make matrix of numbers with vector<vector give way please use namespace std

Here are several ways to create a matrix of numbers using `vector<vector<>>` with `namespace std`:

Basic Example

cpp

```cpp
#include <iostream>
#include <vector>

using namespace std;

int main() {
    // Create a 3x3 matrix
    vector<vector<int>> matrix = {
        {1, 2, 3},
        {4, 5, 6},
        {7, 8, 9}
    };

    // Print the matrix
    for (int i = 0; i < matrix.size(); i++) {
        for (int j = 0; j < matrix[i].size(); j++) {
            cout << matrix[i][j] << " ";
        }
        cout << endl;
    }

    return 0;
}
```

Dynamic Matrix Creation

cpp

```cpp
#include <iostream>
#include <vector>

using namespace std;
```

```cpp
int main() {
    int rows = 4, cols = 5;

    // Method 1: Create with initial value
    vector<vector<int>> matrix1(rows, vector<int>(cols, 0)); // 4x5 matrix filled w
ith 0

    // Method 2: Create empty and push back
    vector<vector<int>> matrix2;

    for (int i = 0; i < rows; i++) {
        vector<int> row;
        for (int j = 0; j < cols; j++) {
            row.push_back(i * cols + j + 1);
        }
        matrix2.push_back(row);
    }

    // Print matrix2
    for (auto& row : matrix2) {
        for (auto& val : row) {
            cout << val << "\t";
        }
        cout << endl;
    }

    return 0;
}
```

Matrix with Different Data Types

cpp

```cpp
#include <iostream>
#include <vector>

using namespace std;

int main() {
    // Matrix of doubles
    vector<vector<double>> doubleMatrix = {
        {1.1, 2.2, 3.3},
        {4.4, 5.5, 6.6}
    };
```

```cpp
    // Matrix of characters
    vector<vector<char>> charMatrix = {
        {'A', 'B', 'C'},
        {'D', 'E', 'F'},
        {'G', 'H', 'I'}
    };

    // Accessing elements
    cout << "Double matrix element [1][2]: " << doubleMatrix[1][2] << endl;
    cout << "Char matrix element [0][1]: " << charMatrix[0][1] << endl;

    return 0;
}
```

Matrix Operations Example

cpp

```cpp
#include <iostream>
#include <vector>

using namespace std;

int main() {
    // Create two matrices
    vector<vector<int>> matrixA = {
        {1, 2, 3},
        {4, 5, 6}
    };

    vector<vector<int>> matrixB = {
        {7, 8, 9},
        {10, 11, 12}
    };

    // Matrix addition
    int rows = matrixA.size();
    int cols = matrixA[0].size();

    vector<vector<int>> result(rows, vector<int>(cols));

    cout << "Matrix Addition Result:" << endl;
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            result[i][j] = matrixA[i][j] + matrixB[i][j];
            cout << result[i][j] << " ";
```

```
        }
        cout << endl;
    }

    return 0;
}
```

Function to Create and Return Matrix

cpp

```cpp
#include <iostream>
#include <vector>

using namespace std;

// Function to create a matrix
vector<vector<int>> createMatrix(int rows, int cols, int initValue = 0) {
    return vector<vector<int>>(rows, vector<int>(cols, initValue));
}

// Function to print any matrix
template<typename T>
void printMatrix(const vector<vector<T>>& matrix) {
    for (const auto& row : matrix) {
        for (const auto& element : row) {
            cout << element << " ";
        }
        cout << endl;
    }
}

int main() {
    // Using the functions
    auto myMatrix = createMatrix(3, 4, 5);

    // Modify some values
    myMatrix[1][2] = 99;
    myMatrix[0][0] = 42;

    cout << "My Matrix:" << endl;
    printMatrix(myMatrix);

    return 0;
}
```

Key Points:

1. `vector<vector<T>>` creates a dynamic 2D array
2. Each inner vector represents a row
3. You can have different row lengths (jagged array)
4. Access elements with `matrix[row][column]`
5. Use range-based for loops for cleaner iteration

