C++ Program Design
-- Review of 1st Part

Junjie Cao @ DLUT

Summer 2018

http://jjcao.github.io/cPlusPlus

面向对象，Class，封装

Procedural programming过程式编程

- Such as C

1. 分解为若干任务或者子任务
2. 为每个任务写一个函数
3. Main函数调用它们，按顺序Sequence执行

- Sequence是明显的，但是hierarchy等级/层次 关系是模糊的ambiguity.
- 难以组织大量的函数，例如大公司再扁平，也要有层次结构：管理层/中层/普通员工

面向过程 vs 面向对象

- 人类在和世界互动的时候，习惯看/想到的概念/名词/物体
  - **属性+行为：Properties + behaviors** (二者不可分割)

- 面向过程把二者分开properties (data) & behaviors (functions)
  - 不利于直观的表示现实世界
  - 需要开发者将二者用合适的方式连接起来，管理起来，增加开发者的负担

- OOP给我们提供了设计object、世界的能力，是一种管理复杂性complexity的工具。
  - 容易写&理解
  - 更加容易重用，扩展，维护：higher degree of code-reusability

从客观事物抽象出类: 属性(名词)+行为(动词)

```
class DateClass {
    int m_month;     int m_day;     int m_year;
public:
    void setDate(int month, int day, int year) {
        m_month = month;    m_day = day;     m_year = year;
    }
    void print() {
        std::cout << m_month << "/" << m_day << "/" << m_year;
    }};
```

Mixing access specifiers

```
void main(){
    DateClass date;
    date.setDate(10, 14, 2020); // okay
    date.print(); // okay
    date.m_year = 1984; // error: ‘DataClass::m_year’: cannot access private member declared in class ‘DataClass ‘
}
```

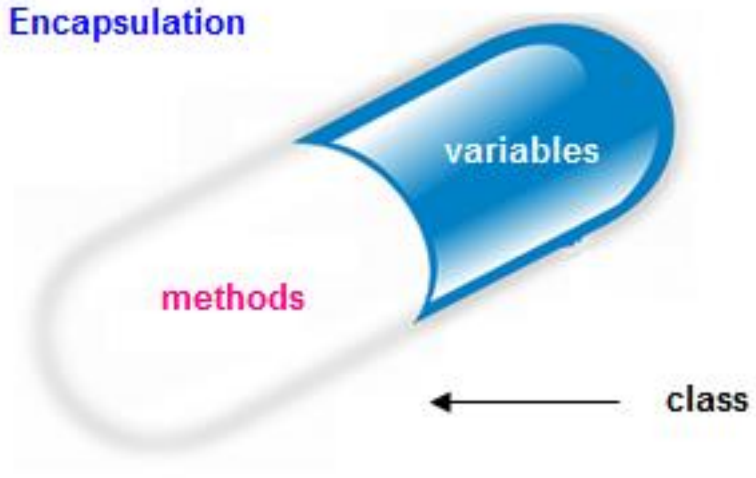
public interface公有接口: setDate(), print()

Rule: 除非有强有力的理由，否则成员都应该是私有的. 所以默认是私有的。

Why make members private?
Encapsulation
封装

Encapsulation封装

- **又称information hiding（信息隐藏）**：用户看不到内部实现细节



- 只能通过**共有接口**访问你提供的对象。
- 这样，用户不必关心内部实现，分工协作能极大的提高社会生产力。

Benefit: encapsulated classes are easier to use and reduce the complexity of your programs

- only need to know public members to use the class
- It doesn't matter how the class was implemented internally
  - a class holding a list of names could have been implemented using a dynamic array of C-style strings, std::array, std::vector, std::map, std::list, or one of many other data structures.
- dramatically reduces the complexity of your programs, and also reduces mistakes
- Imagine how much more complicated C++ would be if you had to understand how std::string, std::vector, or std::cout were implemented in order to use them!

Benefit: encapsulated classes help protect your data and prevent misuse

- two variables have an intrinsic connection

```
class MyString{
    char *m_string; // we'll dynamically allocate our string here
    int m_length; // we need to keep track of the string length
};
```

- If m_length were public, anybody could change the length of the string without changing m_string (or vice-versa) => inconsistent state

- use public member functions can ensure that m_length and m_string are always set appropriately

Benefit: encapsulated classes help protect your data and prevent misuse

- two variables have an intrinsic connection

```
class IntArray{
public:
    int m_array[10];
};

IntArray array;
array.m_array[16] = 2; // invalid array index, now we overwrote memory that we don't own
```

- How to solve this?

```cpp
class IntArray
{
private:
    int m_array[10]; // user can not access this directly any more
public:
    void setValue(int index, int value){
        // If the index is invalid, do nothing
        if (index < 0 || index >= 10)
            return;

        m_array[index] = value;
    }
};
```

Benefit: encapsulated classes are easier to change

```
class Something{
public:
    int m_value1;
    int m_value2;
    int m_value3;
};

int main(){
    Something something;
    something.m_value1 = 5;
    std::cout << something.m_value1 << '\n';
};
```

Nothing can be changed

```
class Something{
private:
    int m_value1;    int m_value2;    int m_value3;

public:
    void setValue1(int value) { m_value1 = value; }
    int getValue1() { return m_value1; }
};

int main(){
    Something something;
    something.setValue1(5);
    std::cout << something.getValue1() << '\n';
```

Same printing result, but chance to change member data

Benefit: encapsulated classes are easier to change

```
class Something{
private:
    int m_value[3]; // note: we changed the implementation of this class!

public:
    // We have to update any member functions to reflect the new implementation
    void setValue1(int value) { m_value[0] = value; }
    int getValue1() { return m_value[0]; }
};
something.setValue1(5);
std::cout << something.getValue1() << '\n';
```

- Program using the code continues to work without any changes!
- They probably wouldn't even notice!

Benefit: encapsulated classes are easier to debug

- Often when a program does not work correctly, it is because one of our member variables has an incorrect value.

- If everyone is able to access the variable directly, tracking down which piece of code modified the variable can be difficult.

- However, if everybody has to call the same public function to modify a value, then you can simply breakpoint that function and watch as each caller changes the value until you see where it goes wrong.

构造与析构

封装(private) =》如何初始化这些成员变量

```
class Foo{
public:
    int m_x;
    int m_y;
};
```

```
int main(){
    Foo foo1 = { 4, 5 }; // initialization list
    Foo foo2 { 6, 7 }; // uniform initialization (C++11)
    return 0;
}
```

However, as soon as we make any member variables private, we're no longer able to initialize classes in this way.

