SESSION

11 The Array

Session Checklist

✔ Introducing the array data type

✔ Using arrays

✔ Initializing an array

✔ Using the most common type of array — the character string

30 Min. To Go

The programs written so far dealt with numbers one at a time. The summing programs input a value from the keyboard and then add it to some total contained in a single variable before reading the next number. If we return to our first analogy, the human program, these programs address one lug nut at a time. However, there are times when we want to hold all of the lug nuts at one time before operating on them.

This session examines how to store a set of values, much like the mechanic can hold or store a number of lug nuts at one time.

What Is an Array?

Let's begin by examining the whys and what fors of arrays. An *array* is a sequence of objects, usually numbers, each of which is referenced by an offset.

Consider the following problem. I need a program that can read a sequence of numbers from the keyboard. I'll use the now-standard rule that a negative number terminates input. After the numbers are read in, and only then, should the program display them on the standard output device.

I could attempt to store numbers in subsequent variables as in:

```
cin > nV1;
if (nV1 >= 0)
{
    cin > nV2;
    if (nV2 >= 0)
    {
     ...
```

You can see that this approach can't handle sequences involving more than just a few numbers.

An array solves the problem nicely:

```
int nV;
int nValues[128];
for (int i = 0; ; i++)
{
    cin > nV;
    if (nV < 0)
    {
        break;
    }
    nValues[i] = nV;
}
```

The second line of this snippet declares an array `nValues`. Array declarations begin with the type of the array members, in this case `int`, followed by the name of the array. The last element of an array declaration is an open and closed bracket containing the maximum number of elements that the array can hold. In this code snippet, `nValues` is declared as a sequence of 128 integers.

This snippet reads a number from the keyboard and stores it in a member of the array `nValues`. An individual element of an array is accessed by providing the name of the array followed by brackets containing an index. The first integer in the array is nValues[0], the second is nValues[1], and so on.

Note

In use, nValues[i] represents the ith element in the array. The index variable *i* must be a counting variable; that is, i must be an int or a long. If nValues is an array of ints, then nValues[i] is an int.

Accessing too far into an array

Mathematicians start counting their arrays with 1. The first member of a mathematical array x is x(1). Most programming languages also start with an offset of 1. C++ arrays begin counting at 0. The first member of a C++ array is nValues[0].

Note

There is a good reason why C++ begins counting from 0, but you have to wait until Session 12 to learn what it is.

Because indexing in a C++ array begins with 0, the last element of a 128-integer array is nArray[127].

Unfortunately for the programmer, C++ does not check whether the index you are using is within the range of the array. C++ is perfectly happy to give you access to nArray[200]. In fact, C++ will even let you access nArray[-15].

As an analogy, suppose that distances on a highway were measured by equally spaced power line poles. (In western Texas this isn't too far from the truth.) Let's call this unit of measure a pole-length. The road to my house begins at the turn off from the main highway and continues to my house in a straight line. The length of this road is exactly 9 pole-lengths.

If we begin numbering poles with the telephone pole at the highway, then the telephone pole next to my house is pole number 10. This is shown in Figure 11-1.

I can access any position along the road by counting poles from the highway. If we measure from the highway to the highway, we calculate a distance of 0 pole-length. The next discrete point is 1 pole-length, and so on, until we get to my house at 9 pole-lengths distance.

I can measure a distance 20 pole-lengths away from the highway. Of course, this location is not on the road. (Remember that the road stops at my house.) In fact, I have no idea what you might find there. You might be on the next highway, you might be out in a field, you might even land in my neighbor's living room. Examining that location is bad enough, but storing something there could be a lot worse. Storing something in a field is one thing, but crashing through my neighbor's living room could get you in trouble.

