CS115 Lab: 2-D Arrays with Functions

Highlights of this lab:

In this lab, you will:

- See a graphical representation of a 2 dimensional array.
- See how to declare 2 dimensional arrays in C++.
- See how to reference 2 dimensional arrays elements.
- See how to pass 2 dimensional arrays to functions.

Lab Exercise:

- Complete a simple C++ program to familiar with 2-Dimensional Array as function arguments.

Click the little computer above for a detailed description.
NOTE: Your lab instructor will tell you what will be marked for this lab.

"New" Unix/Linux Commands

Here are some more Unix/Linux commands that you need to learn. Remember that you can enter **man *command*** to get a complete description of any Unix/Linux command and its options.

Command	Description
cal *[month #] year*	Prints a calendar of the specified year, e.g. **cal 2010**. If a month number is specified, prints only that month, e.g. **cal 3 2010** (for March 2010)
cat *file1 [file2 ...]*	Con**cat**enate (join together) specified files and direct the output to the standard output device - the screen. This command is commonly used to display the contents of one file on the screen. (It's simpler than getting in and out of an editor.)
date	Print the current time and date.
who	Lists who is logged into a machine. It provides information such as the user's login name and the time when the user logged on.
w	Lists who is logged into a machine. Provides information such as the user's login name and the time when the user logged on. It also provides information about what the user is currently doing.
sort	Sorts the input stream or the contents of files. To sort the contents of a file, use **sort filename**.
wc	Displays the number of lines, words and characters in a file. To display only the number of lines, you can use **wc -l**.
file *file*	Perform tests on a file to determine its type. Useful if you want to make sure a file is not an executable before you try to edit it.
cmp *file1 file2*	Compare two files to see if they are the same. Reports just the first difference unless you specify -l
diff *file1 file2*	Displays the differences between *file1* and *file2*. This lists the changes necessary to convert *file1* to *file2*.
find *path option*	Search down directories for a file. e.g. **find . / -name gold.cpp** would search in the current directory and in all subdirectories for the file called *gold.cpp*
grep [option] string [file(s)]	Search for a string pattern in a file.There are several options, e.g. **grep** `namespace *.cpp` would search the current directory for the string "namespace" in all .cpp files and show the lines in each file where the string occurs. e.g. **grep -n** `namespace *.cpp` would perform the same search but also give the line numbers in which the string was found.
ps	Lists the processes that are running for a terminal. To see all the processes that are running for you, use **ps -fu yourusername**. This command is often used with kill.
kill *[option] processid*	Kill the process specified. e.g. **kill -9 1455** would perform a "sure kill" (option 9) on process id "1455". This is a handy command if you change your mind after sending a job to the printer and want to delete it from the queue. See the **lpq** command to see how you can query the print queue for process ids.
lpq -P*[printername]*	Query the specified printer to see active jobs. Reports process ids of jobs. e.g. **lpq -Pcl122**
quota -v	Show how much disk space you are using ("usage") on a multi-user Unix system and what your limit is ("quota"). The numbers given refer to kilobytes of space.

Notes:

- Some commands such as sort, cat, and wc will accept input from the keyboard.
  If you type these commands, without specifying an argument, your command prompt will not return until you press CTRL-d (which indicates an end of file on Unix)

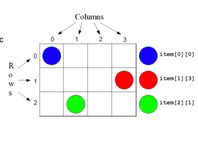
Graphical Representation of 2-Dimensional Arrays

A familiar example of a two dimensional array is what you see when you use a spreadsheet program.There are cells organized into a grid of rows and columns.

In C++ the first row is numbered **0** and the first column is numbered **0**. These numbers are enclosed in square brackets. When you refer to a specific cell location you would specify:

```
item[row_number][column_number]
```

The blue ball in the top left corner of the diagram to the right here would be `item[0][0]`. If you wanted to refer to the empty cell in the bottom right corner you would refer to `item[2][3]`.



Format for Declaring 2-Dimensional Arrays

As just explained, the subscripts for an array reference are enclosed in square brackets. Use these also when you want to declare the array.

e.g. General Format: `data_type variable_name [ row_size ][column_size]`
e.g. Declare an integer array of 2 rows, 3 columns: `int nums[2][3];`

Often you will see a constant declared to hold the array size, and then this constant used in the array declaration.
I.e.

```
const unsigned int R_SIZE = 2;
const unsigned int C_SIZE = 3;
int nums[R_SIZE][C_SIZE];
```

Remember that array elements can also be initialized when the array is declared. Examine the following code segment carefully to determine the precise syntax required.

```
int nums[2][3] =
    {
    {1, 2, 3},
    {4, 5, 6}
    };
```

Format for Referencing 2-Dimensional Arrays

When you reference a particular array element, use a number, constant, variable or expression in the brackets.

- `nums[0][1];`
- `nums[A_SIZE][A_SIZE];`
- `nums[i][j];`
- `nums[i+1][j+3];`

Of course the values in the brackets must be within the defined limits of the array size. Refering to `nums[788][0]` would result in an error if `nums` was only declared as having 2 rows. The `0` is OK though, because the numbering starts at 0, not 1.

