CEE432/CEE532/MAE541
Developing Software for Engineering Applications

Lecture 10: Objects 202
(Chapter 9)

Motivation

- Learn more advanced features so that we can develop powerful and easy-to-use server code.

this pointer

- Special pointer that provides an object access to itself.

```
void CPoint::Display (const std::string& szBanner)
{
    // display the current coordinates
    std::cout << szBanner
              << "[X,Y] Coordinates = ["
              << this->m_fXCoor << ","
              << (*this).m_fYCoor << "].\n";
}
```

Better example to follow ….

Friend classes and functions

- A `friend` class has access to (another) classes' `private` member functions and variables.

```
class CPoint
{
    friend class CTriangle; // CTriangle has access to CPoint
    public:
        CPoint ();
      ….
    private:
      float m_fXCoor;
      float  m_fYCoor;
};
```

Client Code

```
#include “point.h”
class CTriangle
{
    public:
        CTriangle ();
        ….
    private:
        CPoint   m_Vertex[3];
};
…..
void CTriangle::ComputeSide ()
{
      float fSide1 = Distance (m_Vertex[0].m_fXCoor,
                               m_Vertex[0].m_fYCoor,
                               m_Vertex[1].m_fXCoor,
                               m_Vertex[1].m_fYCoor);
….
}
```

Friend classes and functions

- A class has to be explicitly declared as a friend for that class to have access to member functions and variables.
- Class A declares class B as a friend.
  - This does not imply class A is a friend of class B.
- Class A declares class C as a friend.
  - This does not imply B and C are friends.
- Friend functions can also be declared.

Operator Overloading

- Certain operators can be overloaded with your class definition.

Syntax

```
returntype classname::operatorx ()
```

Example

```
CPoint operator+ (const CPoint&) const;
```

Sample Client Code

```
P3 = P1 + P2;
P4 = P1 + P2 + P3;
```

Operator Overloading

```
class CPoint
{
    public:
…
        // overloaded operators
        CPoint& operator= (const CPoint&);
        CPoint operator+ (const CPoint&) const;
        CPoint operator- (const CPoint&) const;
        bool operator!= (const CPoint&) const;
        bool operator== (const CPoint&) const;
};
```

Operator Overloading

```
CPoint P1 (1.1f, -4.5f);
CPoint P2 (3.2f, 0.0f), P3;

if (P1 == P2) // == is overloaded
    std::cout << “Same location\n”;

else

    P3 = P1 - P2; // - is overloaded
```

Operator Overloading

```
bool CPoint::operator== (const CPoint& PRight) const
{
    return (m_fXCoor == PRight.m_fXCoor &&
            m_fYCoor == PRight.m_fYCoor);
}


CPoint CPoint::operator- (const CPoint& PRight) const
{
    CPoint PResult;    // this object will be returned
    // subtract the coordinates
    PResult.m_fXCoor = m_fXCoor - PRight.m_fXCoor;
    PResult.m_fYCoor = m_fYCoor - PRight.m_fYCoor;

    return PResult;
}
```

Copy ctor is called with – operator.

Operator Overloading

```
#include <iostream>
class CPoint
{
    // friend overloaded operator functions
    friend std::istream &operator>> (istream&, CPoint&);
    friend std::ostream &operator<< (ostream&, const CPoint&);

…..

};
```

Operator Overloading

```
ostream &operator<< (ostream& ofs, const CPoint& Point)
{
    // display the current coordinates
    ofs << Point.m_fXCoor << ","
        << Point.m_fYCoor << ".\n";

    return ofs;
}
istream &operator>> (istream& ifs, CPoint& Point)
{
    // get the coordinate values
    std::cout << "X Coordinate: ";
    ifs >> Point.m_fXCoor;
    std::cout << "Y Coordinate: ";
    ifs >> Point.m_fYCoor;

    return ifs;
}
```

Operator Overloading

`<< operator`

```
// these two statements are equivalent
P3.Display ("Point P3: ");
std::cout << "Point P3: " << P3 << "\n";
```

`>> operator`

```
std::cin >> P1;
```

`static` member variable

- There are times when a member variable defined in a class needs to be accessed by **all** the objects of that class.

`static` member function

- static member functions are member functions that do not access an object’s data

Forward class definition

- When one class is a friend of another, it is common for both classes to refer to the other class in the class definitions. This requires the use of **`forward`** declaration.

