MECH215
Lecture Notes 7

Higher Dimensional Arrays

C++ allows for 2-dimensional and higher dimensions arrays. A 2D array is commonly referred to as a matrix in math, but in programming, it is always referred to as an array.

Here is a pictorial representation of a 2-d array consisting of three rows and four columns:

```
        col 0   col 1   col 2  col 3
       -----------------------------
row 0 | 0        1        2      3
      |
row 1 | 1        2        3      4
      |
row 2 | 2        3        4      5
```

In C++, we would declare a 2-d array with the following declaration:

```
int array_2d[3][4];
```

The first number in [] refers to the number of rows, the second number enclosed in [] specifies the number of columns. NOTE: IN C++, ARRAY INDICES START FROM 0. The element found in row 0 and column 0 would be referred to a array_2d[0][0].

Here is an example program which initializes and prints out the values of a 3 by 4 matrix:

```
// author: Ted Obuchowicz and John Peach
// print2DArray.cpp
// example program illustrating use of 2D arrays

#include <iostream>

using namespace std;

int main() {

    //  declare a 2 dimensional array of integers
    const unsigned short ROW    = 3;        // Number of rows
    const unsigned short COLUMN = 4;        // Number of columns
    int array2d[ROW][COLUMN];

    // Populate each cell with the sum of the row and column index
    for(unsigned short row = 0; row < ROW; row++) {
```

```
        for(unsigned short column = 0; column < COLUMN; column++) {
            array2d[row][column] = row + column;
        }
    }

    // print out the array formatted into rows and columns
    for(unsigned short row = 0; row < ROW; row++) {
        for(unsigned short column = 0; column < COLUMN; column++) {

            // Print the value and a tab to align everything
            cout << array2d[row][column] << "\t";

            // After the last column, put and end of line
            if ( column  == COLUMN - 1) {
                cout << endl;
            }
        }
    }

    return 0;
}
```

The program output is:

```
0    1    2    3
1    2    3    4
2    3    4    5
```

How 2d Arrays Are Stored In Memory

Main memory in a computer is viewed by the compiler as a linear sequence of bytes. Hence, a two-dimensional array must be mapped onto this linear array of storage locations by the compiler. The compiler stores EACH ROW of a 2D array in successive memory locations. For example, our 2D array, from the previous program, can be accessed as if it was a 1D array by doing the following:

```
int* array1d = &array2d[0][0];
```

If we were to print out the elements in array1d we would have:

array1d[0] => 0
array1d[1] => 1
array1d[2] => 2
array1d[3] => 3
array1d[4] => 1
array1d[5] => 2
array1d[6] => 3
array1d[7] => 4

array1d[8] => 2
array1d[9] => 3
array1d[10] => 4
array1d[11] => 5

Such a mapping of the elements of a two-dimensional array onto a one-dimensional array is called "ROW-MAJOR FORM". Most programming languages use row-major form. However, FORTRAN, an older but very popular language used in engineering uses column-major form. Components of C++ and FORTRAN can be mixed together in the same application. However, access to arrays must be adjusted. This is not hard to do but you must remember to do it.

The compiler uses the following address translation mechanism when accessing two dimensional arrays stored in row-major form in main memory

```
array2d[i][j] = array1d[ i*(number of columns) + j ];
```

For example, in the above example, there are 4 columns in our 2D array, thus

array2d[0][2] is found in array1d[0*4 + 2] = array1d[2]

and

array[1][3] is found in array1d[1*4 + 3] = array1d[7]

It is important to note that the name of the array is associated with the starting address of the first element in the array. In reality, the compiler simply translates [row][column] indices of a 2D array into an offset from some starting address of where the elements of the array are stored in.

