MULTI-DIMENSIONAL ARRAYS

Sukhbir Tatla

sukhbir.tatla@senecacollege.ca

Introduction

- An array is a structured collection of components (called array elements):
- Arrays are all of the same data type, given a single name, and stored in adjacent memory locations
- The individual components are accessed by using the array name together with an integral valued index in square brackets
- The index indicates the position of the component within the collection

Declaration of an Array

- The index is also called the subscript

- In C++, the first array element always has subscript 0, the second array element has subscript 1, etc.

- The base address of an array is its beginning address in memory

SYNTAX

```
DataType  ArrayName[ConstIntExpression];
```

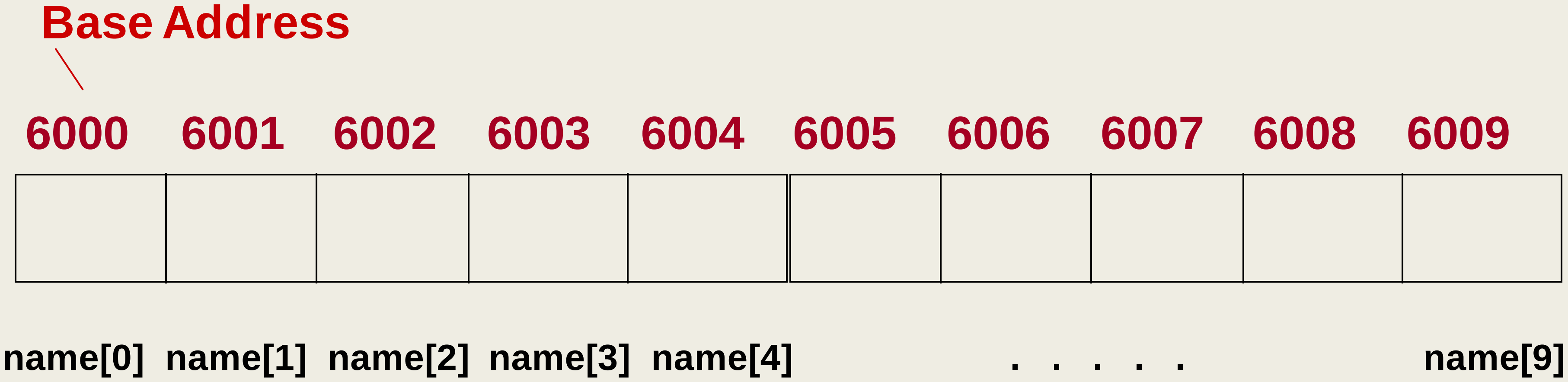
Array Example

- Declare an array called `name` which will hold up to 10 individual **char** values

number of elements in the array

```
char  name[10];        // Declaration allocates memory
```

Base Address



6000 6001 6002 6003 6004 6005 6006 6007 6008 6009

name[0] name[1] name[2] name[3] name[4] . . . . . name[9]

Assigning Values to Individual Array Elements

```
float temps[5]; int m = 4; // Allocates memory
temps[2] = 98.6;
temps[3] = 101.2;
temps[0] = 99.4;
temps[m] = temps[3] / 2.0;
temps[1] = temps[3] - 1.2;
// What value is assigned?
```

7000	7004	7008	7012	7016
99.4	?	98.6	101.2	50.6
temps[0]	temps[1]	temps[2]	temps[3]	temps[4]

What values are assigned?

```
float temps[5]; // Allocates memory
int m;

for (m = 0; m < 5; m++)
{
     temps[m] = 100.0 + m * 0.2 ;
}
```

7000	7004	7008	7012	7016
?	?	?	?	?
temps[0]	temps[1]	temps[2]	temps[3]	temps[4]

Variable Subscripts

```
float temps[5];     // Allocates memory
int m = 3;
 . . . . . .
```

What is temps[m + 1] ?

What is temps[m] + 1 ?

