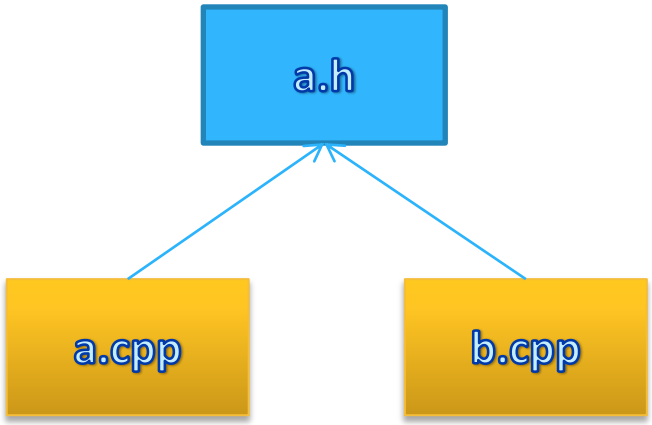
# Practice of Programming
# Q&A

**Haopeng Chen**

***RE**liable, **IN**telligent and **S**calable Systems Group (**REINS**)*
Shanghai Jiao Tong University
Shanghai, China
e-mail: qzheng2010@hotmail.com

- 函数重复定义问题



# Q&A



- 头文件末尾不要忘记 “;”
  - 否则会报编译错误
- Newpoint.h

```
#include "Point.h"
class Newpoint : Point{
        Image im;
public:
        Newpoint(int x, int y);
};
```

- Newpoint.cpp

```
#include <cmath>
#include <sstream>
#include "Simple_window.h"  // get access to our window library
#include "Graph.h"          // get access to our graphics library facilities
#include "Newpoint.h"

Newpoint::Newpoint(int xx, int yy):im(Point(), "c:\1.png"){
        x=xx;
        y=yy;
}
```

# Q&A



- 冒号初始化与括号初始化的区别
- Newpoint.cpp

```
Newpoint::Newpoint(int xx, int yy):im(Point(), "c:\1.png"){
        x=xx;
        y=yy;
}
```

或

```
Newpoint::Newpoint(int xx, int yy){
        x=xx;
        y=yy;
        im(Point(), “c:\1.png”);
}
```

或

```
Newpoint::Newpoint(int xx, int yy):x(xx),y(yy),im(Point(), "c:\1.png"){}
```

# Q&A



- 冒号初始化与括号初始化基本没什么区别
  - 但在非静态const类型以及引用型成员变量必须在初始化列表里面初始化，不能在{}里面初始化
  - 从父类继承而来的属性不能进行冒号初始化

- 探究

```
int a=10;
int a(10);
```

  - 括号赋值只能在变量定义并初始化中,不能用在变量定义后再赋值

```
int a;
.....
a=10;        \\正确
int b;
.....
b(10);       \\错误
```

From：http://www.cppblog.com/luyulaile/archive/2011/02/14/140059.html

# Q&A



- 类构造函数的作用是创建一个类的对象时，调用它来构造这个类对象的数据成员
  - 一要给数据成员分配内存空间
  - 二要给数据成员初始化，按数据成员在类中声明的顺序进行构造

- 冒号初始化与函数体初始化的区别在于:
  - 冒号初始化是给数据成员分配内存空间时就进行初始化
  - 函数体初始化是在所有的数据成员被分配内存空间后才进行的

From：http://www.cppblog.com/luyulaile/archive/2011/02/14/140059.html

# Q&A



```
class student {
 public :
        student ()
 protected:
        const int a;
        int &b;
 }
 student ::student (int i,int j)
 {
        a=i;
        b=j;
 }
```

- Student类不能通过编译
- 因为常量初始化时必须赋值，它的值是不能再改变的，与常量一样引用初始化也需要赋值，定义了引用后，它就和引用的目标维系在了一起,也是不能再被赋值的。
- Student类的构造函数应为:

```
student ::student(int i,int j):a(i),b(j){}
```

