Goals

- Process data stored as a vector of vectors (rows and columns)
- Use nested for loops
- Show a couple matrix operations
Vector of Vectors

- Data that conveniently presents itself in a tabular format is represented well with a `vector of vectors`
- General form
  ```cpp
  vector <vector<type> > identifier (rows,
    vector<type> (cols, initialValue optional));
  ```
- Example declaration (the space is needed `> >`)
  ```cpp
  vector <vector<int> > nums(5,
    vector<int> (3, -1));
  // All 15 values (5 rows, 3 columns) are -1
  ```
- Set the value of the first row and column to 9
  ```cpp
  nums[0][0] = 9;
  ```
• **Change a few more elements**

```cpp
nums[1][1] = 8;
nums[2][2] = 7;
nums[3][1] = 6;
nums[4][0] = 5;
```

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>-1</td>
<td>8</td>
<td>-1</td>
</tr>
<tr>
<td>-1</td>
<td>-1</td>
<td>7</td>
</tr>
<tr>
<td>-1</td>
<td>6</td>
<td>-1</td>
</tr>
<tr>
<td>5</td>
<td>-1</td>
<td>-1</td>
</tr>
</tbody>
</table>

• **Nested for loops are often used with a vector of vectors**

```cpp
for (int row = 0; row < nums.size(); row++) {
    for (int col = 0; col < nums[0].size(); col++) {
        cout.width(4);
        cout << nums[row][col];
    }
    cout << endl;
}
```

**Output:**

```
9   -1   -1
-1   8   -1
-1  -1   7
-1   6   -1
5   -1   -1
```
Matrix Operations

- A matrix is a rectangular vector of numbers, symbols, or expressions, arranged in rows and columns
- The data structure best suited to represent a matrix in C++? A vector of vectors
- This presentation shows of a couple of the operations on this data structure
Build a 6x6 Matrix

vector<vector<int>> nums(6, vector<int>(6));
// all 36 values (6 rows, 6 columns) are garbage

// Initialize to arbitrary values: 1..36
int count = 1;
for (int row = 0; row < nums.size(); row++) {
    for (int col = 0; col < nums[0].size(); col++) {
        nums[row][col] = count;
        count++;
    }
}

Matrix:

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
<td>18</td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
<td>24</td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>26</td>
<td>27</td>
<td>28</td>
<td>29</td>
<td>30</td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>32</td>
<td>33</td>
<td>34</td>
<td>35</td>
<td>36</td>
<td></td>
</tr>
</tbody>
</table>
Scalar Multiplication

- Scalar multiplication takes a number (a "scalar") and multiplies it on every element in the matrix

```cpp
int scalar = 2; // Multiply all elements by 2
for (int row = 0; row < nums.size(); row++) {
    for (int col = 0; col < nums[0].size(); col++) {
        nums[row][col] *= scalar;
    }
}
```

Matrix:

```
    2  4  6  8 10 12
14 16 18 20 22 24
26 28 30 32 34 36
38 40 42 44 46 48
50 52 54 56 58 60
62 64 66 68 70 72
```
• The transpose of a Matrix makes all row the columns and all columns the rows

```cpp
// Create a transposed version
vector<vector<int> > transpose(6, vector<int>(6));
for (int row = 0; row < nums.size(); row++) {
    for (int col = 0; col < nums[0].size(); col++) {
        transpose[col][row] = nums[row][col];
    }
}
```

```cpp
// Copy the transposed matrix back into nums
for (int row = 0; row < nums.size(); row++) {
    for (int col = 0; col < nums[0].size(); col++) {
        nums[row][col] = transpose[row][col];
    }
}
```
Before and After

- The first row (before) is now the first column (after)
- The last row (before) is now the last column (after)
- The 2nd row (before) is now the 2nd column (after)
- ... and so on

<table>
<thead>
<tr>
<th>before</th>
<th>after</th>
</tr>
</thead>
<tbody>
<tr>
<td>2  4  6  8 10 12</td>
<td>2  14  26  38  50  62</td>
</tr>
<tr>
<td>14 16 18  20  22 24</td>
<td>4  16  28  40  52  64</td>
</tr>
<tr>
<td>26 28 30  32  34 36</td>
<td>6  18  30  42  54  66</td>
</tr>
<tr>
<td>38 40 42  44  46 48</td>
<td>8  20  32  44  56  68</td>
</tr>
<tr>
<td>50 52 54  56  58 60</td>
<td>10 22 34 46 58 70</td>
</tr>
<tr>
<td>62 64 66  68  70 72</td>
<td>12 24 36 48 60 72</td>
</tr>
</tbody>
</table>
Primitive Arrays

- There are similarities between vector and primitive C++ arrays
- There are also similarities between a vector of vectors and primitive C++ arrays with two subscripts

```cpp
vector<vector<int>> table(4, vector<int>(6));  // Ugh
string table[4][6];  // Simpler declaration
```

- Elements are accessed the same way

```cpp
table[0][0] = "Upper left";
table[3][5] = "Lower right";
```
Array Initializers

- Primitive arrays are easier to declare
  - feel free to use them instead of vectors
  - A lot of code uses the primitive C++ arrays
- Arrays also have initializers, values between {} separated by commas

```cpp
double oneSubscript[] = {1.5, 0.9, 0.03, 4.2};

int twoSubscripts[2][3] = {{1, 2, 3}, {4, 5, 6}};
```
Arrays with more than 2 Subscripts?

• Yes, they are possible and sometimes useful

```c
double threeD[2][35][10];

threeD[1][34][9] = 87;
```