n"
             << "Append a negative number to the end of the list.\n";

        sum = 0;
        int next;
        cin >> next;
        while (next >= 0)
        {
            sum = sum + next;
            cin >> next;
        }

        cout << "Total = " << sum << endl;
    }
```

Sample Dialogue

```
Enter production data for plant number 1
Enter number of units produced by each department.
Append a negative number to the end of the list.
1 2 3 -1
Total = 6

Enter production data for plant number 2
Enter number of units produced by each department.
Append a negative number to the end of the list.
0 2 3 -1
Total = 5

Enter production data for plant number 3
Enter number of units produced by each department.
Append a negative number to the end of the list.
2 -1
Total = 2
```

(continued)

DISPLAY 7.6 Test of Function input_data *(part 3 of 3)*

```
Enter production data for plant number 4
Enter number of units produced by each department.
Append a negative number to the end of the list.
-1
Total = 0

Total production for each of plants 1 through 4:
6 5 2 0
Test Again?(Type y or n and Return): n
```

```
void scale(int a[], int size)
{
    for (int index = 0; index < size; index++)
        a[index] = round(a[index]/1000.0 );
}
```

Notice that we divided by `1000.0`, not by `1000` (without the decimal point). If we had divided by `1000`, we would have performed integer division. For example, `2600/1000` would give the answer `2`, but `2600/1000.0` gives the answer `2.6`. It is true that we want an integer for the final answer after rounding, but we want `2600` divided by `1000` to produce `3`, not `2`, when it is rounded to a whole number.

We now turn to the definition of the function `round`, which rounds its argument to the nearest integer. For example, `round(2.3)` returns `2`, and `round(2.6)` returns `3`. The code for the function `round`, as well as that for `scale`, is given in Display 7.7. The code for `round` may require a bit of explanation.

The function `round` uses the predefined function `floor` from the library with the header file `cmath`. The function `floor` returns the whole number just below its argument. For example, `floor(2.1)` and `floor(2.9)` both return `2`. To see that `round` works correctly, let's look at some examples. Consider `round(2.4)`. The value returned is

```
floor(2.4 + 0.5)
```

which is `floor(2.9)`, and that is `2.0`. In fact, for any number that is greater than or equal to `2.0` and strictly less than `2.5`, that number plus `0.5` will be less than `3.0`, and so `floor` applied to that number plus `0.5` will return `2.0`. Thus, `round` applied to any number that is greater than or equal to `2.0` and strictly less than `2.5` will return `2`. (Since the function declaration for `round` specifies that the type for the value returned is *`int`*, the computed value of `2.0` is type cast to the integer value `2` without a decimal point using *`static_cast<int>`*.)

Now consider numbers greater than or equal to `2.5`, for example, `2.6`. The value returned by the call `round(2.6)` is

```
floor(2.6 + 0.5)
```

DISPLAY 7.7 The Function `scale`

```
//Demonstration program for the function scale.
#include <iostream>
#include <cmath>

void scale(int a[], int size);
//Precondition: a[0] through a[size - 1] each has a nonnegative value.
//Postcondition: a[i] has been changed to the number of 1000s (rounded to
//an integer) that were originally in a[i], for all i such that 0 <= i <= size - 1.

int round(double number);
//Precondition: number >= 0.
//Returns number rounded to the nearest integer.

int main( )
{
    using namespace std;
    int some_array[4], index;
    cout << "Enter 4 numbers to scale: ";
    for (index = 0; index < 4; index++)
        cin >> some_array[index];
    scale(some_array, 4);
    cout << "Values scaled to the number of 1000s are: ";
    for (index = 0; index < 4; index++)
        cout << some_array[index] << " ";
    cout << endl;
    return 0;
}

void scale(int a[], int size)
{
    for (int index = 0; index < size; index++)
        a[index] = round(a[index]/1000.0);
}

//Uses cmath:
int round(double number)
{
    using namespace std;
    return static_cast<int>(floor(number + 0.5));
}
```

Sample Dialogue

```
Enter 4 numbers to scale: 2600 999 465 3501
Values scaled to the number of 1000s are: 3 1 0 4
```

which is floor(3.1) and that is 3.0. In fact, for any number that is greater than or equal to 2.5 and less than or equal to 3.0, that number plus 0.5 will be greater than 3.0. Thus, round called with any number that is greater than or equal to 2.5 and less than or equal to 3.0 will return 3.

Thus, round works correctly for all arguments between 2.0 and 3.0. Clearly, there is nothing special about arguments between 2.0 and 3.0. A similar argument applies to all nonnegative numbers. So, round works correctly for all nonnegative arguments.

Testing scale

Display 7.7 contains a demonstration program for the function scale, but the testing programs for the functions round and scale should be more elaborate than this simple program. In particular, they should allow you to retest the tested function several times rather than just once. We will not give the complete testing programs, but you should first test round (which is used by scale) in a driver program of its own, and then test scale in a driver program. The program to test round should test arguments that are 0, arguments that round up (like 2.6), and arguments that round down like 2.3. The program to test scale should test a similar variety of values for the elements of the array.

The Function graph

The complete program for producing the desired bar graph is shown in Display 7.8. We have not taken you step-by-step through the design of the function graph because it is quite straightforward.

DISPLAY 7.8 Production Graph Program *(part 1 of 3)*

```
//Reads data and displays a bar graph showing productivity for each plant.
#include <iostream>
#include <cmath>
const int NUMBER_OF_PLANTS = 4;

void input_data(int a[], int last_plant_number);
//Precondition: last_plant_number is the declared size of the array a.
//Postcondition: For plant_number = 1 through last_plant_number:
//a[plant_number - 1] equals the total production for plant number plant_number.

void scale(int a[], int size);
//Precondition: a[0] through a[size - 1] each has a nonnegative value.
//Postcondition: a[i] has been changed to the number of 1000s (rounded to
//an integer) that were originally in a[i], for all i such that 0 <= i <= size - 1.

void graph(const int asterisk_count[], int last_plant_number);
//Precondition: asterisk_count[0] through asterisk_count[last_plant_number - 1]
//have nonnegative values.
//Postcondition: A bar graph has been displayed saying that plant
//number N has produced asterisk_count[N - 1] 1000s of units, for each N such that
//1 <= N <= last_plant_number
```

(continued)

DISPLAY 7.8 Production Graph Program *(part 2 of 3)*

```
void get_total(int& sum);
//Reads nonnegative integers from the keyboard and
//places their total in sum.
int round(double number);
//Precondition: number >= 0.
//Returns number rounded to the nearest integer.

void print_asterisks(int n);
//Prints n asterisks to the screen.

int main( )
{
    using namespace std;
    int production[NUMBER_OF_PLANTS];

    cout << "This program displays a graph showing\n"
         << "production for each plant in the company.\n";
    input_data(production, NUMBER_OF_PLANTS);
    scale(production, NUMBER_OF_PLANTS);
    graph(production, NUMBER_OF_PLANTS);
    return 0;
}

//Uses iostream:
void input_data(int a[], int last_plant_number)
<The rest of the definition of input_data is given in Display 7.6.>

//Uses iostream:
void get_total(int& sum)
<The rest of the definition of get_total is given in Display 7.6.>

void scale(int a[], int size)
<The rest of the definition of scale is given in Display 7.7.>

//Uses cmath:
int round(double number)
<The rest of the definition of round is given in Display 7.7.>

//Uses iostream:
void graph(const int asterisk_count[], int last_plant_number)
{
    using namespace std;
    cout << "\nUnits produced in thousands of units:\n";
    for (int plant_number = 1;
         plant_number <= last_plant_number; plant_number++)
    {
        cout << "Plant #" << plant_number << " ";
        print_asterisks(asterisk_count[plant_number - 1]);
        cout << endl;
    }
}
```

(continued)

DISPLAY 7.8 Production Graph Program *(part 3 of 3)*

```
//Uses iostream:
void print_asterisks(int n)
{
    using namespace std;
    for (int count = 1; count <= n; count++)
        cout << "*";
}
```

Sample Dialogue

```
This program displays a graph showing
production for each plant in the company.
Enter production data for plant number 1
Enter number of units produced by each department.
Append a negative number to the end of the list.
2000 3000 1000 -1
Total = 6000

Enter production data for plant number 2
Enter number of units produced by each department.
Append a negative number to the end of the list.
2050 3002 1300 -1
Total = 6352

Enter production data for plant number 3
Enter number of units produced by each department.
Append a negative number to the end of the list.
5000 4020 500 4348 -1
Total = 13868

Enter production data for plant number 4
Enter number of units produced by each department.
Append a negative number to the end of the list.
2507 6050 1809 -1
Total = 10366

Units produced in thousands of units: Plant #1 ******
Plant #2 ******
Plant #3 **************
Plant #4 **********
```

SELF-TEST EXERCISES

13. Write a function definition for a function called `one_more`, which has a formal parameter for an array of integers and increases the value of each array element by one. Add any other formal parameters that are needed.

14. Consider the following function definition:

```
void too2(int a[ ], int how_many)
{
    for (int index = 0; index < how_many; index++)
        a[index] = 2;
}
```

Which of the following are acceptable function calls?