From：http://www.cppblog.com/luyulaile/archive/2011/02/14/140059.html

# Q&A



```
class teach
  {
     public :
        teach(char *p="name",int a=0)
                          .                    .
                          .
     protected:
           char name[30];
           int age;
  }
teach ::teach(char*p,int a)
{
     strcopy(name ,p);
     age=a;
    cout>>name>>endl;
}
```

```
class student
 {
      public:
         student (char *p="name");
                          .
                          .
                          .
      protected;
         char name[30];
         teach teacher;
   };
   student::student(char *p)
   {
         strcopy(name,p);
         cout>>name>>endl;
   }
```

- 在上面的程序中通不过编译，编译系统会告诉你teacher这个类对象缺默认构造函数，因为在teach 类中没有定义默认的构造函数。

From：http://www.cppblog.com/luyulaile/archive/2011/02/14/140059.html

```
student::student(char *p,char *pl,int ag):teacher(pl,ag)
{
        strcopy(name,p);
        cout>>name>>endl;
 }
```

- 在没有默认构造函数的时候，如果一个类对象是另一个类的数据成员，那么初始化这个数据成员，就应该放到冒号后面。

# Practice of Programming 2
# Subclassing

**Haopeng Chen**

***RE**liable, **IN**telligent and **S**calable Systems Group (**REINS**)*
Shanghai Jiao Tong University
Shanghai, China
e-mail: qzheng2010@hotmail.com

# 内容提纲



- 回顾
  - 子类
  - 继承与虚函数
- 设计原则
  - public继承是is-a关系
  - 避免遮蔽继承而来的名字
  - 不要重新定义继承而来的非虚函数
  - 通过复合来构建has-a关系
  - private继承
  - 谨慎使用多重继承

# 子类型



- 如果编写的程序被证明非常好，但是想在其中添加新特性，这时怎么办？
  - 直接修改这个程序吗？
  - 或者创建该程序的一个副本，然后在这个副本上修改
  - 以使得该程序可以被复用？
- 在现实生活中，我们都想将一样事物扩展为另一样事物
- 在程序设计中
  - 我们通过子类型化(sub-typing)来实现此目的

# 子类型



- 如果我们有两个类型 S和T
- S是T的子类型，记作 S<:T，当且仅当：
  - 对于希望是 T 类型对象的任何实例
  - 都可以向其提供 S 类型的对象
    - 不需要修改原来的任何计算代码以保证程序的正确性

- 如果 S 是 T 的子类型，那么：
  - T类型的对象可以被S类型的对象所替换
    - 并没有改变程序的任何属性，例如正确性

# 替换原则



- 如果S<:T，那么也可以说：
  - T 是 S 的超类型(supertype)
- 超类型不能替换子类型
  - 所有的苹果都是水果，但是并非所有水果都是苹果
- 这与类型转换不同
- 例如：
  - 当需要double类型时
  - 可以将int转换为double，从而改变int的物理表示
- 在需要使用超类型的地方使用子类型
  - 并不会将其转换为超类型，而是将其当做超类型使用

# 子类化



- C++拥有支持子类型化的机制
  - 成为子类化(subclassing)，有时也称为继承(inheritance)
- 其基本思想是：
  - 如果有一个ADT类foo
  - 并且需要创建其子类型bar，那么就可以声明：
- 这在声明：
  - bar是一个foo，可能还包含额外的状态
  - 并且可能还包含新的或重定义的成员函数

# 如何定义MaxIntSet



- 这个函数有什么缺点？
- 它是否总是会返回某个值呢？
- 如果用某个“偏执(paranoid)”的编译器编译这个程序
  - 它会警告你max可能在任何情况下都不会返回一个值

- 我们需要什么？
  - 需要在导出类中访问基类(base class)的成员
  - 同时需要阻止其他从基类外部的访问
- 如何解决呢？
  - 在基类中使用protected修饰符

