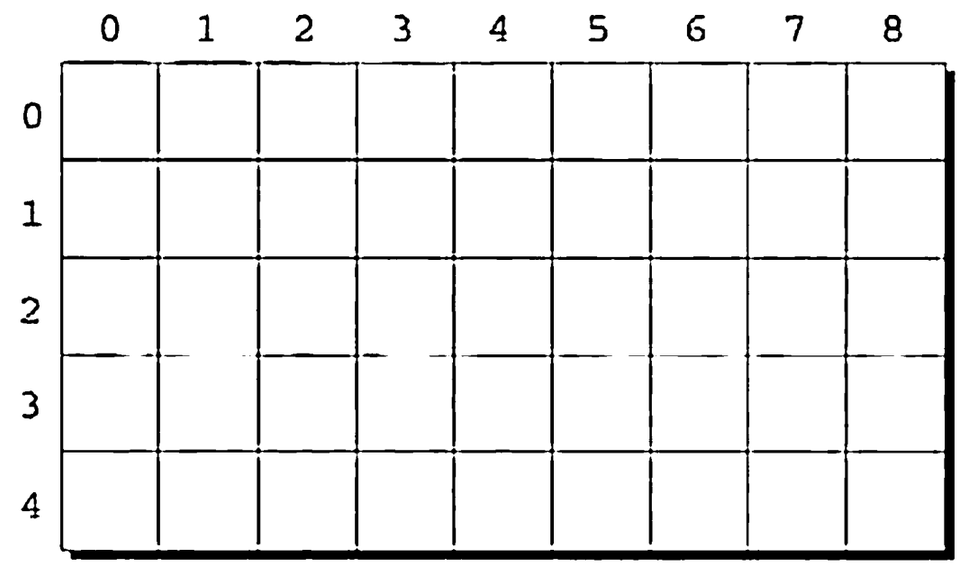
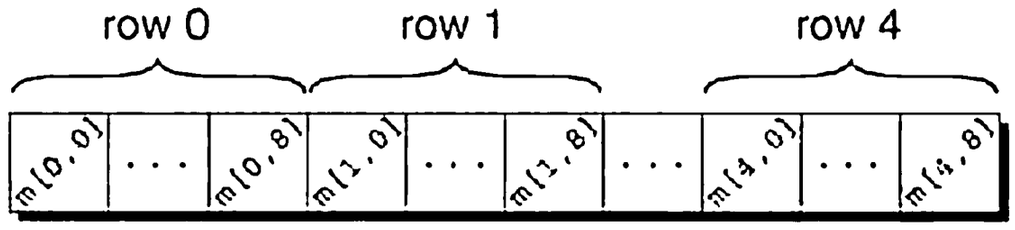


To access the element of `m` in row `i`, column `j`, we must write `m[i][j]`. The expression `m[i]` designates row `i` of `m`, and `m[i][j]` then selects element `j` in this row.

Resist the temptation to write `m[i,j]` instead of `m[i][j]`. C treats the comma as an operator in this context, so `m[i,j]` is the same as `m[j]`.

comma operator ➤ 6.3

Although we visualize two-dimensional arrays as tables, that's not the way they're actually stored in computer memory. C stores arrays in ***row-major order,*** with row 0 first, then row 1, and so forth. For example, here's how the `m` array is stored:



We'll usually ignore this detail, but sometimes it will affect our code.

Just as `for` loops go hand-in-hand with one-dimensional arrays, nested `for` loops are ideal for processing multidimensional arrays. Consider, for example, the problem of initializing an array for use as an identity matrix. (In mathematics, an *identity matrix* has 1's on the main diagonal, where the row and column index are the same, and 0's everywhere else.) We'll need to visit each element in the array in some systematic fashion. A pair of nested `for` loops—one that steps through every row index and one that steps through each column index—is perfect for the job:

```
#define N 10

double ident[N][N];
int row, col;

for (row = 0; row < N; row++)
  for (col = 0; col < N; col++)
    if (row == col)
      ident[row][col] = 1.0;
    else
      ident[row][col] = 0.0;
```

Multidimensional arrays play a lesser role in C than in many other programming languages, primarily because C provides a more flexible way to store multidimensional data: arrays of pointers.

arrays of pointers ➤ 13.7