It does make sense: if you can't directly access a variable (because it's private), you shouldn't be able to directly initialize it. => constructor

Constructors

1. 是特殊的成员函数
2. 当一个class的对象object被实例化instantiated的时候，自动地被调用
3. 同名（同大小写）
4. 没有返回值类型（not even void）

Default constructors默认构造函数

- 没有参数或者参数都有默认值

```
class Fraction{
    int m_numerator;    int m_denominator;
public:
    Fraction(){ // default constructor
        m_numerator = 0; m_denominator = 1;
    }
    int getNumerator() { return m_numerator; }
};
Fraction frac; // Since no arguments, calls Fraction() default constructor
std::cout << frac.getNumerator() << "/" << frac.getDenominator() << '\n';
```

减少不必要的构造函数

```
Fraction() { // default constructor
    m_numerator = 0;    m_denominator = 1;
}

Fraction(int numerator, int denominator=1) {
    assert(denominator != 0);
    m_numerator = numerator;
    m_denominator = denominator;
}
```

```
Fraction(int numerator=0, int denominator=1) {
    assert(denominator != 0);
    m_numerator = numerator;
    m_denominator = denominator;
}
```

封装的好处：防止数据被滥用，维持相关数据、操作的合法性和一致性。

- if you do have other non-default constructors in your class, but no default constructor, C++ will not create an empty default constructor for you

```
class Date{
private:    int m_year;    int m_month;    int m_day;
public:
    Date(int year, int month, int day){ // not a default constructor
        m_year = year;    m_month = month;    m_day = day; }
    // No default constructor provided
};

Date date; // error: Can't instantiate object because default constructor doesn't exist
Date today(2020, 10, 14); // today is initialized to Oct 14th, 2020
```

Quiz time - Write a class named Ball.

- The following sample program should compile:

```
Ball def; def.print();
Ball blue("blue"); blue.print();
Ball twenty(20.0); twenty.print();
Ball blueTwenty("blue", 20.0); blueTwenty.print();
```

color: black, radius: 10
color: blue, radius: 10
color: black, radius: 20
color: blue, radius: 20

怎么办？

```
class Something
{
const int m_value;
public:
Something()
{//error C2789: 'Something::m_value': an object of const-qualified type must be initialized

m_value = 1; //error C2166: l-value specifies const object
}

};             // it is assignment赋值, not initialization初始化
```

member initializer lists成员初始化列表

```
class Something{
private:
    const int m_value;
public:
    Something(){
        m_value = 1; // error: const vars can not be assigned to
    }
};

const int m_value; // error: const vars must be initialized with a value
m_value = 5; //  error: const vars can not be assigned to
```

Member initializer lists

```
class Something{
    int m_value1;
    double m_value2;
    char m_value3;
public:
    Something() : m_value1(1), m_value2(2.2), m_value3('c')
     // directly initialize our member variables
    {
    // No need for assignment here
    }
```

Overlapping and delegating constructors

```
class Foo
{
public:
    Foo() {
        // code to do A
    }

    Foo(int value) {
        // code to do A
        // code to do B
    }
};
```

Using a separate function

```
class Foo {
private:
    void DoA() { // code to do A }

public:
    Foo() {  DoA();   }

    Foo(int nValue) {
        DoA();
        // code to do B
    }
};
```

code duplication is kept to a minimum.

you may find yourself in the situation where you want to write a member function to re-initialize a class back to default values.

```
class Foo{
public:
    Foo(){ Init(); }

    Foo(int value){ Init();
        // do something with value
    }

    void Init() {   // code to init Foo }
};
```

Delegating constructors in C++11

```
class Employee{
private:
    int m_id;    std::string m_name;
public:
    Employee(int id, std::string name):
        m_id(id), m_name(name) {   }
    // All three of the following constructors use delegating constructors to minimize r
edundant code
    Employee() : Employee(0, ””) { }
    Employee(int id) : Employee(id, ””) { }
    Employee(std::string name) : Employee(0, name) { }
};
```

a hidden pointer named “this”

- “When a member function is called, how does C++ keep track of which object it was called on?”
- `simple.setID(2);`
- `setID(&simple, 2); // note that simple has been changed from an object prefix to a function argument!`
- `void setID(int id) { m_id = id; }`
- `void setID(Simple* const this, int id) { this->m_id = id; }`

Chaining objects

```
class Calc{
private:  int m_value;

public:
    Calc() { m_value = 0; }

    void add(int value) { m_value += value; }
    void sub(int value) { m_value -= value; }
    void mult(int value) { m_value *= value; }

    int getValue() { return m_value; }
};
```

Chaining objects

- `Calc calc;`
- `calc.add(5); // returns void`
- `calc.sub(3); // returns void`
- `calc.mult(4); // returns void`
- `std::cout << calc.getValue() << '\n';`

- `calc.add(5).sub(3).mult(4);`
- `Calc& add(int value) { m_value += value; return *this; }`
- `Calc& sub(int value) { m_value -= value; return *this; }`
- `Calc& mult(int value) { m_value *= value; return *this; }`

Const class objects and member functions

Static member variables

Friend functions and classes
不对称，不传递

Operator

Overloading operators for operands of different types

```
class Cents{
...
// add Cents + int using a friend function
friend Cents operator+(const Cents &c1, int value);
// add int + Cents using a friend function
friend Cents operator+(int value, const Cents &c1);
...
};
```

Overloading the I/O operators

```
class Point{
    double m_x, m_y, m_z;
public:
    Point(double x, double y, double z): m_x(x)… { }

    double getX() { return m_x; } …};


      Point point(5.0, 6.0, 7.0);
      std::cout << "Point(" << point.getX() << ", " <<
          point.getY() << ", " <<
          point.getZ() << ")";
```

```cpp
class Point{
public:
     void print(){
     std::cout << "Point(" << m_x << ", " << m_y << ", " << m_z << ")";
     }
};
```

- `std::cout << "My point is: ";`
- `point.print();`
- `std::cout << " in Cartesian space.\n";`

```cpp
cout << "My point is: " << point << " in Cartesian space.\n";
```

Overloading重载 operator<<

- std::cout << point.
- operator操作符 is <<
- operands操作数?
  - 左操作数std::cout// std::ostream is the type for object std::cout
  - 右操作数point

- `friend std::ostream& operator<< (std::ostream &out, const Point &point);`

```
class Point{
    friend std::ostream& operator<< (std::ostream &out, const Point &point);
};

std::ostream& operator<< (std::ostream &out, const Point &point)
{
    out << "Point(" << point.m_x << ", " << point.m_y << ", " << point.m_z << ")";

    return out;
}
```

The trickiest part here is the return type.