Template Classes: Class definition

```
template <class T>
class CPoint
{
    public:
        void SetValues (T, T);
….
    private:
        T m_fXCoor;
        T m_fYCoor;
};
```

Template Classes: Member Functions

```
template <class T>
CPoint<T>::CPoint (const CPoint<T>& P)
{
…
}

template <class T>
void CPoint<T>::SetValues (T fX, T fY)
{
…
}
```

Template Classes: Client Code

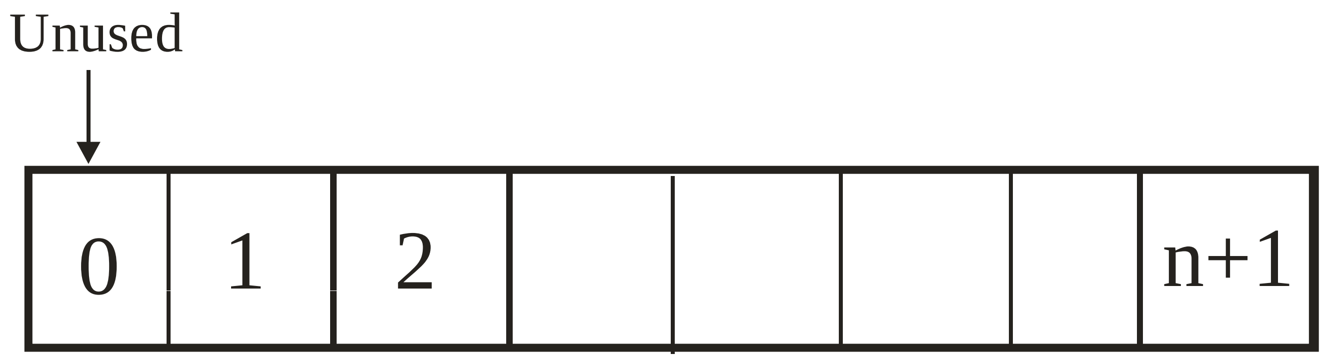
```
#include “point.h”
….
CPoint<int> nPA, nPB;
CPoint<float> fPA, fPB;
```

`CVector` class

- Features
  - Template class
  - Dynamic memory allocation
  - Does not differentiate between row and column vector
  - Indexing starts at 1. Valid indices are between 1 and the size of the vector.
  - Overload the **()** and **=** operators
  - Bounds checking

CVector class

Memory Map



```
CVector<double> dVA(4);
dVA(1) = 12.4; // same as dVA[1] not
               // dVA[0]
```

`CMatrix` class

- Features
  - Template class
  - Dynamic memory allocation
  - Has **n** rows and **m** columns
  - Indexing starts at 1. Valid indices are between 1 and the # of rows or columns.
  - Overload the **()** and **=** operators
  - Bounds checking

`CMatrix` class

```
CVector<double> dMA(4,3);
….
dMA(i,j) = 12.4;
dMA(2,1) = dMA(1,3) + dMA(i,2);
```

