C++ Programming

Ch. 11 Working with Classes

Spring 2014

Myung-Il Roh

Department of Naval Architecture and Ocean Engineering
Seoul National University

Ch. 11 Ch. 11 Working with Classes

Contents

- ☑ Elements of the Object Model
- ☑ Operator Overloading
- ☑ Introducing Friends
- ☑ Overloaded Operators
- ☑ State Members
- ☑ Automatic Conversions and Type Casts for Classes
- ☑ Summary

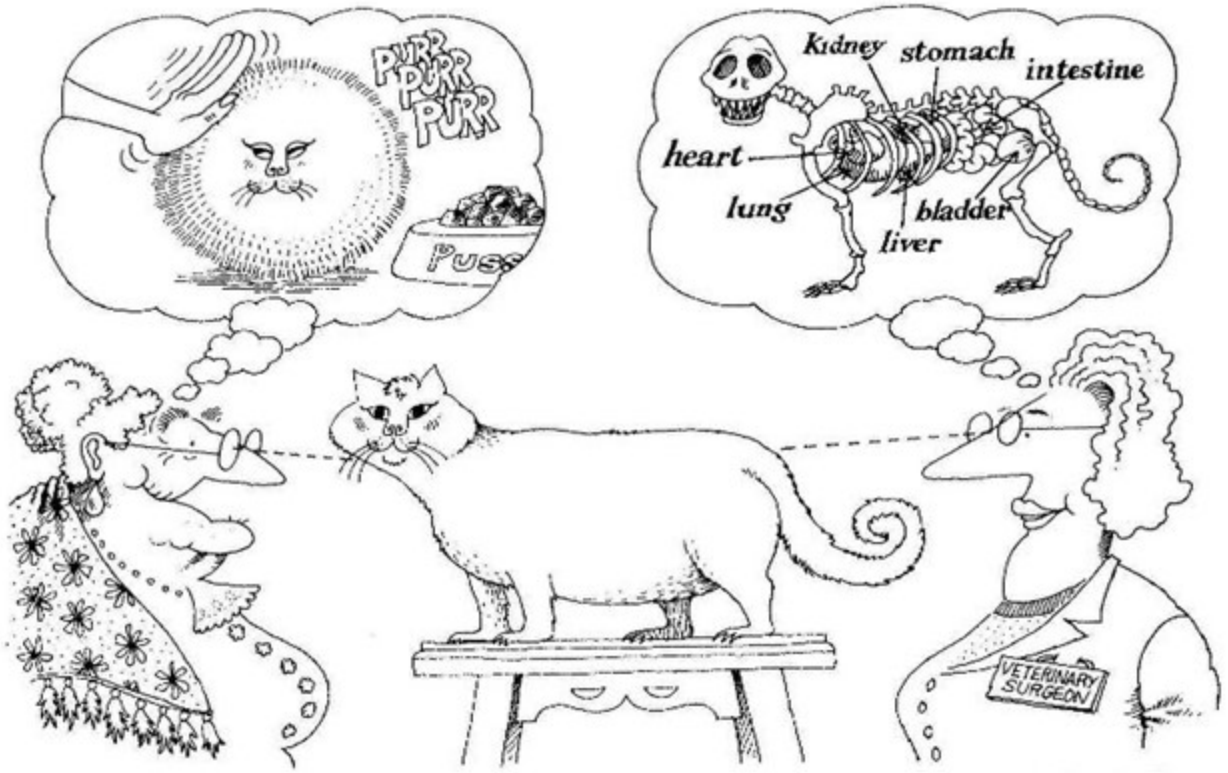
Elements of the Object Model

☑ The following elements become
and fundamentals of object oriented language and programming.

☑ These ideas are applied to various fields such as designing program languages, analyzing problem spaces, and coding.