- `friend Cents operator+(const Cents &c1, const Cents &c2);`
- `friend std::ostream& operator<< (std::ostream &out, const Point &point);`

- 需要把输出命令 “chain”串起来, 如 std::cout << point << std::endl;
  - If returning void => void << std::endl;
  - If returning ostream& =>std::cout << std::endl;

为了能把+串起来用，这样可以不？

```
Cents& operator+(const Cents &c1, const Cents &c2)
{
    Cents c(c1.m_cents + c2.m_cents);
    return c;
}
```

基于成员函数的操作符重载

1. 必须是 **左操作数 left operand** 的成员函数.
2. 左操作数成为该成员函数的隐式的***this对象**
3. 所有其他操作数成为该成员函数的参数

```
class Cents{
public:
   Cents operator+(int value); // Overload Cents + int
};

Cents Cents::operator+(int value){
    return Cents(m_cents + value);
}
```

overload an operator as a friend or a member?

- **某些操作符不能用普通函数（友元）重载**
  - The assignment (=), subscript ([]), function call (()), and member selection (->) operators must be overloaded as member functions, because the language requires them to be.
- **某些操作符不能被重载成成员函数**
  - we are not able to overloaded operator<< as a member function.
  - Because the overloaded operator must be added as a member of the left operand.

重载类型转换操作符：typecasts

```
class Cents{
public:
    // Overloaded int cast
    operator int() { return m_cents; }
};
```

```
class Dollars{
   int m_dollars;
public:
    Dollars(int dollars=0)  {
        m_dollars = dollars;
    }
     // Allow us to convert Dollars into Cents
     operator Cents() { return Cents(m_dollars * 100); }
};
```

```
void printCents(Cents cents)
{
    std::cout << cents; // cents will be implicitly cast to an int here
}

Dollars dollars(9);
printCents(dollars); // dollars will be implicitly cast to a Cents here
```

什么时候需要写一个typecasts，什么时候需要写一个printCents(Dollars doll)?

重载赋值操作符assignment operator

- **Assignment vs Copy constructor**
  - The purpose of the copy constructor and the assignment operator are **almost equivalent** -- both copy one object to another.
  - However, the copy constructor **initializes new** objects,
  - whereas the assignment operator **replaces the contents of existing** objects.

- Overloading the assignment operator (operator=) is fairly straightforward
- …

The copy constructor

- `Fraction fiveThirds(5, 3); // Direct initialize a Fraction, calls Fraction(int, int) constructor`
- `Fraction fCopy(fiveThirds); // Direct initialize -- with what constructor?`
- `std::cout << fCopy < '\n';`

- 如果我们不提供，C++会生成一个public的，实现**Memberwise initialization逐个成员的初始化**

Default assignment operator

- 和其它操作符不一样，如果我们不提供，编译器会提供一个默认的
- 这个默认的，实现逐个成员的赋值**memberwise assignment，和**默认复制构造函数做的memberwise initialization类似。

```
Fraction& operator= (const Fraction &fra):m_numerator(fra.m_nume
rator), m_denominator(fra.m_denominator){} //这么写可以不?
```

- Just like other constructors and operators, you can prevent assignments from being made by making your assignment operator private or using the delete keyword:

```
// Overloaded assignment
Fraction& operator= (const Fraction &fraction) = delete; // no c
opies through assignment!
```

Issues due to self-assignment

```
int main(){
    MyString alex("Alex", 5); // Meet Alex
    alex = alex; // Alex is himself
    std::cout << alex; // Say your name, Alex
```

```
alex = alex; // Alex is himself
```

```
// A simplistic implementation of operator= (do not use)
MyString& MyString::operator= (const MyString &str)
{
    if (m_data) delete m_data;

    m_length = str.m_length;
    // copy the data from str to the implicit object
    m_data = new char[str.m_length];

    for (int i=0; i < str.m_length; ++i)
        m_data[i] = str.m_data[i];

        return *this; // return the existing object so we can chain this operator
}
```

You’ll probably get garbage output (or a crash). What happened?

Detecting and handling self-assignment

```
// A better implementation of operator=
Fraction& Fraction::operator= (const Fraction &fraction)
{
    // self-assignment guard
    if (this == &fraction)
        return *this;

    // do the copy
    m_numerator = fraction.m_numerator;
    m_denominator = fraction.m_denominator;

    // return the existing object so we can chain this operator
    return *this;
}
```

Shallow vs. deep copying

Shallow vs. deep copying

- 因为C++不够了解你的class，所以它提供的默认复制构造函数和默认赋值操作符，都只做逐个成员的浅复制：a memberwise copy (also known as a **shallow copy**.

- This means that C++ copies each member of the class individually (using the assignment operator for overloaded operator=, and direct initialization for the copy constructor).

- 当class是简单的，即不包含动态申请的内存的时候，这种方式OK

```
class MyString{
private:
    char *m_data;     int m_length;
public:
MyString(const char *source="") {
assert(source); // make sure source isn't a null string
// Plus one character for a terminator
m_length = strlen(source) + 1;

// Allocate a buffer equal to this length
m_data = new char[m_length];

// Copy the parameter string into our internal buffer
for (int i=0; i < m_length; ++i)     m_data[i] = source[i];
```

```
    }
   ~MyString() // destructor
   {
       // We need to deallocate our string
       delete[] m_data;
   }
   char* getString() { return m_data; }
   int getLength() { return m_length; }
};
```

浅拷贝shallow copy

- C++提供的复制构造函数是这样的：

```
MyString::MyString(const MyString &source) :
    m_length(source.m_length), m_data(source.m_data)
{}
```

Now, consider the following snippet of code:

```
int main()
{
    MyString hello("Hello, world!");
    {
        MyString copy = hello; // use default copy constructor
    } // copy gets destroyed here

    std::cout << hello.getString() << '\n'; // this will have un
defined behavior

    return 0;
}
```

Deep copying

```
MyString::MyString(const MyString& source){// Copy constructor
    m_length = source.m_length; // because m_length is not a pointer, we can shallow copy it
   // m_data is a pointer, so we need to deep copy it if it is non-null
    if (source.m_data)
   {
        m_data = new char[m_length];
        for (int i=0; i < m_length; ++i)
            m_data[i] = source[i];
    }
    else
        m_data = 0;
}
```

```
MyString& MyString::operator=(const MyString & source){// Assignment oper
ator
    // check for self-assignment
     if (this == &source)        return *this;
     delete[] m_data;  // first we need to deallocate any value that this string is holding!
     m_length = source.m_length;
    // m_data is a pointer, so we need to deep copy it if it is non-null
     if (source.m_data)     {
         m_data = new char[m_length];
         for (int i=0; i < m_length; ++i)  m_data[i] = source[i];
     }
     else m_data = 0;

     return *this;
}
```