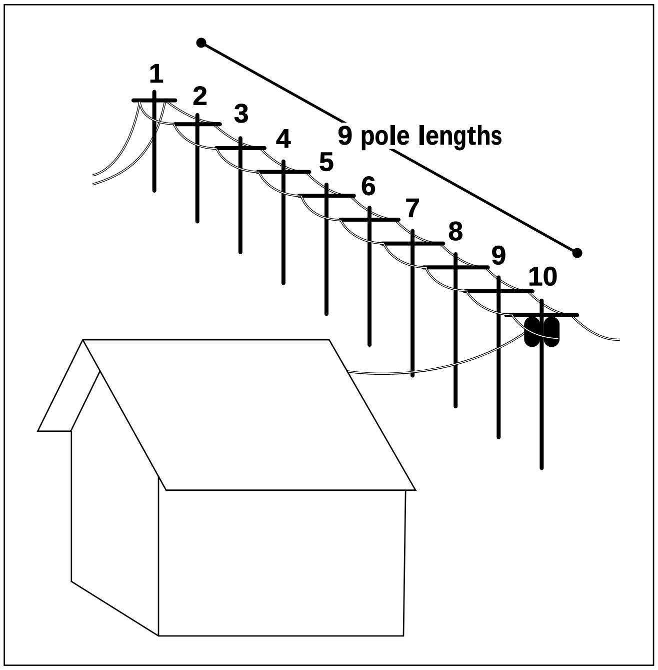

Figure 11-1
It takes 10 telephone poles to measure off a distance of 9 pole-lengths.

By analogy, reading `array[20]` of a 10-element array returns a more or less random value. Writing to `array[20]` has unpredictable results. It may do nothing, it may lead to erratic behavior, or it may crash the program.

Note

The most common incorrect location to access in the 128-element `nArray`, is `nArray[128]`. While only one element beyond the end of the array, reading or writing this location is just as dangerous as any other incorrect address.

20 Min. To Go

An array in practice

The following program solves the initial problem posed. This program inputs a sequence of integer values from the keyboard until the user enters a –1. The program then displays the numbers input and reports their sum.

```
// ArrayDemo - demonstrate the use of arrays
//             by reading a sequence of integers
//             and then displaying them in order
#include <stdio.h>


#include <iostream.h>

// prototype declarations
int sumArray(int nArray[], int nSize);
void displayArray(int nArray[], int nSize);

int main(int nArg, char* pszArgs[])
{




    // input the loop count
    int nAccumulator = 0;
    cout << "This program sums values entered"
         << "by the user\n";
    cout << "Terminate the loop by entering "
         << "a negative number\n";

    // store numbers into an array
    int nInputValues[128];
    int nNumValues = 0;
    do
    {
        // fetch another number
        int nValue;
        cout << "Enter next number: ";
        cin  > nValue;

        // if it's negative...
        if (nValue < 0)    // Comment A
        {
            // ...then exit
            break;
        }

        // ...otherwise store the number
        // into the storage array
        nInputValues[nNumValues++] = nValue;
    } while(nNumValues < 128); // Comment B



    // now output the values and the sum of the values
    displayArray(nInputValues, nNumValues);
    cout << “The sum is “


         << sumArray(nInputValues, nNumValues)
         << “\n”;
    return 0;
}

// displayArray - display the members of an
//                array of length nSize
void displayArray(int nArray[], int nSize)
{
    cout << “The value of the array is:\n”;
    for (int i = 0; i < nSize; i++)
    {
        cout.width(3);
        cout << i << “: “ << nArray[i] << “\n”;
    }
    cout << “\n”;
}

// sumArray - return the sum of the members of an
//            integer array
int sumArray(int nArray[], int nSize)
{
    int nSum = 0;
    for (int i = 0; i < nSize; i++)
    {
        nSum += nArray[i];
    }
    return nSum;
}
```

The program ArrayDemo begins with a prototype declaration of the functions `sumArray()` and `displayArray()`. The main body of the program contains the typical input loop. This time, however, the values are stored in the array `nInputValues`, with the variable `nNumValues` storing a count of the number of values stored in the array. The program stops reading values if the user inputs a negative number (Comment A), or if the number of elements in the array is exhausted (that's the test near Comment B).

Note

The array `nInputValues` is declared as 128 integers long. You might think that this should be enough for anyone, but don't count on it. Writing more data than an array's limit permits causes your program to perform erratically, and often to crash. No matter how large you make the array, always put a check to make sure that you do not exceed the limits of the array.

The main function ends by displaying the contents of the array and the sum.

The `displayArray()` function contains the typical `for` loop used to traverse an array. Notice again, that the index is initialized to 0 and not to 1. In addition, notice how the `for` loop that terminates before `i` is equal to `nSize`.

In a similar fashion, the `sumArray()` function loops through the array, adding each member to the total contained in `nSum`. Just to keep nonprogrammers guessing, the term *iterate* is used to mean traverse through a set of objects, such as an array. We say that "the `sumArray()` function iterates through the array."