```
int my_array[29];
too2(my_array, 29);
too2(my_array, 10);
too2(my_array, 55);
"Hey too2. Please, come over here."
int your_array[100];
too2(your_array, 100);
too2(my_array[3], 29);
```

15. Insert *const* before any of the following array parameters that can be changed to constant array parameters:

```
void output(double a[], int size);
//Precondition: a[0] through a[size - 1] have values.
//Postcondition: a[0] through a[size - 1] have been
//written out.

void drop_odd(int a[ ], int size);
//Precondition: a[0] through a[size - 1] have values.
//Postcondition: All odd numbers in a[0] through
//a[size - 1] have been changed to 0.
```

16. Write a function named `out_of_order` that takes as parameters an array of *double*s and an *int* parameter named `size` and returns a value of type *int*. This function will test this array for being out of order, meaning that the array violates the following condition:

```
a[0] <= a[1] <= a[2] <= ...
```

The function returns −1 if the elements are not out of order; otherwise, it will return the index of the first element of the array that is out of order. For example, consider the declaration

```
double a[10] = {1.2, 2.1, 3.3, 2.5, 4.5,
                7.9, 5.4, 8.7, 9.9, 1.0};
```

In this array, `a[2]` and `a[3]` are the first pair out of order, and `a[3]` is the first element out of order, so the function returns `3`. If the array were sorted, the function would return `-1`.

7.3 PROGRAMMING WITH ARRAYS

Never trust to general impressions, my boy, but concentrate yourself upon details.

SIR ARTHUR CONAN DOYLE, *A Case of Identity (Sherlock Holmes)*

In this section we discuss partially filled arrays and give a brief introduction to sorting and searching of arrays. This section includes no new material about the C++ language, but does include more practice with C++ array parameters.

Partially Filled Arrays

Often the exact size needed for an array is not known when a program is written, or the size may vary from one run of the program to another. One common and easy way to handle this situation is to declare the array to be of the largest size the program could possibly need. The program is then free to use as much or as little of the array as is needed.

Partially filled arrays require some care. The program must keep track of how much of the array is used and must not reference any indexed variable that has not been given a value. The program in Display 7.9 illustrates this point. The program reads in a list of golf scores and shows how much each score differs from the average. This program will work for lists as short as one score, as long as ten scores, and for any length in between. The scores are stored in the array `score`, which has ten indexed variables, but the program uses only as much of the array as it needs. The variable `number_used` keeps track of how many elements are stored in the array. The elements (that is, the scores) are stored in positions `score[0]` through `score[number_used - 1]`.

The details are very similar to what they would be if `number_used` were the declared size of the array and the entire array were used. In particular, the variable `number_used` usually must be an argument to any function that manipulates the partially filled array. Since the argument `number_used` (when used properly) can often ensure that the function will not reference an illegal array index, this sometimes (but not always) eliminates the need for an argument that gives the declared size of the array. For example, the functions `show_difference` and `compute_average` use the argument `number_used` to ensure that only legal array indexes are used. However, the function `fill_array` needs to know the maximum declared size for the array so that it does not overfill the array.

DISPLAY 7.9 Partially Filled Array *(part 1 of 2)*

```
//Shows the difference between each of a list of golf scores and their average.
#include <iostream>
const int MAX_NUMBER_SCORES = 10;

void fill_array(int a[], int size, int& number_used);
//Precondition: size is the declared size of the array a.
//Postcondition: number_used is the number of values stored in a.
//a[0] through a[number_used - 1] have been filled with
//nonnegative integers read from the keyboard.

double compute_average(const int a[], int number_used);
//Precondition: a[0] through a[number_used - 1] have values; number_used> 0.
//Returns the average of numbers a[0] through a[number_used - 1].

void show_difference(const int a[],int number_used);
//Precondition: The first number_used indexed variables of a have values.
//Postcondition: Gives screen output showing how much each of the first
//number_used elements of a differs from their average.

int main( )
{
    using namespace std;
    int score[MAX_NUMBER_SCORES], number_used;

    cout << "This program reads golf scores and shows\n"
         << "how much each differs from the average.\n";

    cout << "Enter golf scores:\n";
    fill_array(score, MAX_NUMBER_SCORES, number_used);
    show_difference(score, number_used);

    return 0;
}
//Uses iostream:
void fill_array(int a[], int size, int& number_used)
{
    using namespace std;
    cout << "Enter up to " << size << " nonnegative whole numbers.\n"
         << "Mark the end of the list with a negative number.\n";
    int next, index = 0;
    cin >> next;
    while ((next >= 0) && (index < size))
    {
        a[index] = next;
        index++;
        cin >> next;
    }

    number_used = index;
}
```

(continued)

DISPLAY 7.9 Partially Filled Array *(part 2 of 2)*

```
double compute_average(const int a[], int number_used)
{
    double total = 0;
    for (int index = 0; index < number_used; index++)
        total = total + a[index];
    if (number_used> 0)
    {
        return (total/number_used);
    }
    else
    {
        using namespace std;
        cout << "ERROR: number of elements is 0 in compute_average.\n"
             << "compute_average returns 0.\n";
        return 0;
    }
}

void show_difference(const int a[], int number_used)
{
    using namespace std;
    double average = compute_average(a, number_used);
    cout << "Average of the " << number_used
         << " scores = " << average << endl
         << "The scores are:\n";
    for (int index = 0; index < number_used; index++)
    cout << a[index] << " differs from average by "
         << (a[index] - average) << endl;
}
```

Sample Dialogue

```
This program reads golf scores and shows
how much each differs from the average.
Enter golf scores:
Enter up to 10 nonnegative whole numbers.
Mark the end of the list with a negative number.
69 74 68 -1

Average of the 3 scores = 70.3333
The scores are:
69 differs from average by -1.33333
74 differs from average by 3.66667
68 differs from average by -2.33333
```

PROGRAMMING TIP Do Not Skimp on Formal Parameters

Notice the function `fill_array` in Display 7.9. When `fill_array` is called, the declared array size `MAX_NUMBER_SCORES` is given as one of the arguments, as shown in the following function call from Display 7.9:

```
fill_array(score, MAX_NUMBER_SCORES, number_used);
```

You might protest that `MAX_NUMBER_SCORES` is a globally defined constant and so could be used in the definition of `fill_array` without the need to make it an argument. You would be correct, and if we did not use `fill_array` in any program other than the one in Display 7.9, we could get by without making `MAX_NUMBER_SCORES` an argument to `fill_array`. However, `fill_array` is a generally useful function that you may want to use in several different programs. We do in fact also use the function `fill_array` in the program in Display 7.10, discussed in the next subsection. In the program in Display 7.10, the argument for the declared array size is a different named global constant. If we had written the global constant `MAX_NUMBER_SCORES` into the body of the function `fill_array`, we would not have been able to reuse the function in the program in Display 7.10. ■

PROGRAMMING EXAMPLE Searching an Array

A common programming task is to search an array for a given value. For example, the array may contain the student numbers for all students in a given course. To tell whether a particular student is enrolled, the array is searched to see if it contains the student's number. The program in Display 7.10 fills an array and then searches the array for values specified by the user. A real application program would be much more elaborate, but this shows all the essentials of the sequential search algorithm. The sequential search algorithm is the most straightforward searching algorithm you could imagine: The program looks at the array elements in the order first to last to see if the target number is equal to any of the array elements.

In Display 7.10, the function `search` is used to search the array. When searching an array, you often want to know more than simply whether or not the target value is in the array. If the target value is in the array, you often want to know the index of the indexed variable holding that target value, since the index may serve as a guide to some additional information about the target value. Therefore, we designed the function `search` to return an index giving the location of the target value in the array, provided the target value is, in fact, in the array. If the target value is not in the array, `search` returns `-1`. Let's look at the function `search` in a little more detail.

The function `search` uses a ***`while`*** loop to check the array elements one after the other to see whether any of them equals the target value. The variable

DISPLAY 7.10 Searching an Array *(part 1 of 2)*

```
//Searches a partially filled array of nonnegative integers.
#include <iostream>
const int DECLARED_SIZE = 20;

void fill_array(int a[], int size, int& number_used);
//Precondition: size is the declared size of the array a.
//Postcondition: number_used is the number of values stored in a.
//a[0] through a[number_used - 1] have been filled with
//nonnegative integers read from the keyboard.

int search(const int a[], int number_used, int target);
//Precondition: number_used is <= the declared size of a.
//Also, a[0] through a[number_used - 1] have values.
//Returns the first index such that a[index] == target,
//provided there is such an index; otherwise, returns -1.

int main( )
{
    using namespace std;
    int arr[DECLARED_SIZE], list_size, target;

    fill_array(arr, DECLARED_SIZE, list_size);

    char ans;
    int result;
    do
    {
        cout << "Enter a number to search for: ";
        cin >> target;

        result = search(arr, list_size, target);
        if (result == -1)
            cout << target << " is not on the list.\n";
        else
            cout << target << " is stored in array position "
            << result << endl
            << "(Remember: The first position is 0.)\n";

        cout << "Search again?(y/n followed by Return): ";
        cin >> ans;
    } while ((ans != 'n') && (ans != 'N'));

    cout << "End of program.\n";
    return 0;
}
//Uses iostream:
void fill_array(int a[], int size, int& number_used)
<The rest of the definition of fill_array is given in Display 7.9.>

```

(continued)

DISPLAY 7.10 Searching an Array *(part 2 of 2)*

```
int search(const int a[], int number_used, int target)
{

    int index = 0;
    bool found = false;
    while ((!found) && (index < number_used))
        if (target == a[index])
            found = true;
        else
            index++;

    if (found)
        return index;
    else
        return -1;
}
```

Sample Dialogue

```
Enter up to 20 nonnegative whole numbers.
Mark the end of the list with a negative number.
10 20 30 40 50 60 70 80 -1
Enter a number to search for: 10
10 is stored in array position 0.
(Remember: The first position is 0.)
Search again?(y/n followed by Return): y
Enter a number to search for: 40
40 is stored in array position 3.
(Remember: The first position is 0.)
Search again?(y/n followed by Return): y
Enter a number to search for: 42
42 is not on the list.
Search again?(y/n followed by Return): n
End of program.
```

`found` is used as a flag to record whether or not the target element has been found. If the target element is found in the array, `found` is set to *`true`*, which in turn ends the *`while`* loop.

Even if we used `fill_array` in only one program, it can still be a good idea to make the declared array size an argument to `fill_array`. Displaying the declared size of the array as an argument reminds us that the function needs this information in a critically important way.

PROGRAMMING EXAMPLE Sorting an Array

One of the most widely encountered programming tasks, and certainly the most thoroughly studied, is sorting a list of values, such as a list of sales figures that must be sorted from lowest to highest or from highest to lowest, or a list of words that must be sorted into alphabetical order. In this section we describe a function called `sort` that sorts a partially filled array of numbers so that they are ordered from smallest to largest.

The procedure `sort` has one array parameter `a`. The array `a` will be partially filled, so there is an additional formal parameter called `number_used`, which tells how many array positions are used. Thus, the declaration and precondition for the function `sort` is