Converting constructors, explicit, and delete

- By default, C++ will treat any constructor as an implicit conversion operator.
- `std::cout << makeNegative(6); // note the integer here`

```
// Default constructor
    Fraction(int numerator=0, int denominator=1) :
        m_numerator(numerator), m_denominator(denominator){
        assert(denominator != 0);
    }
```

- Constructors eligible to be used for implicit conversations are called **converting constructors**.
- Prior to C++11, only constructors taking one parameter could be converting constructors.
- However, with the new uniform initialization syntax in C++11, constructors taking multiple parameters can now be converting constructors.

explicit

```
class MyString{public:
// explicit keyword makes this constructor ineligible for implic
it conversions
explicit MyString(int x){ m_string.resize(x);}
};
int main(){
MyString x = 'x'; // compile error, since MyString(int) is now e
xplicit and nothing will match this
std::cout << x;
```

However, note that making a constructor explicit only prevents implicit conversions. Explicit conversions (via direct or uniform initialization or explicit casts) are still allowed:

```
MyString x('x'); // allowed, even though MyString(int) is explicit
```

- In our MyString case, we really want to completely disallow ‘x’ from being converted to a string (whether implicit or explicit, since the results aren’t going to be intuitive). One way to partially do this is to add a MyString(char) constructor, and make it private:

```
private:
MyString(char) // objects of type MyString(char) can't be constructed from outside the class
{ }
```

- However, this constructor can still be used from inside the class.
- A better way to resolve the issue is to use the “delete” keyword (introduced in C++11) to delete the function:

The delete keyword

```
class MyString
{
private:
std::string m_string;

public:
        MyString(char) = delete; // any use of this constructor
is an error
```

Quiz time

- Write a class that holds a string. Overload operator() to return the substring that starts at the index of the first parameter, and includes however many characters are in the second parameter.

- The following code should run:

```
int main()
{
    Mystring string("Hello, world!");
    std::cout << string(7, 5); // start at index 7 and return 5 characters

    return 0;
}
```

Inheritance

Composition组合

- 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?

- 保持每一个class相对简单
- 每个子类都是独立的self-contained，使得它们容易被重用。
    - reuse our `CPU`

- 聚焦于一个任务，而不是多个
    - **存储和维护数据(eg. `CPU`),**
    - **协调子类 (eg. `PersonalComputer`).**
    - **不要都自己干了.**

复合关系的使用

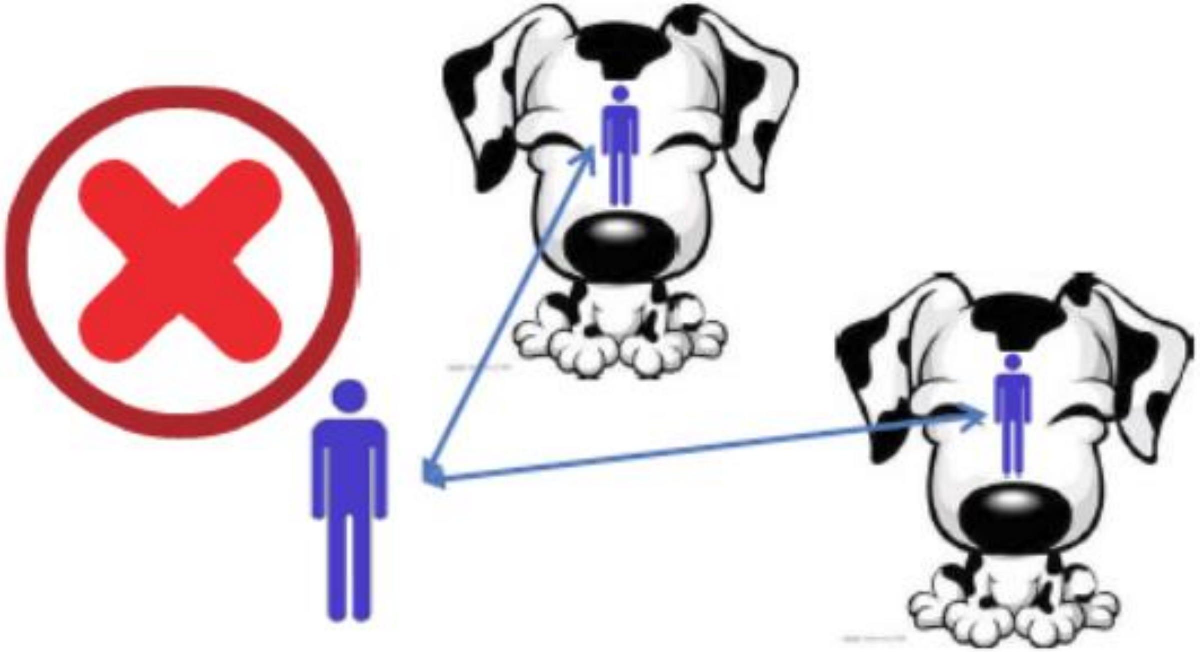
C:\Documents and Settings\hyz\桌面\3_01.gif

➢ 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;
};
```



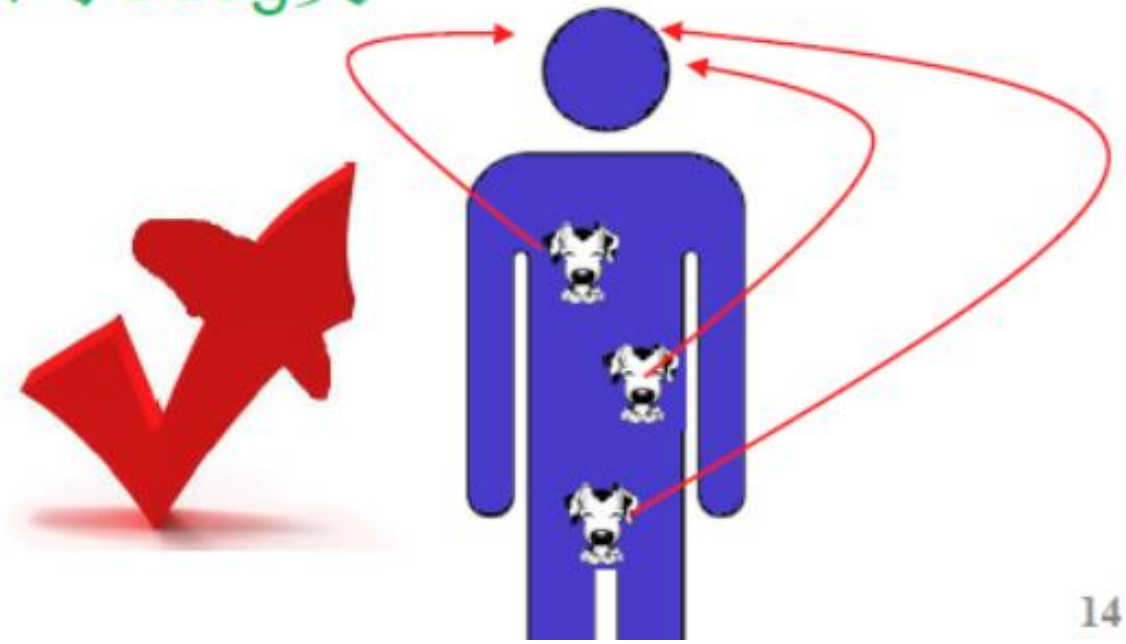
复合关系的使用

- 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];
};
```