7000	7004	7008	7012	7016
100.0	100.2	100.4	100.6	100.8
temps[0]	temps[1]	temps[2]	temps[3]	temps[4]

A Closer Look at the Compiler

```
float temps[5]; // Allocates memory
```

- To the compiler, the value of the identifier `temps` is the base address of the array
- We say `temps` is a pointer (because its value is an address); it "points" to a memory location

7000	7004	7008	7012	7016
100.0	100.2	100.4	100.6	100.8
temps[0]	temps[1]	temps[2]	temps[3]	temps[4]

Initializing in a Declaration

```
int ages[5] = { 40, 13, 20, 19, 36 };


for (int m = 0; m < 5; m++)
{
    cout  << ages[m];
}
```

6000	6002	6004	6006	6008
40	13	20	19	36
ages[0]	ages[1]	ages[2]	ages[3]	ages[4]

Passing Arrays as Arguments

- In C++, No Aggregate Array Operations. The only thing you can do with an entire array as a whole (aggregate) is to pass it as an argument to a function
- Arrays are *always* passed by reference as the arguments to a function.
- Whenever an array is passed as an argument, its base address is sent to the called function
- Generally, functions that work with arrays require two items of information:
  - *The beginning memory address of the array (base address) and*
  - *The number of elements to process in the array*

Example with Array Parameters

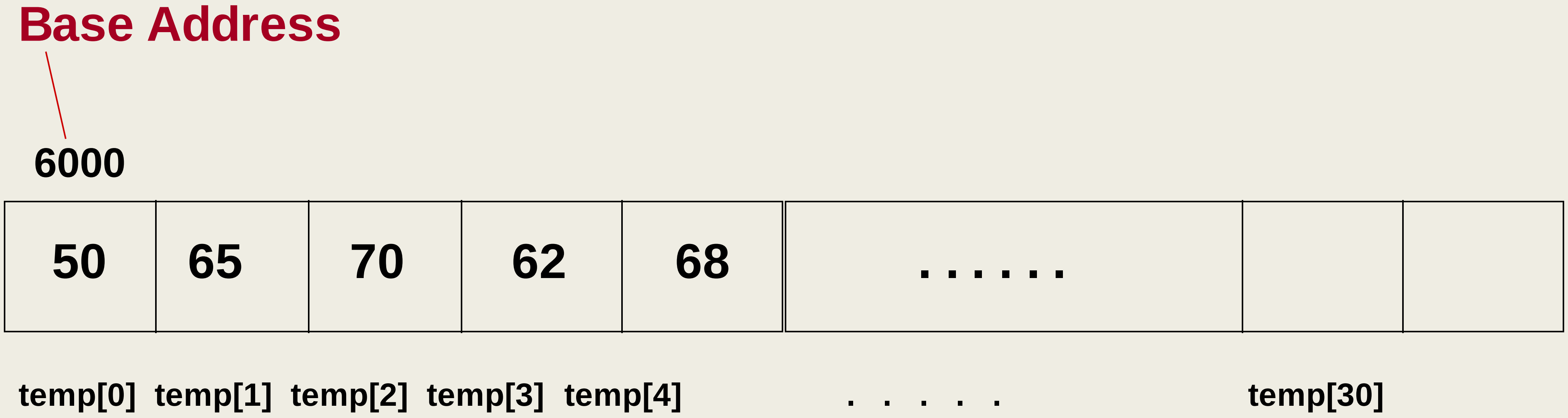
```
#include <iomanip>
#include <iostream>
using namespace std;
void  Obtain (int[], int); // Prototypes here
void  FindWarmest (const  int[],  int , int&);
void  FindAverage (const  int[],  int , int&);
void  Print (const  int[], int);

int main ( )
{
    // Array to hold up to 31 temperatures
    int  temp[31];
    int  numDays, average, hottest, m;
  cout  <<  “How many daily temperatures? ”;
  cin  >>  numDays;
```

