CSE 232, Lab Exercise 7

Command Line

Process Management (&, bg/fg)

So far, we've been teaching you how to run programs at the command line. Running the program and waiting for it to finish works okay for short/fast programs, but if you have a program that takes minutes or hours to finish, you don't want to have to wait that long to do other things at the terminal. Instead, there is a way to run the program in the *background*, so that it is running but not blocking your access to the terminal.

Executing a Background Job

Let us say we have a long-running program called a.out. Normally you would run the program like so:

```
./a.out
```

But if you did that your terminal would be blocked until it finished. No commands can be executed until it is done. If instead you add a ampersand (&) *after* the command, the execution is run in the background.

```
./a.out &
```

The job, `a.out`, will now run but the terminal is freed for your commands.

What jobs do I have?

You can run the command `jobs` to see the status of your suspended (and running) jobs. It reports it in the following way (I have a job called top running in background)

```
>jobs
[1]+  Stopped                 top
```

The [1] is your local job number. We will use that in a minute.

Suspending a Running Job and Running in the Background

Lets say you already were running a.out in the foreground (without the ampersand). You already know about CTRL-C to kill a running program. Here's a new one, CRTL-Z. CNTRL-Z doesn't kill a program, but instead suspends it (pauses it), giving you back control of your terminal. Whatever was running is now stopped, but can be restarted.

After suspending a job, you can have the suspended job run in background using the command `bg`

Take a Background/Suspended Job and Run It In The Foreground

If you want to see the output from a suspended/background job, you can do so with the `fg` command. By default, fg makes the most recent background job run in the foreground. But if you supply the job ID number (from the jobs command), you can select a specific background job to be moved to the foreground like so:

```
fg %1
```

Killing Background Jobs

To kill a specific background job, use the `kill` command and the job id. Like so:

```
kill %2
```

For example, the top command tells you about the present status of your computer. It continuously prints info about processes, memory usage etc. I can do the following:

★ Make a program that outputs all the numbers between 1 and 1 million. Run the program in the background and demonstrate moving it to the foreground and killing it to the TA.

The Problem

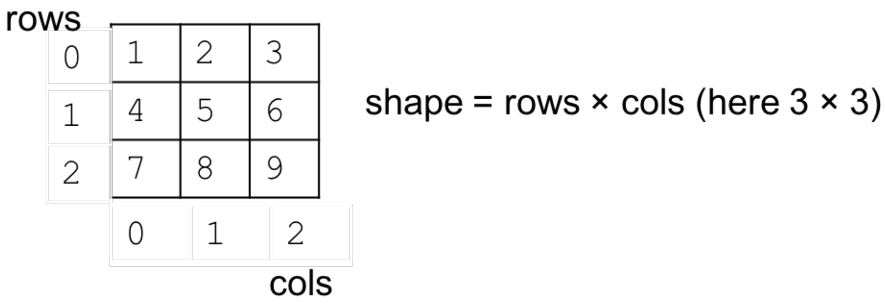
We are going to work on 2D vectors.

2D vector as a matrix

You remember matrices, don't you? We are going to do some simple manipulation of a matrix, namely: adding two matrices and multiplying a matrix by a scalar. You watched the 2D vector video, right? RIGHT???

Matrix

A **matrix** is a 2-dimensional (rows and columns) data structure. It has a shape indicated by the number of rows and the number of columns. Though I suppose a matrix could have uneven sized rows, this doesn't usually happen in practice so a matrix is always rectangular, potentially square (based on its shape).