# protected



- 如果需要在导出类中访问基类(base class)的成员
  - 同时需要阻止其他从基类外部的访问
- 怎么解决呢？
  - 在基类中使用protected修饰符
- 如果一个成员是protected的，那就意味着：
  - 它可以被这个类以及所有的导出类的成员访问
- Protected 数据成员使得导出类变得异常脆弱
  - 这使得我们需要好好考虑是否值得这么做

# 扩展功能



- 如果需要多个方法该怎么办呢？
  - 例如，我们希望能够在IntSet类的子类MaxIntSet中处理空集合
- 通过修改新的导出类中的方法
  - 或者重载/重定义这些方法
- 子类型必须遵守替换原则
- 这就意味着新方法必须执行旧方法中的所有操作
  - 但是可以做更多的事情
- 类似地，它对调用者的要求不能比旧方法多
  - 但是可以更少
- 换句话说，能够正常使用超类型的代码
  - 必须仍旧能够正常使用超类型

# 子类不是子类型



```
class PosIntSet : public IntSet {
  // OVERVIEW: a subclass, but not a subtype
 public:
  void insert(int v);
  // EFFECTS: if v is non-negative and s has room to include it, s=s+{v}
  // if v is negative throw int -1; if s is full thrown int MAXELTS
 };

 void PosIntSet::insert(int v) {
  if (v < 0) throw -1;
  IntSet::insert(v);
 }
```

# 子类不是子类型



- 能够正确使用IntSet的代码在使用PosIntSet时有可能会失败
  - 它并没有遵守替换原则

- C++使用的是“子类”替换，而不是“子类型”替换
- 这意味着编译器允许子类：
  - 用于任何希望使用超类型的地方
  - 即使它是不可替换的

- 例如，下面的代码是完全合法的：

```
PosIntSet s;
IntSet& r = s;
```

# 明显类型和实际类型



- 当代码中包含引用时，需要区分明显类型和实际类型
  - Apparent type vs. Actual type
- 明显类型：引用的声明类型
- 实际类型：引用的真实类型

- 注意：即使编译器“看到”这种赋值
  - 它也不认为它了解的比“声明”的更多
- 在这个例子中，r = s 不会改变 r 的明显类型
  - 这对于方法选择有十分重要的含义

# 方法的静态与动态选择



- C++静态地选择要运行的方法
  - 基于在编译时了解的情况
  - 或者基于明显类型

```
PosIntSet s;
IntSet& r = s;
```

- 打破了s集合的表示不变式，这十分糟糕

- 静态
  - 在程序执行之前编译器即知道
- 动态
  - 不能静态地知道，只有在运行时才能了解到

# 静态与动态实例



```
void foo(IntSet &bar);
IntSet s;
PosIntSet p;
foo(s);
foo(p);
```

- 由于同一个引用同时指向了IntSet和PosIntSet
  - 所以编译器无法选择

- 有办法解决吗？
  - 有，通过使用虚函数强制编译器在运行时选择

# 虚函数



- 在函数声明前增加关键字"*virtual*”

```
class IntSet {
        …
  public:
        …
        virtual void insert(int v);
        …
};
```