```
void sort(int a[], int number_used);
//Precondition: number_used <= declared size of the array a.
//Array elements a[0] through a[number_used - 1] have values.
```

The function `sort` rearranges the elements in array `a` so that after the function call is completed the elements are sorted as follows:

```
a[0] ≤ a[1] ≤ a[2] ≤ ... ≤ a[number_used - 1]
```

The algorithm we use to do the sorting is called selection sort. It is one of the easiest of the sorting algorithms to understand.

One way to design an algorithm is to rely on the definition of the problem. In this case the problem is to sort an array a from smallest to largest. That means rearranging the values so that `a[0]` is the smallest, `a[1]` the next smallest, and so forth. That definition yields an outline for the selection sort algorithm:

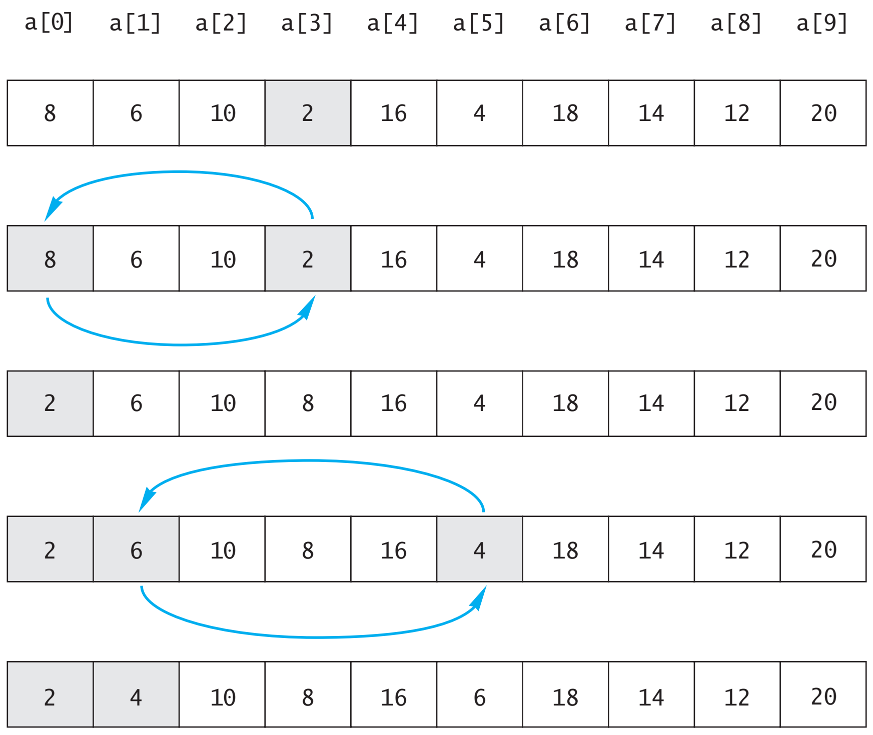
```
for (int index = 0; index < number_used; index++)
    Place the indexth smallest element in a[index]
```

VideoNote
Selection Sort Walkthrough

There are many ways to realize this general approach. The details could be developed using two arrays and copying the elements from one array to the other in sorted order, but one array should be both adequate and economical. Therefore, the function `sort` uses only the one array containing the values to be sorted. The function `sort` rearranges the values in the array `a` by interchanging pairs of values. Let us go through a concrete example so that you can see how the algorithm works.

Consider the array shown in Display 7.11. The algorithm will place the smallest value in `a[0]`. The smallest value is the value in `a[3]`. So the algorithm interchanges the values of `a[0]` and `a[3]`. The algorithm then looks for the next smallest element. The value in `a[0]` is now the smallest element and so the next smallest element is the smallest of the remaining elements `a[1]`, `a[2]`, `a[3]`, …, `a[9]`. In the example in Display 7.11, the next smallest element is in `a[5]`, so the algorithm interchanges the values of `a[1]` and `a[5]`. This positioning of the second smallest element is illustrated in the fourth and fifth array pictures in Display 7.11. The algorithm then positions the third smallest element, and so forth.

DISPLAY 7.11 Selection Sort



As the sorting proceeds, the beginning array elements are set equal to the correct sorted values. The sorted portion of the array grows by adding elements one after the other from the elements in the unsorted end of the array. Notice that the algorithm need not do anything with the value in the last indexed variable, `a[9]`. That is because once the other elements are positioned correctly, `a[9]` must also have the correct value. After all, the correct value for `a[9]` is the smallest value left to be moved, and the only value left to be moved is the value that is already in `a[9]`.

The definition of the function `sort`, included in a demonstration program, is given in Display 7.12. `sort` uses the function `index_of_smallest` to find the index of the smallest element in the unsorted end of the array, and then it does an interchange to move this element down into the sorted part of the array.

The function `swap_values`, shown in Display 7.12, is used to interchange the values of indexed variables. For example, the following call will interchange the values of `a[0]` and `a[3]`:

```
swap_values(a[0], a[3]);
```

The function `swap_values` was explained in Chapter 5.

DISPLAY 7.12 Sorting an Array *(part 1 of 2)*

```
//Tests the procedure sort.
#include <iostream>

void fill_array(int a[], int size, int&number_used);
//Precondition: size is the declared size of the array a.
//Postcondition: number_used is the number of values stored in a.
//a[0] through a[number_used - 1] have been filled with
//nonnegative integers read from the keyboard.

void sort(int a[], int number_used);
//Precondition: number_used <= declared size of the array a.
//The array elements a[0] through a[number_used - 1] have values.
//Postcondition: The values of a[0] through a[number_used - 1] have
//been rearranged so that a[0] <= a[1] <= ... <= a[number_used - 1].

void swap_values(int &v1, int &v2);
//Interchanges the values of v1 and v2.

int index_of_smallest(const int a[], int start_index, int number_used);
//Precondition: 0 <= start_index < number_used. Referenced array elements have
//values.
//Returns the index i such that a[i] is the smallest of the values
//a[start_index], a[start_index + 1], ..., a[number_used - 1].

int main( )
{
    using namespace std;
    cout << "This program sorts numbers from lowest to highest.\n";

    int sample_array[10], number_used;
    fill_array(sample_array, 10, number_used);
    sort(sample_array, number_used);

    cout << "In sorted order the numbers are:\n";
    for (int index = 0; index < number_used; index++)
    cout << sample_array[index] << " ";
    cout << endl;

    return 0;
}

//Uses iostream:
void fill_array(int a[], int size, int&number_used)
```

<The rest of the definition of fill_array is given in Display 7.9.>

```
void sort(int a[], int number_used)
{
    int index_of_next_smallest;
    for (int index = 0; index < number_used - 1; index++)
```

(continued)

DISPLAY 7.12 Sorting an Array *(part 2 of 2)*

```
        {//Place the correct value in a[index]:
            index_of_next_smallest =
                          index_of_smallest(a, index, number_used);
            swap_values(a[index], a[index_of_next_smallest]);
            //a[0] <= a[1] <=...<= a[index] are the smallest of the original array
            //elements. The rest of the elements are in the remaining positions.
        }
    }

    void swap_values(int& v1, int& v2)
    {
        int temp;
        temp = v1;
        v1 = v2;
        v2 = temp;
    }

    int index_of_smallest(const int a[], int start_index, int number_used)
    {
        int min = a[start_index],
            index_of_min = start_index;
        for (int index = start_index + 1; index < number_used; index++)
            if (a[index] < min)
            {
                min = a[index];
                index_of_min = index;
                //min is the smallest of a[start_index] through a[index]
            }

        return index_of_min;
    }
```

Sample Dialogue

```
This program sorts numbers from lowest to highest.
Enter up to 10 nonnegative whole numbers.
Mark the end of the list with a negative number.
80 30 50 70 60 90 20 30 40 -1
In sorted order the numbers are:
20 30 30 40 50 60 70 80 90
```

PROGRAMMING EXAMPLE Bubble Sort

VideoNote
Bubble Sort Walkthrough

The selection sort algorithm that we just described is not the only way to sort an array. In fact, computer scientists have devised scores of sorting algorithms! Some of these algorithms are more efficient than others and some work only for particular types of data. Bubble sort is a simple and general sorting algorithm that is similar to selection sort.

If we use bubble sort to sort an array in ascending order, then the largest value is successively "bubbled" toward the end of the array. For example, if we start with an unsorted array consisting of the following integers:

Initial array:	`{3, 10, 9, 2, 5}`

Then after the first pass we will have moved the largest value, 10, to the end of the array:

After first pass:	`{3, 9, 2, 5, 10}`

The second pass will move the second largest value, 9, to the second to last index of the array:

After second pass:	`{3, 2, 5, 9, 10}`

The third pass will move the third largest value, 5, to the third to last index of the array (where it already is):

After third pass:	`{2, 3, 5, 9, 10}`

The fourth pass will move the fourth largest value, 3, to the fourth to last index of the array (where it already is):

After fourth pass:	`{2, 3, 5, 9, 10}`

At this point the algorithm is done. The remaining number at the beginning of the array doesn't need to be examined since it is the only number left and must be the smallest. To design a program based on bubble sort note that we are placing the largest item at index `length-1`, the second largest item at `length-2`, the next at `length-3`, etc. This corresponds to a loop that starts at index `length-1` of the array and counts down to index 1 of the array. We don't need to include index 0 since that will contain the smallest element. One way to implement the loop is with the following code, where variable `i` corresponds to the target index:

```
for (int i = length-1; i > 0; i--)
```

The "bubble" part of bubble sort happens inside each iteration of this loop. The bubble step consists of another loop that moves the largest number toward the index `i` in the array. First, the largest number between index 0 and index `i` will be bubbled up to index `i`. We start the bubbling procedure by comparing the number at index 0 with the number at index 1. If the number at index 0 is larger than the number at index 1 then the values are swapped so we end up with the largest number at index 1. If the number at index 0 is less than or equal to the number at index 1 then nothing happens. Starting with the following unsorted array:

Initial array: `{3, 10, 9, 2, 5}`

Then the first step of the bubbling procedure will compare 3 to 10. Since 10 > 3 nothing happens and the end result is the number 10 is at index 1:

After step 1: `{3, 10, 9, 2, 5}`

The procedure is repeated for successively larger values until we reach `i`. The second step will compare the numbers at index 1 and 2, which is values 10 and 9. Since 10 is larger than 9 we swap the numbers resulting in the following:

After step 2: `{3, 9, 10, 2, 5}`

The process is repeated two more times:

After step 3: `{3, 9, 2, 10, 5}`
After step 4: `{3, 9, 2, 5, 10}`

This ends the first iteration of the bubble sort algorithm. We have bubbled the largest number to the end of the array. The next iteration would bubble the second largest number to the second to last position, and so forth, where variable `i` represents the target index for the bubbled number. If we use variable `j` to reference the index of the bubbled item then our loop code looks like this:

```
for (int i = length-1; i > 0; i--)
    for (int j = 0; j < i; j++)
```

Inside the loop we must compare the items at index `j` and index `j+1`. The largest should be moved into index `j+1`. The completed algorithm is shown below and a complete example in Display 7.13.

```
for (int i = length-1; i > 0; i--)
   for (int j = 0; j < i; j++)
      if (arr[j] > arr[j+1])
      {
            int temp = arr[j+1];
            arr[j+1] = arr[j];
            arr[j] = temp;
      }
```

DISPLAY 7.13 Bubble Sort Program

```
//DISPLAY 7.13 Bubble Sort Program
//Sorts an array of integers using Bubble Sort.
#include <iostream>

void bubblesort(int arr[], int length);
//Precondition: length <= declared size of the array arr.
//The array elements arr[0] through a[length - 1] have values.
//Postcondition: The values of arr[0] through arr[length - 1] have
//been rearranged so that arr[0] <= a[1] <=  <= arr[length - 1].

int main()
{
    using namespace std;
    int a[] = {3, 10, 9, 2, 5, 1};

    bubblesort(a, 6);
    for (int i=0; i<6; i++)
    {
         cout << a[i] << " ";
    }
    cout << endl;
    return 0;
}

void bubblesort(int arr[], int length)
{
       // Bubble largest number toward the right
       for (int i = length-1; i > 0; i--)
              for (int j = 0; j < i; j++)
                     if (arr[j] > arr[j+1])
                     {
                             // Swap the numbers
                             int temp = arr[j+1];
                             arr[j+1] = arr[j];
                             arr[j] = temp;
                     }
}
```

Sample Dialogue

```
1 2 3 5 9 10
```

SELF-TEST EXERCISES

17. Write a program that will read up to ten nonnegative integers into an array called `number_array` and then write the integers back to the screen. For this exercise you need not use any functions. This is just a toy program and can be very minimal.

18. Write a program that will read up to ten letters into an array and write the letters back to the screen in the reverse order. For example, if the input is

    ```
    abcd.
    ```

    then the output should be

    ```
    dcba
    ```

    Use a period as a sentinel value to mark the end of the input. Call the array `letter_box`. For this exercise you need not use any functions. This is just a toy program and can be very minimal.

19. Following is the declaration for an alternative version of the function search defined in Display 7.12. In order to use this alternative version of the search function, we would need to rewrite the program slightly, but for this exercise all you need to do is to write the function definition for this alternative version of `search`.

    ```
    bool search(const int a[], int number_used,
    int target, int& where);
    //Precondition: number_used is <= the declared size of the
    //array a; a[0] through a[number_used - 1] have values.
    //Postcondition: If target is one of the elements a[0]
    //through a[number_used - 1], then this function returns
    //true and sets the value of where so that a[where] ==
    //target; otherwise this function returns false and the
    //value of where is unchanged.
    ```

7.4 MULTIDIMENSIONAL ARRAYS

Two indexes are better than one.

FOUND ON THE WALL OF A COMPUTER SCIENCE DEPARTMENT RESTROOM

C++ allows you to declare arrays with more than one index. In this section we describe these multidimensional arrays.

Multidimensional Array Basics

It is sometimes useful to have an array with more than one index, and this is allowed in C++. The following declares an array of characters called `page`. The array `page` has two indexes: The first index ranges from `0` to `29`, and the second from `0` to `99`.

```
char page[30][100];
```

The indexed variables for this array each have two indexes. For example, `page[0][0]`, `page[15][32]`, and `page[29][99]` are three of the indexed variables for this array. Note that each index must be enclosed in its own set of square brackets. As was true of the one-dimensional arrays we have already seen, each indexed variable for a multidimensional array is a variable of the base type.

An array may have any number of indexes, but perhaps the most common number of indexes is two. A two-dimensional array can be visualized as a two-dimensional display with the first index giving the row and the second index giving the column. For example, the array indexed variables of the two-dimensional array `page` can be visualized as follows:

```
page[0][0], page[0][1], ..., page[0][99]
page[1][0], page[1][1], ..., page[1][99]
page[2][0], page[2][1], ..., page[2][99]
                    .
                    .
                    .
page[29][0], page[29][1], ..., page[29][99]
```

You might use the array `page` to store all the characters on a page of text that has 30 lines (numbered 0 through 29) and 100 characters on each line (numbered 0 through 99).

A multidimensional array is an array of arrays

In C++, a two-dimensional array, such as `page`, is actually an array of arrays. The example array `page` is actually a one-dimensional array of size 30, whose base type is a one-dimensional array of characters of size 100. Normally, this need not concern you, and you can usually act as if the array `page` is actually an array with two indexes (rather than an array of arrays, which is harder to keep track of). There is, however, at least one situation where a two-dimensional array looks very much like an array of arrays, namely, when you have a function with an array parameter for a two-dimensional array, which is discussed in the next subsection.

Multidimensional Array Parameters

The following declaration of a two-dimensional array is actually declaring a one-dimensional array of size 30, whose base type is a one-dimensional array of characters of size 100:

Multidimensional Array Declaration

SYNTAX

```
Type Array_Name[Size_Dim_1][Size_Dim_2]...[Size_Dim_Last];
```

EXAMPLES

```
char page[30][100];
int matrix[2][3];
double three_d_picture[10][20][30];
```

An array declaration, of the form shown, defines one indexed variable for each combination of array indexes. For example, the second of the sample declarations defines the following six indexed variables for the array matrix:

```
matrix[0][0], matrix[0][1], matrix[0][2],
matrix[1][0], matrix[1][1], matrix[1][2]
```

```
char page[30][100];
```

Viewing a two-dimensional array as an array of arrays will help you to understand how C++ handles parameters for multidimensional arrays. For example, the following function takes an array argument, like `page`, and prints it to the screen:

```
void display_page(const char p[][100], int size_dimension_1)
{
    for (int index1 = 0; index1 < size_dimension_1; index1++)
    {//Printing one line:
        for (int index2 = 0; index2 < 100; index2++)
            cout << p[index1][index2];
        cout << endl;
    }
}
```

Notice that with a two-dimensional array parameter, the size of the first dimension is not given, so we must include an *int* parameter to give the size of this first dimension. (As with ordinary arrays, the compiler will allow you to specify the first dimension by placing a number within the first pair of square brackets. However, such a number is only a comment; the compiler ignores any such number.) The size of the second dimension (and all other dimensions if there are more than two) is given after the array parameter, as shown for the parameter

```
const char p[ ][100]
```

Multidimensional Array Parameters

When a multidimensional array parameter is given in a function heading or function declaration, the size of the first dimension is not given, but the remaining dimension sizes must be given in square brackets. Since the first dimension size is not given, you usually need an additional parameter of type *int* that gives the size of this first dimension. Below is an example of a function declaration with a two-dimensional array parameter p:

```
void get_page(char p[][100], int size_dimension_1);
```

If you realize that a multidimensional array is an array of arrays, then this rule begins to make sense. Since the two-dimensional array parameter

```
const char p[ ][100]
```

is a parameter for an array of arrays, the first dimension is really the index of the array and is treated just like an array index for an ordinary, one-dimensional array. The second dimension is part of the description of the base type, which is an array of characters of size 100.

PROGRAMMING EXAMPLE Two-Dimensional Grading Program

Display 7.14 contains a program that uses a two-dimensional array, named `grade`, to store and then display the grade records for a small class. The class has four students and includes three quizzes. Display 7.15 illustrates how the array `grade` is used to store data. The first array index is used to designate a student, and the second array index is used to designate a quiz. Since the students and quizzes are numbered starting with 1 rather than 0, we must subtract 1 from the student number and subtract 1 from the quiz number to obtain the indexed variable that stores a particular quiz score. For example, the score that student number 4 received on quiz number 1 is recorded in `grade[3][0]`.

Our program also uses two ordinary one-dimensional arrays. The array `st_ave` will be used to record the average quiz score for each of the students. For example, the program will set `st_ave[0]` equal to the average of the quiz scores received by student 1, `st_ave[1]` equal to the average of the quiz scores received by student 2, and so forth. The array `quiz_ave` will be used to record the average score for each quiz. For example, the program will set `quiz_ave[0]` equal to the average of all the student scores for quiz 1, `quiz_ave[1]` will record the average

DISPLAY 7.14 Two-Dimensional Array *(part 1 of 3)*

