Multi-dimensional Arrays, basal decay and dynamic allocation

Declaring a multi-dimensional array is simple enough, with or without an initializer:

```
int a[3][2];
int b[3][2] = {{1, 2}, {3, 4}, {5, 6}}; // All elements specified
int c[][2] = {{1,2}, {9, 10}}; // leading dimension inferred
```

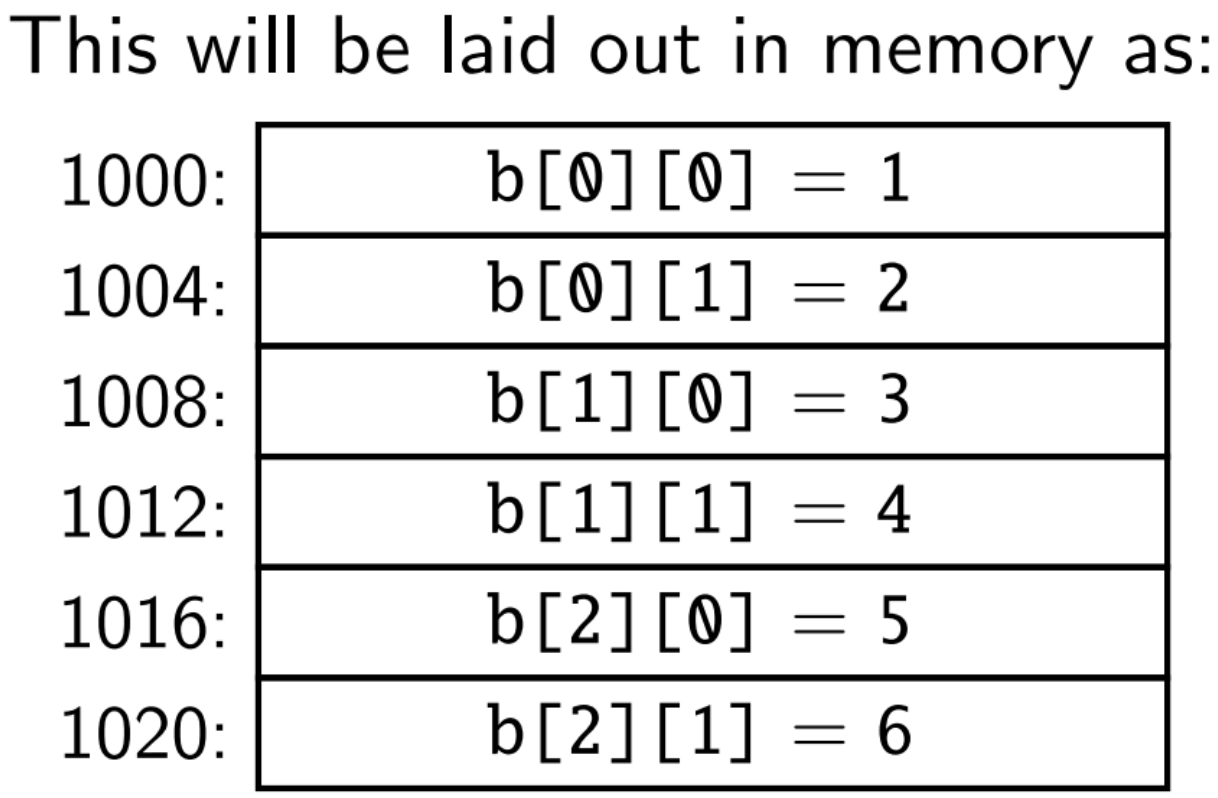
However, if you write:
`int d[][] = {{1,2}, {9, 10}}; // both dimensions inferred`

and then compile it, `gcc` complains: `note: declaration of 'd' as multidimensional array must have bounds for all dimensions except the first`

This is what all versions of the C standard demand

Multi-dimensional arrays are laid out in a "flattened" form in memory. If we declare:
`int b[3][2] = {{1, 2}, {3, 4}, {5, 6}};`

This will be laid out in memory as:



That is: `b[i][j]` is located in memory at an offset of `2*i + j` elements (each an 4-byte int) beyond the start of the array

C's convention of having the first index cause the largest jumps is known as row-major layout

Code to access multi-dimensional arrays is perfectly normal. Given:

```
#define ROWS 3
#define COLS 2
int b[ROWS][COLS] = {{1, 2}, {3, 4}, {5, 6}};
```

You might form the sum of all elements of all rows as:

```
int sum = 0;
for( int i=0; i<ROWS; i++ ) {
        for( int j=0; j<COLS; j++ ) {
                sum += b[i][j];
        }
}
```

You might form the sum of the product of all elements on each row as:

```
int sum = 0;
for( int i=0; i<ROWS; i++ ) {
  int prod = 1;
  for( int j=0; j<COLS; j++ ) {
    prod *= b[i][j];
  }
  sum += prod;
}
```

What about Basal decay for multi-dimensional arrays?