Example with Array Parameters continued

```
     Obtain(temp, numDays);
  // Call passes value of numDays and address temp
     cout  <<  numDays  <<  “ temperatures”  << endl;
     Print (temp, numDays);

     FindAverage (temp, numDays, average);

     FindWarmest (temp, numDays, hottest);

     cout  <<  endl  <<  “Average was:  ” << average

          << endl;

     cout  <<  “Highest was:  ”  << hottest  << endl;

    return 0;

}    // Main Ends
```

Memory Allocated for Array

```
int temp[31]; // Array to hold up to 31 temp
```



Base Address

6000

50	65	70	62	68	. . . . . .		
temp[0]	temp[1]	temp[2]	temp[3]	temp[4]	. . . . .		temp[30]

More about Array Indexes

- Array indexes can be any integral type including char and enum types
- The index must be within the range 0 through the declared array size minus one
- It is the programmer's responsibility to make sure that an array index does not go out of bounds
- The index value determines which memory location is accessed
- Using an index value outside this range causes the program to access memory locations outside the array

Parallel Arrays

- Parallel arrays are two or more arrays that have the same index range and whose elements contain related information, possibly of different data types.

- EXAMPLE:

```
const  int  SIZE = 50;
int        idNumber[SIZE];
float      hourlyWage[SIZE];
```

parallel arrays

Parallel Arrays

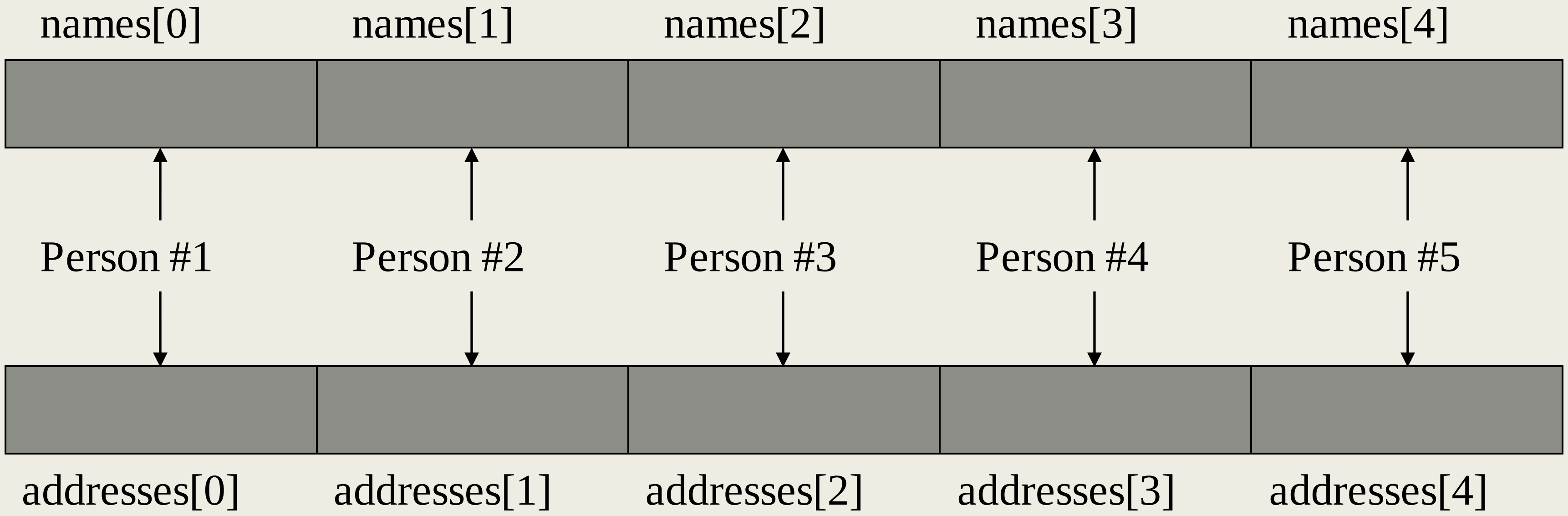
- By using the same subscript, you can build relationships between data stored in two or more arrays.

```
String names [5];
String addresses [5];
```

- The names array stores the names of five persons
- The addresses array stores the addresses of the same five persons.
- The data for one person is stored at the same index in each array.

