# **C++** February 5, 2015

#### Mike Spertus

mike spertus@symantec.com

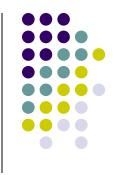


## Matrix example



- For the rest of the quarter, we are going to create a linear algebra library to see how to combine the C++ features we have been learning to make an awesome library
- We are (very) loosely inspired by the matrix classes from the Origin C++ libraries

#### **Matrices**



 A matrix is just a two dimensional array of numbers (picture from Wikipedia)

$$\begin{bmatrix} 1 & 9 & -13 \\ 20 & 5 & -6 \end{bmatrix}$$
.

- Matrices are used in all branches of science, statistics, economics, math, etc. and can be added, multiplied, or have their determinants taken
- I will give you all necessary formulas

### **Initializing matrices**



 We would like our Matrix class to have a natural initializer like the following 2x3 matrix

```
• Matrix<2,3> m = \{ \{2, 4, 6\}, \{1, 3, 5\} \};
```

## **Another kind of constructor Initializer lists**



 In C++98, one of the problems with using vectors was that it was difficult to initialize them

```
• // Yuck!
int init[] = { 0, 1, 1, 2, 3, 5};
vector<int> v(init,
init+sizeof(init)/sizeof(init[0]);
```

# Can't user-defined types be as easy to init as built-in ones?



- In C++11, you can initialize vectors as easily as C arrays
  - vector<int>  $v = \{0, 1, 1, 2, 3, 5\};$
- How does vector<T> do this?
- It has a constructor that takes a
   std::initializer\_list<T>, which
   represents a "braced initializer of Ts" expression
- Initializer lists have begin (), end (), and size() methods so your constructor can iterate through their value.

## Initializer list constructor for matrix



- The code to initialize as mentioned previously is on chalk
- Note from the slide above that we initialize matrices with "initializer lists of initializer lists"
  - The initializer list contains an initializer list for each row

```
Matrix(initializer_list<initializer_list<double>> init) {
   auto dp = data.begin();
   for (auto row : init) {
     std::copy(row.begin(), row.end(), dp->begin());
     dp++;
   }
}
```

#### **Matrix arithmetic**



- In order to easily add and multiply matrices, we would like to be able to tell "+" and "\*" about matrices
- This is called operator overloading

### **Operator overloading**



- You can overload operators just like functions
- The following operators can be overloaded:
- Unary operators:

Binary operators:

## Which operators can't be overloaded?



- ., .\*, ?:, ::
- Fame and fortune await for the one who figures out how to overload "operator.()"

# Operator overloading examples



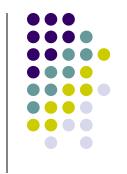
```
class myString {
 myString(const char *cp);
  char operator[](size t idx) const;
  myString operator+(myString &addend) const;
 myString operator+=(myString &addend);
  inline friend myString
    operator+(const myString &s1, const myString &s2) {
// Alternatively
myString
operator+(const myString &s1, const myString &s2);
```

## Which way of overloading addition is better?



- Consider "Hello " + myString("World")
- Doesn't work for the member function
  - The first argument isn't even a class, so the compiler wouldn't know where to look for a member function.
- What about myString("World") + "Hello"
  - Works for both
- Using a global function makes sure both arguments are treated the same way, which fits the intuition that addition operators, which are generally commutative, should apply the same rules to each arguments.

### Do the same way for printing



- ostream & operator<<(ostream &os, myString const &ms) {...}</li>
- If you want to be fancy template<typename charT, typename traits> basic\_ostream<charT, traits> &operator<<(basic\_ostream<charT, traits> &os, myString const &ms) {...}

# How does an I/O manipulator get invoked



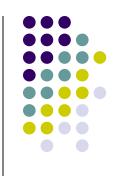
 Recall that endl is defined (as modulo some template complication that is irrelevant here) follows

```
ostream &
endl(ostream &os)
{
   os << '\n';
   os.flush();
   return os;
}</pre>
```

How come "cout << endl;" actually behaves as "endl (cout)"?</li>

#### **Another overload!**

```
ostream &
operator<<
  (ostream&os,
   ostream&(*manip)(ostream &))
{
   return manip(os);</pre>
```

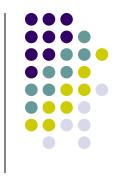


## How does a smart pointer work?



- Overloading operator->() of course
- operator->() overloads with a unique rule
  - Keep doing -> until it is illegal

#### shared\_ptr

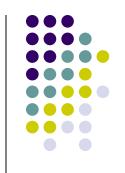


- shared\_ptr is a reference counted pointer.
- Inside shared\_ptr, we have something like:

```
template<class T>
class shared_ptr {
   // Returns the wrapped pointer
   T *operator->();
};
```

 Because the -> is applied again, it acts just like the wrapped pointer except that it maintains a reference count.