Here is a program which prints out the 12 addresses which are used to store our 3 x 4 matrix:

```
// author: Ted Obuchowicz and John Peach
// print2DArrayAddress.cpp
// example program illustrating use of 2D arrays

#include <iostream>

using namespace std;

int main() {

    //  declare a 2 dimensional array of integers
    const unsigned short ROW    = 3;        // Number of rows
    const unsigned short COLUMN = 4;        // Number of columns
    int array2d[ROW][COLUMN];
    int* array1d = &array2d[0][0];
```

```
    // Populate each cell with the sum of the row and column index
    for(unsigned short row = 0; row < ROW; row++) {
        for(unsigned short column = 0; column < COLUMN; column++) {
            array2d[row][column] = row + column;
        }
    }

    // print out the array formatted into rows and columns
    for(unsigned short row = 0; row < ROW; row++) {
        for(unsigned short column = 0; column < COLUMN; column++) {
            // print out single element's indexes and address
            cout <<  "Element found at row = [" << row << "]"
                 <<  " and column = [" << column << "]"
                 <<  " stored at address "
                 <<  (unsigned long) &array2d[row][column] << endl;
        }
    }

    return 0;
}
```

The output produced by this program is:

```
Element found at row = [0] and column = [0] stored at address 140737404452064
Element found at row = [0] and column = [1] stored at address 140737404452068
Element found at row = [0] and column = [2] stored at address 140737404452072
Element found at row = [0] and column = [3] stored at address 140737404452076
Element found at row = [1] and column = [0] stored at address 140737404452080
Element found at row = [1] and column = [1] stored at address 140737404452084
Element found at row = [1] and column = [2] stored at address 140737404452088
Element found at row = [1] and column = [3] stored at address 140737404452092
Element found at row = [2] and column = [0] stored at address 140737404452096
Element found at row = [2] and column = [1] stored at address 140737404452100
Element found at row = [2] and column = [2] stored at address 140737404452104
Element found at row = [2] and column = [3] stored at address 140737404452108
```

Let us examine this output to dig a little deeper into the address translation mechanism employed by the compiler. We see the first element array2d[0][0] is stored in memory location with address 140737404452064 (we used the cast operator to convert the hexadecimal address returned by the & operator into an unsigned long for readability purposes).

If we use our address translation formula:

array2d[0][1] = array1d[0*4 + 1] = array1d[1] // 4 is the number of columns

The compiler treats the name of an array as the starting address in main memory where the first element of the array is stored. Hence, the label we call "array2d" really represents address 140737404452064. What the compiler really does during the address conversion process is:

array2d[i][j] = starting address + sizeof(int)*((i*4)+j) // 4 is the number of columns

As an example,

```
array_2d[0][1] = 140737404452064 + 4*(0*4 + 1)
               = 140737404452064 + 4*(1)
               = 140737404452068
```

Note: The first 4 in "4*(0*4 + 1)" is the size of an int in bytes and the second 4 is the number of columns in the array. Recall that the sizeof function is used to determine the number of bytes that a given data type occupies. An int occupies 4 bytes of
main memory.

As a final example:

```
array_2d[2][2] = 140737404452064 + 4*(2*4 + 2)
               = 140737404452064 + 4*(10)
               = 140737404452104
```

which is the correct address.

Passing Higher-Dimensional Arrays to Functions

When passing a two-dimensional (and higher dimensional) array to a function, C++ requires that the second (and higher) dimensions be passed directly (within the []). The first dimension may be omitted from the first set of []. The reason for this is that the compiler requires the size of the second and higher dimensions in order to perform the address translation mechanism discussed previously. However, if the dimensions of the matrix are not known at compile time, it is possible to pass the memory address of the first element in the array plus parameters that give the dimensions. The, the programmer can calculate the memory location for a desired cell manually.

