An important point about array arguments: A function has no way to check that we've passed it the correct array length. We can exploit this fact by telling the function that the array is smaller than it really is. Suppose that we've only stored 50 numbers in the b array, even though it can hold 100. We can sum just the first 50 elements by writing

```
total = sum_array(b, 50);    /* sums first 50 elements */
```

sum_array will ignore the other 50 elements. (Indeed, it won't know that they even exist!)

Be careful not to tell a function that an array argument is *larger* than it really is:

```
total = sum_array(b, 150);    /*** WRONG ***/
```

In this example, sum_array will go past the end of the array, causing undefined behavior.

Another important thing to know is that a function is allowed to change the elements of an array parameter, and the change is reflected in the corresponding argument. For example, the following function modifies an array by storing zero into each of its elements:

```
void store_zeros(int a[], int n)
{
  int i;

  for (i = 0; i < n; i++)
    a[i] = 0;
}
```

The call

```
store_zeros(b, 100);
```

will store zero into the first 100 elements of the array b. This ability to modify the elements of an array argument may seem to contradict the fact that C passes arguments by value. In fact, there's no contradiction, but I won't be able to explain why until Section 12.3.

Q&A If a parameter is a multidimensional array, only the length of the first dimension may be omitted when the parameter is declared. For example, if we revise the sum_array function so that a is a two-dimensional array, we must specify the number of columns in a, although we don't have to indicate the number of rows:

```
#define LEN 10

int sum_two_dimensional_array(int a[][LEN], int n)
{
  int i, j, sum = 0;
```