The `using namespace std;` directive allows you to use `vector`, `cout`, etc. without the `std::` prefix.

int n = 3, m = 4;
vector<vector<int>> mat(n, vector<int>(m));
vector<vector<int>> mat;

of this type

For the declarations:

```cpp
int n = 3, m = 4;
vector<vector<int>> mat(n, vector<int>(m));
vector<vector<int>> mat;
```

This creates a **conflict** because you're declaring `mat` twice. Let me explain both versions separately:

Version 1: Fixed Size Matrix

```cpp
int n = 3, m = 4;
vector<vector<int>> mat(n, vector<int>(m));
// Creates a 3x4 matrix with all elements initialized to 0
// mat.size() = 3 (rows)
// mat[0].size() = 4 (columns)
```

Initial state:

```
text

mat = {
    {0, 0, 0, 0},
    {0, 0, 0, 0},
    {0, 0, 0, 0}
}
```

Version 2: Empty Matrix

```cpp
vector<vector<int>> mat;
// Creates an empty matrix with 0 rows
// mat.size() = 0
```

Initial state:

```
text

mat = {}  // empty
```

Correct Usage Examples:

Example 1: Pre-initialized fixed size

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

int main() {
    int n = 3, m = 4;
    vector<vector<int>> mat(n, vector<int>(m));

    // Initialize with values
    int count = 1;
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            mat[i][j] = count++;
        }
    }
```

```
    // Print matrix
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            cout << mat[i][j] << "\t";
        }
        cout << endl;
    }

    return 0;
}
```

Output:

```
text

1    2    3    4
5    6    7    8
9    10   11   12
```

Example 2: Empty matrix with dynamic growth

```
cpp

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

int main() {
    vector<vector<int>> mat;  // Empty matrix

    // Add rows dynamically
    mat.push_back({1, 2, 3});      // Row 0
    mat.push_back({4, 5});         // Row 1 (jagged - only 2 elements)
    mat.push_back({6, 7, 8, 9});   // Row 2 (4 elements)

    // Print jagged matrix
    for (int i = 0; i < mat.size(); i++) {
        for (int j = 0; j < mat[i].size(); j++) {
            cout << mat[i][j] << " ";
        }
        cout << endl;
    }
```

```
    return 0;
}
```

Output:

```
text

1 2 3
4 5
6 7 8 9
```

Example 3: Resizing an empty matrix

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

int main() {
    vector<vector<int>> mat;  // Empty

    // Resize to 3 rows
    mat.resize(3);

    // Now resize each row to different sizes
    mat[0].resize(2, 10);  // Row 0: 2 elements, all 10
    mat[1].resize(4, 20);  // Row 1: 4 elements, all 20
    mat[2].resize(3, 30);  // Row 2: 3 elements, all 30

    // Print
    for (auto& row : mat) {
        for (auto& val : row) {
            cout << val << " ";
        }
        cout << endl;
    }

    return 0;
}
```

Output:

```
text
```

```
10 10
20 20 20 20
30 30 30
```

Which one to choose?

Use `vector<vector<int>> mat(n, vector<int>(m))` when:

- You know the exact dimensions beforehand
- You want a rectangular matrix (all rows same length)
- You want all elements initialized (default 0)

Use `vector<vector<int>> mat` when:

- You don't know the dimensions initially
- You need a jagged array (rows of different lengths)
- You want to build the matrix dynamically
- You want to add rows one by one

Common Mistake:

Don't declare the same variable twice:

```cpp
// WRONG - compilation error
vector<vector<int>> mat(n, vector<int>(m));
vector<vector<int>> mat;  // Error: redefinition of 'mat'
```

Correct way if you need both:

```cpp
// Create fixed size matrix
vector<vector<int>> fixedMat(n, vector<int>(m));

// Create empty matrix
vector<vector<int>> dynamicMat;
```

```
// Use them separately
```