CompSci 372 – Tutorial

Part 2

C/C++ Programming

C/C++ Programming

- C++ in Cantonese is pronounced
  "C ga ga".
  Need I say more?
  (Mark D. Glewwe)

- Writing in C or C++
  is like running a chain saw
  with all the safety guards removed
  (Bob Gray)

C/C++ Programming

- Introduction to Microsoft Visual C/C++
  - http://www.cs.auckland.ac.nz/compsci372s2c/burkhardLectures/IntroductionTo.NET.pdf
  - http://www.cs.auckland.ac.nz/compsci372s2c/burkhardLectures/IntroductionCandC++.pdf
  - Changes:
    - .NET (2003) → Visual Studio 2008
    - “Solution” → “Project”, but suffix still “.sln”

C/C++ Programming

- Difference in Debug/Release
- Compiler errors/warnings
  - One error might cause a chain reaction
    → always check the first issue first
  - The problem might be in the line before
    and have nothing to do with the error message
  - Define _CRT_SECURE_NO_WARNINGS to avoid confusing warning messages

C/C++ Programming

- Task: A class for a square Matrix
  - Constructor, Destructor
  - Dynamic size → new[], delete[] operators (formerly known as malloc/free)
  - Methods for
    - Reading/writing elements
    - Adding another matrix
    - Printing

Matrix Class

- Header file

```
#pragma once

#include <iostream>
using namespace std;

class CMatrix
{
public:
    CMatrix(void);
    ~CMatrix(void);

    void setSize(int size);
    float& operator()(int row, int column);

    void add(CMatrix&);
    void operator+=(CMatrix&);

    friend ostream& operator<<(ostream&, CMatrix&);

private:
    int     size;
    float** data;
};
```

Matrix Class

- Constructor

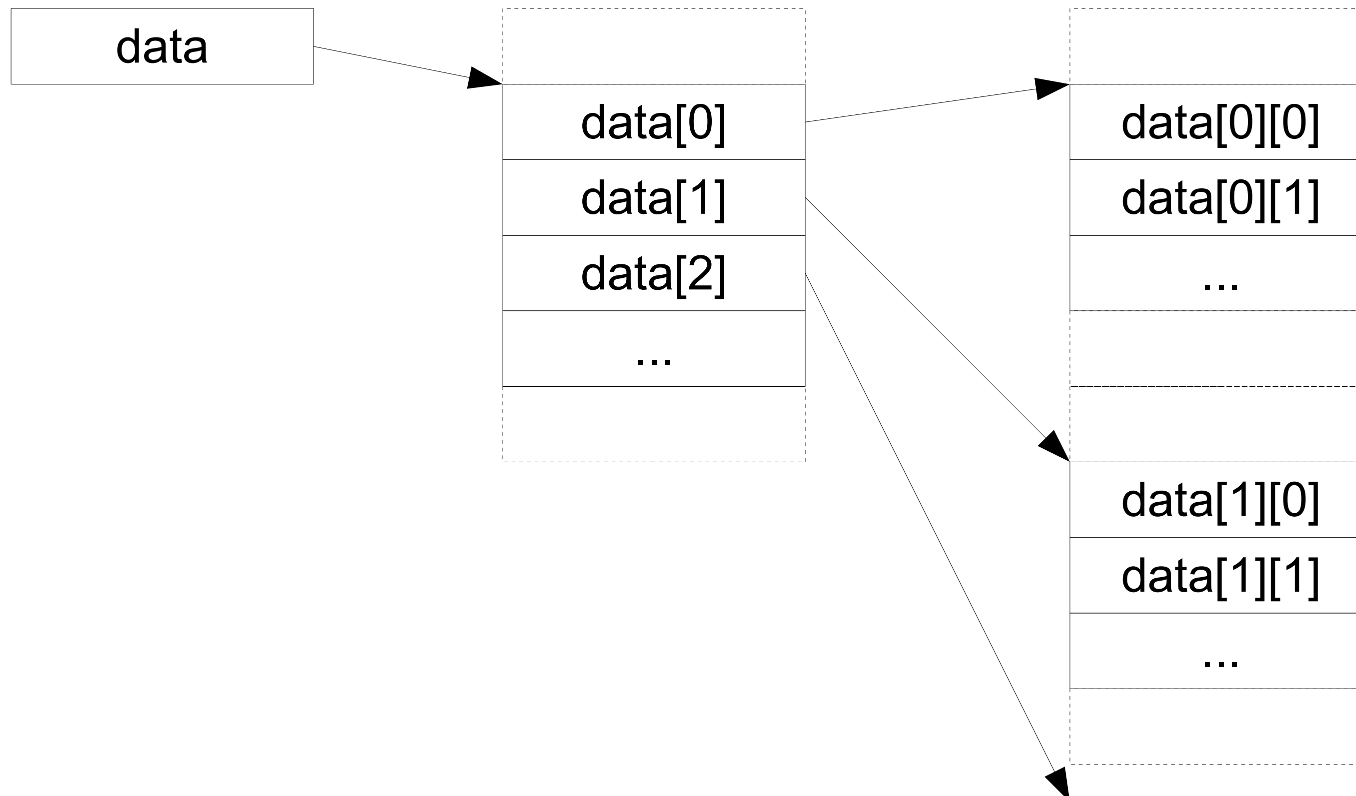
```
#include "Matrix.h"

CMatrix::CMatrix(void)
{
    size = 0;
    data = NULL;
}
```

