the notations int *arr and int arr[] have the identical meaning when (and only when) used in a function heading or function prototype. Both mean that arr is a pointer-to-int. However, the array notation version (int arr[]) symbolically reminds us that arr not only points to an int, it points to the first int in an array of ints. We'll use the array notation when the pointer is to the first element of an array, and we'll use the pointer notation when the pointer is to an isolated value. Don't forget that the notations int *arr and int arr[] are not synonymous in any other context. For example, you can't use the notation int tip[] to declare a pointer in the body of a function.

Given that the variable arr actually is a pointer, the rest of the function makes sense. As you might recall from the discussion of dynamic arrays in Chapter 4, you can use the bracket array notation equally well with array names or with pointers to access elements of an array. Whether arr is a pointer or an array name, the expression arr[3] means the fourth element of the array. And it probably will do no harm at this point to remind you of the following two identities:

arr[i] == *(ar + i)    // values in two notations
&arr[i] == ar + I      // addresses in two notations

Remember, adding 1 to a pointer, including an array name, actually adds a value equal to the size, in bytes, of the type to which the pointer points. Pointer addition and array subscription are two equivalent ways of counting elements from the beginning of an array.

**Implications of Using Arrays As Arguments**

Let's look at the implications of Listing 7.5. The function call sum_arr(cookies, ArSize) passes the address of the first element of the cookies array and the number of elements of the array to the sum_arr() function. The sum_arr() function assigns the cookies address to the pointer variable arr and assigns ArSize to the int variable n. This means Listing 7.5 doesn't really pass the array contents to the function. Instead, it tells the function where the array is (the address), what kind of elements it has (the type), and how many elements it has (the n variable). (See Figure 7.4.) Armed with this information, the function then uses the original array. Pass an ordinary variable, and the function works with a copy. But pass an array, and the function works with the original. Actually, this difference doesn't violate C++'s pass-by-value approach. The sum_arr() function still passes a value that's assigned to a new variable. But that value is a single address, not the contents of an array.