We know that, when it comes to a 1-D array, the array name decays to a basal pointer to the first element, as in:

```
double array[10];
double *bp = array;
```

Our basal pointer bp then works interchangeably with array, i.e. `bp[i] == array[i]`

But what happens with multi-dimensional arrays? Given:
`double mat[4][3] = { {1,2,3}, {4,5,6}, {7,8,9}, {10,11,12} };`

What does `mat` basally decay to? When we say “basal pointer to the first element”: what is the first element of `mat`, and in particular what is `mat`’s basal pointer type?

Our basal pointer to `mat` must still work interchangeably with `mat`, i.e. given a suitable base pointer `bp` (of a type we don’t yet know), we want `bp[i][j]` to work just like `mat[i][j]`

The C compiler has to be able to do the correct array element memory calculations through the basal pointer to deliver the correct elements

A few slides ago we said that `mat[i][j]` is located in memory at an offset of `NCOLS*i + j` cells beyond the start of the array, where NCOLS is the number of columns

So if `bp`'s type doesn't contain the value of NCOLS then C can't do the `bp[i][j]` calculations correctly: So any possible candidate for the basal decay type of a 2-D array needs to know that `NCOLS=3`

Type of `bp`: `double (*bp)[3]`: Pointer to array 3 of double

An Example:

```
double (*bp)[3] = mat;
double x = bp[1][2];
printf( "bp[1][2] = %g\n", x );
assert( x == 6 );
```

This compiles and prints the right answer

We can make it less hideous by using `typedef` to define a `rowt` type, giving:

```
typedef double rowt[3];
rowt *bp = mat;
```

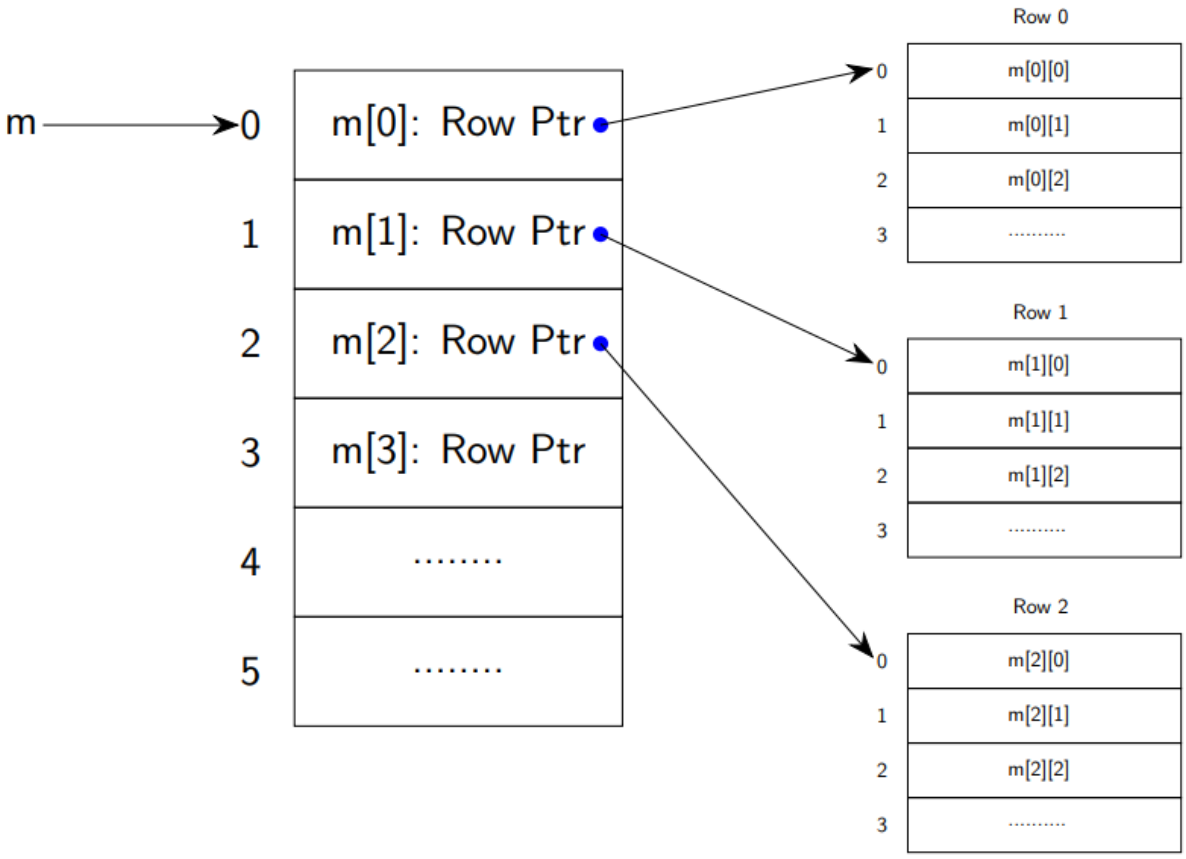
So far all our 2D arrays have been fixed size. But what if we want to be able to create, and work with, 2D arrays with variable numbers of rows and columns?