Parallel Arrays

Relationship between names and addresses array elements.



names[0] names[1] names[2] names[3] names[4]

Person #1 Person #2 Person #3 Person #4 Person #5

addresses[0] addresses[1] addresses[2] addresses[3] addresses[4]

- Parallel arrays are useful when storing data of unlike types.

Two-Dimensional Arrays

- **Two-Dimensional Array:** A collection of a fixed number of components arranged in two dimensions.
  - *All components are of the same type*
- The syntax for declaring a two-dimensional array is:

  dataType arrayName[rowsize][colsize];

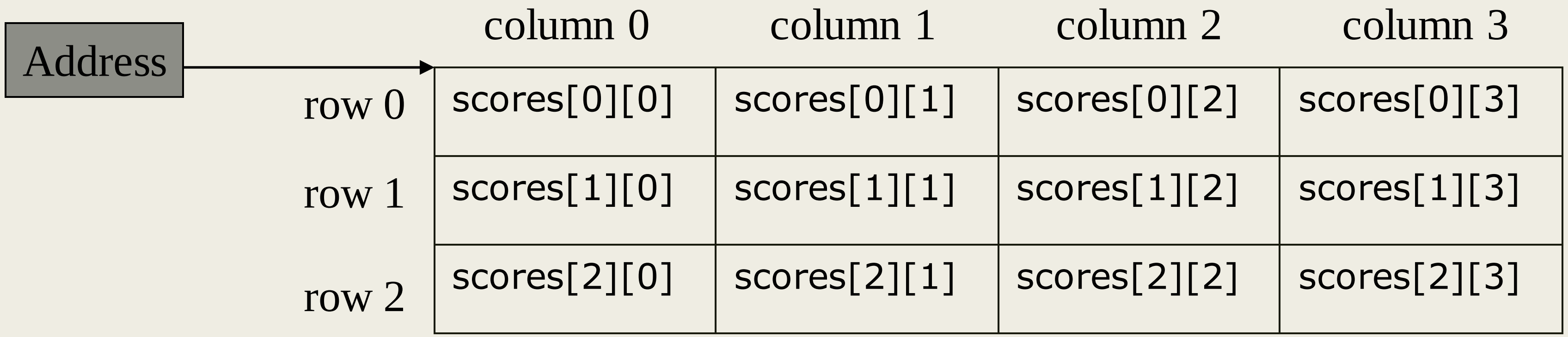
- *Where rowsize and colsize are expressions yielding positive integer values*

- The two expressions rowsize and colsize specify the number of rows and the number of columns, respectively, in the array

- Two-dimensional arrays are sometimes called **matrices or tables.**

Accessing Two-Dimensional Array Elements

The *scores* variable holds the address of a 2D array of doubles.



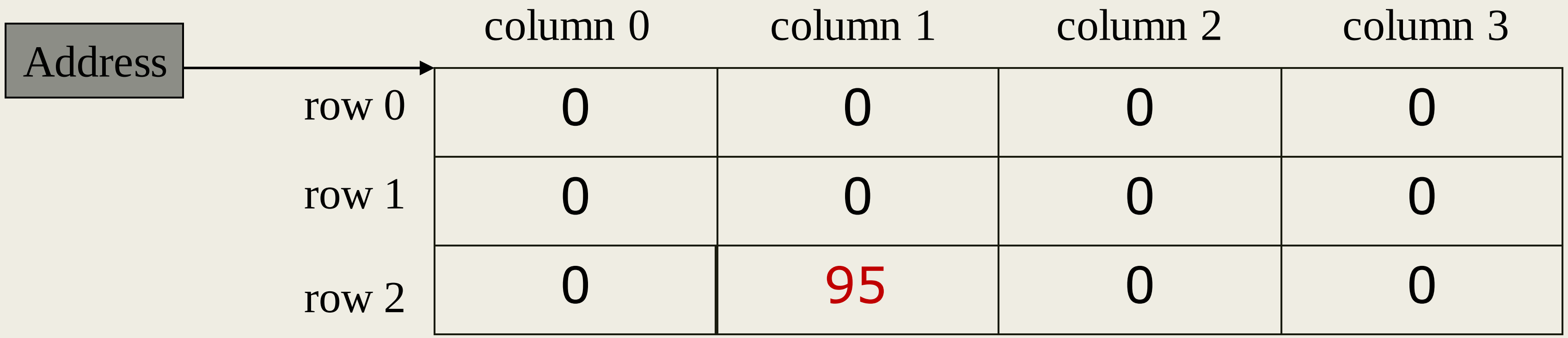
	column 0	column 1	column 2	column 3
row 0	scores[0][0]	scores[0][1]	scores[0][2]	scores[0][3]
row 1	scores[1][0]	scores[1][1]	scores[1][2]	scores[1][3]
row 2	scores[2][0]	scores[2][1]	scores[2][2]	scores[2][3]

