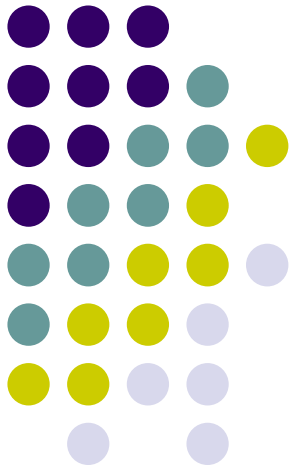
C++

February 12, 2013

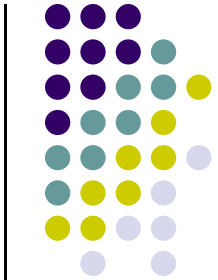
Mike Spertus

mike_spertus@symantec.com

YIM: spertus



Operator overloading



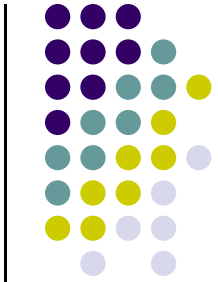
- The following operators can be overloaded:
- Unary operators:

  + - * & ~ ! ++ -- -> ->*

- Binary operators:

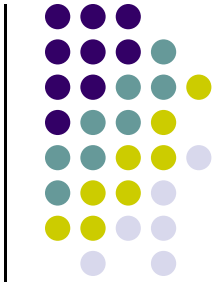
  + - * / % ^ & | << >>
  += -= *= /= %= ^= &= |= <<= >>=
  < <= > >= == != && ||
  , [] ()
  new new[] delete delete[]

Which operators can't be overloaded?



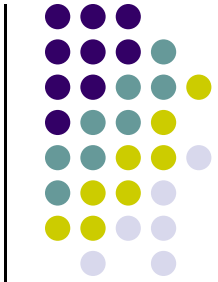
- `.`, `.*`, `?:`, `::`
- Fame and fortune await for the one who figures out how to overload "operator.()"

Operator overloading examples



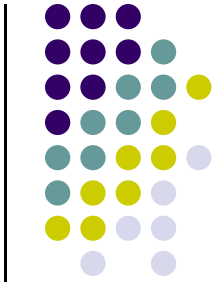
```
class myString {
  myString(const char *cp);
  char operator[](size_t idx) const;
  myString operator+(myString &addend) const;
  myString operator+=(myString &addend);
  inline friend myString
    operator+(const myString &s1, const myString &s2)
{
  }
};
// Alternatively
myString
operator+(const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

Which way of overloading addition is better?



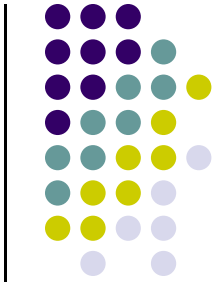
- Consider `"Hello " + myString("World")`
- Doesn't work for the member function
  - The first argument isn't even a class, so the compiler wouldn't know where to look for a member function.
- Using a global function makes sure both arguments are treated the same way, which fits the intuition that addition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

How does an I/O manipulator get invoked



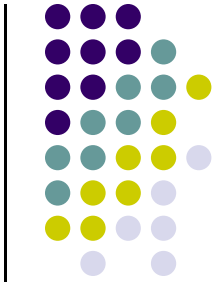
- Recall that `endl` is defined (as modulo some template complication that is irrelevant here) follows
-
```
ostream &
endl(ostream &os)
{
    os << '\n';
    os.flush();
    return os;
}
```
- How come "`cout << endl;`" actually behaves as "`endl(cout)`"?

Another overload!



```
ostream &
operator<<
 (ostream&os,
  ostream&(*manip)(ostream &))
{
    return manip(os);
}
```

shared_ptr

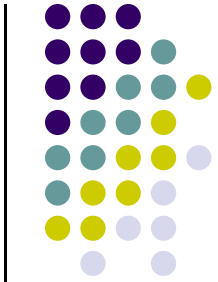


- shared_ptr is a reference counted pointer.
- Inside shared_ptr, we have something like:

```
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.

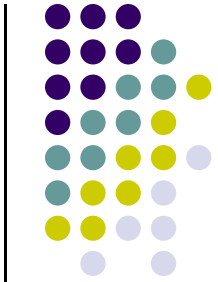
Flow of control is hard to follow but memory is easy to manage



```
class A;
int f()
{
    shared_ptr<A> ap1(new A());
    shared_ptr<A> ap2(new A());
    return ap1->i + ap2->i;
}
```

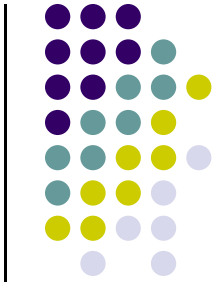
- Deletes automatically no matter what

Example



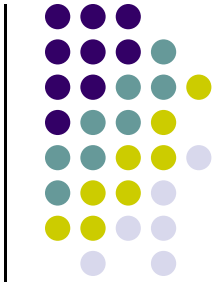
```
#include <boost/shared_ptr.hpp>
int f()
{
  shared_ptr<A> ap(new A);
  ap->m(1);
  g(ap);
} // the A object is automatically deleted
  // when we leave scope unless someone
  // else is using it
```

How does a smart pointer work?