Initializing an array

An array may be initialized at the time it is declared.

Note

An uninitialized variable initially contains random values.

The following code snippet demonstrates how this is done:

```
float fArray[5] = {0.0, 1.0, 2.0, 3.0, 4.0};
```

This initializes `fArray[0]` to 0, `fArray[1]` to 1, `fArray[2]` to 2, and so on.

The number of initialization constants can determine the size of the array. For example, we could have determined that `fArray` has five elements just by counting the values within the braces. C++ can count as well. The following declaration is identical to the one above:

```
float fArray[] = {0.0, 1.0, 2.0, 3.0, 4.0};
```

It is not necessary to repeat the same value over and over to initialize a large a array. For example, the following initializes all 25 locations in `fArray` to 1.0:

```
float fArray[25] = {1.0};
```

Why use arrays?

On the surface, the ArrayDemo program doesn't do anything more than our earlier, nonarray-based programs did. True, this version can replay its input by displaying the numbers input before calculating the sum, but this hardly seems earth shattering.

Yet, the capability to redisplay the input values hints at a significant advantage to using arrays. Arrays enable the program to process a series of numbers multiple times. The main program was capable of passing the array of input values to `displayArray()` for display and then repass the same numbers to `sumArray()` for addition.

Arrays of arrays

Arrays are adept at storing sequences of numbers. Some applications require sequences of sequences. A classic example of this matrix configuration is the spreadsheet. Laid out like a chessboard, each element in the spreadsheet has both an x and a y offset.

C++ implements the matrix as follows:

```
int nMatrix[2][3];
```

This matrix is 2 elements in one dimension and 3 elements in another dimension, equalling 6 elements. As you might expect, one corner of the matrix is `nMatrix[0][0]`, while the other corner is `nMatrix[1][2]`.

Note

Whether you consider `nMatrix` to be 10 elements long in the *x* dimension or in the *y* dimension is a matter of taste.

A matrix may be initialized in the same way that an array is:

```
int nMatrix[2][3] = {{1, 2, 3}, {4, 5, 6}};
```

This initializes the three-element array `nMatrix[0]` to 1, 2, and 3 and the three-element array `nMatrix[1]` to 4, 5, and 6, respectively.

Arrays of Characters

The elements of an array can be of any C++ variable type. Arrays of floats, doubles, and longs are all possible; however, arrays of characters have particular significance.

An array of characters containing my first name would appear as:

```
char sMyName[] = {‘S’, ‘t’, ‘e’, ‘p’, ‘h’, ‘e’, ‘n’};
```

The following small program displays my name to the MS-DOS window, the standard output device.

```
// CharDisplay - output a character array to
//               standard output, the MS-DOS window
#include <stdio.h>


#include <iostream.h>

// prototype declarations
void displayCharArray(char sArray[], int nSize);

int main(int nArg, char* pszArgs[])
{
    char cMyName[] = {‘S’, ‘t’, ‘e’, ‘p’, ‘h’, ‘e’, ‘n’};
    displayCharArray(cMyName, 7);
    cout << “\n”;
    return 0;
}

// displayCharArray - display an array of characters
//                    by outputing one character at
//                    a time
void displayCharArray(char sArray[], int nSize)
```

```
{
    for(int i = 0; i< nSize; i++)
    {
        cout << sArray[i];
    }
}
```

This program works fine, but it is inconvenient to pass the length of the array around with the array itself. We avoided this problem when inputting integers from the keyboard by making up the rule that a negative number terminated input. If we could make the same rule here, we wouldn't need to pass the size of the array — we would know that the array was complete when we encountered the special code character.

Let's use the code 0 to mark the end of a character array.

Note

The character whose value is 0 is not the same thing as 0. The value of 0 is 0x30. The character whose value is 0 is often written as `\0` just to make the distinction clear. Similarly, the character `\y`'is the character whose numeric value is y. The character `\0` is known as the null character.

Using that rule, the previous small program becomes:

```
// DisplayString - output a character array to
//                 standard output, the MS-DOS window
#include <stdio.h>


#include <iostream.h>

// prototype declarations
void displayString(char sArray[]);

int main(int nArg, char* pszArgs[])
{
    char cMyName[] =
              {'S', 't', 'e', 'p', 'h', 'e', 'n', '\0'};
    displayString(cMyName);
    cout << "\n";
    return 0;
}
```

```
// displayString - display a character string
//                 one character at a time
void displayString(char sArray[])
{
    for(int i = 0; sArray[i] != 0; i++)
    {
        cout << sArray[i];
    }
}
```

The bolded declaration of `cMyName` declares the character array with the extra character '\0' on the end. The displayString program iterates through the character array until a null character is encountered.