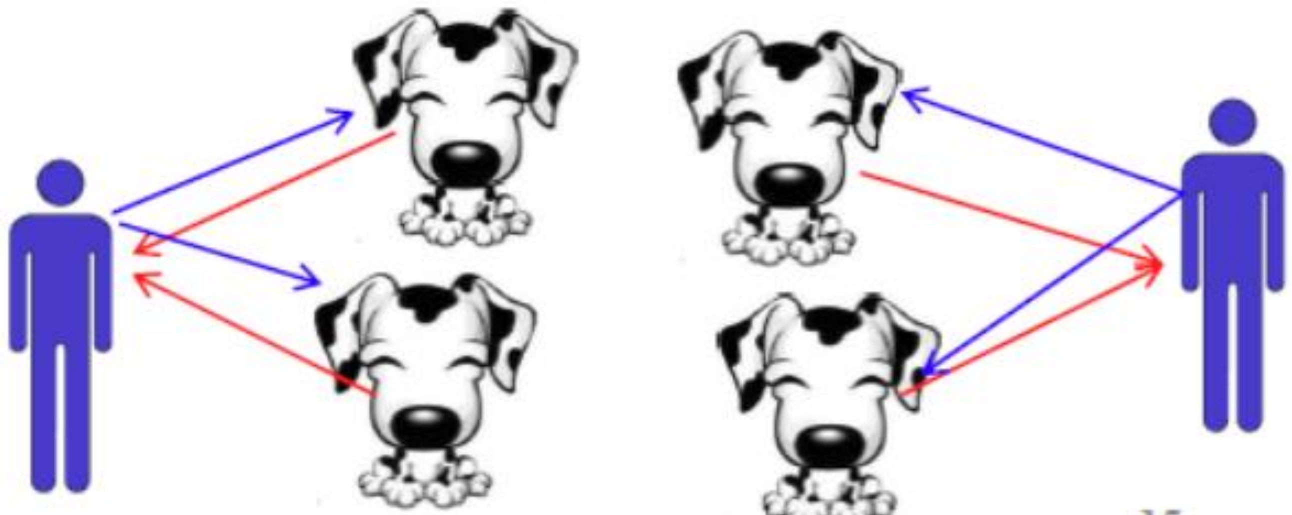
复合关系的使用

- 正确的写法：

  为“狗”类设一个“业主”类的对象指针；

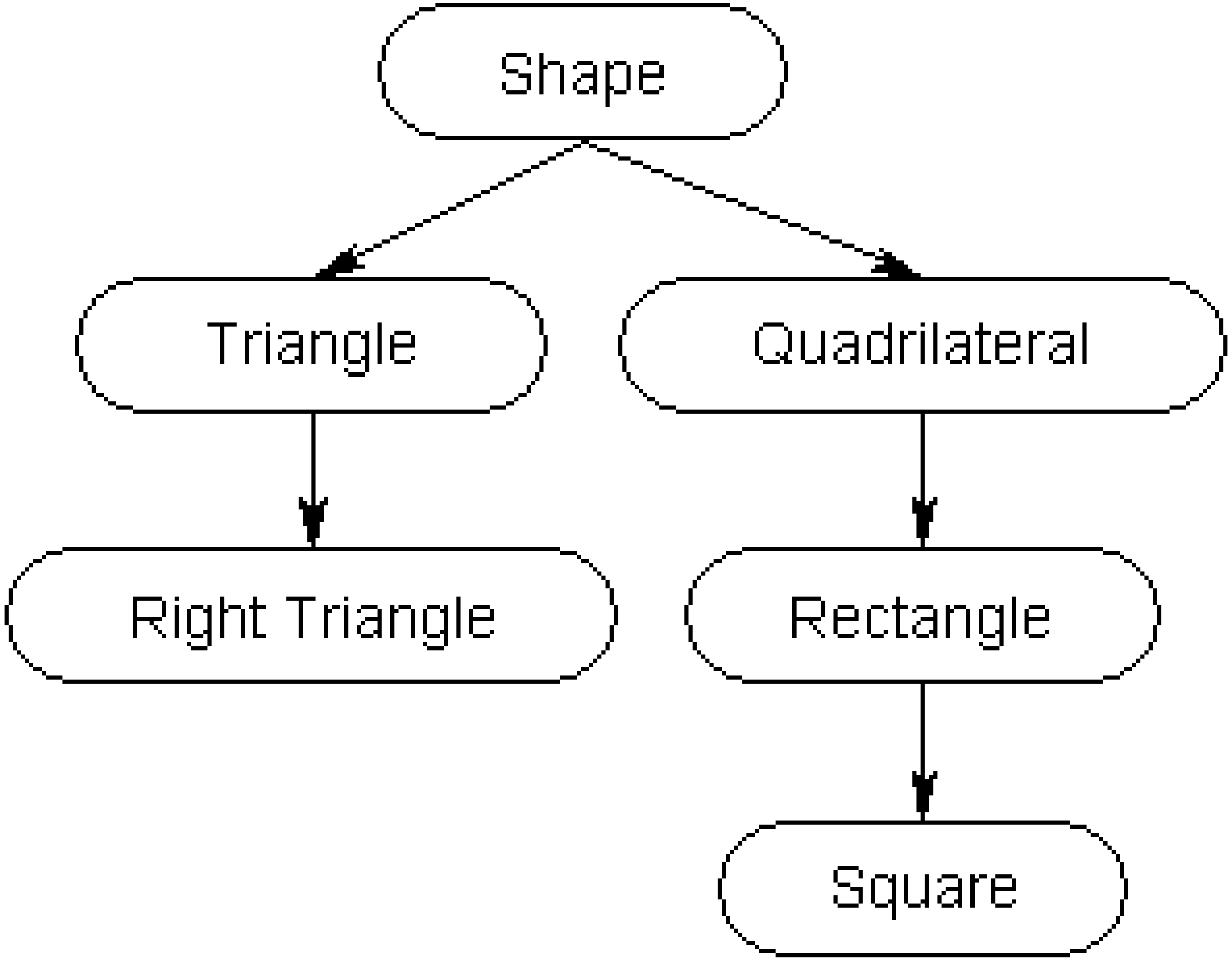
  为“业主”类设一个“狗”类的对象指针数组。

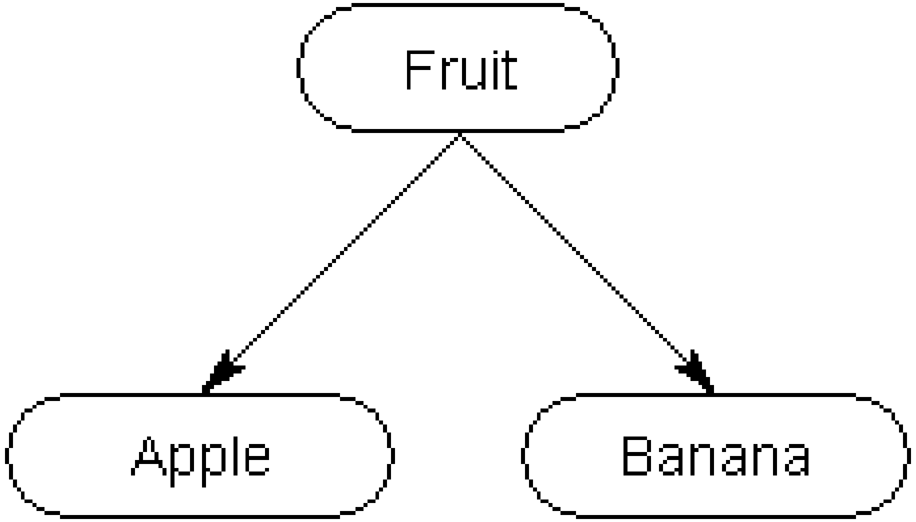
```
class CMaster;    //CMaster必须提前声明，不能先
                  //写CMaster类后写Cdog类

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



How to construct complex classes

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