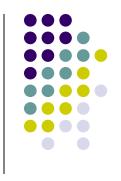
## Flow of control is hard to follow but memory is easy to manage



```
class A;
int f()
   shared ptr<A> ap1(new A());
   shared ptr<A> ap2(new A());
   return ap1->i + ap2->i;
```

Deletes automatically no matter what





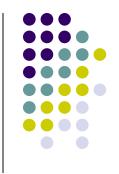
```
#include <boost/shared ptr.hpp>
int f()
  auto ap = make shared<A>();
 ap - > m(1);
 g(ap);
} // the A object is automatically deleted
 // when we leave scope unless someone
 // else is using it
```

### **Best practice**



- All dynamic duration objects should be owned by a smart pointer
- Not all uses need to be through a smart pointer, but the owner needs to be one

### Overloading operator++()



- To overload ++x, writeX &X::operator++() { ... }
- To overload x++, writeX &X::operator++(int) {...}
- The int argument isn't really there. Don't use it! The signature just gives a way to distinguish preincrement and postincrement

#### Overload && and ||



- The built-in operator&& (logical and) has "short circuit evaluation"
  - If the left argument is false, the right one isn't evaluated because it can't make the && true.
  - i != 0 && 5/i > 0 // Will never divide by 0
- User-defined overloads of && and || never short circuit. Both arguments are evaluated no matter what.

## Overloading operators for matrices



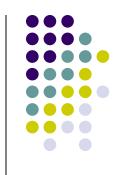
 The sample Matrix.h on chalk overloads matrix multiplication with the (complicated) rule for multiplying matrices

## Specializing templates



- The "secret sauce" for C++ templates is that if the general "generic" definition of the template isn't really what you want for a particular set of template parameters, you can override it for that particular case with a specialization
- Think of this as the compile-time analog to object orientation where you also override a more general method in a more specialized derived class

#### **Matrix determinants**



- The determinant is a number that represents "how much a matrix transformation expands its input"
  - Don't worry if you don't understand this
- We will just use the formula to calculate them
- The general formula is here
  - http://en.wikipedia.org/wiki/Laplace\_expansion
- But I will give you the special case you need for the homework

## Full specialization



- A function, class, or member can be fully specialized
- See the definition of Matrix<1,1>::determinant() in Matrix.h

### Partial specialization



- Only classes may be partially specialized
- Template class:

```
template<class T, class U>
class Foo { ... };
```

Partial specialization:

```
template<class T>
class Foo<T, int> {...};
```

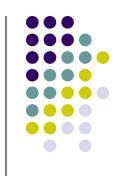
 You can tell the second is a specialization because of the <> after the class name

## Partial specialization



- The partially specialized class has no particular relation to the general template class
  - In particular, you need to either redefine (bad) or inherit (good) common functionality
  - For example, see PSMatrix.h

### **Overloading**



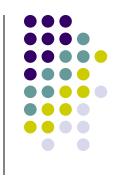
- functions cannot be partially specialized, so overloading is used instead
- For example, see OverloadMatrix.h

## Compilation of template methods



- A method of a template class is only compiled if it is used
  - Indeed this is true for any kind of template
- That's why in the Matrix example Matrix<int, 1, 1> objects can be instantiated even though Matrix<int, 1, 1>::minor(int, int) doesn't compile
- This has interesting implications for static members

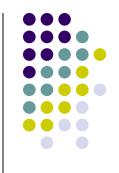
#### **Exercise 5-1**



- Modify Matrix.h to let you add matrices
- To add two matrices, they both have to have the same number of rows and columns
- Just add the corresponding elements to get a new matrix with the same number of rows and columns
- See

<u>http://www.purplemath.com/modules/mtrxadd</u>
<u>.htm</u> for an example

#### Exercise 5-2



- For each of the following programs, modify them to have a direct (i.e., specialized or overloaded implementation) of determinants for 2x2 matrices.
  - Matrix
  - PSMatrix
  - OverloadMatrix
- The formula for the determinant of the 2x2 matrix m is

```
m(0,0)*m(1,1) - m(1,0)*m(0,1)
```

 Test how much your code changed the execution time for the programs. What do you conclude?

#### **Exercise 5-3: Extra credit**



- Some of the calculations in determinant seem to copy matrices a lot
- Can you modify the matrix class to be (efficiently) movable
- Does that improve the benchmark?

#### **Exercise 5-4**



- Create a ComplexInt class that acts like a complex integer (c.r is the real part. c.i is the imaginary part)
  - Define multiplication and addition for complex integers
  - Ensure that "cout << c;" prints something like 5+3i.

#### **Exercise 5-5: Extra credit**



- Look up how to create user-defined literals
- Create a user-defined literal to make it easier to enter complex numbers

```
ComplexInt ci = 6 + 3_i;
```

#### **Exercise 5-6: Extra Credit**



What happens when you do

```
ComplexInt ci(2, 7); // 2 + 7i
cout << setw(10) << ci << endl;</pre>
```

- Is this the desired behavior?
- If not, how would you fix it?