Program Design
-- Composition & Inheritance

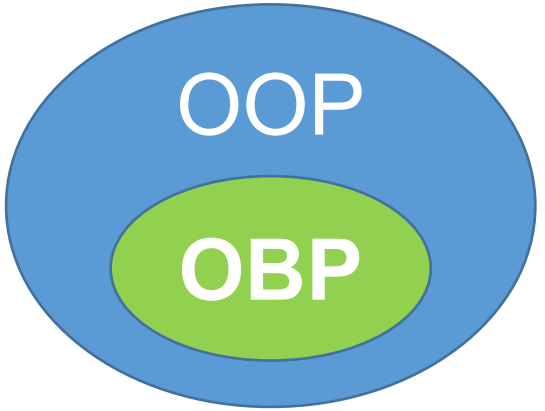
Junjie Cao @ DLUT

Summer 2022

https://github.com/jjcao-school/c

Object-Based vs Object-Oriented

- People think of the world in terms of interacting **objects**
- Object-Based: **Encapsulation** (define composite datatypes using classes: fields + methods)
- Object-Oriented:
  - Encapsulation
  - **Inheritance**: reusing code between related types
  - **Polymorphism**: determining at runtime which functions to call on it based on its type



Composition

- In real-life, complex objects are often built from smaller, simpler objects.
- *has-a* relationship
  - *PC has-a* CPU, a motherboard

```
#include "CPU.h"
#include "Motherboard.h"
#include "RAM.h"

class PersonalComputer{
private:
  CPU m_cCPU;   Motherboard m_cMotherboard;
  RAM m_cRAM;
};
```

Initializing class member variables

```
PersonalComputer::PersonalComputer(int nCPUSpeed,
                                   char *strMotherboardModel,
                                   int nRAMSize)
    : m_cCPU(nCPUSpeed),
      m_cMotherboard(strMotherboardModel),
      m_cRAM(nRAMSize)
{
}
```

Why use composition?

- Each individual class can be kept relatively simple and straightforward, focused on performing one task.
- Each subobject can be self-contained, which makes them reusable.
  - reuse our **CPU**

- each class should be built to accomplish a single task. That task should either be
  - **the storage and manipulation of some kind of data (eg. CPU),**
  - **OR the coordination of subclasses (eg. PersonalComputer).**
  - **Not both.**

Aggregation聚合

- in a **composition**, the complex object "**owns**" all of the subobjects it is composed of.
- When a composition is destroyed, all of the **subobjects are destroyed** as well.
  - If you destroy a PC, you would expect it's RAM and CPU to be destroyed as well.

- An **aggregation** is a specific type of composition where **no ownership** between the complex object and the subobjects is implied.
- When an aggregate is destroyed, the subobjects are **not destroyed**.
  - math department of a school, made up of teachers.
  - The department should be an aggregate.
  - When the department is destroyed, the teachers should still exist independently (they can go get jobs in other departments).

Aggregation

```
class Teacher
{
private:
    string m_strName;
public:
    Teacher(string strName)
        : m_strName(strName)
    {}

    string GetName()
    { return m_strName; }
};
```

```
class Department
{
private:
    Teacher *m_pcTeacher; // This dept h
olds only one teacher

public:
    Department(Teacher *pcTeacher=NU
LL)   : m_pcTeacher(pcTeacher)
    {}
};
```

```
int main(){
    // Create a teacher outside the scope of the Department
    Teacher *pTeacher = new Teacher("Bob"); // create a teacher
    {
        // Create a department and use the constructor parameter to pass
        // the teacher to it.
        Department cDept(pTeacher);

    } // cDept goes out of scope here and is destroyed

    // pTeacher still exists here because cDept did not destroy it
    delete pTeacher;
}
```

Compositions vs. Aggregations

- Compositions:
  - Typically use normal member variables
  - Can use pointer values if the composition class automatically handles allocation/deallocation
  - Responsible for creation/destruction of subclasses
- Aggregations:
  - Typically use pointer variables that point to an object that lives outside the scope of the aggregate class
  - Can use reference values that point to an object that lives outside the scope of the aggregate class
  - Not responsible for creating/destroying subclasses