# 虚函数



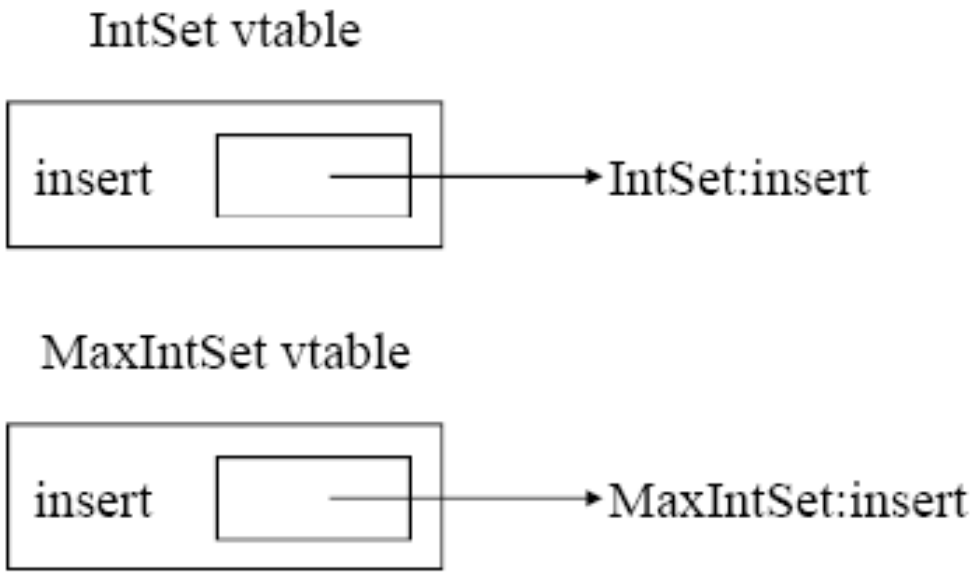
- 告知编译器某个类可能会覆盖我的实现：
  - 总是在运行时检查需要调用哪个版本