Accessing Two-Dimensional Array Elements

The *scores* variable holds the address of a 2D array of `doubles`.

Accessing one of the elements in a two-dimensional array requires the use of both subscripts.

```
scores[2][1] = 95;
```



Address

	column 0	column 1	column 2	column 3
row 0	0	0	0	0
row 1	0	0	0	0
row 2	0	95	0	0

Accessing Two-Dimensional Array Elements

- Programs that process two-dimensional arrays can do so with nested loops.
- To print out the scores array:

```
for (int row = 0; row < 3; row++)
{

for (int col = 0; col < 4; col++){

  cout << setw(5) << scores[row][col] << “ “;
  }
cout << endl;
}
```

Number of rows, not the largest subscript

Number of columns, not the largest subscript

Example:

```
int main ()
{
   // Declaration and initialization of variables and Array
    const int DIVS = 3;          // Three divisions in the company
    const int QTRS = 4;          // Four quarters in the Division
    double totalSales = 0.0;             // Accumulator
    double sales [DIVS][QTRS];
    cout <<"This program will calculate the total sales of ";
    cout << “All the company's divisions:  " << endl ;
    cout << "Enter the following sales data:");
```

Example:

```
// For input values in Two dimensional Array
for (int div = 0; div < DIVS; div++)          {
        for (int qtr = 0; qtr < QTRS; qtr++){
              cout <<"Division " << (div + 1) << ", Quarter " << (qtr + 1) << ": $";
              cin >> sales[div][qtr];
        }
     }
// Display output of Two Dimensional Array using 2 nested iterations.
for (int div = 0; div < DIVS; div++)   {
        for (int qtr = 0; qtr < QTRS; qtr++)  {
           totalSales += sales[div][qtr];
        }
     }
     cout << "The total sales for the company " <<  "are $ " << totalSales;
}
```

Processing Two-Dimensional Arrays

- A two-dimensional array can be processed in three different ways:

  1. *Process the entire array*
  2. *Process a particular row of the array, called row processing*
  3. *Process a particular column of the array, called column processing*

- Each row and each column of a two-dimensional array is a one-dimensional array

- When processing a particular row or column of a two-dimensional array

  – *we use algorithms similar to processing one-dimensional arrays.*

Processing Two-Dimensional Arrays

- Two-dimensional arrays are stored in row order
  - *The first row is stored first, followed by the second row, followed by the third row and so on*
- When declaring a two-dimensional array as a formal parameter
  - *can omit size of first dimension, but not the second*
- Number of columns must be specified

Initializing a Two-Dimensional Array

- Initializing a two-dimensional array requires enclosing each row's initialization list in its own set of braces.

  int numbers [][] = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};