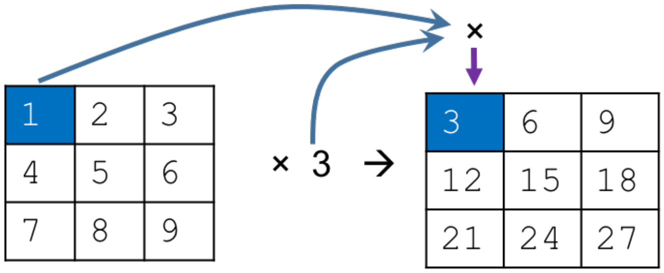


Matrix operations

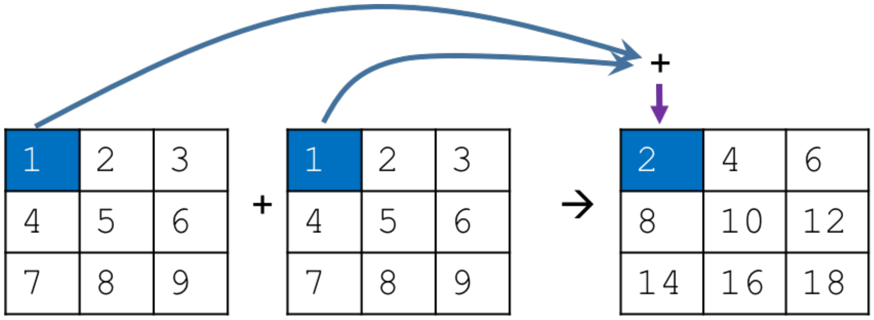
We will perform two operations on our matrices, yielding a new matrix as a result.

The first is **scalar multiplication**. Regardless of the size or shape, if the matrix is not empty we multiply the scalar value by every entry in the matrix, yielding a new matrix. We do this for every entry in the matrix.

The second is **addition**. The shape of the two matrices ***must be the same*** for addition to go forward. If the shapes are the same and they are both not empty, we add the same row/col element of each argument matrix into the same row/col element of a new matrix, yielding the new matrix. We do this for every element in the two matrices.



Scalar Multiplication



Addition of two matrices

Requirements

We provide a lab07_functions.h As always, you will submit a lab07_functions.cpp to Mimir for testing. Write your own main to test your code.

We will use a vector<vector<long>> as the underlying representation of our matrix. This means that the top level vector has, as elements, another vector.

In lab07_functions.h we provide two using definitions to make things a little easier, to wit:

```
using matrix_row = vector<long>;
using matrix = vector<matrix_row>;
```

This is really a big win! We need only say that the type of vector in a matrix_row is a long and then, if we are careful, can easily change the type of our entire code set by just changing that one template.

Function Declarations

The functions are clearly described in the lab07_functions.h file provided, read them there.

Printing

I think printing the 2D matrix is actually kind of hard. Here are some tips to help out:

In the include file iomanip is an io manipulator setw. It sets the width for an output element:

- Unlike every other manipulator, it requires you to run it each time you use it.
- If you say something like cout << setw(5) << 123; then 5 spaces are reserved for output, 3 of which are occupied by 123 and two of which are just blank spaces (the default, you can change that with setfill)
- Two other manipulators are left and right for left or right justification respectively. Thus
    - cout << right << setw(5) << 123; prints 2 spaces and 123
    - cout << left << setw(5) << 123; prints 123 and 2 spaces
- If you use an ostringstream (and you should), then any endl in the stringstream counts as a character in the stream. That might matter when you try to match up with Mimir output.
- My code used:
    - ostringstream to capture the output and then convert to a string
    - setw to set the width, where the default is 3
    - right to get the elements right justified so they look better

Other Hints

1. Write your own local lab07_main.cpp so you can test your code. It would be a good idea to starting doing your testing locally!

    a. As a side note, I see far too many people ***writing all the code before testing***. That's crazy! Write a function, test a function, write the next function, test that function, etc. This is the way you figure things out, one by one.

2. You can make a temporary row (of type `matrix_row`) and `push_back` values on to that. You can then `push_back` the row onto a matrix (of type `matrix`). You can reuse the row in the your loop, but remember to `.clear()` it first.
3. Testing is on Mimir. Here is a change! Some of the test cases are hidden so you can't hardcode to the test. By hidden I mean that you can see if you passed but not what the input/output pairs are. The cases are provided as pairs:
    a. The first of the pair you can see the input output testing
    b. The second of the pair, you can see if you passed the test but not the input/output pair.