In most programs, 2-dimensional arrays go hand in hand with nested FOR loops because that is a quick easy way to reference all of the array elements in a 2-D array. For example, here's a little program segment that initializes a 3 x 5 integer array called nums.

```
int nums[3][5];
int i, j;

for ( i = 0; i < 3; i++)      // march down the rows
{
    for ( j = 0; j < 5; j++)  // march across the columns
    {
        nums[i][j] = 0;    // set array element to zero
    }
}
```

This starts at the `zero'th` row, and accesses each column of that row from left to right.
I.e. `nums[0][0] nums[0][1] nums[0][2] nums[0][3] nums[0][4]`

It then moves down to the next row and works accross the columns for that row.
I.e. `nums[1][0] nums[1][1] nums[1][2] nums[1][3] nums[1][4]`

Here is a complete program summarizing everything discussed so far.

```
/*******************************************************
*
*  FileName:    2Darray.cpp
*  Author:      Ada Lovelace
*  Purpose:
*               Demonstrate manipulation of a 2D array.
*
*******************************************************/
#include <iostream>
using namespace std;

int main()
{
    int nums[3][5];
    int i, j;

    for ( i = 0; i < 3; i++)        // march down the rows
    {
        for ( j = 0; j < 5; j++)    // march across the columns
        {
            nums[i][j] = 0;       // set array element to zero
        }
    }
}
```

Passing 2-D Arrays to Functions

Before we examine how to pass a 2-D array in detail here's a sample program that takes the initialization loop from the last program and turns it into a function:

```
#include <iostream>
using namespace std;

const unsigned int R_SIZE = 2;
const unsigned int C_SIZE = 3;

void initArray( int twoDArray[][C_SIZE] );  // function prototype

int main()
{
   int nums[R_SIZE][C_SIZE];
   initArray(nums);

}

// Function:  initArray
// Purpose:   To initialize an array.
// Parameters: Base address of an array.
// Returns:   void
// ------------------------------------------------------
void initArray(int twoDArray[][C_SIZE])
{

   int i, j;

   for ( i = 0; i < R_SIZE; i++)
   {
       for ( j = 0; j < C_SIZE; j++)
       {
          twoDArray[i][j] = 0;
       }
   }
} // end initArray function
```

Now, let's take a closer look at how those arrays are passed. In C++, arrays are **not** passed by ***value*** to functions, they are passed by ***reference***. Because of this, you do **not** have to use the **&** reference character. You simply pass the ***base address*** of an array to a function. To do this, just supply the name of the array like this:

```
initArray(nums);
```

Notice that only the array name `nums` appears as an argument; it is not followed by any subscripts at all.

Before we talk about declaring the function remember how we declared the array:

```
const unsigned int R_SIZE = 2
const unsigned int C_SIZE = 3
int nums[R_SIZE][C_SIZE];
```

We used constants for the number of rows and columns.

You may have noticed in the review at the beginning of this section, that the ***column size*** is specified in both:

- the function prototype at the top of the program, and
  `void initArray( int twoDArray[][C_SIZE]);`
- the function declaration line.
  `void initArray(int twoDArray[][C_SIZE])`

This is because arrays are stored in memory in ***one row after another***. The function needs to know how many columns there are in a 2D array so it can find the next row. It's as if 2D arrays were stored as a big 1D array. In a 2D array cells referenced like this:

```
   int array[R_SIZE][C_SIZE];
   array[3][2];
```

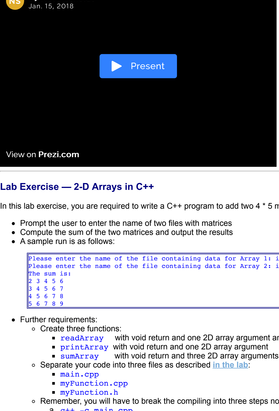
are actually being referenced like this:

```
   int array[R_SIZE*C_SIZE];
   array[3*C_SIZE + 2];
```

but the compiler doesn't know what value to multiply by unless you supply it in the function definition.

It is also important to note that **arrays are not a legal return type**.

Separate Compilation and Linking



Lab Exercise — 2-D Arrays in C++

In this lab exercise, you are required to write a C++ program to add two 4 * 5 matrices and then output the results to the screen.

- Prompt the user to enter the name of two files with matrices
- Compute the sum of the two matrices and output the results
- A sample run is as follows:

```
Please enter the name of the file containing data for Array 1: input1.txt
Please enter the name of the file containing data for Array 2: input2.txt
The sum is:
2 3 4 5 6
3 4 5 6 7
4 5 6 7 8
5 6 7 8 9
```

- Further requirements:
  - Create three functions:
    - `readArray` with void return and one 2D array argument and one string argument
    - `printArray` with void return and one 2D array argument
    - `sumArray` with void return and three 2D array arguments
  - Separate your code into three files as described in the lab:
    - `main.cpp`
    - `myFunction.cpp`
    - `myFunction.h`
  - Remember, you will have to break the compiling into three steps now:
    a. `g++ -c main.cpp`
    b. `g++ -c myFunction.cpp`
    c. `g++ main.o myFunction.o -o main`
  - The two input files are available by clicking: input1.txt and input2.txt

[im error occurred while processing this directive]