Managing 2D Arrays

Simple Dynamic Allocation

A 2-dimensional `int` array of any fixed size can be allocated and initialized as follows:

```
const int Col = 4;
int Row = 2;

int (*A)[Col];
A = new int[Row][Col];

for (int i = 0; i < Row; i++)
   for (int j = 0; j < Col; j++)
      A[i][j] = i;
```

Spec addition:

> in order to make your lives a bit easier, we will guarantee that the maze for MazeCrawler will never have more than 20 columns.

Note: you must still allocate the array dynamically!!

The primary shortcoming of the approach above is that the column dimension MUST be a constant.

Passing a 2D array to a function also involves a requirement…

Passing a 2D Array to a Function

A 2-dimensional array parameter requires that the column dimension of the array be specified in the function prototype:

Actual number of columns in array A.

```
void init2D(int A[][C], int nRows, int nCols) {

   for (int i = 0; i < Row; i++)
      for (int j = 0; j < Col; j++)
         A[i][j] = i;
}
```

The reason relates to the manner in which the elements of the array A will be laid out in memory…

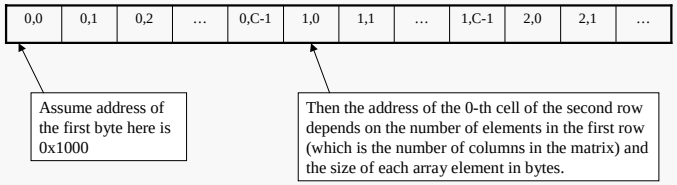
Example: 2D Array Parameter


```
#include <iostream>
#include <iomanip>
using namespace std;
const int Col = 4;
void init2D(int A[][Col], int nRows, int nCols) ;

void main() {
   int Row = 2;
   int (*A)[Col];
   A = new int[Row][Col];

   init2D(A, Row, Col);

   for (int i = 0; i < Row; i++) {
      for (int j = 0; j < Col; j++)
         cout << setw(5) << A[i][j];
      cout << endl;
   }
}

void init2D(int A[][Col], int nRows, int nCols) {

   for (int i = 0; i < nRows; i++)
      for (int j = 0; j < nCols; j++)
         A[i][j] = i;
}
```


Memory Layout for 2D Arrays

First, physical memory is simply a one-dimensional array of bytes, indexed by addresses.

C++ (like most languages) specifies that 2D arrays are to be mapped into this one-dimensional environment by storing the array cells row-by-row (known as row-major order). Assuming that the array has R rows and C columns, that results in a layout something like:



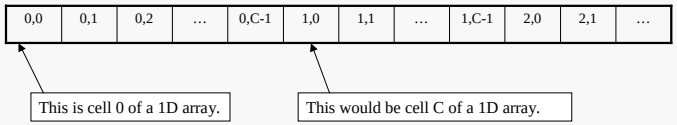
When you make an array reference, like `A[4][3]`, the compiler must generate code to calculate the address of the specified cell at runtime. The formula will necessarily involve the column dimension C of the matrix A.

There MUST be a Better Way

Well… of course… but as usual, it's a matter of picking which type of pain you wish to endure.

One approach is to allocate an array of pointers, viewed say as pointers to the columns in a 2D matrix, and then use each pointer to dynamically allocate a column of the desired size. That's syntactically (and conceptually) ugly but it WILL work.

Another approach is to simply allocate a one-dimensional array, say `A1D`, of the correct total size and then treat it as a 2D array. How? By taking on the management of the array indexing arithmetic yourself. (See the previous memory layout again.)



Now, if you want to logically refer to `A[i][j]`, you just have to do some simple arithmetic to calculate the correct index `k` and refer to `A1D[k]` instead.

2D Array Encapsulation

Of course, if you're going to do that, there's an advantage to be gained by wrapping the one-dimensional array inside a class and hiding all the messy translation details inside its member functions:

```
class Array2D {
private:
   int  *Data;
   int  nRows;                       // (logical) number of rows
   int  nCols;                       // (logical) number of columns
   int  Map(int R, int C) const;     // translate [R][C] to right 1D index
   bool ValidRC(int R, int C) const;  // check if [R][C] specify a cell
public:
   Array2D();
   Array2D(int R, int C);            // allocate array of R*C cells
   Array2D(const Array2D& Source);   // copy constructor and assignment
   Array2D& operator=(const Array2D& Source);
   bool Set(int R, int C, int Value); // store Value at A[R][C] (logically)
   int  Get(int R, int C) const;     // get Value at A[R][C] (logically)
   ~Array2D();
};
```

The idea is to allow the user to act as if there's really a normal 2D array under the hood, while supporting pure dynamic allocation (no requirements that any dimension be a constant). Of course, we also provide protection against array bounds errors as well.

Array2D Construction and Destruction

The constructor will provide a (logical) 2D array with any dimensions:

```
Array2D::Array2D(int R, int C) {

   nRows = R;
   nCols = C;
   Data = new int[nRows * nCols];
   if (Data == NULL) {
      nRows = nCols = 0;
   }
}
```

And destruction poses no new issues since `Data` is a simple one-dimensional array:

```
Array2D::~Array2D() {
   delete [] Data;
}
```


Array2D Implementation

The remainder of the implementation is left to the reader. A few notes…

The copy constructor is necessary in order to pass `Array2D` objects as parameters or use them as return values. The assignment overload is not used in the following example, but it's cheap to provide since a copy constructor must be written anyway.

The function `Map()` that provides the translation from 2D to 1D indices is simple. Draw a few pictures and play around with indices and you'll see the correct formula.

The `ValidRC()` function does not use `Map()` (or any other translation code).

It's fairly trivial to turn `Array2D` into a template.

Array2D Use


```
#include <iostream>
#include <iomanip>
using namespace std;
#include "Array2D.h"

void init2D(Array2D& A, int Rows, int Cols);
void print2D(const Array2D A, int Rows, int Cols, ostream& Out);

void main() {

   int R = 3, C = 5;
   Array2D myArray(R, C);

   init2D(myArray, R, C);
   print2D(myArray, R, C, cout);
}

void init2D(Array2D& A, int Rows, int Cols) {

   int row, col;
   for (row = 0; row < Rows; row++)
      for (col = 0; col < Cols; col++)
         A.Set(row, col, row);
}

// . . . continues . . .
```


Array2D Use


```
// . . . continued . . .

void print2D(const Array2D A, int Rows, int Cols, ostream& Out) {

   int row, col;
   for (row = 0; row < Rows; row++) {
      for (col = 0; col < Cols; col++)
         cout << setw(5) << A.Get(row, col);
      cout << endl;
   }
}
```

```
    0    0    0    0    0
    1    1    1    1    1
    2    2    2    2    2
```