Elements of the Object Model
- Abstraction

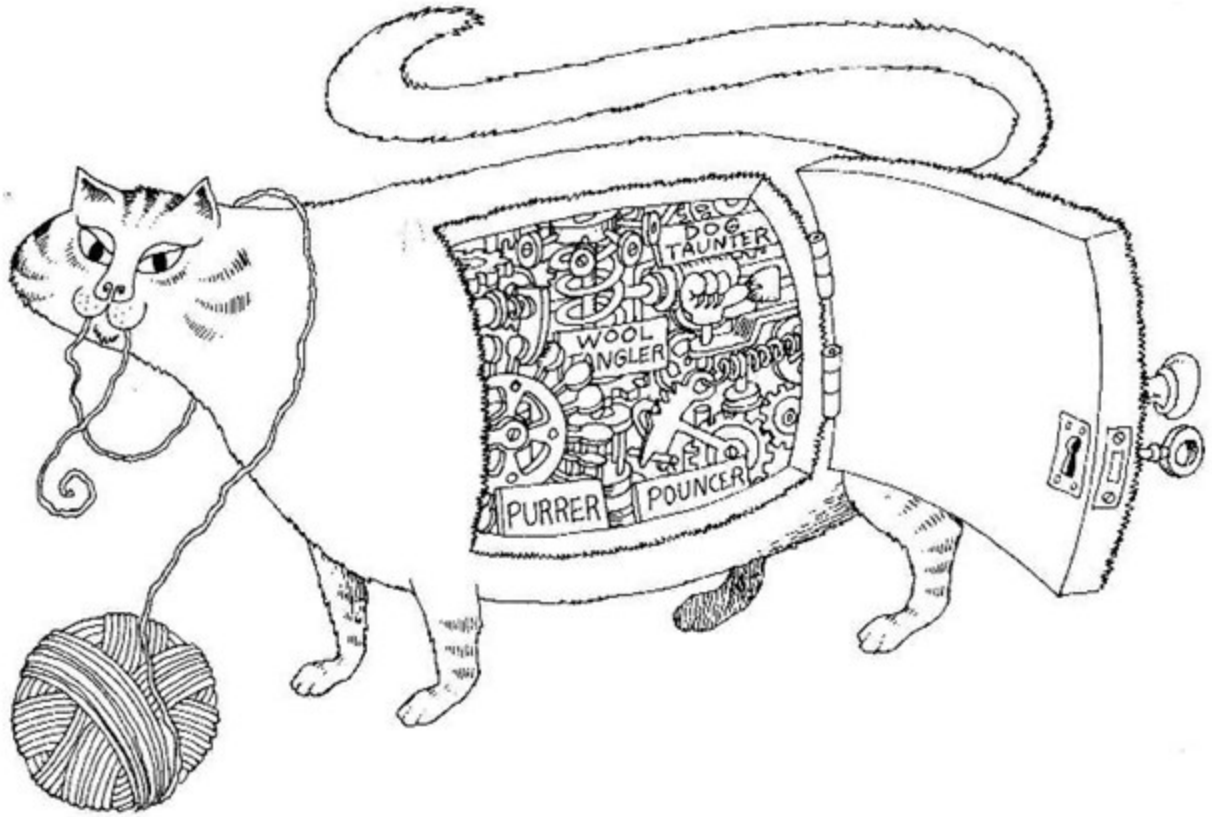
- ☑ Abstraction is from specific instances of those ideas at work. It means, for objects.
- ☑ Abstraction focuses upon the essential characteristics of some object, relative to the perspective of the viewer.