- 在PosIntSet的定义中不需要加上virtual关键字
  - 因为“虚”是可继承的，就像别的成员一样

# 虚表

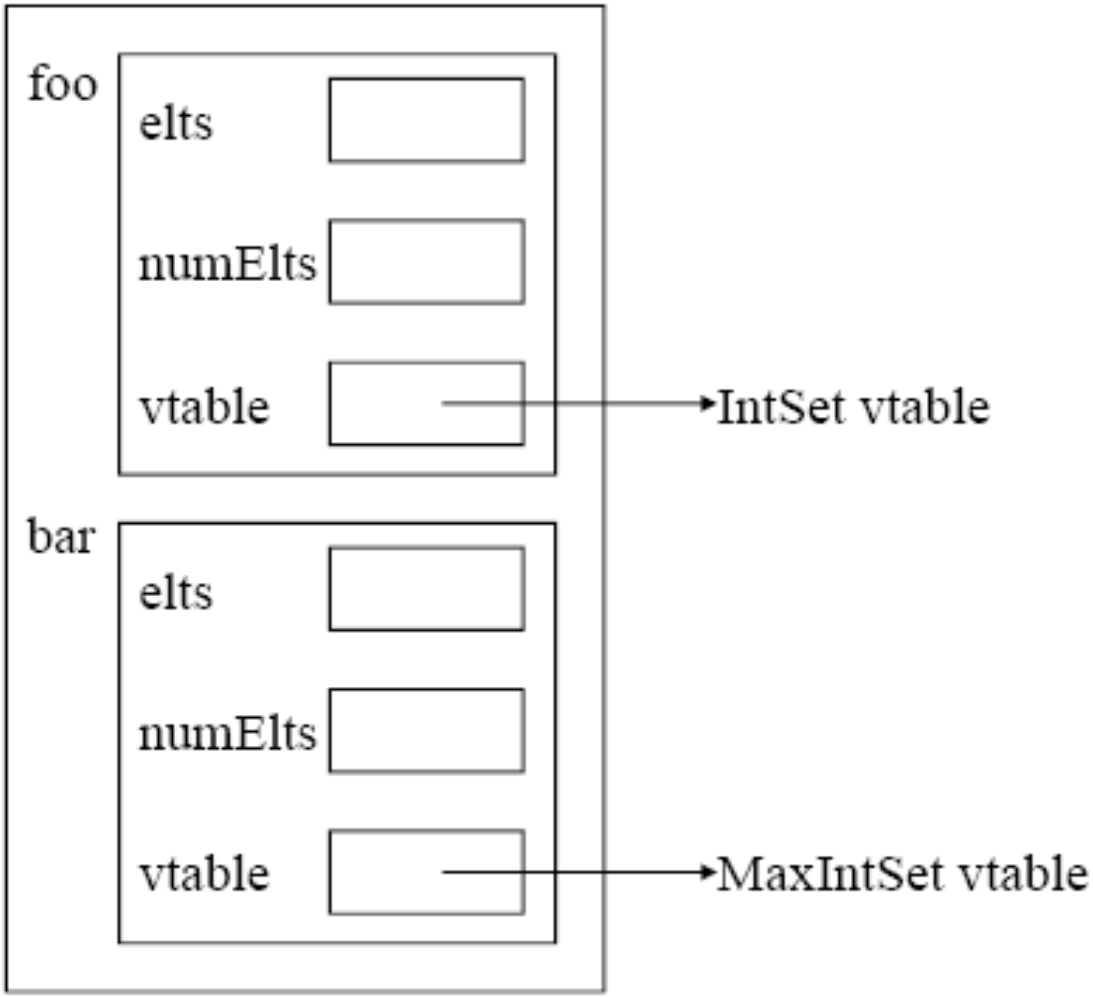


- 对于拥有虚函数的类
  - 编译器将创建一个“vtable”，即“虚表(virtual table)”
  - 其中包含一组函数指针
  - 每个指针指向一个虚函数
- 这些指针都将初始化为指向恰当的实现



# IntSet foo; MaxIntSet bar;





# 延迟绑定和多态



- 虚函数强制编译器
  - 将选择函数实现的工作延迟到运行时
- 这种函数选择的延迟成为延迟绑定(late binding)

- 这种将一个函数的多个实现关联起来的机制
  - 称为多态(Polymorphism)

- 多态、延迟绑定、虚函数实际上是一回事

# 链式初始化



- 如果我们正在创建一个导出类的实例
  - 基类的构造器将先被调用
  - 然后导出类的构造器才被调用
- 这条规则可以递归地作用
  - 知道我们到达导出链的根类(root class)

# 链式初始化



- 因此，如果我们有下面的类：

```
class foo {
        int f;
    public:
        foo();
};

class bar : public foo {
        int r;
    public:
        bar();
};
```

# 链式初始化



```
class baz : public bar {
        int z;
    public:
        baz();
}

baz *bzp = new baz;
```

# 链式初始化



- 将会指向下面的操作
  - 为baz创建内存空间(三个整数，z、r 和 f )
  - 调用foo()的构造器()
  - 调用bar()的构造器()
  - 调用baz()的构造器()

- 这意味着导出类的构造器可以认为：
  - 它已经是其基类的一个“正确(proper)”实例了

# 为什么不把所有方法都变成虚函数？



- 虚函数的优点已经很明显了
- 但是它也有缺点！

- 主要缺点之一：开销
  - 使用了更多的存储空间
  - 延迟绑定是“操作中处理(on the fly)”的，因此程序运行较慢
- 因此，虚函数如果不是必需的，则不应该使用

# 扩展类型兼容性



- 假如：
- Derived是Base的导出类
  - Derived对象可以赋值给Base类型的对象
  - 但是反之不可以！

# 扩展类型兼容性



```
class Pet {
    public:
        string name;
        virtual void print() const;
};

class Dog : public Pet {
    public:
        string breed;
        virtual void print() const;
};
```

- 成员name和breed是公有的，这只是为了举例，属非典型

# 使用Pet和Dog类



- 现在假设有如下声明:

```
Dog vdog;
Pet vpet;
```

- 任何是狗的对象都是宠物:

```
vdog.name = "Tiny";
vdog.breed = "Great Dane";
vpet = vdog;
```

- 这些语句是允许的
- 子类型可以赋值给父类型，但是反之不然
  - 宠物不都(必须)是狗

# 切片问题



- 注意，赋值给 vpet 的对象，就丢失了其breed域：

  cout << vpet.breed;

- 这称为切片问题
- 看起来似乎是合适的
  - Dog被当做了Pet变量，因此它不应该被当做Pet
    - 因此也就不需要有“dog”特有的属性
- 但是对于严谨的计算机科学家来说，这么做是有问题的
  - 因为信息丢失了

