C doesn't require that subscript bounds be checked; if a subscript goes out of range, the program's behavior is undefined. One cause of a subscript going out of bounds: forgetting that an array with n elements is indexed from 0 to $n - 1$, not 1 to n. (As one of my professors liked to say, "In this business, you're always off by one." He was right, of course.) The following example illustrates a bizarre effect that can be caused by this common blunder:

```
int a[10], i;

for (i = 1; i <= 10; i++)
  a[i] = 0;
```

With some compilers, this innocent-looking `for` statement causes an infinite loop! When `i` reaches 10, the program stores 0 into `a[10]`. But `a[10]` doesn't exist, so 0 goes into memory immediately after `a[9]`. If the variable `i` happens to follow `a[9]` in memory—as might be the case—then `i` will be reset to 0, causing the loop to start over.

An array subscript may be any integer expression:

```
a[i+j*10] = 0;
```

The expression can even have side effects:

```
i = 0;
while (i < N)
  a[i++] = 0;
```

Let's trace this code. After `i` is set to 0, the `while` statement checks whether `i` is less than `N`. If it is, 0 is assigned to `a[0]`, `i` is incremented, and the loop repeats. Note that `a[++i]` wouldn't be right, because 0 would be assigned to `a[1]` during the first loop iteration.

Be careful when an array subscript has a side effect. For example, the following loop—which is supposed to copy the elements of the array `b` into the array `a`—may not work properly:

```
i = 0;
while (i < N)
  a[i] = b[i++];
```

The expression `a[i] = b[i++]` accesses the value of `i` and also modifies `i` elsewhere in the expression, which—as we saw in Section 4.4—causes undefined behavior. Of course, we can easily avoid the problem by removing the increment from the subscript:

```
for (i = 0; i < N; i++)
  a[i] = b[i];
```