继承关系 vs. 复合关系

is a vs has a

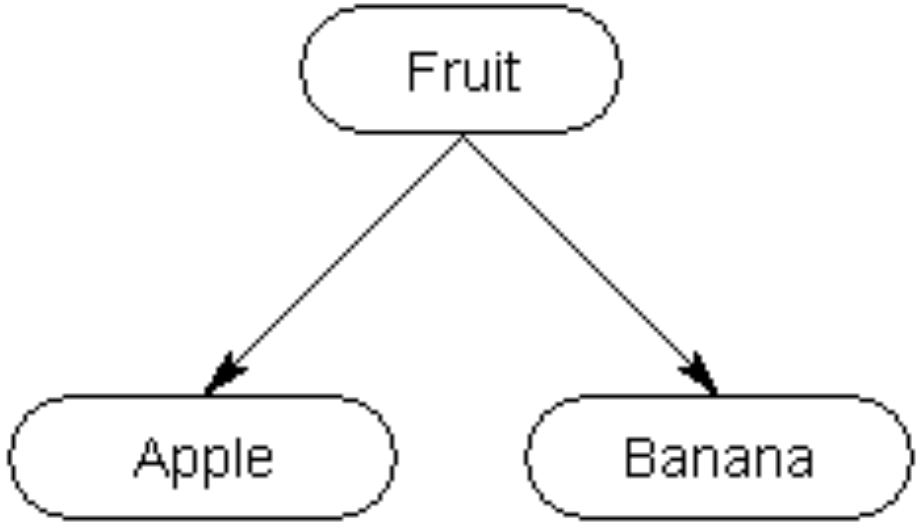
Is-a vs. has-a

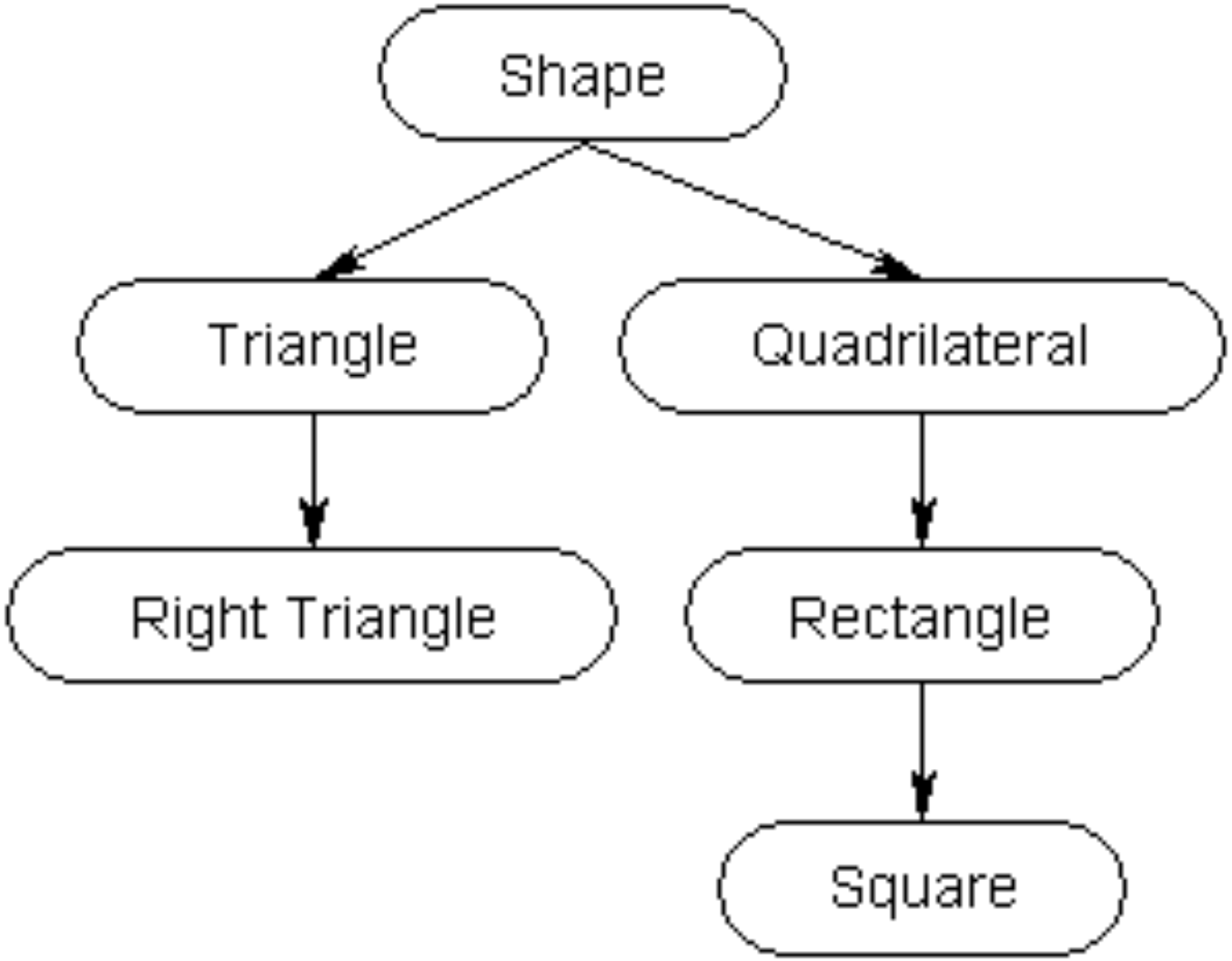
- Every instance of A **is a** B instance. (Car is a Vehicle)
- Every A object **has a** B object. (Vehicle has a string object, i.e. license)

- Is a: inheritance
- Has a: data members

How to construct complex classes

- Has-a
  - Composition
  - Aggregation
- Is-a: Inheritance
  - **parent** or **base**
  - **child** or **derived** object





类之间的两种关系

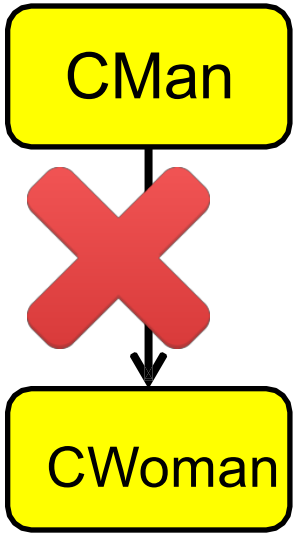
- 继承：“是”关系。
  - 基类A，B是基类A的派生类。
  - 逻辑上要求：“一个B对象也是一个A对象”。