The following program fails to compile since we did not specify the size of the second dimension directly to the function:

```
// author: Ted Obuchowicz and John Peach and John Peach
// pass2DArrayError.cpp
// example program illustrating use of improperly defined array size

#include <iostream>

using namespace std;

const int COLUMN_SIZE = 4;
const int ROW_SIZE    = 3;

void PrintMatrix(int some_array[][], int row, int col);
```

```
// When passing multidimensional arrays to a function, the
// size of each dimension other than the first must be specified.
// This program will not compile!
// The following error will be reported by g++:
//   declaration of ‘data’ as multidimensional array must have
//   bounds for all dimensions except the first

int main() {

    //  declare a 2 dimensional array of integers
    int array2d[ROW_SIZE][COLUMN_SIZE];

    for(int row = 0; row < ROW_SIZE; row++) {
        for(int column = 0; column < COLUMN_SIZE; column++) {
            array2d[row][column] = row + column;
        }
    }

    // print out the array formatted into rows and columns
    PrintMatrix(array2d, ROW_SIZE, COLUMN_SIZE);

    return 0;
}

void PrintMatrix(int data[][], int row, int column) {

    // Cycle through the rows and column
    for(int i = 0; i < row; i++) {
        for(int j = 0; j < column; j++) {

            // print an element
            cout << data[i][j] << "\t";

            // Force a new line after the last column in every row
            if ( j  == column - 1 ) {
                cout << endl;
            }
        }
    }
}
```

The program generates the following compile time errors:

```
#> g++ pass2DArrayError.cpp
pass2DArrayError.cpp:12:29: error: declaration of ‘data’ as multidimensional array
must have bounds for all dimensions except the first
pass2DArrayError.cpp:12:30: error: expected ‘)’ before ‘,’ token
pass2DArrayError.cpp:12:32: error: expected unqualified-id before ‘int’
pass2DArrayError.cpp:40:29: error: declaration of ‘data’ as multidimensional array
must have bounds for all dimensions except the first
pass2DArrayError.cpp:40:30: error: expected ‘)’ before ‘,’ token
pass2DArrayError.cpp:40:32: error: expected unqualified-id before ‘int’
```

The solution is to pass the second dimension directly to the function. The following program declares a global constant integer called COLUMN_SIZE (set to 4), this constant is included in the second set of [] when passing the array to the function:

```
// author: Ted Obuchowicz and John Peach and John Peach
// pass2DArray.cpp
// example program illustrating use of improperly defined array size

#include <iostream>

using namespace std;

const int COLUMN_SIZE = 4;
const int ROW_SIZE    = 3;

void PrintMatrix(int some_array[][COLUMN_SIZE], int row, int col);

// When passing multidimensional arrays to a function, the
// size of each dimension other than the first must be specified.

int main() {

    //  declare a 2 dimensional array of integers
    int array2d[ROW_SIZE][COLUMN_SIZE];

    for(int row = 0; row < ROW_SIZE; row++) {
        for(int column = 0; column < COLUMN_SIZE; column++) {
            array2d[row][column] = row + column;
        }
    }

    // print out the array formatted into rows and columns
    PrintMatrix(array2d, ROW_SIZE, COLUMN_SIZE);

    return 0;
}
```

```
void PrintMatrix(int data[][COLUMN_SIZE], int row, int column) {

    // Cycle through the rows and column
    for(int i = 0; i < row; i++) {
        for(int j = 0; j < column; j++) {

            // print an element
            cout << data[i][j] << "\t";

            // Force a new line after the last column in every row
            if ( j  == column - 1 ) {
                cout << endl;
            }
        }
    }
}
```

Note in the function prototype we pass the parameters:

```
void PrintMatrix(int data[][COLUMN_SIZE], int row, int column);
```

data[][COLUMN_SIZE],
row, and
column.

The parameters row and column are used to control two nested for loops which access the array elements using array index notation:
`array2d[row][column]`

It is necessary to pass the second dimension size directly to the function as in int data[] [COLUMN_SIZE] since the value of the second dimension is needed by the compiler to perform the address translation mechanism.

As a byproduct of this requirement of passing higher dimensions sizes of an array to a function is that we would have to write different functions for different sized multidimensional arrays. There is a way of getting around this restriction, but it requires the use of pointers which is our next topic of discussion.