A Matrix Class

During the course of writing the `Matrix` class we will cover some interesting C++ topics. Specifically:

- constructors and destructors,
- operator overloading,
- the rule of three,
- returning references,
- overloading using `const`.

Basic functionality

The `Matrix` class will store a 2-dimensional array of doubles and will have the following data members (all private).

(i) `int nrows`. The number of rows.

(ii) `int nrows`. The number of columns.

(iii) `double* data`. A pointer to the first cell.

(iv) `double* endPointer`. A pointer to one after the last cell.

The pointer data will point to a single chunk of memory of length nrows $\times$ ncols. The cell (i, j) will be stored at the location `data+(j*nRows)+i`.

Declarations

```
private:

    /*  The number of rows in the matrix */
    int nrows;
    /*  The number of columns */
    int ncols;
    /*  The data in the matrix */
    double* data;
    /*  Pointer to one after the end of the data */
    double* endPointer;
```

Simple methods

```
/*  The number of rows in the matrix */
int nRows() const {
    return nrows;
}

/*   The number of columns  in the matrix */
int nCols() const {
    return ncols;
}
```

These methods are inlined.

Get and set methods

```
/*  Retrieve the value at the given index */
double get( int i, int j ) const {
    return data[ offset(i, j ) ];
}

/*  Set the value at the given index */
void set( int i, int j, double value ) {
    data[ offset(i, j ) ] = value;
}
```

```
int offset( int i, int j ) const {
    ASSERT( i >=0 && i<nrows && j>=0 && j<ncols );
    return j*nrows + i;
}
```

A constructor

```
    Matrix( int nrows, int ncols, bool zeros=1 );
```

```
Matrix::Matrix( int nrows, int ncols, bool zeros )
    : nrows( nrows ), ncols( ncols ) {
    int size = nrows*ncols;
    data = new double[size];
    endPointer = data+size;
    if (zeros) {
        // memset is an optimized low level function
        // that should be faster than looping
        memset( data, 0, sizeof( double )*size );
    }
};
```

Calling delete[]

Our `Matrix` will be removed from memory under the following circumstances:

(i) If the `Matrix` was created by `new`, it will be removed from memory when `delete` is called.

(ii) If the `Matrix` was created by `new []`, it will be removed from memory when `delete []` is called.

(iii) If the `Matrix` was created on the stack as a local variable, it will be removed from memory when the local variable is no longer needed (i.e., when it goes out of scope).

(iv) If the `Matrix` is a member variable of another object, this will happen when the containing object is deleted.

We need to call `delete[]` when one of these happens. Use a destructor.

A Destructor for `Matrix`

```
class Matrix {
...
    ~Matrix() {
        delete[] data;
    }
...
}
```

Needn't be inline.

Writing a destructor

To write a destructor for your class you must follow these rules.

(i) A destructor is declared and defined just like a function except...

(ii) It must have the same name as the class except with the the addition of a tilde ~.

(iii) It must have no return value (not even `void`).

(iv) It must have no parameters.

(v) It must not be `const`.

Rules for destructors

- All classes that you wish to subclass should have a `virtual` destructor.
- Whenever you write a destructor, other than an empty `virtual` destructor, you must abide by *the rule of three*

You will notice that our `Matrix` class does not have a `virtual` destructor, therefore, you must not subclass it. The same applies to many standard classes. For example, you should never subclass `vector<double>`, no matter how tempted you may feel.

When is a destructor needed?

- Whenever you call `new` or `new[]` in the constructor and don't use a `shared_ptr`
- When you obtain a resource in the constructor that you must release in the destructor:
  - a chunk of memory;
  - a lock on a file that prevents others writing to the file;
  - a print job that you've started;
  - a connection to a database.
- Not very often in mathematical code. In practice typically only if you are using a C-programming interface.

Additional constructors

- A default constructor that creates a 1×1 matrix containing the number zero.
- A constructor that takes a `std::vector<double>` and constructs a corresponding column vector. It has an optional additional argument you can use if you want to create a row vector.
- A constructor that takes a single scalar and creates a 1×1 matrix.
- A constructor that takes a string describing the contents of the matrix.

```
Matrix m("1,2,3;4,5,6");
ASSERT( m.nRows()==2 );
ASSERT( m.nCols()==3 );
```

Const pointers

```
/*  Access a pointer to the first element */
const double* begin() const {
    return data;
}
/*  Access a pointer to the element after last */
const double* end() const {
    return endPointer;
}
/*  Access a pointer to the first element */
double* begin() {
    return data;
}
/*  Access a pointer to the element after last */
double* end() {
    return endPointer;
}
```

- The two begin functions differ by the `const` on the end
- If you call the function on a `const` matrix you are given a `const` pointer.