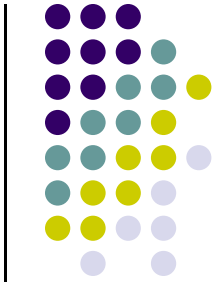
- Overloading operator->() of course
- operator->() overloads with a unique rule
  - Keep doing -> until it is illegal

Best practice



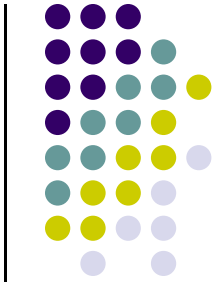
- All dynamic duration objects should be owned by a smart pointer
- Not all uses need to be through a smart pointer, but the owner needs to be one

Overloading operator++()



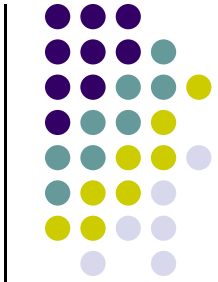
- To overload ++x, write
  X &X::operator++() { … }
- To overload x++, write
  X &X::operator++(int) {…}
- The int argument isn't really there. Don't use it! The signature just gives a way to distinguish preincrement and postincrement

Overload && and ||



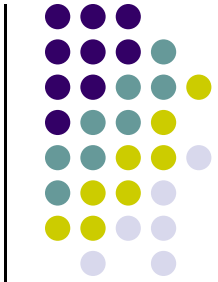
- The built-in operator&& (logical and) has “short circuit evaluation”
  - If the left argument is false, the right one isn’t evaluated because it can’t make the && true.
  - i != 0 && 5/i > 0 // Will never divide by 0
- User-defined overloads of && and || never short circuit. Both arguments are evaluated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

Specialization and overloading



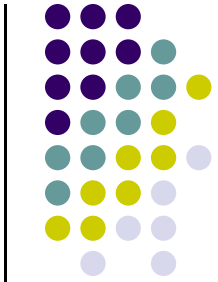
- These are demonstrated in chalk

Full specialization



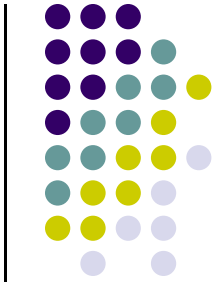
- A function, class, or member can be fully specialized
- See the definition of IntegerMatrix<1,1>::determinant() in IntegerMatrix.cpp

Partial specialization



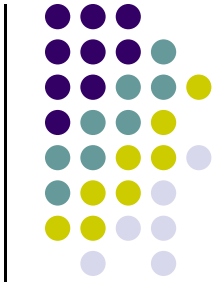
- Only classes may be partially specialized
- Template class:
```
template<class T, class U>
class Foo { ... };
```
- Partial specialization:
```
template<class T>
class Foo<T, int> {...};
```
- You can tell the second is a specialization because of the <> after the class name

Partial specialization



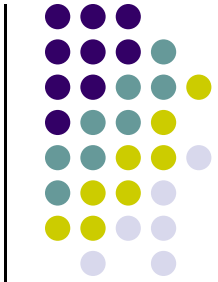
- The partially specialized class has no particular relation to the general template class
  - In particular, you need to either redefine (bad) or inherit (good) common functionality
  - For example, see PSMatrix.cpp

Overloading



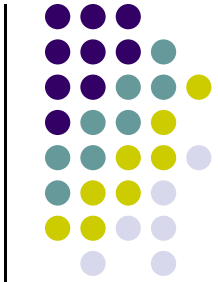
- functions cannot be partially specialized, so overloading is used instead
- For example, see OverloadMatrix.cpp

Compilation of template methods



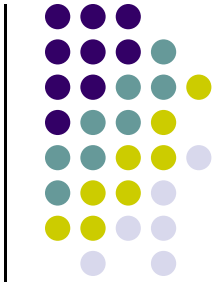
- A method of a template class in only compiled if it is used
  - Indeed this is true for any kind of template
- That's why in the Matrix example `Matrix<int, 1, 1>` objects can be instantiated even though `Matrix<int,1,1>::minor(int, int)` doesn't compile
- This has interesting implications for static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

Exercise 6-1



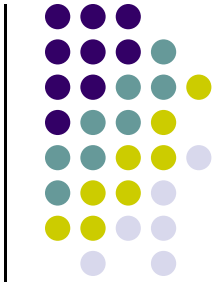
- Create a ComplexInt class that acts like a complex integer (c.r is the real part. c.i is the imaginary part)
  - Define multiplication and addition for complex integers
  - Ensure that “cout << c;” prints something like 5+3i.

Exercise 6-2



- Create a template version of the complex class

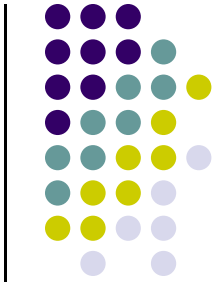
Exercise 6-3



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

  ```
  m(0,0)*m(2,2) – m(1,0)*m(0,1)
  ```
- Test how much your code changed the execution time for the programs. What do you conclude?

Exercise 6-4: Very hard extra credit



- Find a legitimate reason for a function to return the address of a local variable