We can dynamically allocate our arrays, building an array of pointers to rows rather than a solid 2D matrix. Note that our row type is now a `double *` (a PtoM), but the matrix type is now a `rowt *` (also a PtoM)

```
typedef double *rowt;
typedef rowt *matrixt;

matrixt makeMatrix( int rows, int cols ) {
  matrixt m = malloc( rows * sizeof(rowt) );
  if( m == NULL ) return NULL;            // failure

  for( int i = 0; i < rows; i++ ) {
    m[i] = malloc( cols*sizeof(double) );
    if( m[i] == NULL ) {                  // failure: must clean up
      for( int j=0; j<i; j++ ) free( m[j] );
      free( m );
      return NULL;
    }
  }
  return m;
}
```




```
matrixt matrix = makeMatrix( rows, cols ); // allocate a matrix: size rows x cols

for( int i=0; i < rows; i++ ) {           // use the matrix normally.
  for( int j=0; j < cols; j++ ) {
    matrix[i][j] = 0.0;
  }
}

freeMatrix( matrix, rows );               // later: free the matrix
```

Accessing the dynamically allocated matrix looks identical to the fixed size ones. `free()`ing our matrix is easy, and needs to know the number of rows:

```
void freeMatrix( matrixt m, int rows ) {
        for( int i = 0; i < rows; i++ ) {
                free( m[i] );
        }
        free( m );
}
```

However, `makeMatrix()` makes a lot of `malloc()` statements (rows + 1), and a lot of the code dealt with cleaning up after a `malloc()` failure. How do we simplify this?

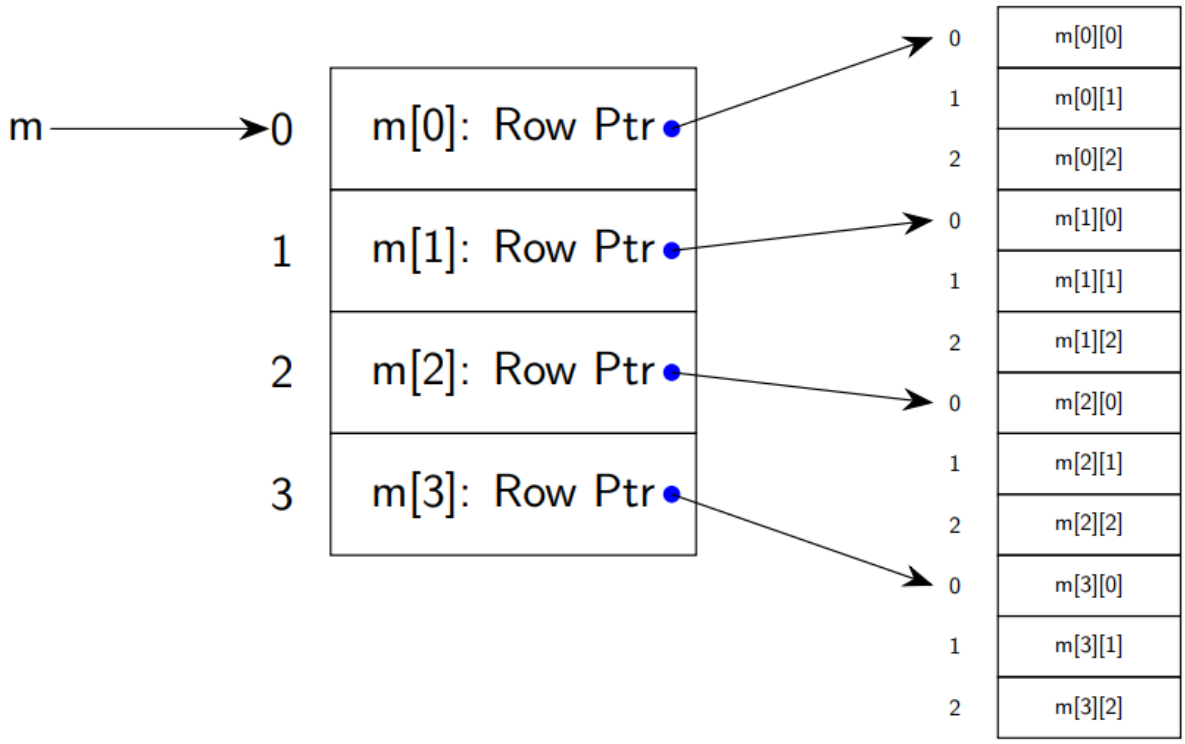
First, we `malloc()`ed the row pointer array as before. But instead of `malloc()`ing each row separately, we could `malloc()` a single chunk big enough for $\text{rows} * \text{cols}$ array elements, and then wire the row pointers up correctly:

```
typedef double *rowt;
typedef rowt *matrixt;

matrixt makeMatrix2( int rows, int cols ) {
        matrixt m = malloc( rows * sizeof(rowt) );
        if( m == NULL ) return NULL; // failure

        m[0] = malloc( rows*cols*sizeof(double) );
        if( m[0] == NULL ) {
                free( m );
                return NULL; // failure
        }

        // now wire up the row pointers correctly
        for( int i = 1; i < rows; i++ ) {
                m[i] = m[i-1] + cols;
        }
        return m;
}
```