Operator overloading

The code we would like to write:

```
Matrix m1("1,2,3;4,5,6");
Matrix m2("2,3,4;5,6,7");

Matrix actual = m1 + m2;

Matrix expected("3,5,7;9,11,13");
expected.assertEquals( actual, 0.001 );
```

By overloading the + operator, we can make this code compile.
In fact we can overload practically every C++ operator to make the matrix class much easier to work with.

Overloading plus

```
/*  Add two matrices
    NB - not a member function  */
Matrix operator+(const Matrix& x, const Matrix& y );
```

Implementation

```
Matrix operator+(const Matrix& x, const Matrix& y ) {
    ASSERT( x.nRows()==y.nRows()
                   && x.nCols()==y.nCols());
    Matrix ret(x.nRows(), x.nCols(), 0 );
    double* dest = ret.begin();
    const double* s1 = x.begin();
    const double* s2 = y.begin();
    const double* end = x.end();
    while (s1!=end) {
        *(dest++) = *(s1++) + *(s2++);
    }
    return ret;
}
```

Adding a scalar

```
/*  Add a scalar to every element of a matrix
    NB - not a member function */
Matrix operator+(const Matrix& m, double scalar );
```

```
/*  Add a scalar to every element of a matrix
    NB - not a member function */
inline Matrix operator+(double scalar,
                        const Matrix& m ) {
    return m+scalar;
}
```

Implementing everything required for operator overloading can be time consuming, but it can result in a class that is very easy to use.

Overloading other arithmetic operators

- Overloading - is much the same as overloading +.
- Overloading * is straightforward too, apart from the fact that there are two possible choices for how to implement it.
- Should it mean matrix multiplication or entrywise multiplication?

Comparison operators

Overloading >, >=, ==, `!=`, <, <= is straightforward. Here's a typical declaration. It takes two `const` references and returns a Matrix of 0's and 1's.

```
/*  Comparison operator
    NB - not a member function */
Matrix operator>(const Matrix& x, const Matrix& s );
```

```
        Matrix test1("1,2;3,4");
        Matrix test2("3,3;3,3");
        Matrix expected("0.0,0.0;1.0,1.0");
        expected.assertEquals( test1>=test2, 0.001);
```

Overloading the << operator

```
/*  Write a matrix to a stream
    NB - not a member function */
std::ostream& operator<<(std::ostream& out,
                         const Matrix& m );
```

The function `operator<<` returns a reference to an `ostream` that we can do some more writing to. This will, in practice, always be the same `ostream` that we pass in as the parameter `out`.

Why return an ostream

```
cout << "To be " << "or not to be";
```

Is equivalent to the following:

```
(cout << "To be ") << "or not to be";
```

Implementation

```
ostream& operator<<(ostream& out, const Matrix& m ) {
    int nRow = m.nRows();
    int nCol = m.nCols();
    out <<"[";
    for (int i=0; i<nRow; i++) {
        for (int j=0; j<nCol; j++) {
            out << m(i,j);
            if (j!=nCol-1) {
                out << ",";
            }
        }
        if (i!=nRow-1) {
            out << ";";
        }
    }
    out <<"]";
    return out;
}
```

Return by reference

- Return by reference is acceptable, so long as you don't return a reference to a local variable
- It is potentially more efficient than return by value.
- Returning a reference allows the user to modify what the reference points to.

Overloading the () operator

Usage example:

```
Matrix m("1,2,3;4,5,6");
ASSERT( m(1,2)==6 ); // read a value
m(1,2)=0; // change the value
```

We've chosen round brackets, you could use square. Many C++ libraries allow you to use either.

Implementation

Note that it must be a member function:

```
double& operator()(int i, int j ) {
    return data[ offset(i,j) ];
}
```

```
const double& operator()(int i, int j ) const {
    return data[ offset(i,j) ];
}
```

Note that we return a reference in order that you can modify cells using () too.

