C++ Examples: Matrix

CS-410, Fall 2004

Michael Weiss

Today's Lecture

Remember "the Big Four" member methods of a C++ class:

```
class Foo {
public:
    Foo(); //default constructor
    Foo(const Foo & f); //copy constructor
    ~Foo(); //destructor
    Foo& Foo::operator=(const Foo& rhs);
        //copy assignment
}
```

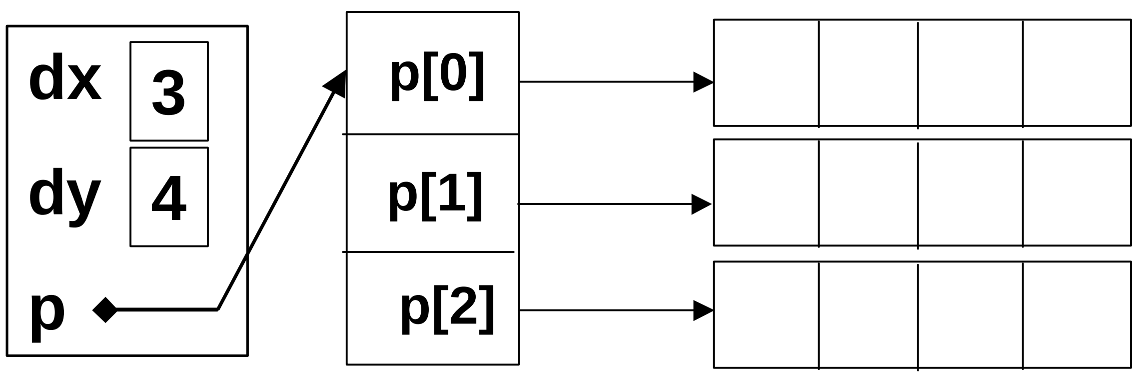
We will illustrate these with a class Matrix.

A Matrix will be a dynamic two-dimensional array.

The class Matrix

We will use a pointer-to-pointer-to-base type structure. Our base type will be long (although the base type doesn't make a big difference to the code).

Matrix



Types:

long* p[i]

long p**

Matrix Class Definition

```
class Matrix {
public:
   Matrix(int sizeX, int sizeY);
   Matrix();
   ~Matrix();
   Matrix(const Matrix& m);
   Matrix& operator=(const Matrix& rhs);
   ...
private:
   int dx, dy;  // dimensions, dx × dy
   long **p;   // pointer to a pointer to a long integer
};
```

Matrix Class Definition

It will be convenient to add a private method to allocate the array p and the p[i] arrays:

```
class Matrix {
private:
        ...
        void allocArrays() {
                p = new long*[dx];
                for (int i = 0; i < dx; i++) {
                        p[i] = new long[dy];
                }
        }
};
```

Matrix Constructor

```
// Dynamically allocate a sizeX × sizeY matrix,
// initialized to all 0 entries
Matrix::Matrix(int sizeX, int sizeY)
: dx(sizeX),dy(sizeY)  {
        allocArrays();
        for (int i = 0; i < dx; i++) {
                for (int j = 0; j < dy; j++) {
                        p[i][j] = 0;
        }
    }
}
```

Matrix Default Constructor

```
Matrix::Matrix() : Matrix(1,1) {}
// We could also have given default arguments
// to our previous constructor:
Matrix::Matrix(int sizeX=1, int sizeY=1)
        // the rest is the same as before
```

Matrix Copy Constructor

```
Matrix::Matrix(const Matrix& m)
: dx(m.dx), dy(m.dy) {
    allocArrays();
    for (int i=0; i<dx; ++i) {
        for (int j=0; j<dy; ++j) {
            p[i][j] = m.p[i][j];
        }
    }
}
```

"this" has been allocated, but not initialized at this point.