- 复合：“有”关系。
  - 类C中“有”成员变量k，k是类D的对象，则C和D是复合关系
  - 逻辑上要求：“D对象是C对象的固有属性或组成部分”。

继承关系的使用

- 写了一个CMan类代表男人
- 后来又发现需要一个CWoman类来代表女人
- CWoman类和CMan类有共同之处

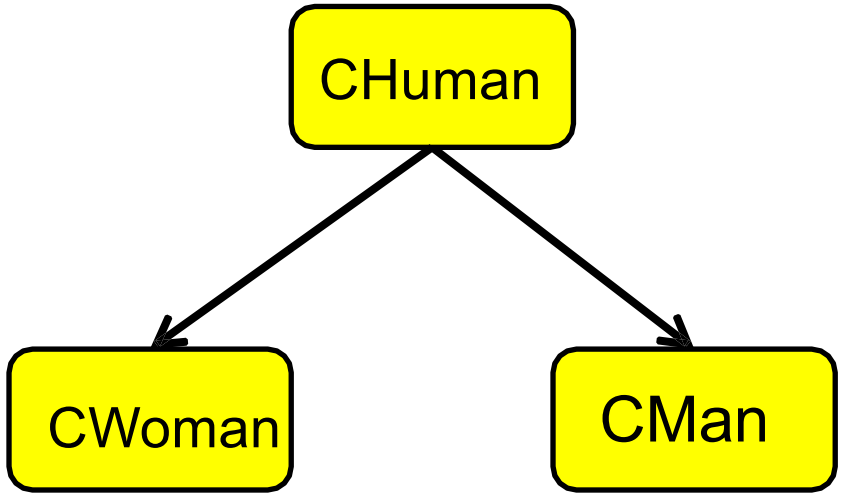
- 就让CWoman类从CMan类派生而来，是否合适？



- 是不合理的!
- 因为“一个女人也是一个男人”从逻辑上不成立!

继承关系的使用

- 好的做法是概括男人和女人共同特点，
- 写一个 CHuman类，代表“人”，
- 然后CMan和CWoman都从CHuman派生。



复合关系的使用

- 几何形体程序中，需要写“ 点 ”类，也需要写“圆”类

```
class CPoint
{
    double x,y;
};
```

```
class CCircle:public CPoint
{
    double r;
};
```

复合关系的使用

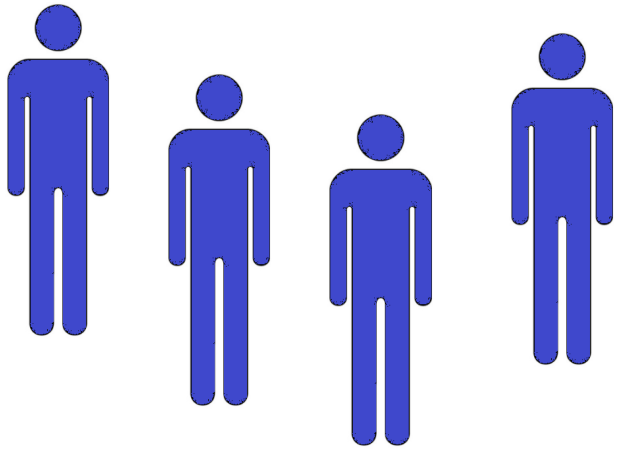
- 几何形体程序中，需要写“点”类，也需要写“圆”类，两者的关系就是复合关系
  - 每一个“圆”对象里都包含(有)一个“点”对象，这个“点”对象就是圆心

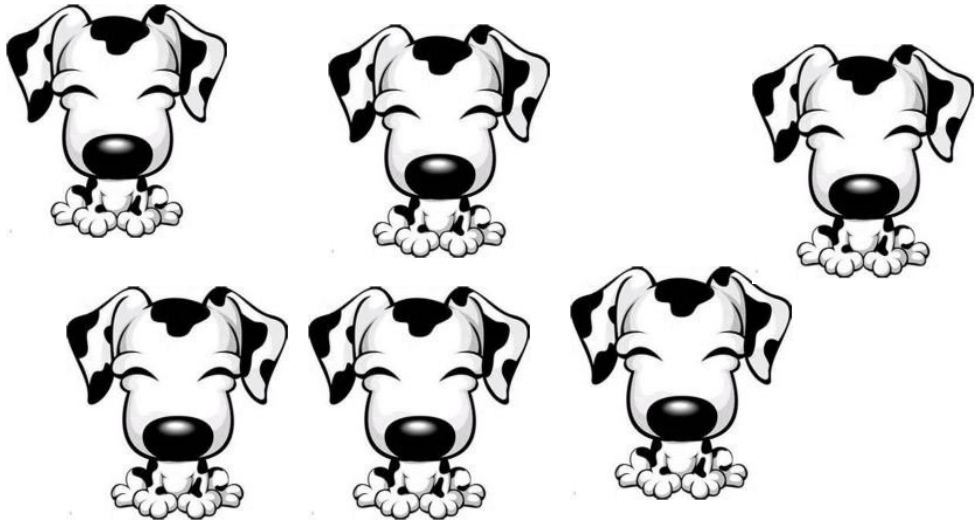
```
class CPoint
{
    double x,y;
    friend class CCircle;
   //便于Ccirle类操作其圆心
};
```

```
class CCircle
{
    double r;
    CPoint center;
};
```

复合关系的使用

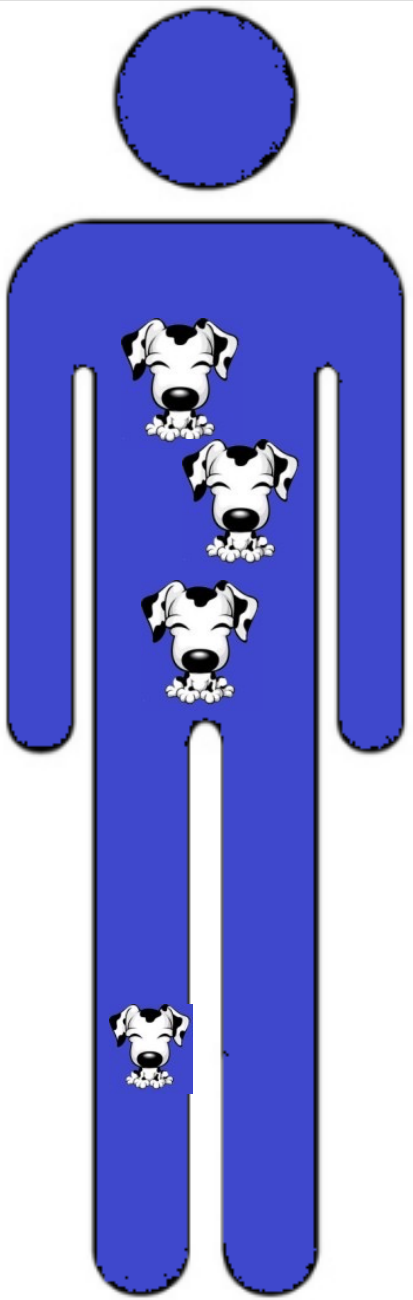
- 如果要写一个小区养狗管理程序，需要写一个“业主”类，还需要写一个“狗”类。
- 而狗是有“主人”的，主人当然是业主(假定狗只有一个主人，但一个业主可以有最多10条狗)