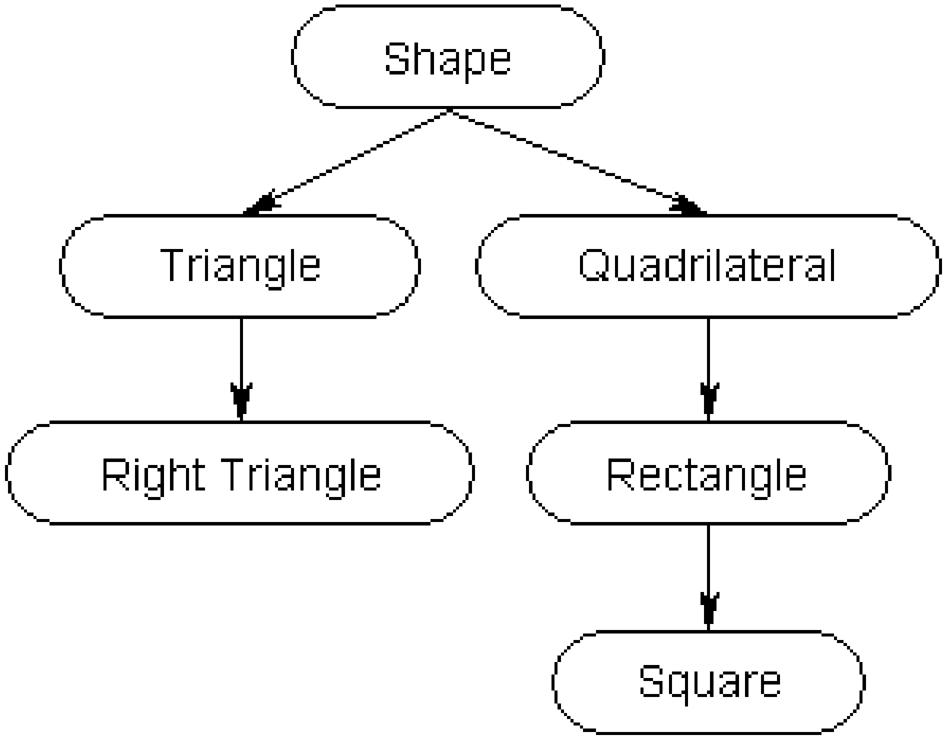




Why the need for inheritance in C++?

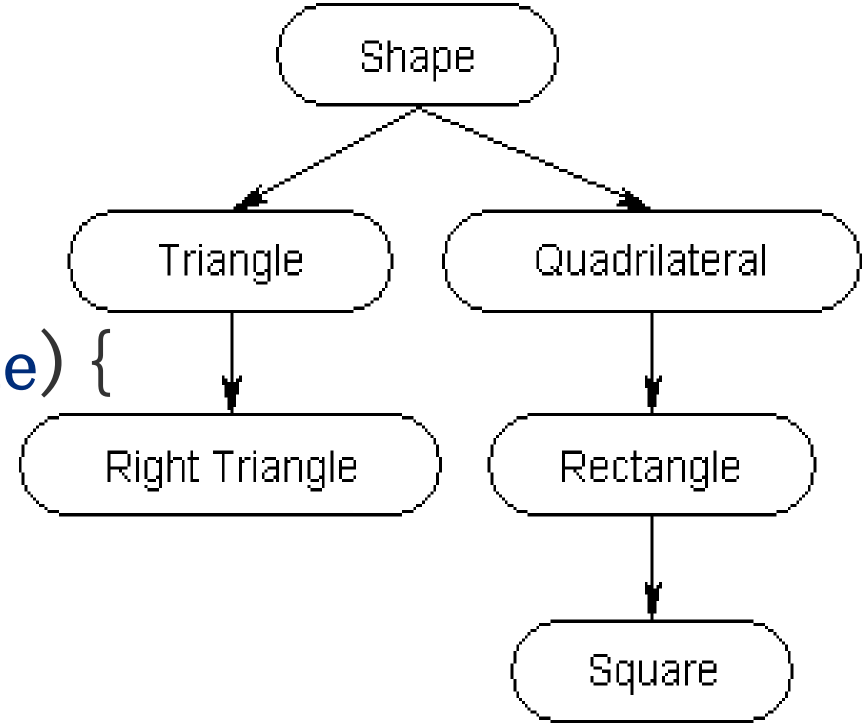
- 重用
  - 重用triangle，拓展功能triangle =》right triangle。

- 如果没有继承，而是把triangle代码拷贝改名为right triangle，然后再拓展功能。会带来**维护成本**的增加：Triangle的升级或者改错，都需要在Right Triangle的代码中再敲一遍，或者重新copy一遍。



派生类构造次序

```
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 successive derived class.

what actually happens when cDerived is instantiated?