- Set Size

```
void CMatrix::setSize(int size)
{
    this->size = size;
    data = new float*[size];
    for ( int row = 0 ; row < size ; row++ )
    {
        data[row] = new float[size];
    }
}
```

Matrix Class

- 2D array
  - A 2D array is an array with pointers to arrays with the values



Matrix Class

- Destructor

```
Cmatrix::~CMatrix(void)
{
    for ( int row = 0 ; row < size ; row++ )
    {
        delete[] data[row];
    }
    delete[] data;
    data = NULL;
}
```

- Be aware:
  - Create with new, destroy with delete
  - Create with new[], destroy with delete[]
  - Don't mix malloc/free and new/delete!

Matrix Class

- Read/write elements

```
float& CMatrix::operator()(int row, int column)
{
    // TODO: check validity of row/column
    return data[row][column];
}
```

  - Dangerous: Direct access to memory
    - + Fast
    - - No protection
  - Java implementation:
    - float getAt(row, col)
    - void setAt(row,col,value)

Matrix Class

- Add

```
void CMatrix::add(CMatrix& matrix)
{
    for ( int row = 0 ; row < size ; row++ )
    {
        for ( int col = 0 ; col < size ; col++ )
        {
            data[row][col] += matrix.data[row][col];
            // or: data[row][col] += matrix(row, col);
        }
    }
}
```

- += operator (“syntactic sugar”)

```
void CMatrix::operator+=(CMatrix& matrix)
{
    add(matrix);
}
```

Matrix Class

- Output

```
ostream& operator<<(ostream& s, CMatrix& matrix)
{
    s << "Matrix: Size=" << matrix.size << endl;
    for ( int row = 0 ; row < matrix.size ; row++ )
    {
        for ( int col = 0 ; col < matrix.size ; col++ )
        {
            s << matrix(row, col) << "\t";
        }
        s << endl;
    }
    return s;
}
```

