CS4115 Week03 Lab Exercise

Lab Objective: The objective of this lab is to write a program, `matmult`, to read in two matrices and then to multiply them, printing the result to the screen. In addition to being a small exercise in writing a C++ program this will stand us in good stead later when we look at other methods for multiplying matrices. Here's a quick summary of the steps that you should follow:

❶ Set up a directory (folder) for this week's lab work

❷ Create the skeleton of your program in a file called `matmult.cc`

❸ Read from the command line the dimensions of the two arrays

❹ Write a function that will read in an array

❺ Write a function that performs the multiplication of two matrices / arrays

❻ Complete this lab by the end of the week for submitting through the `handin` mechanism.

In Detail

❶ Refer back to last week's lab sheet for instructions on how to create a weekly lab directory. This week it will be `week03` or, to give it its full path, `~/cs4115/labs/week03`. Speaking of paths, you may recall the `./lin` vs. `lin` issue from last week and your `PATH` variable being adjusted to include `.`, the current working directory. I am going on the assumption that you have adjusted your `PATH` variable so from now on I will simply refer to the execution of programs without the leading `./`.

❸ `matmult` will run by being given the dimensions of the two arrays to multiply together. These 4 values will be read as command-line arguments when the `matmult` command is first given. So a typical invocation of the program might be

```
matmult 5 6 6 9
```

This would specify that a matrix of size 5×6 is to be multiplied by a matrix of size 6×9. When I say a matrix of size 5×6 I mean 5 rows of 6 numbers each. The numbers underneath each other form a column so we can talk about a matrix of "5 rows and 6 columns."

The four numbers on the command-line can be accessed in the same style as last week

```
int n = atoi(argv[1]);
```

except that you will need to do this 4 times, each accessing a different element, 1 to 4, of the command-line arguments array, `argv`. Of course you will need to call the variables different names, too!

Did you know...

`argv[0]` holds the name of the command that is currently running; the remaining elements of the array `argv[]` hold the arguments given on the command line. The count of arguments is given by `argc` which is also to be found in the declaration of the function `main()`.

❹ Your first task should be to read a matrix from the keyboard into an array in our program. This is a perfect task for assigning to a separate C++ function. You might want to call it something like `readArr()`. What should the parameter list be of this function? You will certainly want to specify the dimensions of the matrix to be read, but you should also pass in the name of the array that the values are to be read in to. Here's how I declared my function:

```
void readArr(int, int, double**);
```

In C++ you must **declare** a function before it is **called**, otherwise the compiler gets nervous about being told to use a function that it doesn't know about. So the declaration above keeps the compiler happy. In our program a typical call of the function might be

```
readArr(6, 4, A);
```

The first arguments are the array row and column counts and the third effectively amounts to saying that `A` is a two-dimensional data structure of `double`s – double-precision floating point numbers.

Before you can read the array, however, you must create a place to store it. If you knew the dimensions of the array you could declare a 2-D array of `double`s called `matrixData` with

```
double matrixData[10][50];
```

but that would be too inflexible as we could never read a matrix larger than that. We need to base our matrix on the dimensions given to us on the command line. How to do this is as follows:

Allocating a 2-D array in g++ is done row by row, where each row is a 1-D array. If the requested matrix is to be `r` rows and `c` columns then we will have a `for` loop that iterates `r` times each time requesting an array of `c` data elements. The C++ operator that requests this memory is called `new()`.
From each call of `new()` the address of the block of memory is given back. What to do with these? *Another* array is needed to keep track of these. So each time `new()` returns its return value is assigned into the next slot of the "rows" array.

2-D array allocation in C++

The first thing you should do in your `readArr()` function is to allocate storage for the matrix (array) of double-precision floating point numbers. Once you have done this you are now ready to read in the elements. You should do this by reading the elements one row at a time, up to `r` of them where each row is composed of `c doubles`. So you will need two nested for-loops, one iterating over the rows and inside that a loop that reads the `c` elements on that row. The line of code to perform the actual input is

```
std::cin>> arr[i][j];
```

assuming that the indices of your loops are `i` and `j` and that the array's name is `arr`. This reads as "take something from the input stream `cin` and place it in the 2-D array `arr` at position `i,j`."

You should now be able to fill in the remaining code to read in the two arrays whose dimensions are specified as command-line arguments to `matmult`. Now to debug it and test it.

I am providing you with a skeleton of the code you need to write. It can be found in `~cs4115/labs/week03/matmult.cc` so you might want to copy that file into your directory and use it as a starting point. The command for that would be:
`cp ~cs4115/labs/week03/matmult.cc matmult.cc`
assuming that you have set your "working directory" to `week03`.

Array indices in C++

In C++ the indices of an array of size n run from 0 to $n-1$. There are still n elements but we just start counting from 0. Valid indices are $i \leq n-1$ (or $i < n$). So the prototypical for-loop is

```
for (int i=0; i<n; i++) {
  // begin loop body
}
```

Namespaces in C++

When writing big programs that draw on several libraries the possibility exists that two libraries will refer to a variable of the same name – say `listLen`. We can distinguish between the two variables of the same name by asking what **namespace** the variable comes from. To read something in to a variable the statement I gave you above was something along the lines of

```
std::cin>> val;
```

This can be broken down as "apply the *input extraction operator*, `>>`, to the input stream `std::cin`, placing the result in `val`. The input stream is in the standard or default namespace, `std`. You can tell the C++ compiler to assume by default that this is where to find `cin` by putting

```
using std::cin;
```

at the top of your program. (If you are using the file from last time as a starting point you will see a few such declarations there already.)