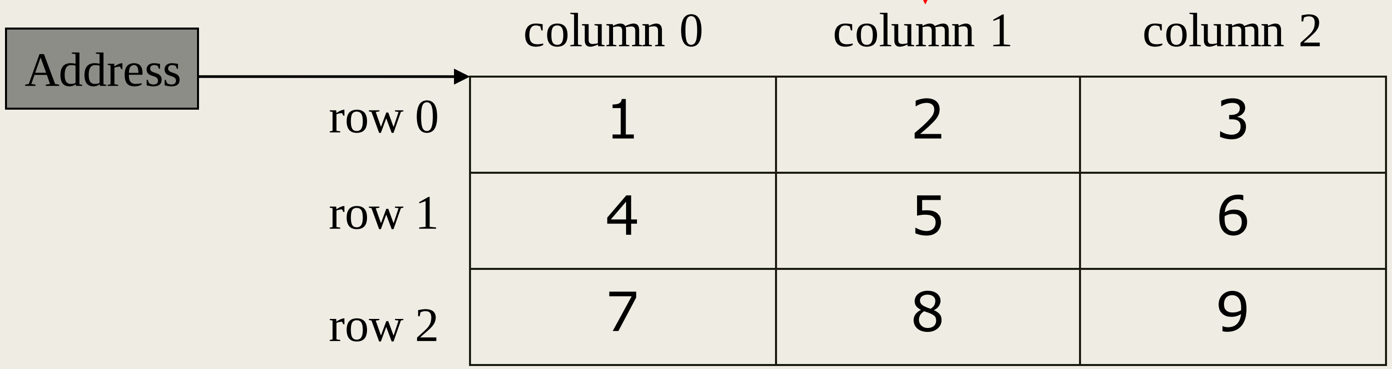
- Java automatically creates the array and fills its elements with the initialization values.
  - *row 0   {1, 2, 3}*
  - *row 1   {4, 5, 6}*
  - *row 2   {7, 8, 9}*
- Declares an array with three rows and three columns.

Initializing a Two-Dimensional Array

The *numbers* variable holds the address of a 2D array of `int` values.

```
int[][] numbers = {{1, 2, 3},
                   {4, 5, 6},
                   {7, 8, 9}};
```

produces:



	column 0	column 1	column 2
row 0	1	2	3
row 1	4	5	6
row 2	7	8	9

Summing The Elements of a Two-Dimensional Array

```
const int NumOfRows = 3;
const int NumOfCols = 4;
int[][] numbers = { { 1, 2, 3, 4 },
                    {5, 6, 7, 8},
                    {9, 10, 11, 12} };
int total;
total = 0;
for (int row = 0; row < NumOfRows; row++)
{
  for (int col = 0; col < NumOfCols; col++)
    total += numbers[row][col];
}

cout << “The total is: ” << total;
```