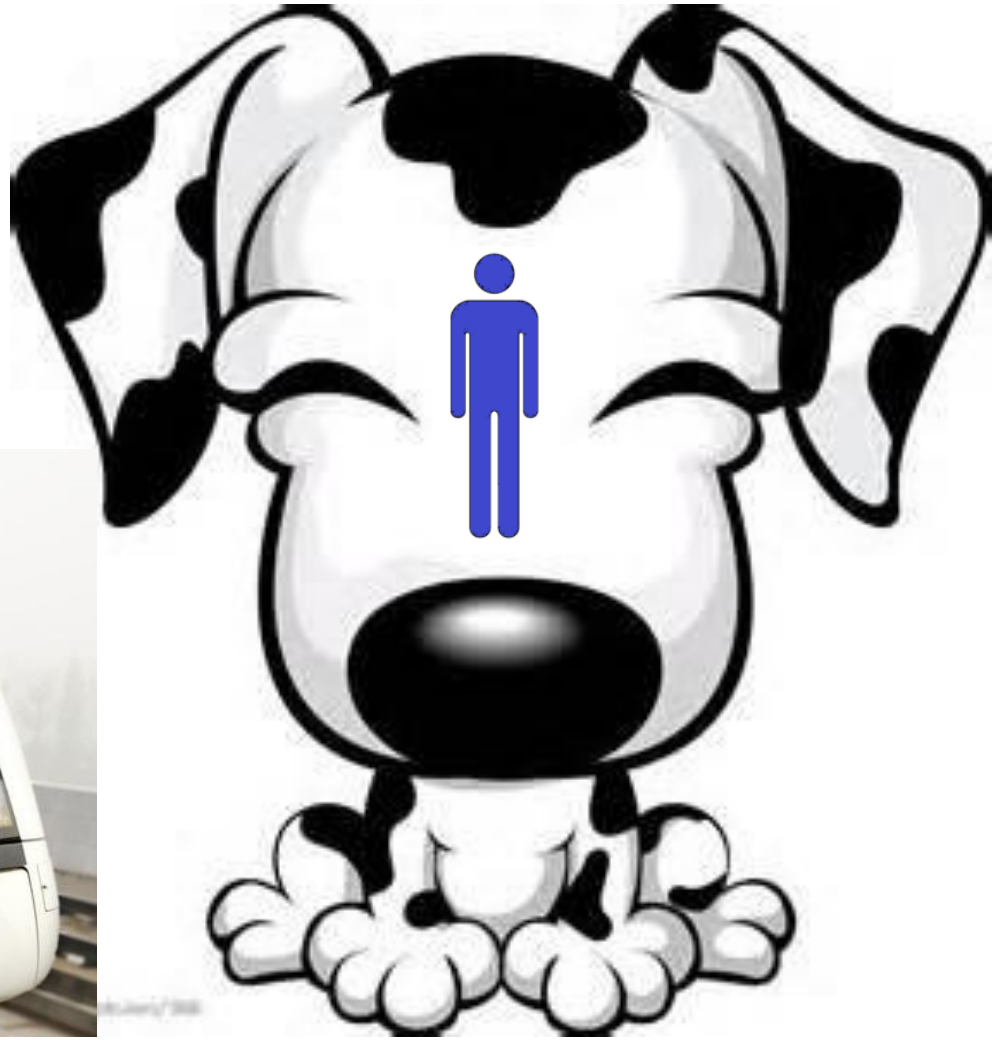


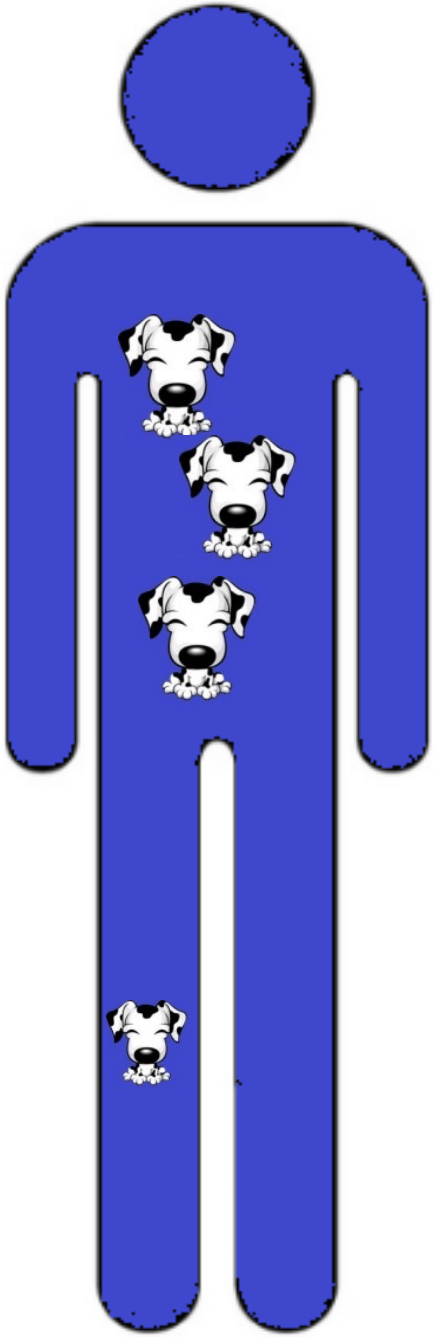
复合关系的使用

```
class CDog;
class CMaster
{
    CDog dogs[10];
};
class CDog
{
    CMaster m;
};
```