Elements of the Object Model
- Encapsulation

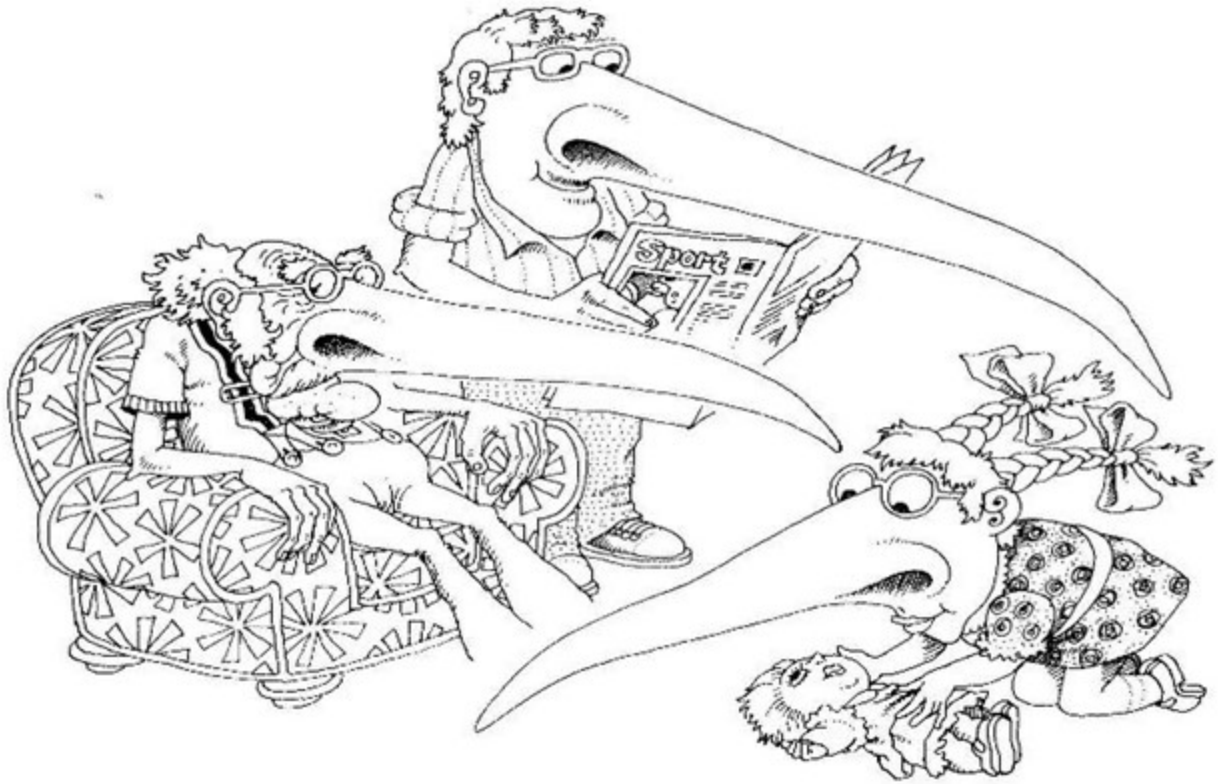
☑ Encapsulation means that
and access to it restricted to members of that class.

☑ Encapsulation of an object.



Elements of the Object Model
- Hierarchy (Inheritance)

- ☑ One well-defined abstraction is used for basis of the other abstractions. Inheritance is when
  , using the same implementation.
- ☑ A subclass may inherit the structure and behavior of its superclass.



Elements of the Object Model
- Polymorphism

☑ Polymorphism is creating
, with the programming context determining which definition is used.

- Ex. Area(radius), Area(width, height)

Operator Overloading (1/2)

- ☑ Example of C++ Polymorphism
  - ■ Adding two arrays element by element
    - ● Ex.

```
for (int i = 0; i < 20; i++)
    evening[i] = sam[i] + janet[i];
```

  - ■ Adding two array objects
    - ● Ex. define arrays with classes, and overload the operator

```
evening = sam + janet;
```

- ☑ We can extend the overloading concept to operators, letting us assign multiple meanings to C++ operators
- ☑ Use of a Special Function Form Called an 'Operator Function'

```
class SP                          // Example of operator overloading
{
    double sale;
public:
    SP operator+(SP&);            // Overloading of '+' operator
};
// Example of using operator overloading
SP sid, sara, district;
district = sid + sara;            // Compiler replaces the statement as below.
                                  // district = sid.operator+(sara);
```

Operator Overloading (2/2)

☑ Vector Class

- Vector class transforms polar coordinates to Cartesian coordinates, and uses Cartesian coordinates for addition.
- We can represent vector addition as C = A + B by using overloaded '+' operator.

```
class Vector
{
private:
    double x, y;
public:
    void show_polar() const;              // Polar Coordinate (53.1 deg, 50 meter)
    void show_vector() const;             // Cartesian Coordinate (x = 30, y = 40)
    ...
    Vector operator +(const Vector &b) const;  // Operator overloading
}

// Example of using operator overloading
Vector A, B, C;
...                       // Initialization
C = A + B;                // Overloaded '+', C = A.operator+(B);
```

Introducing Friends (1/2)

☑ Overloaded operators can be

.

☑ Therefore, for using overloaded operators.

```
// Function definition
Vector Vector::Vector operator*(double n) const
{
    return Vector(n * x, n * y);
}

// Example of using operator overloading
Vector bar = piano * 2.0;     // OK

Vector bar = 2.0 * piano;     // Vector bar = 2.0.operator*(piano) ➡ Not allowed!
```

Introducing Friends (2/2)

- ☑ Definition:

- ☑ Varieties of 'friends'
  - Friend functions
  - Friend classes
  - Friend member functions
- ☑ To do this, we declare a function with keyword 'friend' on the class declaration

```
class Vector {
    ...
public:
    friend Vector operator*(double n, const Vector &a); // Friend declaration
}
// Function definition
Vector operator*(double n, const Vector &a)
{
    return Vector(n * a.x, n * a.y);
}
// Example of using the friend function
Vector bar = 2.0 * piano;  // Now available!
```

Overloaded Operators
- Restrictions (1/2)

☑ Some restrictions for operator overloading

- Operators must be valid C++ operators.