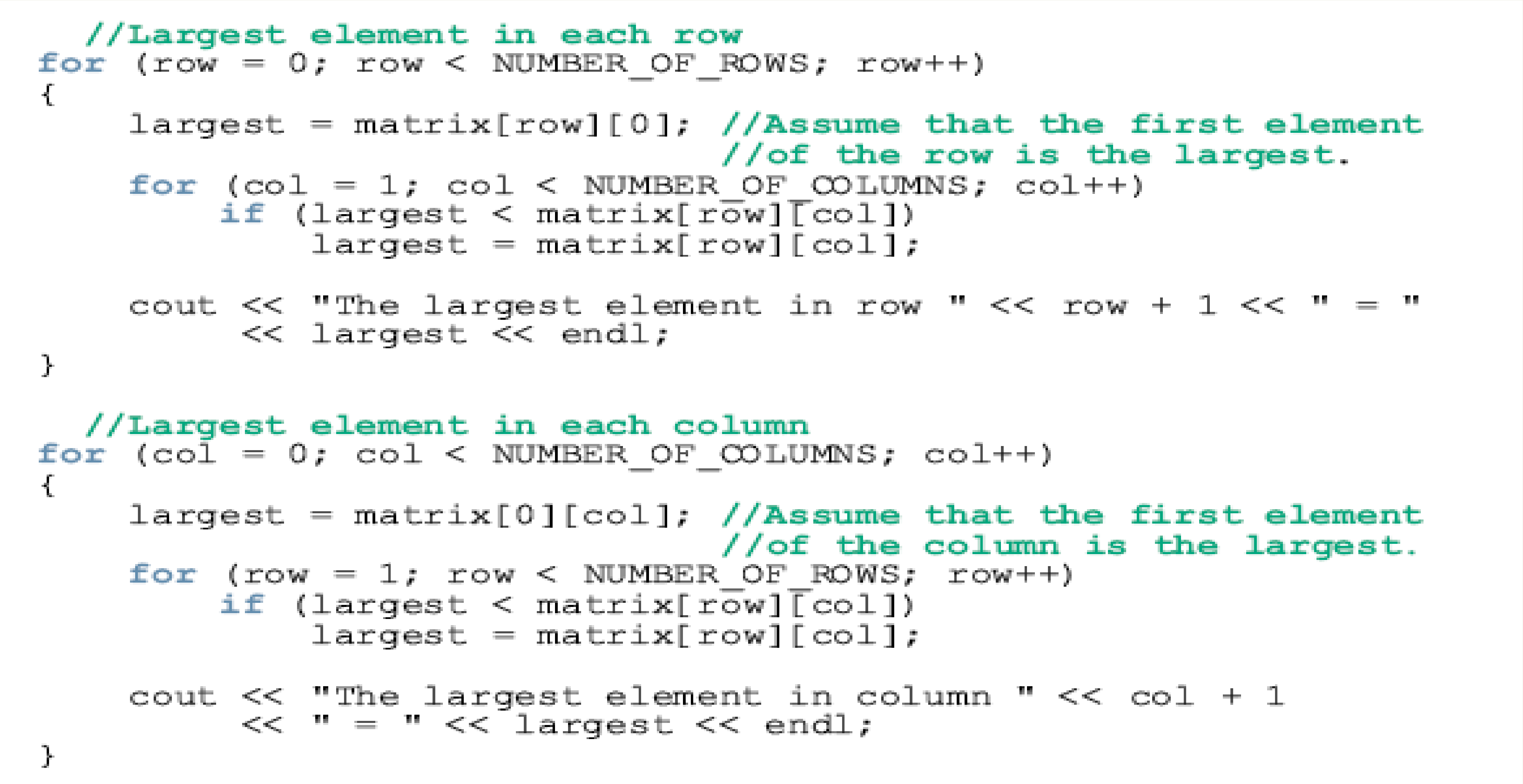
Summing The Rows of a Two-Dimensional Array

```
const int NumOfRows = 3;
const int NumOfCols = 4;
int[][] numbers = { { 1, 2, 3, 4 },
                    {5, 6, 7, 8},
                    {9, 10, 11, 12} };
int total;

for (int row = 0; row < NumOfRows; row++)
{
  total = 0;
  for (int col = 0; col < NumOfCols; col++)
        total += numbers[row][col];
  cout << “The total is: ” << total;
}
```

Summing The Columns of a Two-Dimensional Array

```
const int NumOfRows = 3;
const int NumOfCols = 4;
int[][] numbers = { { 1, 2, 3, 4 },
                    {5, 6, 7, 8},
                    {9, 10, 11, 12} };
int total;

for (int col = 0; col < NumOfCols; col++)
{
   total = 0;
  for (int row = 0; row < NumOfRows; row++)
         total += numbers[row][col];
   cout << “The total is: ” << total;
}
```