和谐号





复合关系的使用

```
class CDog;
class CMaster
{
    CDog dogs[10];
};
class CDog
{
    CMaster m;
};
```

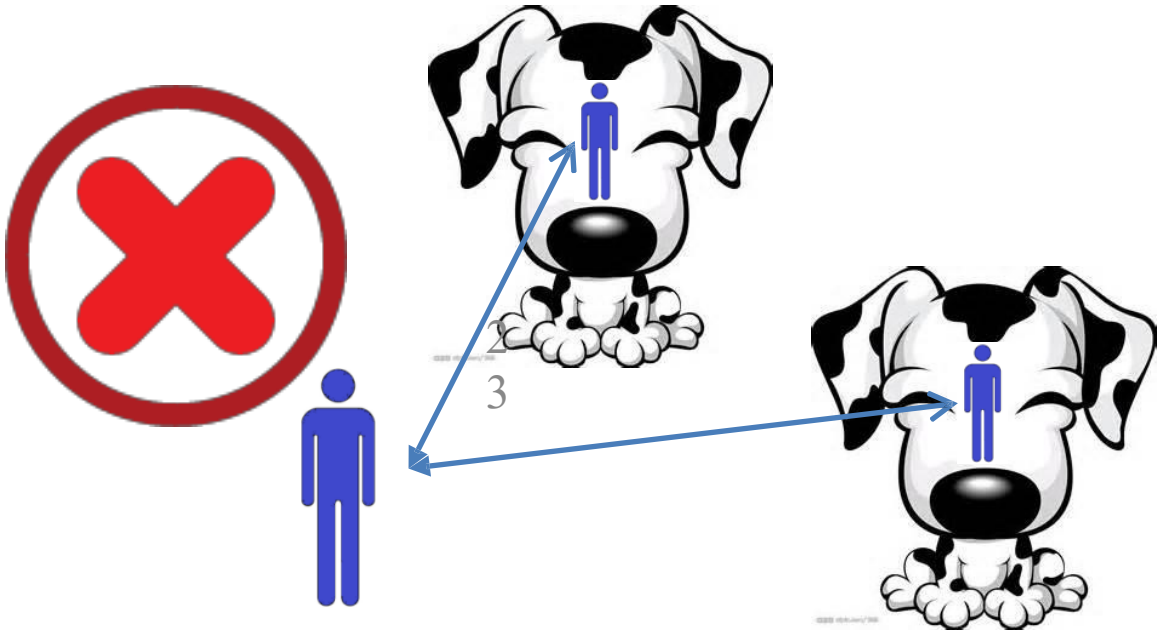
复合关系的使用

- 另一种写法：

  为“狗 ”类设一个“业主”类的成员对象；

  为“业 主 ”类设一个“狗”类的对象指针数组。

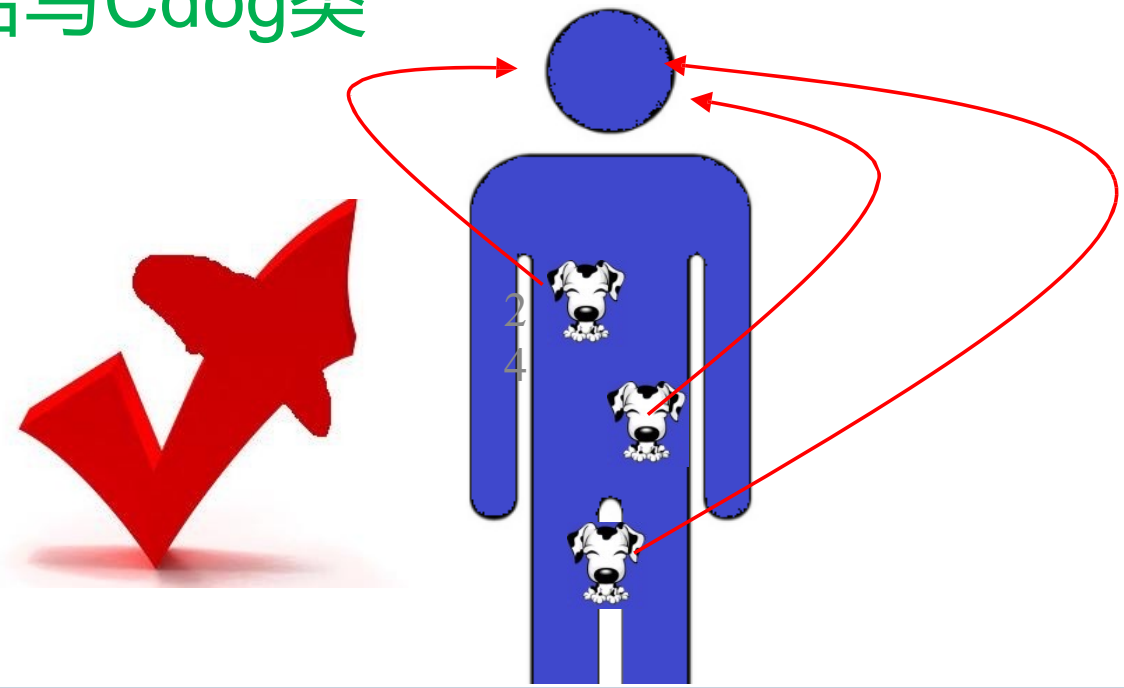
```
class CDog;
class CMaster  {
    CDog * dogs[10];
};
class CDog   {
    CMaster m;
};
```