```
//Reads quiz scores for each student into the two-dimensional array grade (but
//the input code is not shown in this display). Computes the average score
//for each student and the average score for each quiz. Displays the quiz scores
//and the averages.
#include <iostream>
#include <iomanip>
const int NUMBER_STUDENTS = 4, NUMBER_QUIZZES = 3;

void compute_st_ave(const int grade[][NUMBER_QUIZZES], double st_ave[]);
//Precondition: Global constants NUMBER_STUDENTS and NUMBER_QUIZZES
//are the dimensions of the array grade. Each of the indexed variables
//grade[st_num - 1, quiz_num - 1] contains the score for student st_num on quiz
//quiz_num.
//Postcondition: Each st_ave[st_num - 1] contains the average for student
//number stu_num.


void compute_quiz_ave(const int grade[][NUMBER_QUIZZES], double quiz_ave[]);
//Precondition: Global constants NUMBER_STUDENTS and NUMBER_QUIZZES
//are the dimensions of the array grade. Each of the indexed variables
//grade[st_num - 1, quiz_num - 1] contains the score for student st_num on quiz
//quiz_num.
//Postcondition: Each quiz_ave[quiz_num - 1] contains the average for quiz number
//quiz_num.


void display(const int grade[][NUMBER_QUIZZES],
const double st_ave[], const double quiz_ave[]);
//Precondition: Global constants NUMBER_STUDENTS and NUMBER_QUIZZES are the
//dimensions of the array grade. Each of the indexed variables grade[st_num - 1,
//quiz_num - 1] contains the score for student st_num on quiz quiz_num. Each
//st_ave[st_num - 1] contains the average for student stu_num. Each
//quiz_ave[quiz_num - 1] contains the average for quiz number quiz_num.
//Postcondition: All the data in grade, st_ave, and quiz_ave has been output.


int main( )
{
    using namespace std;
    int grade[NUMBER_STUDENTS][NUMBER_QUIZZES];
    double st_ave[NUMBER_STUDENTS];
    double quiz_ave[NUMBER_QUIZZES];


<The code for filling the array grade goes here, but is not shown.>
```

(continued)

DISPLAY 7.14 Two-Dimensional Array *(part 2 of 3)*

```
        compute_st_ave(grade, st_ave);
        compute_quiz_ave(grade, quiz_ave);
        display(grade, st_ave, quiz_ave);
        return 0;
    }

    void compute_st_ave(const int grade[][NUMBER_QUIZZES], double st_ave[])
    {
        for (int st_num = 1; st_num <= NUMBER_STUDENTS; st_num++)
        {//Process one st_num:
            double sum = 0;
            for (int quiz_num = 1; quiz_num <= NUMBER_QUIZZES; quiz_num++)
                sum = sum + grade[st_num - 1][quiz_num - 1];
            //sum contains the sum of the quiz scores for student number st_num.
            st_ave[st_num - 1] = sum/NUMBER_QUIZZES;
            //Average for student st_num is the value of st_ave[st_num-1]
        }
    }


    void compute_quiz_ave(const int grade[][NUMBER_QUIZZES], double quiz_ave[])
    {
        for (int quiz_num = 1; quiz_num <= NUMBER_QUIZZES; quiz_num++)
        {//Process one quiz (for all students):
            double sum = 0;
            for (int st_num = 1; st_num <= NUMBER_STUDENTS; st_num++)
                sum = sum + grade[st_num - 1][quiz_num - 1];
            //sum contains the sum of all student scores on quiz number quiz_num.
            quiz_ave[quiz_num - 1] = sum/NUMBER_STUDENTS;
            //Average for quiz quiz_num is the value of quiz_ave[quiz_num - 1]
        }
    }


    //Uses iostream and iomanip:
    void display(const int grade[][NUMBER_QUIZZES],
        const double st_ave[], const double quiz_ave[])
    {
        using namespace std;
        cout.setf(ios::fixed);
        cout.setf(ios::showpoint);
        cout.precision(1);
        cout << setw(10) << "Student"
             << setw(5) << "Ave"
             << setw(15) << "Quizzes\n";
        for (int st_num = 1; st_num <= NUMBER_STUDENTS; st_num++)
        {//Display for one st_num:
```

(continued)

DISPLAY 7.14 Two-Dimensional Array *(part 3 of 3)*

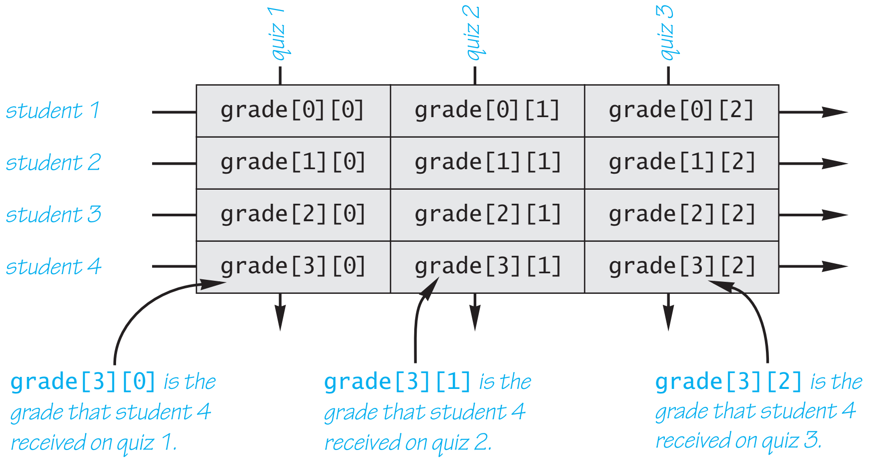
```
            cout << setw(10) << st_num
                 << setw(5) << st_ave[st_num - 1] << " ";
            for (int quiz_num = 1; quiz_num <= NUMBER_QUIZZES; quiz_num++)
                cout << setw(5) << grade[st_num - 1][quiz_num - 1];
            cout << endl;
        }

        cout << "Quiz averages = ";
        for (int quiz_num = 1; quiz_num <= NUMBER_QUIZZES; quiz_num++)
            cout << setw(5) << quiz_ave[quiz_num - 1];
        cout << endl;
    }
```

Sample Dialogue

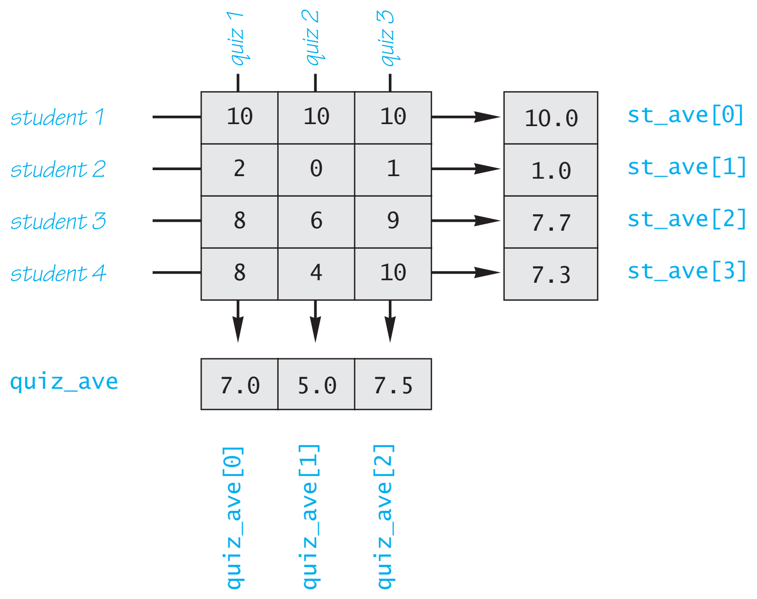
<The dialogue for filling the array grade is not shown.>

```
Student       Ave                Quizzes

1            10.0          10      10      10

2             1.0           2       0       1

3             7.7           8       6       9

4             7.3           8       4      10

Quiz averages =           7.0     5.0     7.5
```

DISPLAY 7.15 The Two-Dimensional Array grade



DISPLAY 7.16 The Two-Dimensional Array grade (Another View)



score for quiz 2, and so forth. Display 7.16 illustrates the relationship between the arrays `grade`, `st_ave`, and `quiz_ave`. In that display, we have shown some sample data for the array `grade`. This data, in turn, determines the values that the program stores in `st_ave` and in `quiz_ave`. Display 7.16 also shows these values, which the program computes for `st_ave` and `quiz_ave`.

The complete program for filling the array `grade` and then computing and displaying both the student averages and the quiz averages is shown in Display 7.14. In that program we have declared array dimensions as global named constants. Since the procedures are particular to this program and could not be reused elsewhere, we have used these globally defined constants in the procedure bodies, rather than having parameters for the size of the array dimensions. Since it is routine, the display does not show the code that fills the array.

PITFALL Using Commas Between Array Indexes

Note that in Display 7.14 we wrote an indexed variable for the two-dimensional array `grade` as `grade[st_num - 1][quiz_num - 1]` with two pairs of square brackets. In some other programming languages it would be written with one pair of brackets and commas as follows: `grade[st_num - 1, quiz_num - 1]`; this is incorrect in C++. If you use `grade[st_num - 1, quiz_num - 1]` in C++ you are unlikely to get any error message, but it is incorrect usage and will cause your program to misbehave. ■

SELF-TEST EXERCISES

20. What is the output produced by the following code?

```
int my_array[4][4], index1, index2;
for (index1 = 0; index1 < 4; index1++)
    for (index2 = 0; index2 < 4; index2++)
        my_array[index1][index2] = index2;
for (index1 = 0; index1 < 4; index1++)
{
    for (index2 = 0; index2 < 4; index2++)
        cout << my_array[index1][index2] << " ";
    cout << endl;
}
```

21. Write code that will fill the array a (declared below) with numbers typed in at the keyboard. The numbers will be input five per line, on four lines (although your solution need not depend on how the input numbers are divided into lines).