Overloading +=

Member declaration

```
Matrix& operator+=( const Matrix& other );
```

You should always return a reference to `*this`. This allows you to write code such as:

```
Matrix a("1,2");
Matrix b("1,2");
(a+=b)+=b;
```

Implementation

```
Matrix& Matrix::operator+=( const Matrix& other ) {
    ASSERT( nRows()==other.nRows()
            && nCols()==other.nCols());
    double* p1=begin();
    const double* p2=other.begin();
    while (p1!=end()) {
        *p1=(*p1) + (*p2);
        p1++;
        p2++;
    }
    return *this;
}
```

The rule of three

Whenever you write a destructor (other than an empty virtual destructor) you must:

- override the assignment operator =;
- write a copy constructor.

In fact if you write any one of these three things:

- a non-trivial destructor;
- a copy constructor;
- an assignment operator =;

then you should write all three.

Overriding the assignment operator

Suppose that we have two variables of type `Matrix` called `a` and `b`. We write

```
a = b;
```

`=` is called the assignment operator because it is used to assign a value to a variable.

- C++ gives classes a default assignment operator.
- If we just copied data, both matrices would share the same data.
- When one matrix was deleted the other would be broken.
- The rule of three says if your class needs a destructor, the default assignment operator won't work.

Assignment operator

```
    Matrix& operator=( const Matrix& other ) {
        delete[] data;
        assign( other );
        return *this;
    }
```

```
void Matrix::assign( const Matrix& other ) {
    nrows = other.nrows;
    ncols = other.ncols;
    int size = nrows*ncols;
    data = new double[size];
    endPointer = data+size;
    memcpy( data, other.data, sizeof( double )*size );
}
```

Rules for the = operator

- The = operator should be defined as a member function.
- It should take a `const` reference and return a reference.
- You should always return `*this`.
- You should abide by the rule of three.

Copy constructor

Using the copy constructor explicitly:

```
Matrix a("1,2;3,4");
Matrix b(a); // copy a
```

- C++ uses the copy constructor if it needs to copy data for pass by value.
- This means that copy constructors are actually called a lot without you noticing it.
- The rule of three tells us that the default copy constructor won't work if your class needs a destructor.

Declaration of the copy constructor:

```
    Matrix( const Matrix& other ) {
        assign( other );
    }
```

- A copy constructor takes a single parameter: a `const` reference to another instance.
- It is not marked as `explicit` despite only taking one parameter.
- It performs whatever tasks are necessary to copy the data from the other reference.

The lazy way of meeting the rule of three

- Make the copy constructor and assignment operator private and don't implement them.
- This means your object can't be passed by value. Since we prefer pass by reference for objects, this often won't be a problem at all.

Other features of Matrix

- Member functions `exp`, `log`, `sqrt`, `pow`, `times`.
- Functions `setCol` and `setRow` to copy individual rows and columns from one matrix to another.
- Functions `row` and `col` to extract a row or column.
- Member function `positivePart` that returns $(x)^+$ for every cell x. This is handy for call options.
- `matlib` has been rewritten throughout so it works with `Matrix` rather than with `std::vector`.
- `matlib` has new functions to make it easy to work with matrices such as `ones` and `zeros`.
- Functions like `meanRows` and `meanCols` have been added to replace `mean`.

Array programming

- A real `Matrix` class hard to write. It may use
  - Vectorisation
  - Clever memory management
  - Multiple threads
  - GPUs
- In array programming you try to rewrite maths computations as matrix calculations. If your `Matrix` class is multi threaded, then your calculation will be too.

Array programming example

```
Matrix UpAndOutOption::payoff(
        const Matrix& prices ) const {
    Matrix max = maxOverRows( prices );
    Matrix didntHit = max < getBarrier();
    Matrix p = prices.col( prices.nCols()-1);
    p -= getStrike();
    p.positivePart();
    p.times(didntHit);
    return p;
}
```

FMLib has been rewritten to use array programming.

Summary

In terms of the C++ language we have studied the following topics:

- `const` pointers.
- How to write two member functions: one that works on `const` instances; one that works on standard instances.
- How to overload operators such as +, ∗ and >=.
- How to overload the << operator to make objects easy to print.
- How to overload =, += and -=.
- How to write a destructor for classes that manage memory and other resources. Note that most classes don't need a destructor.
- The rule of three: whenever we write a destructor we write a copy constructor and override =.