Matrix Class

- Main program + output

```
// Tutorial 2, Matrix class

#include <iostream>
using namespace std;

#include "Matrix.h"

void doTest(CMatrix& m1, CMatrix& m2)
{
	cout << "Set size of m1" << endl;
	m1.setSize(2);
	cout << m1 << endl;
	cout << "Set values of m1" << endl;
	m1(0,0) = 0; m1(0,1) = 1;
	m1(1,0) = 2; m1(1,1) = 3;
	cout << m1 << endl;
	cout << "Element at (1,1) = " << m1(1, 1) << endl;
	cout << endl;
	cout << "Set size and values of m2" << endl;
	m2.setSize(2);
	m2(0,0) = 10; m2(0,1) = 11;
	m2(1,0) = 12; m2(1,1) = 13;
	cout << m2 << endl;
	cout << "Add m1 and m2" << endl;
	m1.add(m2);
	// same result: m1 += m2;
	cout << m1 << endl;
}

int main(int argc, char* argv)
{
	CMatrix m1;
	CMatrix m2;
	doTest(m1, m2);
}
```

```
Set size of m1
Matrix: Size=2
-4.31602e+008   -4.31602e+008
-4.31602e+008   -4.31602e+008

Set values of m1
Matrix: Size=2
0         1
2         3

Element at (1,1) = 3

Set size and values of m2
Matrix: Size=2
10        11
12        13

Add m1 and m2
Matrix: Size=2
10        12
14        16
```

Memory garbage

Matrix2 Class

- Type Polymorphism
  - “Allow different data types to be handled using a uniform interface”
  - Keyword: virtual
  - Change the one critical method, but don't touch the rest
  - Let the new class appear like the old one
  - No need to change the doTest() routine.

Matrix2 Class

- Small changes to header file of CMatrix

```
#pragma once

#include <iostream>
using namespace std;

class CMatrix
{
public:
    CMatrix(void);
    virtual ~CMatrix(void);

    virtual void setSize(int size);
    float& operator()(int row, int column);

    void add(CMatrix&);
    void operator+=(CMatrix&);

    friend ostream& operator<<(ostream&, CMatrix&);

protected:
    int     size;
    float** data;
};
```

Important for polymorphism to work

To allow the derived class the access to the data

Matrix2 Class

- Create new Matrix2 class

```
#pragma once

#include "Matrix.h"

class CMatrix2 : public CMatrix
{
public:
    virtual void setSize(int size);
};
```

```
#include "Matrix2.h"

void CMatrix2::setSize(int size)
{
    // old class does its work
    CMatrix::setSize(size);
    // then we clear the matrix values
    for ( int row = 0 ; row < size ; row++ )
    {
        for ( int col = 0 ; col < size ; col++ )
        {
            data[row][col] = 0.0f;
        }
    }
}
```

Matrix2 Class

- Main program + output

```
// Tutorial 2, Matrix class

#include <iostream>
using namespace std;

#include "Matrix2.h"

void doTest(CMatrix& m1, CMatrix& m2)
{
	cout << "Set size of m1" << endl;
	m1.setSize(2);
	cout << m1 << endl;
	cout << "Set values of m1" << endl;
	m1(0,0) = 0; m1(0,1) = 1;
	m1(1,0) = 2; m1(1,1) = 3;
	cout << m1 << endl;
	cout << "Element at (1,1) = " << m1(1, 1) << endl;
	cout << endl;
	cout << "Set size and values of m2" << endl;
	m2.setSize(2);
	m2(0,0) = 10; m2(0,1) = 11;
	m2(1,0) = 12; m2(1,1) = 13;
	cout << m2 << endl;
	cout << "Add m1 and m2" << endl;
	m1.add(m2);
	// same result: m1 += m2;
	cout << m1 << endl;
}

int main(int argc, char* argv)
{
	CMatrix2 m1;
	CMatrix2 m2;
	doTest(m1, m2);
}
```

```
Set size of m1
Matrix: Size=2
0         0
0         0

Set values of m1
Matrix: Size=2
0         1
2         3

Element at (1,1) = 3

Set size and values of m2
Matrix: Size=2
10        11
12        13

Add m1 and m2
Matrix: Size=2
10        12
14        16
```

It works

Question

- What would happen, if
  - We would not add “virtual” to Matrix.h?
  - We would not declare the members protected?
- How is a 3D array represented in memory?