```
int a[4][5];
```

22. Write a function definition for a *void* function called echo such that the following function call will echo the input described in Self-Test Exercise 21 and will echo it in the same format as we specified for the input (that is, four lines of five numbers per line):

```
echo(a, 4);
```

CHAPTER SUMMARY

- An array can be used to store and manipulate a collection of data that is all of the same type.
- The indexed variables of an array can be used just like any other variables of the base type of the array.
- A *for* loop is a good way to step through the elements of an array and perform some program action on each indexed variable.
- The most common programming error made when using arrays is attempting to access a nonexistent array index. Always check the first and last iterations of a loop that manipulates an array to make sure it does not use an index that is illegally small or illegally large.
- An array formal parameter is neither a call-by-value parameter nor a call-by-reference parameter, but a new kind of parameter. An array parameter is similar to a call-by-reference parameter in that any change that is made to the formal parameter in the body of the function will be made to the array argument when the function is called.

- The indexed variables for an array are stored next to each other in the computer's memory so that the array occupies a contiguous portion of memory. When the array is passed as an argument to a function, only the address of the first indexed variable (the one numbered 0) is given to the calling function. Therefore, a function with an array parameter usually needs another formal parameter of type *int* to give the size of the array.
- When using a partially filled array, your program needs an additional variable of type *int* to keep track of how much of the array is being used.
- To tell the compiler that an array argument should not be changed by your function, you can insert the modifier *const* before the array parameter for that argument position. An array parameter that is modified with a *const* is called a **constant array parameter.**
- If you need an array with more than one index, you can use a multidimensional array, which is actually an array of arrays.

Answers to Self-Test Exercises

1. The statement `int a[5];` is a declaration, where `5` is the number of array elements. The expression `a[4]` is an access into the array defined by the previous statement. The access is to the element having index 4, which is the fifth (and last) array element.

2. a. `score`

   b. `double`

   c. `5`

   d. `0 through 4`

   e. Any of `score[0], score[1], score[2], score[3], score[4]`

3. a. One too many initializers

   b. Correct. The array size is 4.

   c. Correct. The array size is 4.

4. `abc`

5.
   ```
   1.1 2.2 3.3
   1.1 3.3 3.3
   ```

   (Remember that the indexes start with 0, not 1.)

6. `2 4 6 8 10 12 14 16 18 0 4 8 12 16`

7. The indexed variables of `sample_array` are `sample_array[0]` through `sample_array[9]`, but this piece of code tries to fill `sample_array[1]` through `sample_array[10]`. The index 10 in `sample_array[10]` is out of range.

8. There is an index out of range. When `index` is equal to 9, `index + 1` is equal to 10, so `a[index + 1]`, which is the same as `a[10]`, has an illegal index. The loop should stop with one less iteration. To correct the code, change the first line of the *for* loop to

   ```
   for (int index = 0; index < 9; index++)
   ```

9.
   ```
   int i, a[20];

   cout << "Enter 20 numbers:\n";

   for (i = 0; i < 20; i++)
       cin >> a[i];
   ```

10. The array will consume 14 bytes of memory. The address of the indexed variable `your_array[3]` is1006.

11. The following function calls are acceptable:

    ```
    tripler(number);
    tripler(a[2]);
    tripler(a[number]);
    ```

    The following function calls are incorrect:

    ```
    tripler(a[3]);
    tripler(a);
    ```

    The first one has an illegal index. The second has no indexed expression at all. You cannot use an entire array as an argument to `tripler`, as in the second call. The section "Entire Arrays as Function Arguments" discusses a different situation in which you can use an entire array as an argument.

12. The loop steps through indexed variables `b[1]` through `b[5]`, but `5` is an illegal index for the array `b`. The indexes are 0, 1, 2, 3, and 4. The correct version of the code is:

    ```
    int b[5] = {1, 2, 3, 4, 5};
        for (int i = 0; i < 5; i++)
            tripler(b[i]);
    ```

13.
    ```
    void one_more(int a[ ], int size)
    //Precondition: size is the declared size of the array a.
    //a[0] through a[size - 1] have been given values.
    //Postcondition: a[index] has been increased by 1
    //for all indexed variables of a.
    {
    ```

```
    for (int index = 0; index < size; index++)
        a[index] = a[index] + 1;
}
```

14. The following function calls are all acceptable:

```
too2(my_array, 29);
too2(my_array, 10);
too2(your_array, 100);
```

The call

```
too2(my_array, 10);
```

is legal, but will fill only the first ten indexed variables of `my_array`. If that is what is desired, the call is acceptable.

The following function calls are all incorrect:

```
too2(my_array, 55);
"Hey too2. Please, come over here."
too2(my_array[3], 29);
```

The first of these is incorrect because the second argument is too large. The second is incorrect because it is missing a final semicolon (and for other reasons). The third one is incorrect because it uses an indexed variable for an argument where it should use the entire array.

15. You can make the array parameter in output a constant parameter, since there is no need to change the values of any indexed variables of the array parameter. You cannot make the parameter in `drop_odd` a constant parameter because it may have the values of some of its indexed variables changed.

```
void output(const double a[ ], int size);
//Precondition: a[0] through a[size - 1] have values.
//Postcondition: a[0] through a[size - 1] have been
//written out.

void drop_odd(int a[], int size);
//Precondition: a[0] through a[size - 1] have values.
//Postcondition: All odd numbers in a[0] through
//a[size - 1] have been changed to 0.
```

16.
```
int out_of_order(double array[], int size)
{
    for (int i = 0; i < size - 1; i++)
        if (array[i] > array[i+1]) //fetch a[i+1] for each i.
        return i+1;
    return -1;
}
```

17.
```
#include <iostream>
using namespace std;
const int DECLARED_SIZE = 10;

int main( )
{
    cout << "Enter up to ten nonnegative integers.\n"
         << "Place a negative number at the end.\n";
    int number_array[DECLARED_SIZE], next, index = 0;
    cin >> next;
    while ( (next >= 0) && (index < DECLARED_SIZE) )
    {
        number_array[index] = next;
        index++;
        cin >> next;
    }

    int number_used = index;
    cout << "Here they are back at you:";
    for (index = 0; index < number_used; index++)
        cout << number_array[index] << " ";
    cout<< endl;
    return 0;
}
```

18.
```
#include <iostream>
using namespace std;
const int DECLARED_SIZE = 10;

int main()
{
    cout << "Enter up to ten letters"
         << " followed by a period:\n";
    char letter_box[DECLARED_SIZE], next;
    int index = 0;
    cin >> next;
    while ( (next != '.') && (index < DECLARED_SIZE) )
    {
        letter_box[index] = next;
        index++;
        cin >> next;
    }
    int number_used = index;
    cout << "Here they are backwards:\n";
    for(index = number_used - 1; index >= 0; index--)
        cout << letter_box[index];
    cout << endl;
    return 0;
}
```

19.
```
bool search(const int a[ ], int number_used,
                              int target, int& where)
{
    int index = 0;
    bool found = false;
    while ((!found) && (index < number_used))
        if (target == a[index])
            found = true;
        else
            index++;
    //If target was found, then
    //found == true and a[index] == target.
    if (found)
        where = index;
    return found;
}
```

20.
```
0 1 2 3
0 1 2 3
0 1 2 3
0 1 2 3
```

21.
```
int a[4][5];
int index1, index2;
for (index1 = 0; index1 < 4; index1++)
    for (index2 = 0; index2 < 5; index2++)
        cin >> a[index1][index2];
```

22.
```
void echo(const int a[][5], int size_of_a)
//Outputs the values in the array a on size_of_a lines
//with 5 numbers per line.
{
    for (int index1 = 0; index1 < size_of_a; index1++)
    {
        for (int index2 = 0; index2 < 5; index2++)
            cout << a[index1][index2] << " ";
        cout << endl;
    }
}
```

PRACTICE PROGRAMS

Practice Programs can generally be solved with a short program that directly applies the programming principles presented in this chapter.

1. Write a function named `firstLast2` that takes as input an array of integers and an integer that specifies how many entries are in the array. The function should return `true` if the array starts or ends with the digit 2. Otherwise it should return `false`. Test your function with arrays of different length and

with the digit 2 at the beginning of the array, end of the array, middle of the array, and missing from the array.

2. Write a function named `countNum2s` that takes as input an array of integers and an integer that specifies how many entries are in the array. The function should return the number of 2's in the array. Test your function with arrays of different length and with varying number of 2's.

3. Write a function named `swapFrontBack` that takes as input an array of integers and an integer that specifies how many entries are in the array. The function should swap the first element in the array with the last element in the array. The function should check if the array is empty to prevent errors. Test your function with arrays of different length and with varying front and back numbers.

4. The following code creates a small phone book. An array is used to store a list of names and another array is used to store the phone numbers that go with each name. For example, Michael Myers' phone number is 333-8000 and Ash Williams' phone number is 333-2323. Write the function `lookupName` so the code properly looks up and returns the phone number for the input target name.