# 切片问题修正



- 在C++中，切片问题是令人讨厌的
  - 它仍旧是一条纯种的狗
  - 我们希望在它被当做Pet时仍旧可以引用到其breed属性
- 有什么方法可以实现这个目的吗？
- 是的，虚函数！
- 但是如何做呢？
  - 可以将虚函数与指向动态变量的指针结合使用

# 切片问题示例



```
Pet *ppet;
Dog *pdog;
pdog = new Dog;
pdog->name = "Tiny";
pdog->breed = "Great Dane";
ppet = pdog;
```

- 不能访问ppet指针指向的对象的breed域

```
cout << ppet->breed; //ILLEGAL!
```

# 切片问题示例



- 必须使用虚成员函数：

  ```
  ppet->print();
  ```

- 调用Dog类中的print成员函数
  - 因为它是虚函数
- C++在绑定调用前会先观察ppet指向何种对象

# 虚函数的内部工作机制



- 不需要了解如何使用它！
  - 信息隐藏的原则
- 虚函数表
  - 编译器创建的
  - 包含指向每个虚成员函数的指针
  - 指向函数的正确的代码位置
- 这种类的对象也都有一个指针
  - 指向虚函数表

# 内容提纲



- 回顾
  - 子类
  - 继承与虚函数
- 设计原则
  - public继承是is-a关系
  - 避免遮蔽继承而来的名字
  - 不要重新定义继承而来的非虚函数
  - 通过复合来构建has-a关系
  - private继承
  - 谨慎使用多重继承

# public继承是 is-a 关系



```
class Person {...};
class Student: public Person {...};
```

- 子类对象可以当做父类对象，但是反之不成立

```
void eat(const Person& p);          // anyone can eat
void study(const Student& s);       // only students study
Person p;                           // p is a Person
Student s;                          // s is a Student
eat(p);                             // fine, p is a Person
eat(s);                             // fine, s is a Student,
                                    // and a Student is-a Person
study(s);                           // fine
study(p);                           // error! p isn't a Student
```

# public继承是 is-a 关系



- 问题

```
class Bird {
public:
  virtual void fly();              // birds can fly
  ...
};

class Penguin:public Bird {        // penguins are birds
  ...
};
```

# public继承是 is-a 关系



- 方案一

```
class Bird {
  ...                          // no fly function is declared
};

class FlyingBird: public Bird {
public:
  virtual void fly();
  ...
};

class Penguin: public Bird {
  ...                          // no fly function is declared
```

# public继承是 is-a 关系



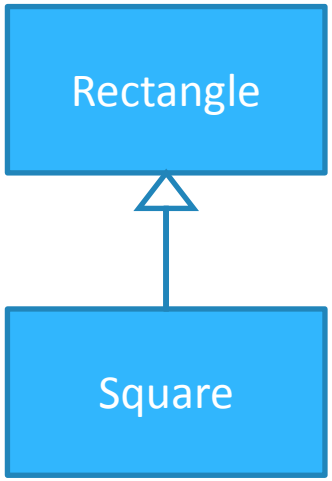
- 方案二

```
void error(const std::string& msg);      // defined elsewhere

class Penguin: public Bird {
public:
  virtual void fly() { error("Attempt to make a penguin fly!");}
  ...
};
```

# public继承是 is-a 关系



- 问题
  - Square是否应该继承自Rectangle？



# public继承是 is-a 关系



```
class Rectangle {
public:
  virtual void setHeight(int newHeight);
  virtual void setWidth(int newWidth);
  virtual int height() const;          // return current values
  virtual int width() const;
   ...
};

void makeBigger(Rectangle& r)          // function to increase r's area
{
   int oldHeight = r.height();
   r.setWidth(r.width() + 10);         // add 10 to r's width
   assert(r.height() == oldHeight);    // assert that r's
}
```