- The overloaded operator must have
  . This prevents us from overloading operators for the standard types.
  - Ex. We can't redefine the minus operator ('-') so that it yields the sum of two integer values instead of their difference.

- We can't use an operator in a manner that violates the syntax rules for the original operator.
  - Ex. We can't overload the binary operator to the unary operator.
    - Such as A+ overloaded operator for A+3 operation.

- We of the operators.
  - Ex. We can't overload '+' operator that has higher precedence that '*' operator.

Overloaded Operators - Restrictions (2/2)

☑ Some restrictions for operator overloading (continued)

- We can't create new operator symbols.
  - Ex. We can't define an operator**() function to denote exponentiation.

- We can't overload the following operators.

sizeof	.	.*	::	?:	typeid
const_cast	dynamic_cast		reinterpret_cast		static_cast

Overloaded Operators
- Overloadable Operators

☑ We can use only member function to overload the following operators (Not allowed for friend function).

Operator	Description
=	Assignment operator
()	Function call operator
[]	Subscripting operator
->	Class member access by pointer operator

☑ Operators that can be overloaded

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

Using a State Member

- ☑ We can use objects selectively by storing the information that describe object's state.
- ☑ For example, we can create both rectangular coordinates and polar coordinates by improving 'Vector' constructor.
- ☑ Also, we can distinguish the state of the object that is rectangular or polar with adding a 'mode'.

```
class Vector
{
private:
    double x, y;
    char mode; // polar? rectangular?
    ......
}

Vector::Vector(double n1, double n2,
char form = 'r')
// default value is
rectangular(Cartesian) coordinates
{
    mode = form;
    if (form == 'r') { ... }
    else if (form == 'p') { ... }
}
```

Automatic Conversions and Type Casts for Classes - Implicit Conversion

☑ Allowance of Implicit Conversion
- C++ constructors that have one argument allow automatic type conversion from the argument type to the class type.

☑ Restriction of Implicit Conversion
- If the class function is declared as explicit 'Stonewt(double lbs);', implicit conversion is not allowed.
- Therefore, in this case 'myCat = 19.6;' is wrong.

```
class Stonewt
{
public:
    Stonewt(double lbs);
    // From double pounds
    ...
};

Stonewt myCat;     // Create an object.
myCat = 19.6;
    // Implicit conversion. Not allowed!
myCat = Stonewt(19.6);
    // Explicit conversion
myCat = (Stonewt)19.6
    // C style type conversion
```

Automatic Conversions and Type Casts for Classes - Conversion Function

- ☑ C++ allowed type conversion from the class type to the default type if we provide a conversion function.
- ☑ Conversion function
  - Must be a class method.
  - Must not specify a return type.
  - Must have no arguments.
- ☑ If just one conversion function is defined, implicit conversion is allowed.
- ☑ In this case, the object is output by using conversion function

```
class Stonewt
{
    // Conversion functions
    operator int() const;
    operator double() const;
};

Stonewt poppins(9, 2.8);
int host = int(poppins);
int hosts = (int)poppins;  // C style

cout << "Poppins: " << int(poppins)
      << " pounds.₩n";
```

Summary

- ☑ Normally, the only way you can access private class members is by using a class method. C++ alleviates that restriction with .
- ☑ C++ extends that describe how particular operators relate to a particular class. An operator function can be a class member function or a friend function.
- ☑ C++ lets us establish to and from class types.

Practice

- ☑ Define Vector class.
- ☑ Define methods for Vector class.
- ☑ Use Vector class.

```
class Vector {                                  // Class declaration
   ...
   public:
      Vector operator+(const Vector &b) const;
}
Vector Vector::operator+(const Vector &b) const {          // Function definition
    double sx, sy;
    sx = x + b.x;
    sy = y + b.y;
    Vector sum = Vector(sx, sy);
    return sum;
}
void main() {                                   // Using operator overloading
    Vector bobo1(20,30);
    Vector bobo2(50,10);
    Vector bobo = bobo1 + bobo2;
}
```