The function `displayString()` is simpler to use than its `displayCharArray()` predecessor. It is no longer necessary to pass along the length of the character array. Further, `displayString()` works when the size of the character string is not known at compile time. For example, this would be the case if the user were entering a string of characters from the keyboard.

I have been using the term *string* as if it were a fundamental type, such as `int` or `float`. At the time of its introduction, I mentioned that *string* is actually a variation of an existing type. As you see here, a *string* is a null-terminated character array.

C++ provides an optional, more convenient means of initializing a string by using double quotes rather than the single quotes used for characters. The line

```
char szMyName[] = "Stephen";
```

is exactly equivalent to the line

```
char cMyName[] = {'S', 't', 'e', 'p', 'h', 'e', 'n', '\0'};
```

in the previous example.

Note

The naming convention used here is exactly that: a convention. C++ does not care; however, the prefix sz stands for zero-terminated string.

Note

The string "Stephen" is eight characters long, not seven —the null character after the n is assumed.

10 Min. To Go

Manipulating Strings

The C++ programmer is often required to manipulate strings.
Although C++ provides a number of string manipulation functions, let's write our own to get an idea of how these functions might work.

Our own concatenate function

Let's begin with a simple, if somewhat lengthy, C++ program to concatenate two strings.

```
// Concatenate - concatenate two strings
//               with a " - " in the middle
#include <stdio.h>


#include <iostream.h>

// the following include file is required for the
// str functions
#include <string.h>

// prototype declarations
void concatString(char szTarget[], char szSource[]);

int main(int nArg, char* pszArgs[])
{
    // read first string...
    char szString1[256];
    cout << "Enter string #1:";
    cin.getline(szString1, 128);

    // ...now the second string...
    char szString2[128];
    cout << "Enter string #2:";
    cin.getline(szString2, 128);

    // ...concatenate a " - " onto the first...
    concatString(szString1, " - ");
    // strcat(szString1, " - ");



    // ...now add the second string...
    concatString(szString1, szString2);
    // strcat(szString1, szString2);

    // ...and display the result
    cout << "\n" << szString1 << "\n";

    return 0;
}

// concatString - concatenate the szSource string
//                to the end of the szTarget string
void concatString(char szTarget[], char szSource[])
{
    // find the end of the first string
    int nTargetIndex = 0;
    while(szTarget[nTargetIndex])
    {
        nTargetIndex++;
    }

    // tack the second to the end of the first
    int nSourceIndex = 0;
    while(szSource[nSourceIndex])
    {
        szTarget[nTargetIndex] =


            szSource[nSourceIndex];
        nTargetIndex++;
        nSourceIndex++;
    }

    // tack on the terminating null
    szTarget[nTargetIndex] = '\0';
}
```

The function `main()` reads two strings using the `getline()` function.

Note **The alternate cin > szString reads up to the first space. Here we want to read until the end of the line.**

Function `main()` concatenates the two strings using the `concatString()` function before outputting the result.

The `concatString()` concatenates the second argument, `szSource`, onto the end of the first argument, `szTarget`.

The first loop within `concatString()` iterates through the string `szTarget` until `nTargetIndex` references the null at the end of the string.

Tip **The loop** `while(value == 0)` **is the same as** `while(value)` **because** `value` **is considered false if it's equal to 0, and true if equal to anything other than 0.**

The second loop iterates through the `szSource` string, copying elements from that string to `szTarget` starting with the first character in `szSource` and the null character in `szTarget`. The loop stops when `nSourceIndex` references the null character in `szSource`.

The `concatString()` function tacks a final null character to the resulting target string before returning.

Note **Don't forget to terminate the strings that you construct programmatically. You will generally know that you forgot to terminate your string if the string appears to contain "garbage" at the end when displayed or if the program crashes when you next try to manipulate the string.**

The result of executing the program is shown below.

```
Enter string #1:This is the first string
Enter string #2:THIS IS THE SECOND STRING

This is the first string - THIS IS THE SECOND STRING
Press any key to continue
```

Never

It is very tempting to write C++ statements such as the following:

```
char dash[] = " - ";
concatString(dash, szMyName);
```

This doesn't work because `dash` is given just enough room to store four characters. The function will undoubtedly overrun the end of the `dash` array.