# public继承是 is-a 关系



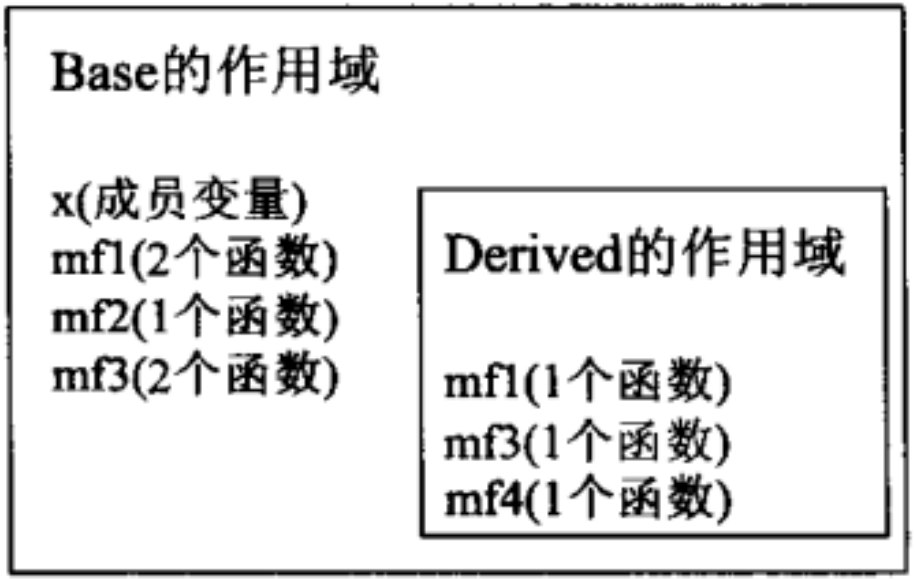
```
class Square: public Rectangle {...};

Square s;
...

assert(s.width() == s.height());  // this must be true for all squares
makeBigger(s);                    // by inheritance, s is-a Rectangle,
                                  // so we can increase its area
                                  // for all squares
```

- 请记住：public继承是is-a关系，每一个导出类对象都是一个基类对象，适用于基类的所有事情都需要适用于导出类。

# 避免遮蔽继承而来的名字



```
class Base {
private:
  int x;
public:
  virtual void mf1() = 0;
  virtual void mf1(int);
  virtual void mf2();
  void mf3();
  void mf3(double);
};

class Derived: public Base {
public:
  virtual void mf1();
  void mf3();
  void mf4();
};
```



# 避免遮蔽继承而来的名字



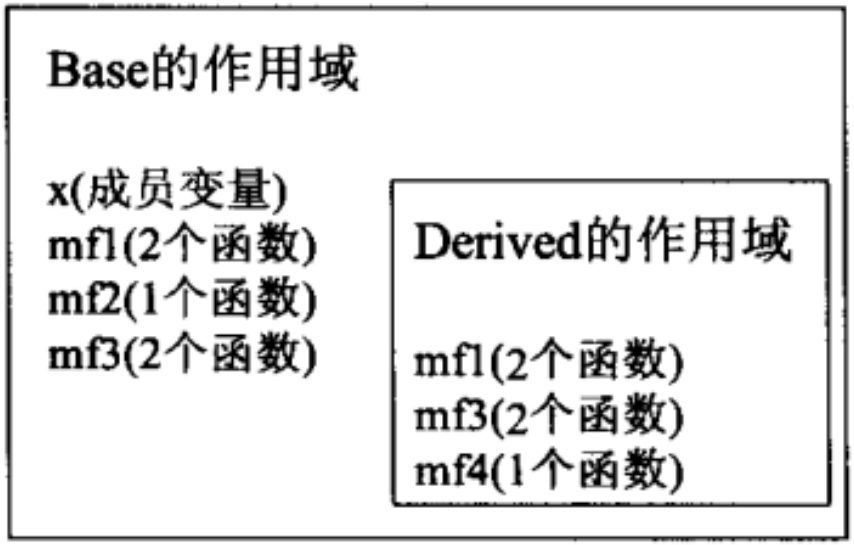
```
class Base {
private:
  int x;
public:
  virtual void mf1() = 0;
  virtual void mf1(int);
  virtual void mf2();
  void mf3();
  void mf3(double);
};

class Derived: public Base {
public:
  virtual void mf1();
  void mf3();
  void mf4();
};
```

```
Derived d;
int x;
...

d.mf1();   // fine, calls Derived::mf1

d.mf1(x); // error! Derived::mf1 hides Base::mf1

d.mf2();   // fine, calls Base::mf2

d.mf3();  // fine, calls Derived::mf3

d.mf3(x);  // error! Derived::mf3 hides Base::mf3
```