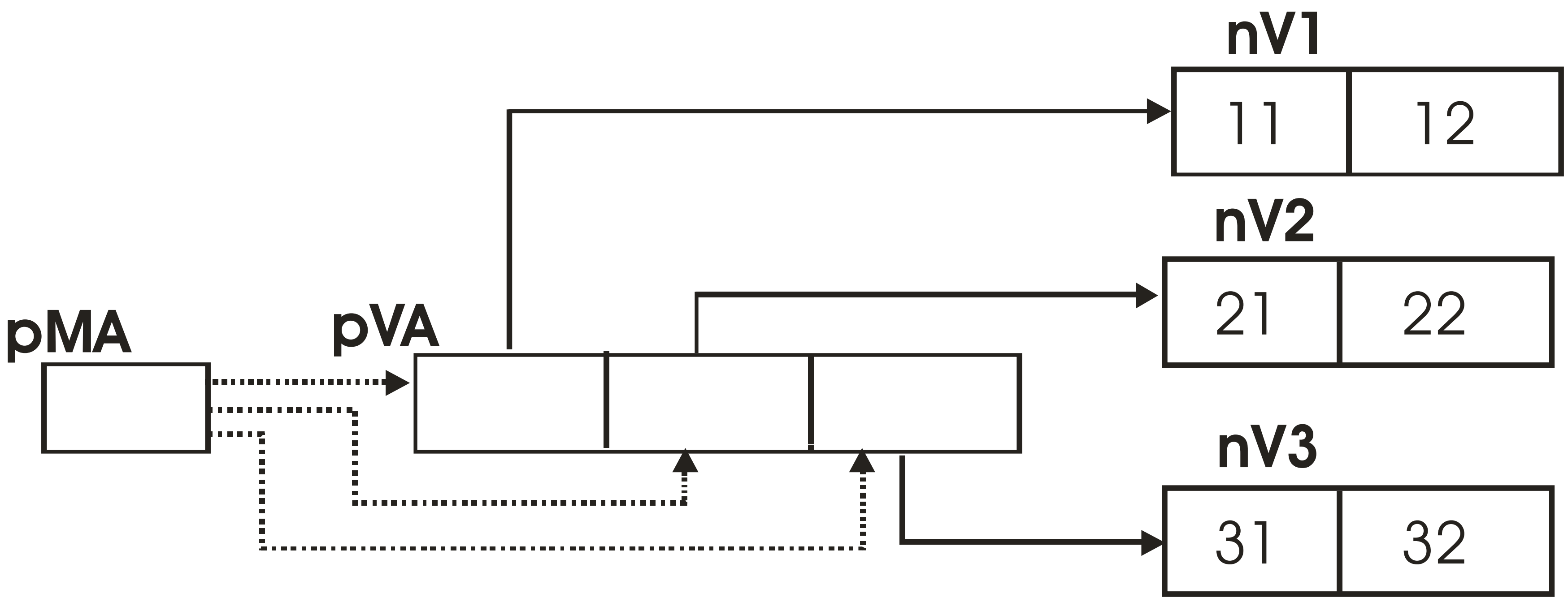
`CMatrix` class Development

- What is a pointer to a pointer?

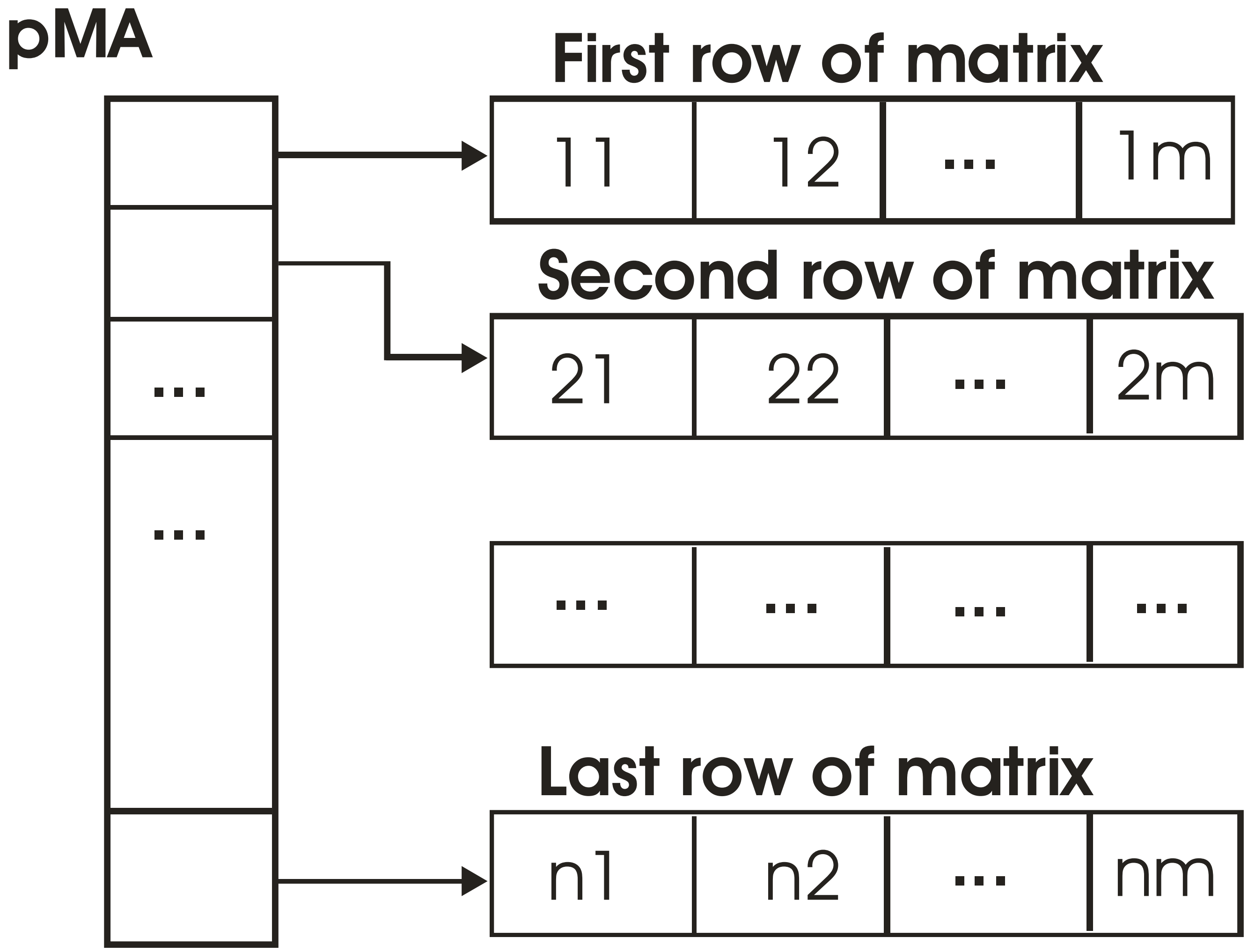
```
int *pVA[3];  // a vector of int pointers
int **pMA;    // pointer to an int pointer
int nV1[2] = {11, 12};
int nV2[2] = {21, 22};
int nV3[2] = {31, 32};
pVA[0] = nV1;    // stores the address of vector nV1
pVA[1] = nV2;    // stores the address of vector nV2
pVA[2] = nV3;    // stores the address of vector nV3

for (int i=0; i < 3; i++)
{
    pMA = &pVA[i];  // grab the address stored in pVA[i]
    for (int j=0; j < 2; j++)
    {
        std::cout << "[" << i << "," << j
                  << "] = " << pMA[0][j] << '\n';
    }
 }
```