复合关系的使用

- 凑合的写法：
  为“ 狗 ”类设一个“业主”类的对象指针； 为“ 业 主 ”类设一个“狗”类的对象数组。

```
class CMaster;    //CMaster必须提前声明，不能先
                  //写CMaster类后写Cdog类
class CDog  {
    CMaster * pm;
};
class CMaster {
    CDog dogs[10];
};
```



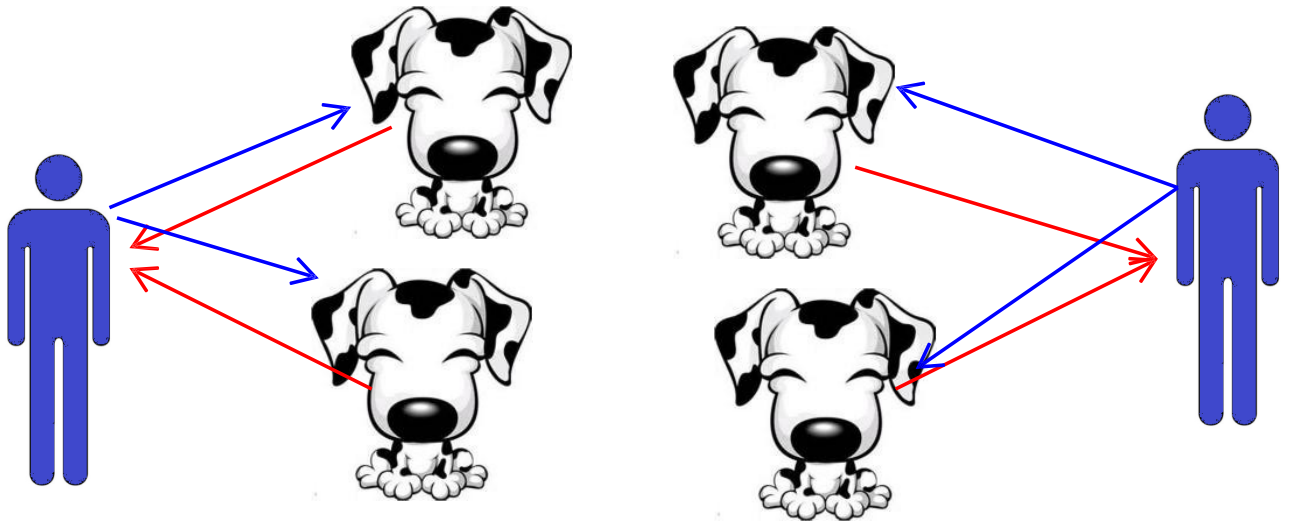
复合关系的使用

- 正确的写法：

  为“狗”类设一个“业主”类的对象指针；

  为“业主”类设一个“狗”类的对象指针数组。

```
class CMaster;     //CMaster必须提前声明，不能先
                                  //写CMaster类后写Cdog类
class CDog  {
    CMaster * pm;
};
class CMaster  {
    CDog * dogs[10];
};
```



Inheritance in C++

Why the need for inheritance in C++?

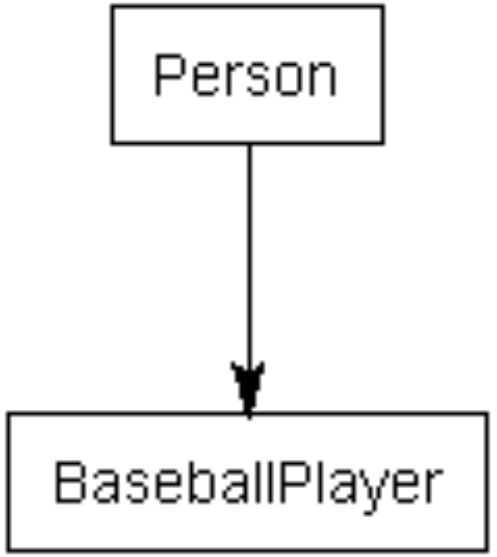
- Reusable
- However, existing code often does not do EXACTLY what you need it to.
  - what if you have a triangle and you need a right triangle?

a) change the existing code to do what you want.
   - no longer be able to use it for its original purpose

b) make a copy of some or all of the existing code and change it to do what we want.
   - maintenance problem: Improvements or bug fixes have to be added to multiple copies of functions

Basic inheritance in C++

```
class Person{
public:
    std::string m_strName;
    int m_nAge;    bool m_bIsMale;

    std::string GetName() { return m_strName; }
    int GetAge() { return m_nAge; }
    bool IsMale() { return m_bIsMale; }

    Person(std::string strName = "", int nAge = 0, bool bIsMale = false)   : m_str
Name(strName), m_nAge(nAge), m_bIsMale(bIsMale)
    { }  };
```