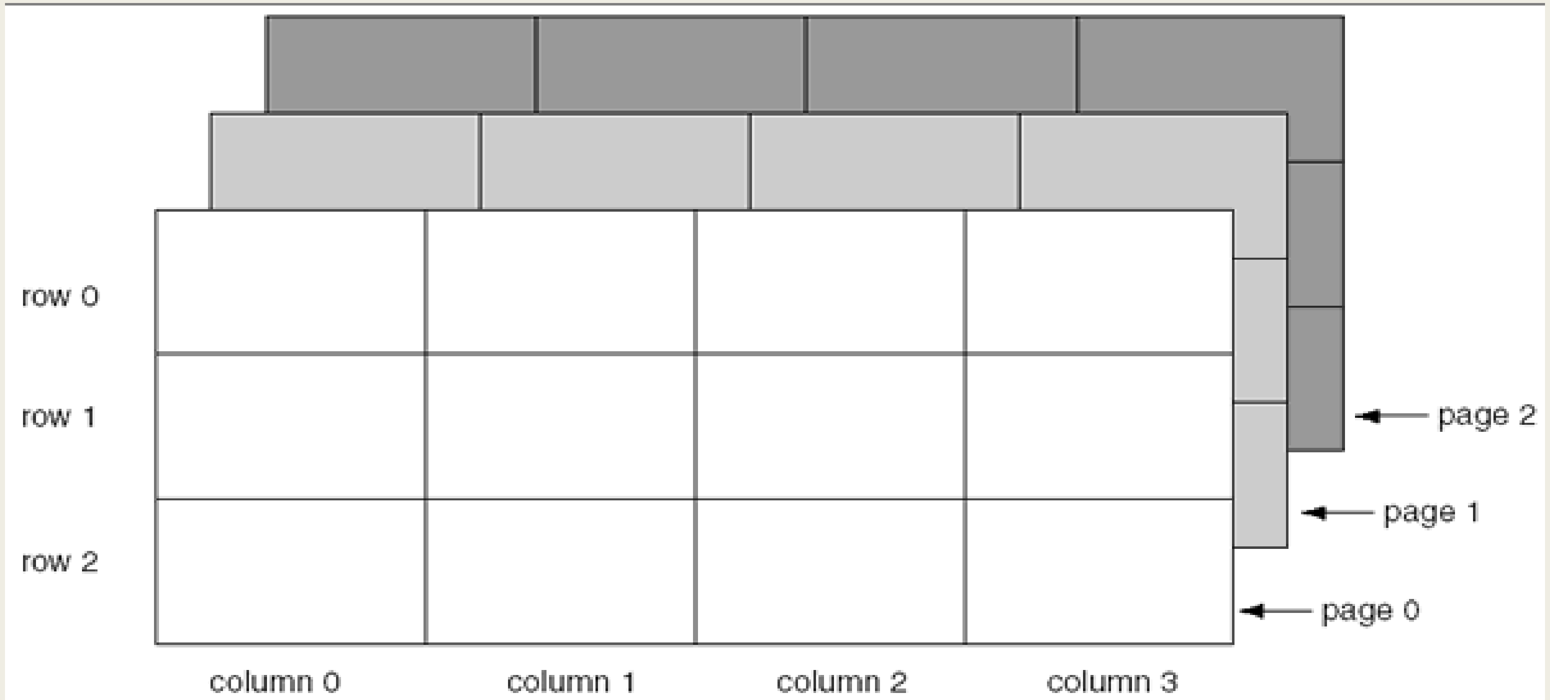
Finding the Largest Element in a Two-Dimensional Array



```
  //Largest element in each row
for (row = 0; row < NUMBER_OF_ROWS; row++)
{
    largest = matrix[row][0]; //Assume that the first element
                              //of the row is the largest.
    for (col = 1; col < NUMBER_OF_COLUMNS; col++)
        if (largest < matrix[row][col])
            largest = matrix[row][col];

    cout << "The largest element in row " << row + 1 << " = "
         << largest << endl;
}

  //Largest element in each column
for (col = 0; col < NUMBER_OF_COLUMNS; col++)
{
    largest = matrix[0][col]; //Assume that the first element
                              //of the column is the largest.
    for (row = 1; row < NUMBER_OF_ROWS; row++)
        if (largest < matrix[row][col])
            largest = matrix[row][col];

    cout << "The largest element in column " << col + 1
         << " = " << largest << endl;
}
```

More Than Two Dimensions

- C++ does not limit the number of dimensions that an array may be.
- More than three dimensions is hard to visualize, but can be useful in some programming problems.



THANK YOU!

Any Questions Please?