1. 申请到cDerived将占用的内存(包括基类和派生类部分).
2. 调用派生类的合适的构造函数
3. 使用合适的基类的构造函数构造基类对象
4. 用成员初始化列表初始化派生类的成员变量
5. 派生类的构造函数的{}内的代码被执行
6. 控制权交还给调用者

初始化基类成员

```
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, int nValue=0)
        : m_dValue(dValue), m_nValue(nValue)
//error C2614: 'Derived': illegal member initialization: 'm_nValue' is not a base or member
    {   }
};
```

Initializing base class members

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

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

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

Adding, changing, and hiding members in a derived class

Adding new functionality

```
class Base{
protected:
int m_nValue;

public:
Base(int nValue)
     : m_nValue(nValue)
{}

void Identify() { cout << "I am a Base" << endl; }
};
```

```
class Derived: public Base
{
public:
    Derived(int nValue)
        :Base(nValue)
    { }

    int GetValue() { return m_nValue; }
};
```

Redefining functionality

```
class Derived: public Base{
public:
    // Here's our modified function
    void Identify() { cout << "I am a Derived" << endl; }
};
```

- Base cBase(5);
- cBase.Identify();
-
- Derived cDerived(7);
- cDerived.Identify()

I am a Base
I am a Derived

Adding to existing functionality
如何调用基类同名成员

```
class Derived: public Base{
public:
    void Derived::Identify(){
        Identify(); // would be Derived::Identify() => infinite loop!
        cout << "I am a Derived"; // then identify ourselves
    }

    void Identify(){
        Base::Identify(); // call Base::Identify() first
        cout << "I am a Derived"; // then identify ourselves
    }
};
```

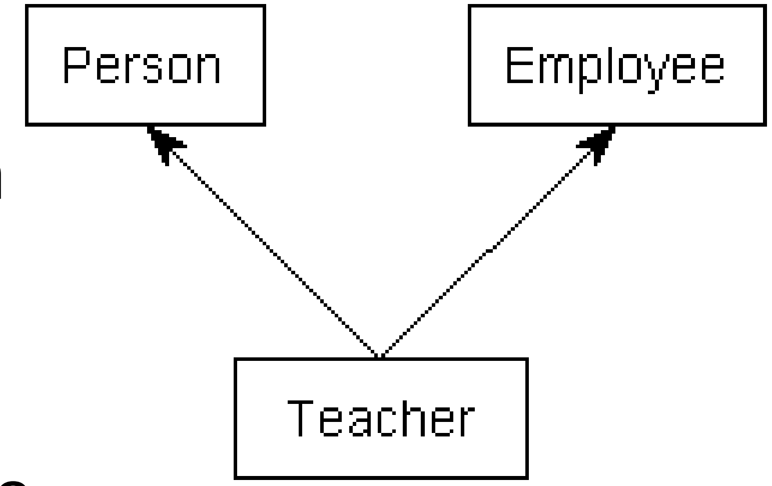
Hiding functionality

- In C++, it is not possible to remove functionality from a class. However, it is possible to hide existing functionality.

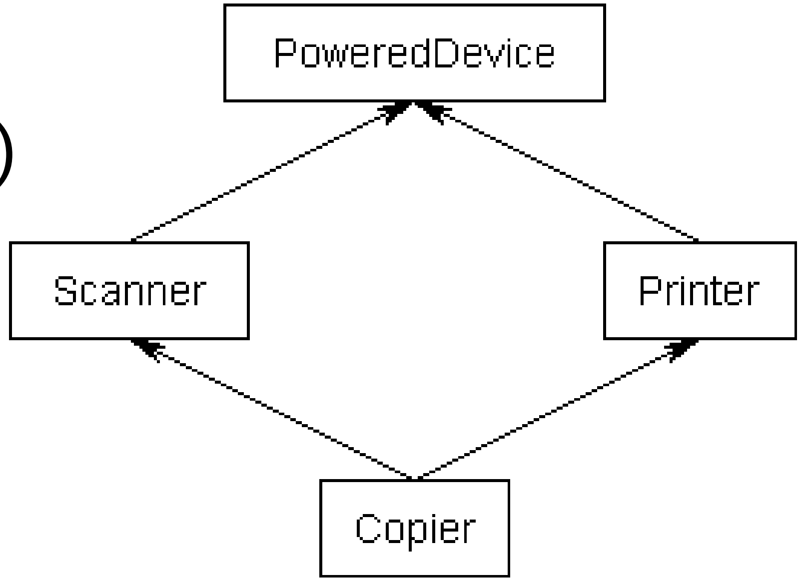
```
class Base{
protected:
    void PrintValue() { cout << m_nValue; }
};
class Derived: public Base{
public:
    Base::PrintValue;
};
        // PrintValue is public in Derived, so this is okay
        cDerived.PrintValue(); // prints 7
```

```
class Base{
public:    int m_nValue;
};

class Derived: public Base{
private:    Base::m_nValue;
};

int main(){
    Derived cDerived(7);

    // The following won't work because m_nValue has been redefi
ned as private
    cout << cDerived.m_nValue;
```

Multiple inheritance



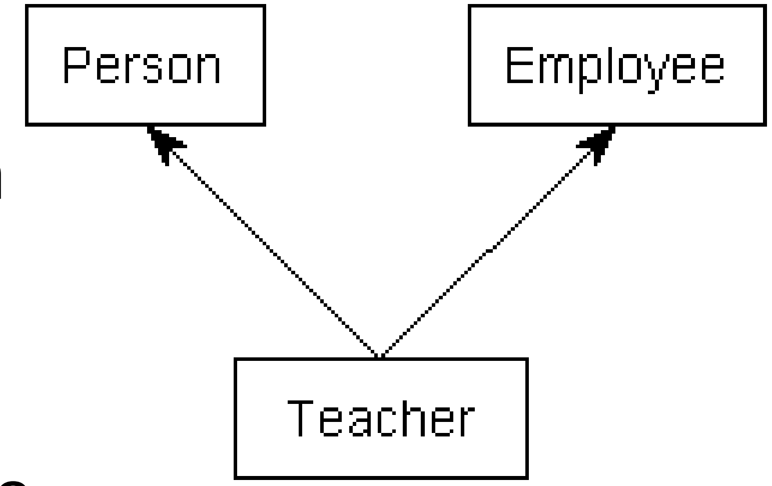
- multiple inheritance introduces a lot of issues that can markedly increase the complexity of programs and make them a maintenance nightmare.
- most of the problems that can be solved using multiple inheritance can be solved using single inheritance as well.



