Lecture 08b
Operator Overloading
Oct. 12th, 2017

A Pitch for Unit Testing
• How many combinations of suffixes are there?
• How many other special cases?
  – Definition of Vowel (and other defs)
  – Region based conditions
  – Preceder based conditions
• How did you test PA4?
• How will you test PA5?

GoogleTest
• For every public method, write a test
  – Each test is almost trivial to write
  – Each test can be used for every assignment
  – Think of them like scaffolding…
• Does not solve everything
  – End-to-end testing is still needed for interactions
• But it will shorten your development time
  – Its hard to find bugs that don’t crash in large code

OK, A Question For You

//a is an Animal (or derived from it)
//lifespan is a virtual function a.lifespan()

// compiler generates (pseudo-code)
//a is the start of the object, and
//therefore also the VFPT ptr
//follow that ptr to get to VFPT
addr = *a

// offset of lifespan calculated at compile time
// same for Animal, Mammal, ...
addr += offset(lifespan)

// this is the actual function call
jump to *addr

class Animal {
public:
  inline bool Behave() {return WarmUp();}
  bool WarmUp() {return false;}
};

class Mammal : public Animal {
public:
  virtual bool WarmUp() (Shiver(); return true;)
};

class Equine : public Mammal {
public:
  virtual bool WarmUp() (Shiver(); Buck(); return true;)
};
Now the Question

Equine e;
std::cout << e.Behave() << std::endl;

• What does this print?
• Why?

Overloading & Java

• Both support polymorphism
  – What is polymorphism again?
• Polymorphism supports overloading
  – Abstracts semantics but not implementation
  – Think of WarmUp() for Animal, Mammal, Fish, Reptile.
• What about the functionality of operators?
  – How does Java abstract operators like I/O?

Overloading & Java (II)

• Through polymorphism and the object class
  – All objects define toString()
  – The I/O functions call toString()
• How would you overload the functionality of other operators?
  – In general, you can’t
• C++ is more general and more powerful
  – And a sharper knife

Overloading

• C++ allows you to define functions (or methods) for operators on non-primitive data types.
  • Example:
    – I create a Complex number class
    – I want complex numbers to behave
      • Like other numbers
      • Like they do in math textbooks
    – So I overload +, -, *, /, etc.
    – Now Complex numbers act like primitive numbers
• Java doesn’t allow operator overloading
  – You cannot define a class that works like a primitive
  – Of course, you can’t pass an object like a primitive, either.
  – In Java, primitives and objects are just different

Syntax of Overloading

• Like any other method or function, except
  – Name is operator op
  – Where ‘operator’ is a keyword
  – Where op is the operator symbol being overloaded

Example of Overloading

class Complex {
  public:
    Complex(double real_part = 0.0, double imaginary_part = 0.0) :
      real(real_part), imaginary(imaginary_part) {}
  
  inline double Real() const {return real;}
  inline double Imaginary() const {return imaginary;}
  inline double& Real() {return real;}
  inline double& Imaginary() {return imaginary;}

  void Add(const Complex& in);
  void Subtract(const Complex& sub_in);
  void Multiply(const Complex& mult_in);
  bool Divide(const Complex& div_in);

  Complex operator + (const Complex& in) const;
  Complex operator - (const Complex& in) const;
  Complex operator * (const Complex& in) const;
  Complex operator / (const Complex& in) const;
};
```cpp
Example of .cpp file
Complex Complex::operator +
    (const Complex& in) const
{
    Complex out(Real() + in.Real(),
        Imaginary() + in.Imaginary());
    return out;
}
```

What can be overloaded?
- Math:
  + - * / % == *= -= /= += %=
- I/O:
  << >>
- Logic:
  & ! ~ <<= >>= ^ ^= & &=
- Indexing:
  () [] ++ -- , > -* new delete new[] delete[]
- You cannot overload . * ? sizeof

Limitations on overloading
- You cannot change the arity of an operator
  += takes two arguments.
  + can take one arguments (as in a = +4)
  + can take two arguments (as in a = 4+4)
  You cannot define a 3 argument +
  You cannot define a 1 argument +=
- You cannot change operator precedence
  - In a*b*c, * happens before +
- General rule: you cannot change how the language is parsed

Functions vs Methods
- You can define an operator to be a method
  - As in the complex example
- Or a function
  - Define it outside of any class
- If you define it as a method
  - Remember the hidden argument
  - += as a method takes one explicit argument (and the hidden argument)
  - Methods have access to protected data
- I use functions when
  - The operator is symmetric (e.g. add)
  - The data is publicly accessible
  - The arguments are of different types
  - There are no side-effects
- Otherwise, I use methods

What about Java?
- Java doesn’t support operator overloading
- Why not?
  - Java is afraid you will misuse/overuse it
  - Programmers have very strong intuitions about operators
    - We all know what + means
    - We all know what <= means
  - Overloading operators with functions that don’t match these intuitions creates confusion
    - Better not to use it than to use it in the wrong case
  - Also, Java doesn’t have templates

When should I overload an operator?
- When a method exactly matches the expectations we have about an operator
- Most common case: I/O
  - If you write a class that is written or read
    - To files
    - To the screen
  - Then you should overload << and/or >>
  - Makes sense for data bases, too
When (continued)

- 2nd most common case: Functors
  - C programs sometimes pass function pointers
  - In C++ it is better to
    - Define a class with a method that performs the function
    - Overload operator () for the class
    - Pass the object

Example Functor

class Add3 {
public:
    int operator() (int value) const
    {return value+3};
};

int main(int argc, char* argv[])
{
    Add3 add3;
    // add3 is now an object I can pass...
    return add3(-2);
}

Note that functors have local memory...

class AddN {
public:
    AddN(int n) : incr(n) []
    {int operator() (int value) const
    {return value+incr};
    protected:
    int incr;
    };

int main(int argc, char* argv[])
{
    AddN add(5);
    return add(-4);
}

When (continued)

- Less common case: Numerics
  - Mathematical objects support math operators
  - Examples:
    - Matrix
    - Tensor
    - Rational
    - Complex
    - Angle
  - Sometimes logic operators, but “iffier”
    - Streams overload !, for example

When not to overload

Company c;
Person p;
cout << c + p;

- What the heck is going on?
  - Did company c hire person p?
  - Did person p buy company c?
  - Are we adding individual and corporate assets?
- Only use overloading when the semantics is obvious!

When not to overload II

- Think about symmetric operators. We expect
  \[(a + b) = (b + a)\]

- If I’m not careful, I can destroy this property
- Only overload reflexive or symmetric operators with reflexive/symmetric methods
When *not* to overload III

- One more rule of thumb about operator overloading…

When in doubt, don’t overload