C++ string-handling functions

C++ provides significant string capability in the > and << stream functions. You will see some of this capability in Session 28. At a more basic level, C++ provides a set of simple functions shown in Table 11-1.

Table 11-1
C++ Library Functions for Manipulating Strings

Name	**Operation**
`int strlen(`*string*`)`	Returns the number of characters in a string
`void strcat(`*target*`, `*source*`)`	Concatenates the source string to the end of the target string
`void strcpy(`*target*`, `*source*`)`	Copies a string to a buffer
`int strstr`	Finds the first occurrence of one string in another
`int strcmp(`*source1*`, `*source2*`)`	Compares two strings
`int stricmp(`*source1*`, `*source2*`)`	Compares two strings without regard to case

In the Concatenate program, the call to `concatString()` could have been replaced with a call to `strcat()`, which would have saved us the need to write our own version:

```
strcat(szString1, " - ");
```

Note

You need to add the statement `#include <string.h>` to the beginning of any program that uses the `str`. . . functions.

Wide characters

The standard C++ `char` type is an 8-bit field capable of representing the values from 0 to 255. There are 10 digits, plus 26 lowercase letters, plus 26 uppercase letters. Even if various umlauted and accented characters are added, there is still more than enough range to represent the Roman alphabet set plus the Cyrillic alphabet.

Problems with the `char` type don't arise until you begin to include the Asian character sets, in particular the Chinese and Japanese kanjis. There are literally thousands of these symbols — many more than the lowly 8-bit character set.

C++ includes support for a newer character type called `wchar` or wide characters. While this is not an intrinsic type like `char`, numerous C++ functions treat it as if it were. For example, `wstrstr()` compares two wide character sets. If you are writing international applications and need access to Asian languages, you will need to use these wide character functions.

Obsolescent Output Functions

C++ also provides a set of lower-level input and output functions. The most useful is the `printf()` output function.

Note

These are the original C input and output functions. Stream input and output didn't come along until after the introduction of C++.

In its most basic form, `printf()` outputs a string to `cout`.

```
printf("This string is output to cout");
```

The `printf()` function performs output using a set of embedded format control commands, each of which begins with a `%` sign. For example, the following prints out the value of an integer and a double variable.

```
int nInt = 1;
double dDouble = 3.5;
printf(“The int value is %i; the float value is %f”,
       nInt, dDouble);
```

The integer value is inserted at the point of the `%i`, whereas the double appears at the location of the `%f`:

```
The int value is 1; the float value is 3.5
```

Done!

Although difficult to use, the `printf()` function provides a level of output control that is difficult to achieve using stream functions.

Review

The array is nothing more than a sequence of variables. Each identical-type variable is accessed by an index to the array—much like the number portion of a house address identifies the houses on a street. The combination of arrays and loop commands, such as `for` and `while` loops, enable a program to easily process a number of elements. By far the most common C++ array type is the zero-terminated character array, commonly known as the character string.

- Arrays enable the program to loop through a number of entries quickly and efficiently using one of C++'s loop commands. For example, the increment portion of the `for` loop is designed to increment an index, while the condition portion is set up to detect the end of the array.
- Accessing elements outside the boundaries of an array is both common and dangerous. It is tempting to access element 128 of an array declared as 128 bytes long; however, because array indices start at 0, the final element is at offset 127, not 128.
- Terminating a character array with a special character enables a function to know where the array ends without the need to carry a character-length field. To facilitate this, C++ considers the character '\0', the character whose bit value is 0, an illegal, terminating, noncharacter. Programmers use the term *character string* or *ASCIIZ strings* for a null-terminated character array.
- Built in an Occidental world, the 8-bit C++ `char` types cannot handle the thousands of special characters required in some Asian languages. To handle these characters, C++ supports a special wide character, often referred to as `wchar`. C++ includes special functions to handle `wchar` strings in the standard C++ library of routines.

Quiz Yourself

1. What is the definition of an array? (See “What Is an Array?”)
2. What is the offset of the first and last elements of an array declared as `myArray[128]`? (See “Accessing Too Far into an Array.”)
3. What is a character string? What is the type of a character string? What terminates a string? (See “Arrays of Characters.”)