Very little changes about how we use our matrix:

```
matrixt matrix = makeMatrix2( rows, cols ); // allocate a matrix:
size rows x cols

for( i=0; i < rows; i++ ) { // use the matrix normally
        for( j=0; j < cols; j++ ) {
                matrix[i][j] = 0.0;
        }
}

freeMatrix2( matrix ); // later: free the matrix
```

`free()`ing a `makeMatrix2()` matrix is a lot simpler - and doesn't need to take the number of rows:

```
void freeMatrix2( matrixt m ) {
        free( m[0] );
        free( m );
}
```

I prefer `makeMatrix2()` and `freeMatrix2()` over the originals, because:
`makeMatrix2()` has only 2 mallocs, and has simpler error checking as a result

`freeMatrix2()` is also simpler - and has fewer parameters

However, **There's More Than One Way To Do It**

To pass the matrix to a function, we must pass the matrix basal pointer, the number of rows and the number of columns

Print a matrix:

```
void printMatrix( matrixt matrix, int rows, int cols ) {
        for( int i = 0; i < rows; i++ ) {
                for( int j = 0; j < cols; j++ ) {
                        printf( "%8.5lf ", matrix[i][j] );
                }
                putchar('\n');
        }
}
```

Create a random matrix:

```
void randomMatrix( matrixt matrix, int rows, int cols ) {
        for( int i = 0; i < rows; i++ ) {
                for( int j = 0; j < cols; j++ ) {
                        matrix[i][j] = (double)rand()/RAND_MAX;
                }
        }
}
```

Add two (same sized) matrices together:

```
void addMatrices( matrixt A, matrixt B, matrixt R, int rows, int
cols ) {
        for( int i = 0; i < rows; i++ ) {
                for( int j = 0; j < rows; j++ ) {
                        R[i][j] = A[i][j] + B[i][j];
                }
```

```
        }
}
```

```
int main( void ) {
  int rows, cols;
  printf( "Enter rows cols: " );
  assert( scanf("%d %d", &rows, &cols) == 2 );

  matrixt ma = makeMatrix2(rows, cols);
  matrixt mb = makeMatrix2(rows, cols);
  matrixt mc = makeMatrix2(rows, cols);

  if( ma==NULL || mb==NULL || mc==NULL ) {
    fprintf( stderr, "Unable to allocate matrices!\n" );
    return 1;
  }

  randomMatrix( ma, rows, cols );
  randomMatrix( mb, rows, cols );
  addMatrices( ma, mb, mc, rows, cols );

  printf( "\n\nA = \n" ); printMatrix( ma, rows, cols );
  printf( "\n\nB = \n" ); printMatrix( mb, rows, cols );
  printf( "\n\nA + B = \n" ); printMatrix( mc, rows, cols );

  freeMatrix2( mc );
  freeMatrix2( mb );
  freeMatrix2( ma );

  return 0;
}
```

Output:

```
Enter rows cols: 4 4

A =
 0.84019  0.39438  0.78310  0.79844
 0.91165  0.19755  0.33522  0.76823
 0.27777  0.55397  0.47740  0.62887
 0.36478  0.51340  0.95223  0.91620


B =
 0.63571  0.71730  0.14160  0.60697
 0.01630  0.24289  0.13723  0.80418
 0.15668  0.40094  0.12979  0.10881
 0.99892  0.21826  0.51293  0.83911


A + B =
 1.47590  1.11168  0.92470  1.40541
 0.92795  0.44044  0.47245  1.57241
 0.43445  0.95491  0.60719  0.73768
 1.36371  0.73166  1.46516  1.75531
```

Running it via `valgrind` demonstrates that it runs without leaking any memory