But if we view a as a one-dimensional array of integers (which is how it's stored), we can replace the pair of loops by a single loop:

```
int *p;
...
for (p = &a[0][0]; p <= &a[NUM_ROWS-1][NUM_COLS-1]; p++)
  *p = 0;
```

The loop begins with p pointing to a[0][0]. Successive increments of p make it point to a[0][1], a[0][2], a[0][3], and so on. When p reaches a[0][NUM_COLS-1] (the last element in row 0), incrementing it again makes p point to a[1][0], the first element in row 1. The process continues until p goes past a[NUM_ROWS-1][NUM_COLS-1], the last element in the array.

Although treating a two-dimensional array as one-dimensional may seem like cheating, it works with most C compilers. Whether it's a good idea to do so is another matter. Techniques like this one definitely hurt program readability, but—at least with some older compilers—produce a compensating increase in efficiency. With many modern compilers, though, there's often little or no speed advantage.

Q&A

Processing the Rows of a Multidimensional Array

What about processing the elements in just one *row* of a two-dimensional array? Again, we have the option of using a pointer variable p. To visit the elements of row i, we'd initialize p to point to element 0 in row i in the array a:

```
p = &a[i][0];
```

Or we could simply write

```
p = a[i];
```

since, for any two-dimensional array a, the expression a[i] is a pointer to the first element in row i. To see why this works, recall the magic formula that relates array subscripting to pointer arithmetic: for any array a, the expression a[i] is equivalent to *(a + i). Thus, &a[i][0] is the same as &(*(a[i] + 0)), which is equivalent to &*a[i], which is the same as a[i], since the & and * operators cancel. We'll use this simplification in the following loop, which clears row i of the array a:

```
int a[NUM_ROWS][NUM_COLS], *p, i;
...
for (p = a[i]; p < a[i] + NUM_COLS; p++)
  *p = 0;
```

Since a[i] is a pointer to row i of the array a, we can pass a[i] to a function that's expecting a one-dimensional array as its argument. In other words, a function that's designed to work with one-dimensional arrays will also work with a row belonging to a two-dimensional array. As a result, functions such as