# 避免遮蔽继承而来的名字



```
class Base {
private:
  int x;
public:
  virtual void mf1() = 0;
  virtual void mf1(int);
  virtual void mf2();
  void mf3();
  void mf3(double);
};

class Derived: public Base {
public:
  using Base::mf1;      // make all things in Base named mf1 and mf3
  using Base::mf3;      // visible (and public) in Derived's scope
  virtual void mf1();
  void mf3();
  void mf4();
};
```



# 避免遮蔽继承而来的名字



```
class Base {
private:
  int x;
public:
  virtual void mf1() = 0;
  virtual void mf1(int);
  virtual void mf2();
  void mf3();
  void mf3(double);
};

class Derived: public Base {
public:
  using Base::mf1;
  using Base::mf3;
  virtual void mf1();
  void mf3();
  void mf4();
};
```

```
Derived d;
int x;
...

d.mf1();   // still fine, still calls Derived::mf1

d.mf1(x); // now okay, calls Base::mf1

d.mf2();   // still fine, still calls Base::mf2

d.mf3();  // fine, calls Derived::mf3

d.mf3(x);  // now okay, calls Base::mf3
```

# 避免遮蔽继承而来的名字



```
class Base {
public:
  virtual void mf1() = 0;
  virtual void mf1(int);
  ...                          // as before

};

class Derived: private Base {
public:
  virtual void mf1()               // forwarding function;
  { Base::mf1(); }
   ...
};

Derived d;
int x;
d.mf1();                       // fine, calls Derived::mf1
d.mf1(x);                      // error! Base::mf1() is hidden
```

# 避免遮蔽继承而来的名字



- 请记住
  - 导出类中的名字会遮蔽基类中的名字，不论是虚函数还是非虚函数
  - 可以使用using指示符或者使用forwarding function来解决问题

# 不要重新定义继承而来的非虚函数



```
class B {
public:
  void mf();
  ...
};

class D: public B {
public:
  void mf();                    // hides B::mf;
  ...
};

D x;                            // x is an object of type D
B *pB = &x;                     // get pointer to x
pB->mf();                       // call mf through pointer
D *pD = &x;                     // get pointer to x
pD->mf();                       // call mf through pointer
pB->mf();                       // calls B::mf
pD->mf();                       // calls D::mf
```

# 通过复合来构建has-a关系



```
class Address { ... };          // where someone lives
class PhoneNumber { ... };

class Person {
public:
  ...
private:
  std::string name;             // composed object
  Address address;               // ditto
  PhoneNumber voiceNumber;       // ditto
  PhoneNumber faxNumber;         // ditto
};
```

# private继承



```
class Person { ... };
class Student: private Person { ... };    // inheritance is now private

void eat(const Person& p);                // anyone can eat
void study(const Student& s);             // only students study

Person p;                                 // p is a Person
Student s;                                // s is a Student

eat(p);                                   // fine, p is a Person
eat(s);                                   // error! a Student isn't a Person
```

# 谨慎使用多重继承



```
class BorrowableItem {          // something a library lets you borrow
public:
  void checkOut();              // check the item out from the library
  ...
};

class ElectronicGadget {
private:
  bool checkOut() const;        // perform self-test, return whether
  ...                           // test succeeds
};

class MP3Player:                // note MI here
  public BorrowableItem,        // (some libraries loan MP3 players)
  public ElectronicGadget
{ ... };                        // class definition is unimportant

MP3Player mp;
mp.checkOut();                  // ambiguous! which checkOut?
```


# Thank You!