```
int main()
{
    using namespace std;
    string names[] = {"Michael Myers",
                      "Ash Williams",
                      "Jack Torrance",
                      "Freddy Krueger"};
    string phoneNumbers[] = {"333-8000","333-2323",
                             "333-6150","339-7970"};
    string targetName, targetPhone;
    char c;
    do
    {
         cout << "Enter a name to find the "
              << "corresponding phone number."
              << endl;
         getline(cin, targetName);
         targetPhone = lookupName(targetName,
                            names, phoneNumbers,4);
         if (targetPhone.length() > 0)
                cout << "The number is: "
                     << targetPhone << endl;
         else
                cout << "Name not found. "
                     << endl;
         cout << "Look up another name? (y/n)"
              << endl;
```

```
        cin >> c;
        cin.ignore();
    } while (c == 'y');
    return 0;
}
```

PROGRAMMING PROJECTS

Programming Projects require more problem-solving than Practice Programs and can usually be solved many different ways. Visit www.myprogramminglab.com to complete many of these Programming Projects online and get instant feedback.

Projects 7 through 11 can be written more elegantly using structures or classes. Projects 12 through 15 are meant to be written using multidimensional arrays and do not require structures or classes. See Chapters 10 and 11 for information on defining classes and structures.

1. There are three versions of this project.

   Version 1 (all interactive). Write a program that reads in the average monthly rainfall for a city for each month of the year and then reads in the actual monthly rainfall for each of the previous 12 months. The program then prints out a nicely formatted table showing the rainfall for each of the previous 12 months as well as how much above or below average the rainfall was for each month. The average monthly rainfall is given for the months January, February, and so forth, in order. To obtain the actual rainfall for the previous 12 months, the program first asks what the current month is and then asks for the rainfall figures for the previous 12 months. The output should correctly label the months.

   There are a variety of ways to deal with the month names. One straightforward method is to code the months as integers and then do a conversion before doing the output. A large `switch` statement is acceptable in an output function. The month input can be handled in any manner you wish, as long as it is relatively easy and pleasant for the user.

   After you have completed this program, produce an enhanced version that also outputs a graph showing the average rainfall and the actual rainfall for each of the previous 12 months. The graph should be similar to the one shown in Display 7.8, except that there should be two bar graphs for each month and they should be labeled as the average rainfall and the rainfall for the most recent month. Your program should ask the user whether she or he wants to see the table or the bar graph and then should display whichever format is requested. Include a loop that allows the user to see either format as often as the user wishes until the user requests that the program end.

Version 2 (combines interactive and file output). For a more elaborate version, also allow the user to request that the table and graph be output to a file. The file name is entered by the user. This program does everything that the Version 1 program does but has this added feature. To read a file name, you must use material presented in the optional section of Chapter 5 entitled "File Names as Input."

Version 3 (all I/O with files). This version is like Version 1 except that input is taken from a file and the output is sent to a file. Since there is no user to interact with, there is no loop to allow repeating the display; both the table and the graph are output to the same file. If this is a class assignment, ask your instructor for instructions on what file names to use.

2. Hexadecimal numerals are integers written in base 16. The 16 digits used are '0' through '9' plus 'a' for the "digit 10", 'b' for the "digit 11", 'c' for the "digit 12", 'd' for the "digit 13", 'e' for the "digit 14", and 'f' for the "digit 15". For example, the hexadecimal numeral d is the same as base 10 numeral 13 and the hexadecimal numeral 1d is the same as the base 10 numeral 29. Write a C++ program to perform addition of two hexadecimal numerals each with up to 10 digits. If the result of the addition is more than 10 digits long, then simply give the output message "Addition Overflow" and not the result of the addition. Use arrays to store hexadecimal numerals as arrays of characters. Include a loop to repeat this calculation for new numbers until the user says she or he wants to end the program.

VideoNote
Solution to Programming Project 7.3

3. Write a function called `delete_repeats` that has a partially filled array of characters as a formal parameter and that deletes all repeated letters from the array. Since a partially filled array requires two arguments, the function will actually have two formal parameters: an array parameter and a formal parameter of type *`int`* that gives the number of array positions used. When a letter is deleted, the remaining letters are moved forward to fill in the gap. This will create empty positions at the end of the array so that less of the array is used. Since the formal parameter is a partially filled array, a second formal parameter of type *`int`* will tell how many array positions are filled. This second formal parameter will be a call-by-reference parameter and will be changed to show how much of the array is used after the repeated letters are deleted.

For example, consider the following code:

```
char a[10];
a[0] = 'a';
a[1] = 'b';
a[2] = 'a';
a[3] = 'c';
int size = 4;
delete_repeats(a, size);
```

After this code is executed, the value of `a[0]` is `'a'`, the value of `a[1]` is `'b'`, the value of `a[2]` is `'c'`, and the value of `size` is 3. (The value of `a[3]` is no longer of any concern, since the partially filled array no longer uses this indexed variable.)

You may assume that the partially filled array contains only lowercase letters. Embed your function in a suitable test program.

4. The standard deviation of a list of numbers is a measure of how much the numbers deviate from the average. If the standard deviation is small, the numbers are clustered close to the average. If the standard deviation is large, the numbers are scattered far from the average. The standard deviation, *S*, of a list of *N* numbers *x* is defined as follows:

$$S = \sqrt{\frac{\sum_{i=1}^{N} = (x_i - \overline{x})^2}{N}}$$

where *x* is the average of the *N* numbers *x*1, *x*2, ... Define a function that takes a partially filled array of numbers as its arguments and returns the standard deviation of the numbers in the partially filled array. Since a partially filled array requires two arguments, the function will actually have two formal parameters: an array parameter and a formal parameter of type `int` that gives the number of array positions used. The numbers in the array will be of type `double`. Embed your function in a suitable test program.

5. Write a program that reads in a list of integers into an array with base type `int`. Provide the facility to either read this array from the keyboard or from a file, at the user's option. If the user chooses file input, the program should request a file name. You may assume that there are fewer than 50 entries in the array. Your program determines how many entries there are. The output is to be a two-column list. The first column is a list of the distinct array elements; the second column is the count of the number of occurrences of each element. The list should be sorted on entries in the first column, largest to smallest.

For example, for the input

```
-12 3 -12 4 1 1 -12 1 -1 1 2 3 4 2 3 -12
```

the output should be

```
 N   Count
 4   2
 3   3
 2   2
 1   4
-1   1
-12  4
```

6. The text discusses the selection sort. We propose a different "sort" routine, the insertion sort. This routine is in a sense the opposite of the selection sort in that it picks up successive elements from the array and *inserts* each of these into the correct position in an already sorted subarray (at one end of the array we are sorting).

   The array to be sorted is divided into a sorted subarray and to-be-sorted subarray. Initially, the sorted subarray is empty. Each element of the to-be-sorted subarray is picked and inserted into its correct position in the sorted subarray.

   Write a function and a test program to implement the selection sort. Thoroughly test your program.

   Example and hints: The implementation involves an outside loop that selects successive elements in the to-be-sorted subarray and a nested loop that inserts each element in its proper position in the sorted subarray.

   Initially, the sorted subarray is empty, and the to-be-sorted subarray is all of the array:

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]
8	6	10	2	16	4	18	14	12	10

   Pick the first element, `a[0]` (that is, 8), and place it in the first position. The inside loop has nothing to do in this first case. The array and subarrays look like this:

sorted	**to-be-sorted**								
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]
8	6	10	2	16	4	18	14	12	10

   The first element from the to-be-sorted subarray is `a[1]`, which has value 6. Insert this into the sorted subarray in its proper position. These are out of order, so the inside loop must swap values in position 0 and position 1. The result is as follows:

sorted		**to-be-sorted**							
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]
6	8	10	2	16	4	18	14	10	12

Note that the sorted subarray has grown by one entry.

Repeat the process for the first to-be-sorted subarray entry, a[2], finding a place where a[2] can be placed so that the subarray remains sorted. Since a[2] is already in place—that is, it is larger than the largest element in the sorted subarray—the inside loop has nothing to do. The result is as follows:

sorted			**to-be-sorted**						
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]
6	8	10	2	16	4	18	14	10	12

Again, pick the first to-be-sorted array element, a[3]. This time the inside loop has to swap values until the value of a[3] is in its proper position. This involves some swapping:

sorted				**to-be-sorted**					
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]
6	8	10<-->2		16	4	18	14	10	12

sorted				**to-be-sorted**					
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]
6	8<--->2		10	16	4	18	14	10	12

sorted				**to-be-sorted**					
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]
6<--->2		8	10	16	4	18	14	10	12

The result of placing the 2 in the sorted subarray is

sorted				**to-be-sorted**					
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]
2	6	8	10	16	4	18	14	10	12

The algorithm continues in this fashion until the to-be-sorted array is empty and the sorted array has all the original array's elements.

7. An array can be used to store large integers one digit at a time. For example, the integer 1234 could be stored in the array `a` by setting `a[0]` to `1`, `a[1]` to `2`, `a[2]` to `3`, and `a[3]` to `4`. However, for this exercise you might find it more useful to store the digits backward, that is, place `4` in `a[0]`, `3` in `a[1]`, `2` in `a[2]`, and `1` in `a[3]`.

   In this exercise you will write a program that reads in two positive integers that are 20 or fewer digits in length and then outputs the sum of the two numbers. Your program will read the digits as values of type *`char`* so that the number 1234 is read as the four characters `'1'`, `'2'`, `'3'`, and `'4'`. After they are read into the program, the characters are changed to values of type *`int`*. The digits will be read into a partially filled array, and you might find it useful to reverse the order of the elements in the array after the array is filled with data from the keyboard. (Whether or not you reverse the order of the elements in the array is up to you. It can be done either way, and each way has its advantages and disadvantages.)

   Your program will perform the addition by implementing the usual paper-and-pencil addition algorithm. The result of the addition is stored in an array of size 20, and the result is then written to the screen. If the result of the addition is an integer with more than the maximum number of digits (that is, more than 20 digits), then your program should issue a message saying that it has encountered "integer overflow." You should be able to change the maximum length of the integers by changing only one globally defined constant. Include a loop that allows the user to continue to do more additions until the user says the program should end.

8. Write a program that will read a line of text and output a list of all the letters that occur in the text together with the number of times each letter occurs in the line. End the line with a period that serves as a sentinel value. The letters should be listed in the following order: the most frequently occurring letter, the next most frequently occurring letter, and so forth. Use two arrays, one to hold integers and one to hold letters. You may assume that the input uses all lowercase letters. For example, the input

   ```
   do be do bo.
   ```

   should produce output similar to the following:

Letter	Number of Occurrences
o	3
d	2
b	2
e	1

Your program will need to sort the arrays according to the values in the integer array. This will require that you modify the function `sort` given in Display 7.12. You cannot use `sort` to solve this problem without changing the function. If this is a class assignment, ask your instructor if input/output should be done with the keyboard and screen or if it should be done with files. If it is to be done with files, ask your instructor for instructions on file names.

9. Write a program to score five-card poker hands into one of the following categories: nothing, one pair, two pairs, three of a kind, straight (in order, with no gaps), flush (all the same suit, for example, all spades), full house (one pair and three of a kind), four of a kind, straight flush (both a straight and a flush). Use two arrays, one to hold the value of the card and one to hold the suit. Include a loop that allows the user to continue to score more hands until the user says the program should end.

10. Write a program that will allow two users to play tic-tac-toe. The program should ask for moves alternately from player X and player O. The program displays the game positions as follows:

```
1 2 3
4 5 6
7 8 9
```

The players enter their moves by entering the position number they wish to mark. After each move, the program displays the changed board. A sample board configuration is as follows:

```
X X O
4 5 6
O 8 9
```

11. Write a program to assign passengers seats in an airplane. Assume a small airplane with seat numbering as follows:

```
1 A B C D
2 A B C D
3 A B C D
4 A B C D
5 A B C D
6 A B C D
7 A B C D
```

The program should display the seat pattern, with an X marking the seats already assigned. For example, after seats 1A, 2B, and 4C are taken, the display should look like this:

```
1 X B C D
2 A X C D
3 A B C D
```

```
4 A B X D
5 A B C D
6 A B C D
7 A B C D
```

After displaying the seats available, the program prompts for the seat desired, the user types in a seat, and then the display of available seats is updated. This continues until all seats are filled or until the user signals that the program should end. If the user types in a seat that is already assigned, the program should say that that seat is occupied and ask for another choice.

12. Write a program that accepts input like the program in Display 7.8 and that outputs a bar graph like the one in that display except that your program will output the bars vertically rather than horizontally. A two-dimensional array may be useful.

13. The mathematician John Horton Conway invented the "Game of Life." Though not a "game" in any traditional sense, it provides interesting behavior that is specified with only a few rules. This project asks you to write a program that allows you to specify an initial configuration. The program follows the rules of LIFE to show the continuing behavior of the configuration.

    LIFE is an organism that lives in a discrete, two-dimensional world. While this world is actually unlimited, we don't have that luxury, so we restrict the array to 80 characters wide by 22 character positions high. If you have access to a larger screen, by all means use it.

    This world is an array with each cell capable of holding one LIFE cell. Generations mark the passing of time. Each generation brings births and deaths to the LIFE community. The births and deaths follow the following set of rules.

    - We define each cell to have eight *neighbor* cells. The neighbors of a cell are the cells directly above, below, to the right, to the left, diagonally above to the right and left, and diagonally below to the right and left.
    - If an occupied cell has zero or one neighbors, it dies of *loneliness*. If an occupied cell has more than three neighbors, it dies of *overcrowding*.
    - If an empty cell has exactly three occupied neighbor cells, there is a *birth* of a new cell to replace the empty cell.
    - Births and deaths are instantaneous and occur at the changes of generation. A cell dying for whatever reason may help cause birth, but a newborn cell cannot resurrect a cell that is dying, nor will a cell's death prevent the death of another, say, by reducing the local population.

    Notes: Some configurations grow from relatively small starting configurations. Others move across the region. It is recommended that for text output you use a rectangular array of *char* with 80 columns and 22 rows to store the LIFE world's successive generations. Use an asterisk * to indicate a living cell, and use a blank to indicate an empty (or dead) cell. If you have a screen with more rows than that, by all means make use of the whole screen.

Examples:

```
***
```

becomes

```
*
*
*
```

then becomes

```
***
```

again, and so on.

Suggestions: Look for stable configurations. That is, look for communities that repeat patterns continually. The number of configurations in the repetition is called the *period*. There are configurations that are fixed, which continue without change. A possible project is to find such configurations.

Hints: Define a *void* function named *generation* that takes the array we call *world*, an 80-column by 22-row array of *char*, which contains the initial configuration. The function scans the array and modifies the cells, marking the cells with births and deaths in accord with the rules listed earlier. This involves examining each cell in turn, either killing the cell, letting it live, or, if the cell is empty, deciding whether a cell should be born. There should be a function *display* that accepts the array *world* and displays the array on the screen. Some sort of time delay is appropriate between calls to *generation* and *display*. To do this, your program should generate and display the next generation when you press Return. You are at liberty to automate this, but automation is not necessary for the program.

14. Redo (or do for the first time) Programming Project 10 from Chapter 6. Your program should first load all boy names and girl names from the file into separate arrays. Search for the target name from the arrays, not directly from the file.

15. Redo (or do for the first time) Programming Project 11 from Chapter 6. Your program should not be hard-coded to create a bar chart of exactly four integers, but should be able to graph an array of up to 100 integers. Scale the graph appropriately in the horizontal and vertical dimensions so the bar chart fits within a 400 by 400 pixel area. You can impose the constraint that all integers in the array are nonnegative. Use the sentinel value of −1 to indicate the end of the values to draw in the bar chart. For example, to

create the bar chart with values 20, 40, 60, and 120, your program would operate on the array:

```
a[0]  =  20
a[1]  =  40
a[2]  =  60
a[3]  =  120
a[4]  =  -1
```

Test your program by creating several bar charts with different values and up to 100 entries and view the resulting SVG files to ensure that they are drawn correctly.

16. A common memory matching game played by young children is to start with a deck of cards that contains identical pairs. For example, given six cards in the deck, two might be labeled "1," two might be labeled "2," and two might be labeled "3." The cards are shuffled and placed face down on the table. The player then selects two cards that are face down, turns them face up, and if they match they are left face up. If the two cards do not match, they are returned to their original position face down. The game continues in this fashion until all cards are face up.

    Write a program that plays the memory matching game. Use 16 cards that are laid out in a 4 X 4 square and are labeled with pairs of numbers from 1 to 8. Your program should allow the player to specify the cards that she would like to select through a coordinate system.

    For example, suppose the cards are in the following layout:

```
     1   2   3   4
   ----------------
 1 | 8   *   *   *
 2 | *   *   *   *
 3 | *   8   *   *
 4 | *   *   *   *
```

    All of the cards are face down except for the pair 8, which has been located at coordinates (1, 1) and (2, 3). To hide the cards that have been temporarily placed face up, output a large number of newlines to force the old board off the screen.

    (*Hint:* Use a two-dimensional array for the arrangement of cards and another two-dimensional array that indicates if a card is face up or face down. Write a function that "shuffles" the cards in the array by repeatedly selecting two cards at random and swapping them. Random number generation is described in Chapter 4.)

17. Your swim school has two swimming instructors, Jeff and Anna. Their current schedules are shown below. An "X" denotes a 1-hour time slot that is occupied with a lesson.

Jeff	Monday	Tuesday	Wednesday	Thursday
11–12	X	X		
12–1		X	X	X
1–2		X	X	
2–3	X	X	X	

Anna	Monday	Tuesday	Wednesday	Thursday
11–12	X	X		X
12–1		X		X
1–2	X	X		
2–3	X		X	X

Write a program with array(s) capable of storing the schedules. Create a main menu that allows the user to mark a time slot as busy or free for either instructor. Also, add an option to output the schedules to the screen. Next, add an option to output all time slots available for individual lessons (slots when at least one instructor is free). Finally, add an option to output all time slots available for group lessons (when both instructors are free).

18. Modify Programming Project 17 by adding menu options to load and save the schedules from a file.

19. Traditional password entry schemes are susceptible to “shoulder surfing” in which an attacker watches an unsuspecting user enter their password or PIN number and uses it later to gain access to the account. One way to combat this problem is with a randomized challenge-response system. In these systems, the user enters different information every time based on a secret in response to a randomly generated challenge. Consider the following scheme in which the password consists of a five-digit PIN number (00000 to 99999). Each digit is assigned a random number that is 1, 2, or 3. The user enters the random numbers that correspond to their PIN instead of their actual PIN numbers.

    For example, consider an actual PIN number of 12345. To authenticate the user would be presented with a screen such as:

```
PIN:    0 1 2 3 4 5 6 7 8 9
NUM:    3 2 3 1 1 3 2 2 1 3
```

The user would enter 23113 instead of 12345. This doesn't divulge the password even if an attacker intercepts the entry because 23113 could correspond to other PIN numbers, such as 69440 or 70439. The next time the user logs in, a different sequence of random numbers would be generated, such as:

```
PIN:    0 1 2 3 4 5 6 7 8 9
NUM:    1 1 2 3 1 2 2 3 3 3
```

Your program should simulate the authentication process. Store an actual PIN number in your program. The program should use an array to assign random numbers to the digits from 0 to 9. Output the random digits to the screen, input the response from the user, and output whether or not the user's response correctly matches the PIN number.

20. The Social Security Administration maintains an actuarial life table that contains the probability that a person in the United States will die (http://www.ssa.gov/OACT/STATS/table4c6.html). The death probabilities from this table for 2009 are stored in the file `LifeDeathProbability.txt` and it is included on the website for the book. There are three values for each row, the age, death probability for a male, and death probability for a female. For example, the first five lines are:

```
0 0.006990 0.005728
1 0.000447 0.000373
2 0.000301 0.000241
3 0.000233 0.000186
4 0.000177 0.000150
```

This says that a 3 year old female has a 0.000186 chance of dying.

Write a program that reads the data into arrays from the file. Next, let the user enter his or her sex and age. The program should simulate to what age the user will live by starting with the death probability for the user's current age and sex. Generate a random number between 0-1; if this number is less than or equal to the death probability then predict that the user will live to the current age. If the random number is greater than the death probability then increase the age by one and repeat the calculation with a new random number for the next probability value.

If the simulation reaches age 120 then stop and predict that the user will live to 120. This program is merely a simulation and will give different results each time it is run, assuming you change the seed for the random number generator.