Matrix Destructor

```
Matrix::~Matrix() {
   for (int i = 0; i < dx; i++) {
         delete [] p[i];
   }
   delete [] p;
}
```

Matrix Copy Assignment

```
Matrix &Matrix::operator =
  (const Matrix &m) {
  if (this == &m) {
    // avoid self-assignment
    return *this;
  } else {
    if (dx != m.dx || dy != m.dy) {
      this->~Matrix();
      dx = m.dx; dy = m.dy;
      allocArrays();
    }
    for (int i = 0; i < dx; i++) {
      for (int j = 0; j < dy; j++) {
        p[i][j] = m.p[i][j];
      }
    }
    return *this;
}
```

The Copy Assignment Return Type

Remember the signature of copy assignment:

```
Foo& Foo::operator=(const Foo& rhs);
```

“this” points at the left-hand-side:

this x = y; rhs

The code makes “this” have the same contents as the rhs. So why not return void?

Answer: “x = y = z”, which is parsed as “x = (y = z)”

The Copy Assignment Return Type

The traditional return type “Foo&” even supports code like this:

++(x=y)

if you’ve defined Foo::operator++(). This copies the contents of y into x, then increments x.

Some More Matrix Operations

We next define matrix addition, output, and element access.

```
class Matrix {
public:
    …
    Matrix operator+(const Matrix & m);
    Matrix& operator+=(const Matrix & m);
    friend ostream &operator<<
           (ostream &out, const Matrix &m);
    long &operator()(int x, int y);
    …
};
```

Matrix Addition

```
Matrix& Matrix::operator+=(const Matrix& m) {
  // x+=y adds the y-entries into the x-entries
  for (int i=0; i<dx; ++i) {
        for (int j=0; j<dy; ++j) {
              p[i][j] += m.p[i][j];
        }
  }
  return *this;
}
```

Matrix Addition

```
Matrix Matrix::operator+(const Matrix& m) {
   Matrix temp(*this); //copy constructor
   return (temp += m);
}
```

The assignment form, +=, does the real work.
The copy constructor does the allocation.
This trick is less useful for matrix multiplication.

+, +=, and =

- In C++, defining operator+ and operator= **does not** automatically give the right meaning to +=.
- This language-design bug is fixed in C#.

Overloading the << Operator

```
cout << text1 << age << text2;

ostream &operator<<(ostream &ostr, string &s);

cout << age << text2;

cout << age << text2;

ostream &operator<<(ostream &ostr, int i);

cout << text2;
…
```

Matrix Output

```
ostream &operator<<
        (ostream &out, const Matrix &m)
{
        for (int i = 0; i < m.dx; ++i) {
                for (int j = 0; j < m.dx; ++j)
                        out << m.p[i][j] << "  ";
                out << endl;
        }
        return out;
}
```

Overloading the << Operator

- operator<< must be a non-member function (an ordinary function), since the first operand is an ostream, and not "this".
- We make operator<< a friend of Matrix, so it has access to m.dx, m.dy, and m.p.
- Friend functions are usually overloaded operators, for just this reason.

Matrix Element Access

We will overload () so we can write things like:

```
long x = myMatrix(1,2);

myMatrix(0,1) = 25;
```

```
class Matrix {
public:
    …
    long &operator()(int x, int y);
    …
};
```

Overloading the () Operator

```
long &Matrix::operator()(int i, int j) {
    return p[i][j];
}
```

Note that operator() returns the matrix element **by reference.** Why?

Answer: so we can put “myMatrix(i,j)” on the left-hand side of an assignment statement:

```
myMatrix(0,1) = 25;
```

Matrix Multiplication

We conclude with matrix multiplication. We want to be able to write two kinds of statements:

```
matProduct = mat1 * mat2;
matDouble = 2 * mat1;
```

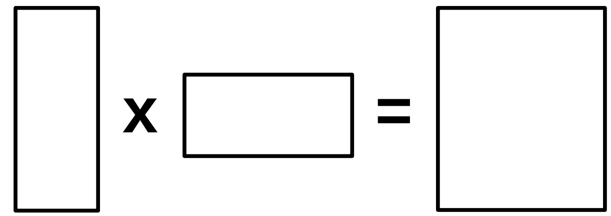
We write two ordinary, non-member functions, and let ordinary overloading pick the right one.