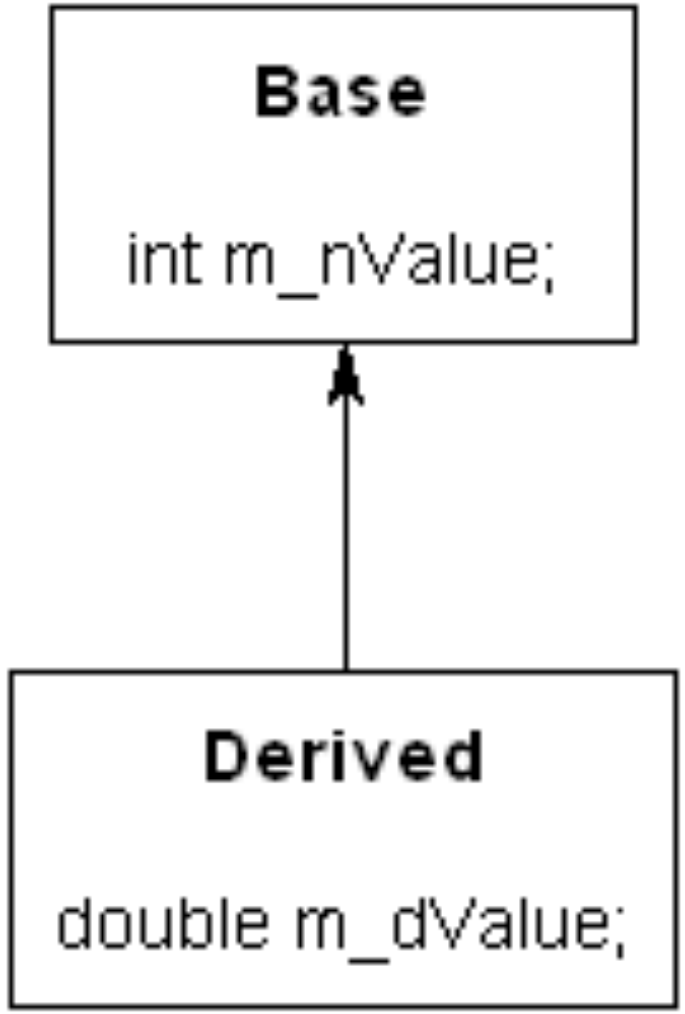



```
class BaseballPlayer : public Person{
public:
    double m_dBattingAverage;
    int m_nHomeRuns;

    BaseballPlayer(double dBattingAverage = 0.0, int nHomeRuns = 0) : m_dBat
tingAverage(dBattingAverage), m_nHomeRuns(nHomeRuns)
    { }
};
```

- BaseballPlayer cJoe;
- cJoe.m_strName = "Joe";
- std::cout << cJoe.GetName() << std::endl;

Order of construction of derived classes

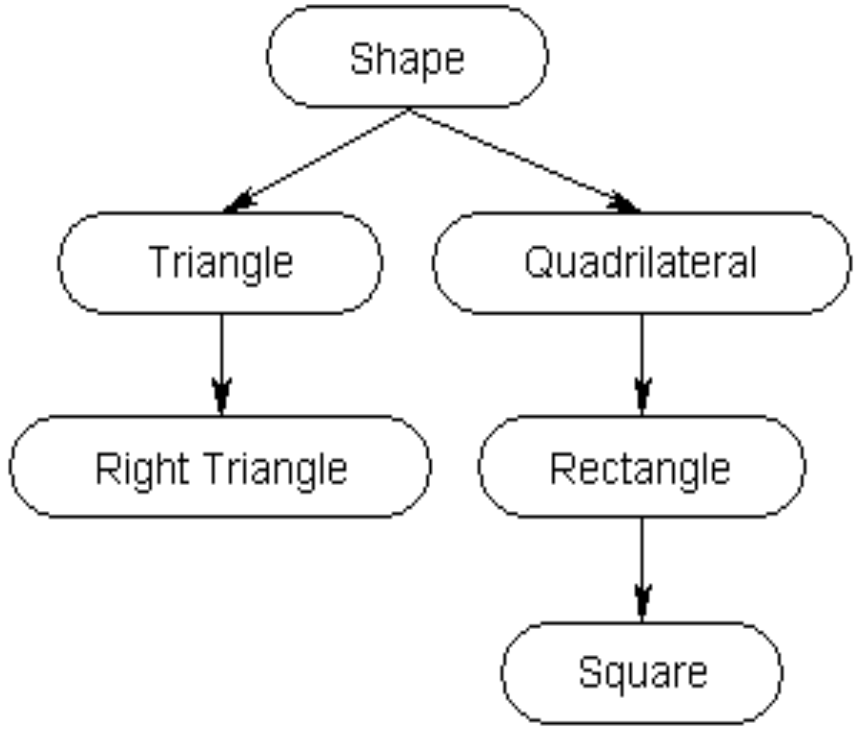
```
class Base{
public:  int m_nValue;
    Base(int nValue=0)
        : m_nValue(nValue){}
};

class Derived: public Base{
public:  double m_dValue;
    Derived(double dValue=0.0)
        : m_dValue(dValue){}
};
```



what happens when we instantiate a derived class

```
int main()
{
    Derived cDerived;

    return 0;
}
```

```
Base(int nValue=0): m_nValue(nValue){
      cout << "Base" << endl;}

Derived(double dValue=0.0): m_dValue(dValue){
      cout << "Derived" << endl;}
```




```
Derived cDerived;
Base
Derived
```

C++ always constructs the “first” or “most base” class first. It then walks through the inheritance tree in order and constructs each suc cessive derived class.

what actually happens when cDerived is instantiated?

1. Memory for cDerived is set aside (enough for both the Base and Derived portions).
2. The appropriate Derived constructor is called
3. **The Base object is constructed first using the appropriate Base constructor**
4. The initialization list initializes variables
5. The body of the constructor executes
6. Control is returned to the caller

Initializing base class members

```
class Derived: public Base
{
public:
    double m_dValue;

    Derived(double dValue=0.0, int nValue=0)
        // does not work
        : m_dValue(dValue), m_nValue(nValue)
    {
    }
};
```

what would happen if m_nValue were const

Initializing base class members

```
class Derived: public Base{
public:
   double m_dValue;

   Derived(double dValue=0.0, int nValue=0)
      : Base(nValue), // Call Base(int) constructor with value nValue!
         m_dValue(dValue)
   {
   }
};
Derived cDerived(1.3, 5); // use Derived(double) cons
tructor
```