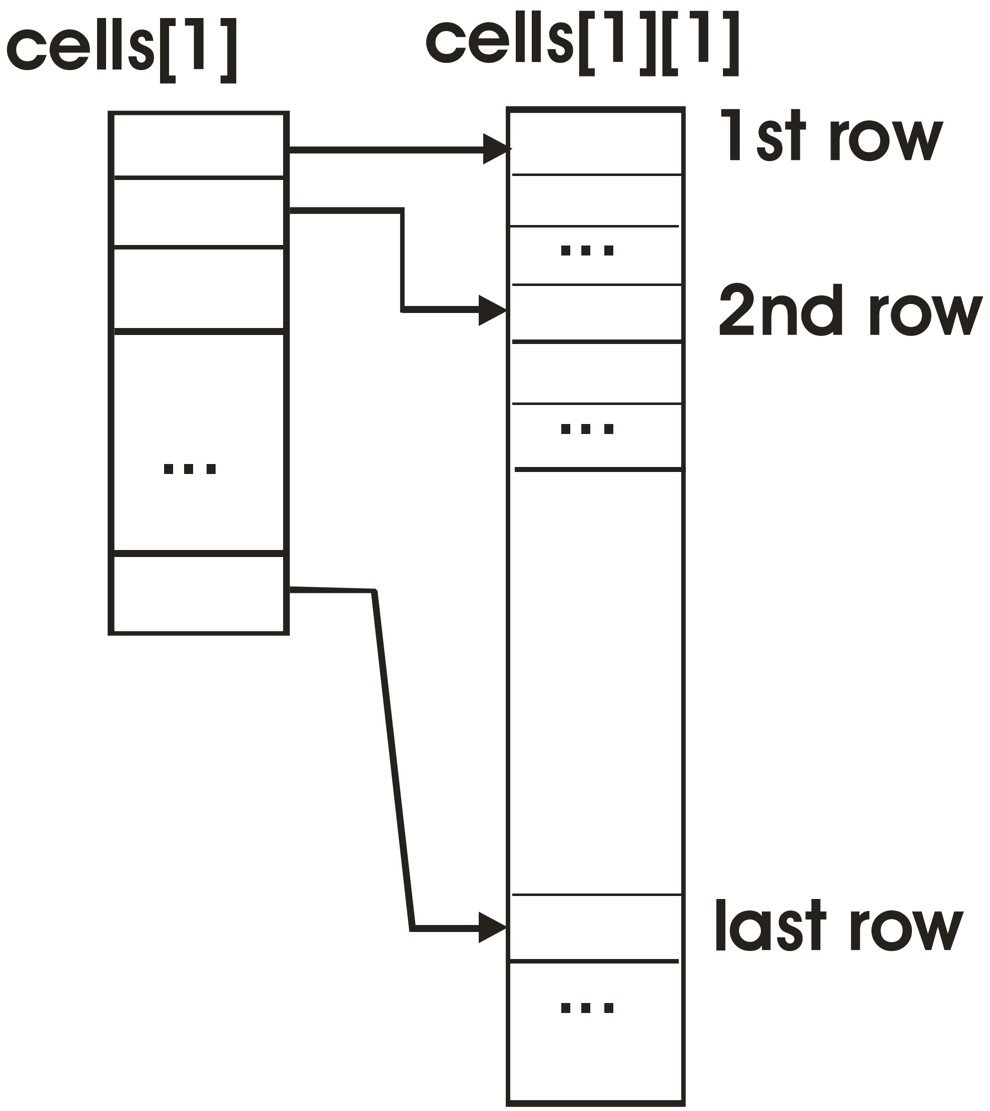
Memory Map



Matrix Storage Memory Map



CMatrix class



Wasted
memory space
not shown.

Programming Tips

- *Tip 9.1: Pass objects including arrays by reference*
- *Tip 9.2: Ask yourself if overloaded operators are really necessary?*
- *Tip 9.3: Need for the Big Three*

`usingthelib` project

- Program shows how to use functions and classes from the **`library`** directory
  - fileop.cpp
  - getinteractive.cpp
  - vectortemplate.h
  - matrixtemplate.h
  - clock.cpp