One step backwards, two steps forwards

No program of any serious size gets written in one go. Bugs in even the smallest program will creep in. And here's the thing: it is much easier to trap these earlier than to wait until the end. Sometimes you have to take a step backwards to take two forwards. You should just accept that when you develop a system of any size you will write code whose sole purpose is to aid debugging and testing. Accept it. Just accept it and move on. We'll all be in the better of it if I, uhhh, I mean you, could just get over it and move on. This code will never be run in the final working system yet is as valuable as any that does. You can call the effort you put in to that the price of getting your system to work faster.

A very useful trick

When you are debugging / testing the program you will be blue in the face from entering the elements of the two matrices, especially if they are large. Here's how you can take the effort out of that: linux and UNIX has a handy feature where you can **redirect** input to your program to come from a file instead of the keyboard. The great thing about this is that you don't do anything in your program. Instead you store your test matrix in a file called, say, `test-mat`. Now you run your program as follows:

```
matmult 5 4 4 8 < test-mat
```

The linux shell magically feeds the contents of the file to your program as though you typed it at the keyboard. Of course it's still your responsibility that what is in the file is of the dimensions you said.
Serious savings of time.

Compilation

The command for compiling will be

```
g++ matmult.cc -o matmult
```

This invocation of `g++` combines the three steps of "compilation", syntax checking, object code generation and program linking into one here. Later we will talk about developing libraries which will require us to separate out the linking step.

By this point you should have a program that allocates storage for two matrices of a specified size and reads them in to memory. All that remains is to write the multiplication function and finally a function to print out the answer.

Make sure that you write your function *declarations* (also called prototypes) at the top of the file and put your function *definitions* towards the bottom of the file.

Matrix multiplication

The first thing to know about multiplying matrices is that not just any matrices can be multiplied together. If we want to do the matrix multiplication

$$C = A \times B$$

the number of columns of A **must** match the number of rows of B. So that means that eventhough we can compute $A \times B$ we may not be able to compute $B \times A$! When we look at the formula for computing an element of the result we'll see why that is but this immediately gives us something that we must check upon getting the 4 numbers from the command line. We need to check that **the col. count of the first equals the row count of the second**. Please put this check in your code.

The trick to multiplying two matrices is to keep in mind that every element of the result comes from multiplying together a **row** of the first and a **column** of the second. (Technically, we take the *dot product* of row i of A and column j of B in order to determine one single element $c_{i,j}$ of C. $c_{i,j}$ is the element of C at row i, column j.) In more compact notation we say

$$c_{i,j} = \sum_{k=1}^{l} a_{i,k} b_{k,j}$$

Look very carefully at the subscripts on a and b and what they are implying. The element $c_{i,j}$ is computed by running a for-loop from 1 to `l` where `l` is the number of elements in the ith row of A which had better be the same the number of elements in the jth row of B.

For a graphical picture of what is going on have a look at Slide 19 here.

An aside

How much work is done in multiplying two matrices? To make life simpler we will say they are each of size $n \times n$?
We have seen that **every** element of the result requires the multiplication of a row of A and a column of B. (A *dot product actually.*) Since every row has n elements that involves n multiplications and $n-1$ additions.
Since there are $n^2 = n \cdot n$ entries in C we will need to do n^3 multiplications.

❻ The command to submit your program is

```
handin -m cs4115 -p w03
```

This will copy your `matmult.cc` file into the class database, compile and run it. It will perform 4 tests to ensure that your output **exactly** matches mine. The input files `m[01].in` that I mentioned earlier will be the first two test multiplications performed. I will then perform two more tests on bigger arrays.

For new users of `handin` it is always an issue how identical do I mean when I say that the programs have to be identical. What I mean is *identical*. How can you tell if your output is identical to mine? The following procedure will guarantee that your output is identical and you should do this at least for each of the test cases you know about, and as many others as you want. In fact what `handin` does is pretty much what is given below.

Comparing outputs

```
matmult 6 6 6 6 < m1.in > m1.out.mine  # your (student) output
~cs4115/labs/week03/matmult 6 6 6 6 < m1.in > m1.out.his  # official output
diff m1.out.mine m1.out.his
```

The `diff` command stays silent – silence is golden, that's the UNIX way – if the two files are **identical**. If it makes any complaints at all, you're in trouble.

Finally, the following is a trace of a successful run of `handin`. You will (hopefully) see something similar to this.

handin trace

```
[d9994115@smyths-19 week03]$ handin -m cs4115 -p w03
Good, the project deadline of 09.00 Monday, 16.02.2015 has not yet expired...
Finished copying files.
Finished compiling.
Finished execution test 1.
Finished execution test 2.
Finished execution test 3.
Finished execution test 4.
Your marks on this project, so far, are:
  1 /   1 for 'matmult.cc' being present
  2 /   2 for compiling 'matmult.cc'
  1 /   1 for Execution Test 1
  2 /   2 for Execution Test 2
  2 /   2 for Execution Test 3
  2 /   2 for Execution Test 4
This was your sixth submission; you are allowed a maximum of 10 submissions.

[d9994115@smyths-19 week03]$
```

Labs that are to be handed in are open for handin at 09.00 on Wednesday of the week it is *assigned* and, in order to get full marks, are due by 15.00 on Thursday of the *following* week; a lateness penalty applies to submissions made until 18.00, Monday after that. This gives you one week to work on each lab that is to be assessed and handin without penalty. The lab sheet will be available for reading earlier in the week so that you can read it beforehand and come to the lab with questions.

Subject to last-minute changes, the planned schedule of lab assignments due for handing in are:

Lab. Week	Assessed	DueDate
Week02	✗	
Week03	✓	Thu, Week04
Week04	✗	
Week05	✗	
Week06	✓	Thu, Week07
Week07	✓	Thu, Week08
Week08	✗	
Week09	✓	Thu, Week10
Week10	✗	
Week11	✓	Thu, Week12