Initializing base class members

1. Memory for cDerived is allocated.
2. The Derived(double, int) constructor is called, where dValue = 1.3, and nValue = 5
3. The compiler looks to see if we've asked for a particular Base class constructor. We have! So it calls Base(int) with nValue = 5.
4. The base class constructor initialization list sets m_nValue to 5
5. The base class constructor body executes
6. The base class constructor returns
7. The derived class constuctor initialization list sets m_dValue to 1.3
8. The derived class constructor body executes
9. The derived class constructor returns

```
Person(std::string strName = "", int nAge = 0, bool bIsMale = false) : m_strName(str
Name), m_nAge(nAge), m_bIsMale(bIsMale)
    {}

class BaseballPlayer : public Person{
public:
    double m_dBattingAverage;    int m_nHomeRuns;

    BaseballPlayer(double dBattingAverage = 0.0, int nHomeRuns = 0)
    : m_dBattingAverage(dBattingAverage), m_nHomeRuns(nHomeRuns)
    { }
```

it makes sense to give our BaseballPlayer a name and age when we create th em, How?

```
BaseballPlayer(std::string strName = "", int nAge = 0,
                                                bool bIsMale = false,
    double dBattingAverage = 0.0, int nHomeRuns = 0)
    : Person(strName, nAge, bIsMale),
      m_dBattingAverage(dBattingAverage),
            m_nHomeRuns(nHomeRuns)
  {}

BaseballPlayer cPlayer("Pedro Cerrano", 32, true, 0.3
42, 42);
```

destructor

- When a derived class is destroyed, each destructor is called in the reverse order of construction.

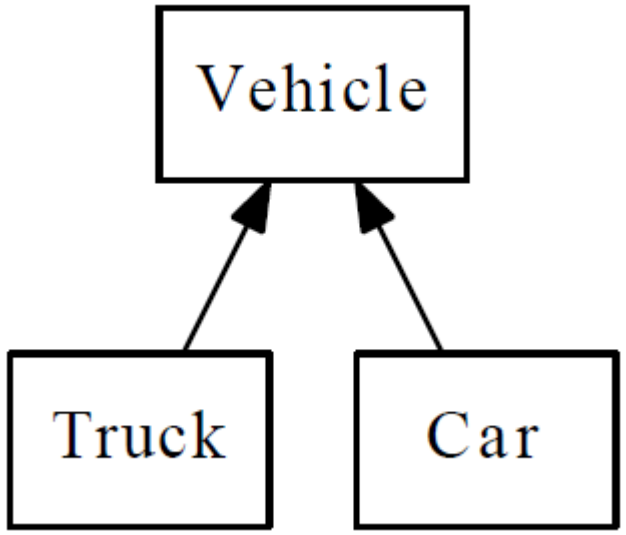
protected

```
class Base
{
public:
    int m_nPublic; // can be accessed by anybody
private:
    int m_nPrivate; // can only be accessed by Base member functions (but not derived classes)
protected:
    int m_nProtected; // can be accessed by Base member functions, or derived classes.
};
```

inheritance type – a complex topic

```
// Inherit from Base publicly
class Pub: public Base{};

// Inherit from Base privately
class Pri: private Base{};

// Inherit from Base protectedly
class Pro: protected Base{};

class Def: Base // Defaults to private inheritance{};
```

Inheritance in Python



Common base class in Python

- The object class:
  - The most basic type of class is an object, which generally all other classes inherit from as their parent.

```
class Dog(object):
    pass

# In Python 3, this is the same as:
class Dog:
    pass
```

class in Python

```
if __name__ == '__main__':
   miles = Dog("Miles", 4, "Jack Russell Terrier")
   buddy = Dog("Buddy", 9, "Dachshund")
   jack = Dog("Jack", 3, "Bulldog")
   jim = Dog("Jim", 5, "Bulldog")

   buddy.speak("Yap") #'Buddy says Yap'
   jack.speak("Woof")#'Jack says Woof'
   jim.speak("Woof")#'jim says Woof'

            class Dog:
               def __init__(self, name, age, breed):
                  self.name = name;self.age = age
                  self.breed = breed
               def speak(self, sth):
                  print("{} says {}".format(self.name, sth))
```

Inheritance in Python

```
# how to implment these? see dogs2.py
buddy.speak() #'Buddy says Yap'
jack.speak()#'Jack says Woof'
jim.speak()#'jim says Woof'
```

```
class Dog:
    def __init__(self, name, age):
        self.name = name;self.age = age
    def speak(self, sth):
        print("{} says {}".format(self.name, sth))
class JackRussellTerrier(Dog):
    def speak(self, sound="Arf"):
        print("{} says {}".format(self.name, sound))
class Dachshund(Dog):
    pass
class Bulldog(Dog):
    pass
```

Inheritance in Python

```
if __name__ == '__main__':
    miles = JackRussellTerrier("Miles", 4)
    buddy = Dachshund("Buddy", 9)
    jack = Bulldog("Jack", 3)
    jim = Bulldog("Jim", 5)

    miles.speak()#'Miles says Arf'
    miles.speak("Grrr")#'Miles says Grrr'
```

Inheritance in Python

```
class Dog:
   def __init__(self, name, age):
      self.name = name;self.age = age
   def speak(self, sth):
      print("{} says {}".format(self.name, sth))
class JackRussellTerrier(Dog):
   def speak(self, sound="Arf"):
      print("{} says {}".format(self.name, sound))
class Dachshund(Dog):
   def speak(self, sound="Yap"):
      super().speak(sound)
class Bulldog(Dog):
   def speak(self, sound="Woof"):
      super().speak(sound)
```

Inheritance in Python

```
if __name__ == '__main__':
    miles = JackRussellTerrier("Miles", 4)
    buddy = Dachshund("Buddy", 9)
    jack = Bulldog("Jack", 3)
    jim = Bulldog("Jim", 5)

    miles.speak()#'Miles says Arf'
    miles.speak("Grrr")#'Miles says Grrr’

    buddy.speak() #'Buddy says Yap'
    jack.speak()#'Jack says Woof'
    jim.speak()#'jim says Woof'
```