Matrix Multiplication

```
class Matrix {
public:
   …
   friend Matrix operator*
      (const Matrix & m1, const Matrix & m2);
   friend Matrix operator*
      (long c, const Matrix & m2);
   friend Matrix operator*
      (const Matrix & m1, long c);
   …
};
```

Matrix Multiplication

```
Matrix operator*(const Matrix& m1, const Matrix& m2) {
    Matrix prod(m1.dx, m2.dy);
    for (int i=0; i<prod.dx; ++i) {
        for (int j=0; j<prod.dy; ++j) {
            for (int k=0; k<m1.dy; ++k) {
                prod.p[i][j] += m1.p[i][k] * m2.p[k][j];
            }
        }
    }
    return prod;
}
```



Matrix Multiplication

```
Matrix operator*(long c, const Matrix& m2) {
    Matrix prod(m2);
    for (int i=0; i<prod.dx; ++i) {
        for (int j=0; j<prod.dy; ++j) {
            prod.p[i][j] = c * m2.p[i][j];
        }
    }
    return prod;
}
Matrix operator*(const Matrix& m2, long c) {
    return c*m2;
}
```

Testing the Code

```
int main() {
    Matrix x(2,1), y(1,2), z(1,1);
    x(0,0) = 1;
    x(1,0) = 2;
    y(0,0) = 3;
    y(0,1) = 4;
    cout << "Matrix x\n" << x
         << "\nMatrix y\n" << y
         << "\nMatrix z\n" << z << endl;
    cout << "x*y = \n" << x*y << endl;
    z = x*y;
    cout << "Matrix z = x*y  (note new dimensions)\n" << z << endl;
    Matrix x2(2,1);
    x2 = 2*x;
    cout << "Matrix x2 = 2*x\n" << x2 << endl;
    cout << "x + x2 = \n" << x+x2 << endl;
    return 0;
}
```

Output

Matrix x
1
2

Matrix y
3 4

Matrix z
0

x*y =
3 4
6 8

Matrix z = x*y (note new dimensions)
3 4
6 8

Matrix x2 = 2*x
2
4

x + x2 =
3
6

Final Remarks

- The full code Matrix.cpp is in the Examples section (see the course homepage)
- **ALL** the code should have had error checking, for example:
```
assert(sizeX > 0);
if (m1.dy != m2.dx) throw MatMulException();
```
  The on-line copy of Matrix.cpp has asserts.
- We didn’t code *= for matrices; this is left as an exercise.
- Another exercise: define a class BinaryTree, and figure out what overloaded operators should be defined for it. (“BinTree x = y + z” ?)

Appendix

The class version of this Powerpoint file had a couple of bugs. These bugs are instructive, especially for those with a Java background. Hence this appendix.

We have three places in the code where we need to allocate the arrays p and the p[i] arrays:

```
Matrix(int sizeX, int sizeY); //constructor
Matrix(const Matrix& m); //copy constructor
Matrix& operator=(const Matrix& rhs);
    //copy assignment
```

Appendix

We'd like to reuse the code that does these allocations. Above, I've introduced the private method allocArrays() to do this.

In Java and in C#, one constructor can call another constructor in the same class:

```
// Java

Matrix(Matrix m) {

   this(m.dx, m.dy);

}
```

```
// C#

Matrix(Matrix m)

: this(m.dx, m.dy) {

}
```

Appendix

I tried to do something similar:

```
Matrix::Matrix(const Matrix& m)
: Matrix(m.dx, m.dy) {...}
```

Illegal!

```
Matrix::Matrix(const Matrix& m) {
  Matrix(m.dx, m.dy);
  ...
}
```

Legal, but just creates an anonymous Matrix object – doesn't affect "this"

Appendix

In operator=, I tried:

```
if (dx != m.dx || dy != m.dy) {
    this->~Matrix();
    Matrix(m.dx, m.dy);
}
```

legal! And does the right thing.

Legal, but just creates an anonymous Matrix object – doesn't affect "this"

Note: “this->Matrix(...)” is **illegal!**

For more on this topic, see:

http://www.parashift.com/c++-faq-lite/ctors.html#faq-10.3