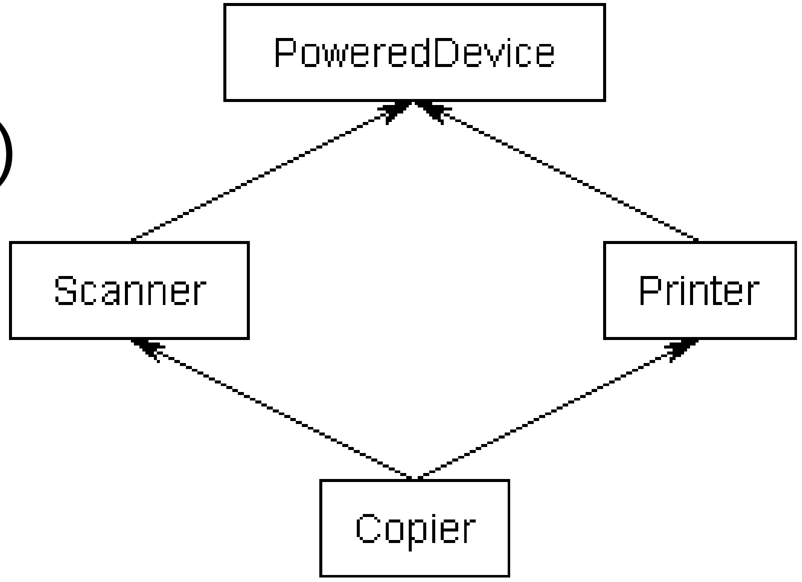
- Many object-oriented languages (eg. Smalltalk, PHP) do not even support multiple inheritance.
- Many relatively modern languages such as Java and C# restricts classes to single inheritance of normal classes, but allow multiple inheritance of interface classes

Multiple inheritance

- multiple inheritance introduces a lot of issues that can markedly increase the complexity of programs and make them a maintenance nightmare.



- most of the problems that can be solved using multiple inheritance can be solved using single inheritance as well.

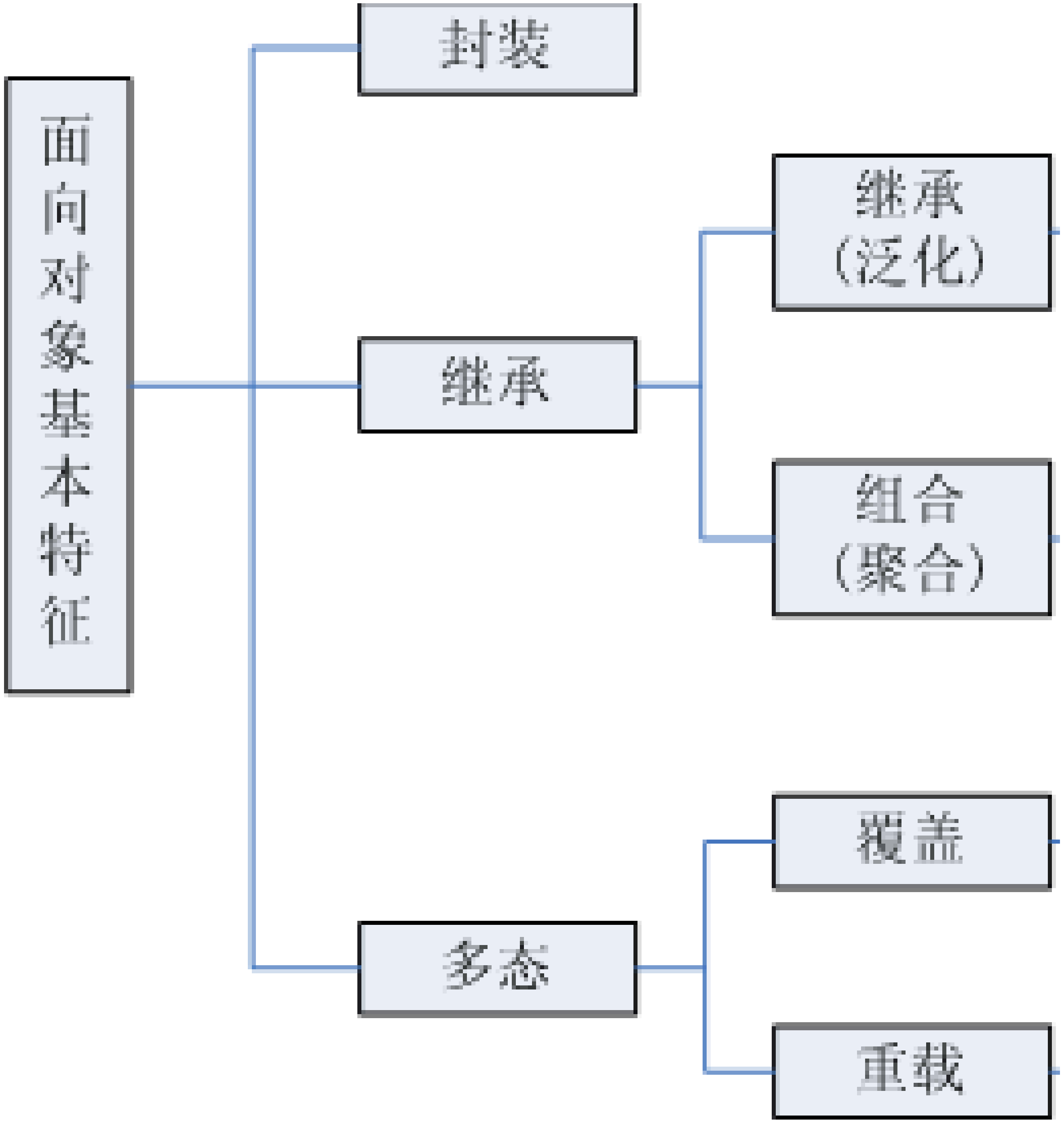


- Many object-oriented languages (eg. Smalltalk, PHP) do not even support multiple inheritance.

- Many relatively modern languages such as Java and C# restricts classes to single inheritance of normal classes, but allow multiple inheritance of interface classes

Polymorphism

- **most important and powerful aspects of inheritance -- virtual functions.**


面向对象基本特征
封装
继承
多态
继承（泛化）
组合（聚合）
覆盖
重载