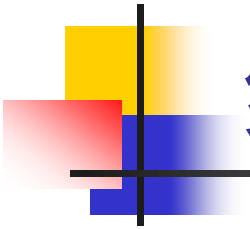


继承

- 继承机制
  - 基于**目标代码**的复用
  - 对事物进行分类
    - 派生类是基类的具体化
    - 把事物（概念）以层次结构表示出来，有利于描述和解决问题
  - 增量开发

单继承

```
//错误声明
class Undergraduated_Student : public Student;
//正确声明
class Undergraduated_Student ;
```

```
class Student
{       int id;
    public:
        char nickname[16];
        void set_ID(int x)  { id = x; }
        void SetNickName(char *s) { strcpy(nickname,s);}
virtual void showInfo()
        { cout << nickname << " : " << id <<endl; }
};
```

protected

id
nickname
dept_no

```
class Undergraduated_Student : public Student
{       int dept_no;
    public:
        void setDeptNo(int x) { dept_no = x; }
        void set_ID(int x) {.....}
        void showInfo()
         { cout << dept_no << ": "<< nickname <<endl; }
    private:
       Student::nickname;
       void SetNickName ();
};
```

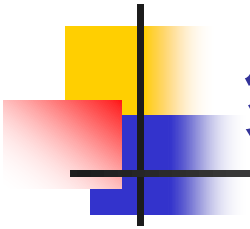
继承方式

public

private、protected

友元和protected

```
class Base (
protected :
    int prot_mem;                          // protected 成员
} ;
class Sneaky : public Base {
    friend void clobber{Sneaky&) ;         //能访问Sneaky::prot_mem
    friend void clobber{Base&);            //不能访问Base::prot_mem
    int j;                                 // j 默认是private

void clobber(Sneaky &s) { s.j =            //正确：clobber 能访问Sneaky对象的
s.prot_mem = 0; )                          private和protected成员

void clobber(Base &b) {                    //错误： clobber 不能访问Base的
b.prot_mem = 0; }                          protected 成员
```



继承

- 派生类对象的初始化
  - 由基类和派生类共同完成
- 构造函数的执行次序
  - 基类的构造函数
  - 派生类对象成员类的构造函数
  - 派生类的构造函数
- 析构函数的执行次序
  - 与构造函数相反

B(const B& b){…} ??

继承

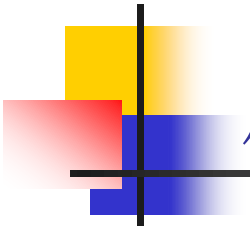
```
class B: public A{
public:
    using A::A; //继承A的构造函数
```

- 基类构造函数的调用
  - 缺省执行基类默认构造函数
  - 如果要执行基类的**非默认构造函数**，则必须在派生类构造函数的**成员初始化表**中指出

```
class A
{    int x;
  public:
     A() { x = 0; }
     A(int i) { x = i; }
};
```

```
class B: public A
{   int y;
public:
    B() { y = 0; }
    B(int i) { y = i; }
    B(int i, int j):A(i)
    {   y = j;  }
};
```

```
B b1;          //执行A::A()和B::B()
B b2(1);       //执行A::A()和B::B(int)
B b3(0,1);     //执行A::A(int)和B::B(int,int)
```



虚函数

- 类型相容
  - 类、类型
  - 类型相容、赋值相容
  - 问题：a、b是什么类型时，a = b 合法？
    - A a;　B b;　　class B: public A
      - 对象的身份发生变化
      - 属于派生类的属性已不存在
    - B* pb;　A* pa = pb;　　class B: public A
    - B　b;　　A &a=b;　　class B: public A
      - 对象身份没有发生变化

虚函数

```
class A
{    int x,y;
 public:
      void f();
};
class B: public A
{    int  z;
public:
       void f();
       void g();
};
```

```
A  a;
B  b;

a = b;      //OK,
b = a;      //Error
a.f();      //A::f()
```

```
A &r_a=b;          //OK
A *p_a=&b;         //OK

B &r_b=a;          //Error
B *p_b=&a;         //Error
```

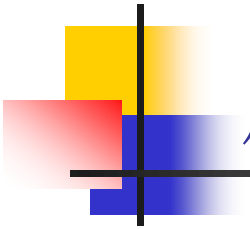
```
func1(A& a)
{ …  a.f();  … }

func2(A *pa)
{ …  pa->f(); …}

func1(b);
func2(&b);
```

把派生类对象赋值给基类对象

基类的引用或指针可以引用或指向派生类对象

A::f?

B::f?



虚函数

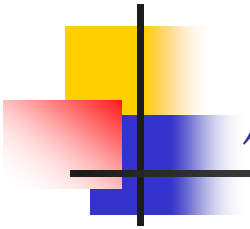
- 前期绑定（Early Binding）
  - 编译时刻
  - 依据对象的静态类型
  - 效率高、灵活性差
- 动态绑定（Late Binding）
  - 运行时刻
  - 依据对象的实际类型（动态）
  - 灵活性高、效率低
- 注重效率
  - 默认前期绑定
  - 后期绑定需显式指出 *virtual*

虚函数

- 定义
  - *virtual*

```
class A
{    ...
   public:
      virtual void f();
};
```

  - 动态绑定
    - *根据*实际引用和指向的对象类型
  - 方法重定义



虚函数

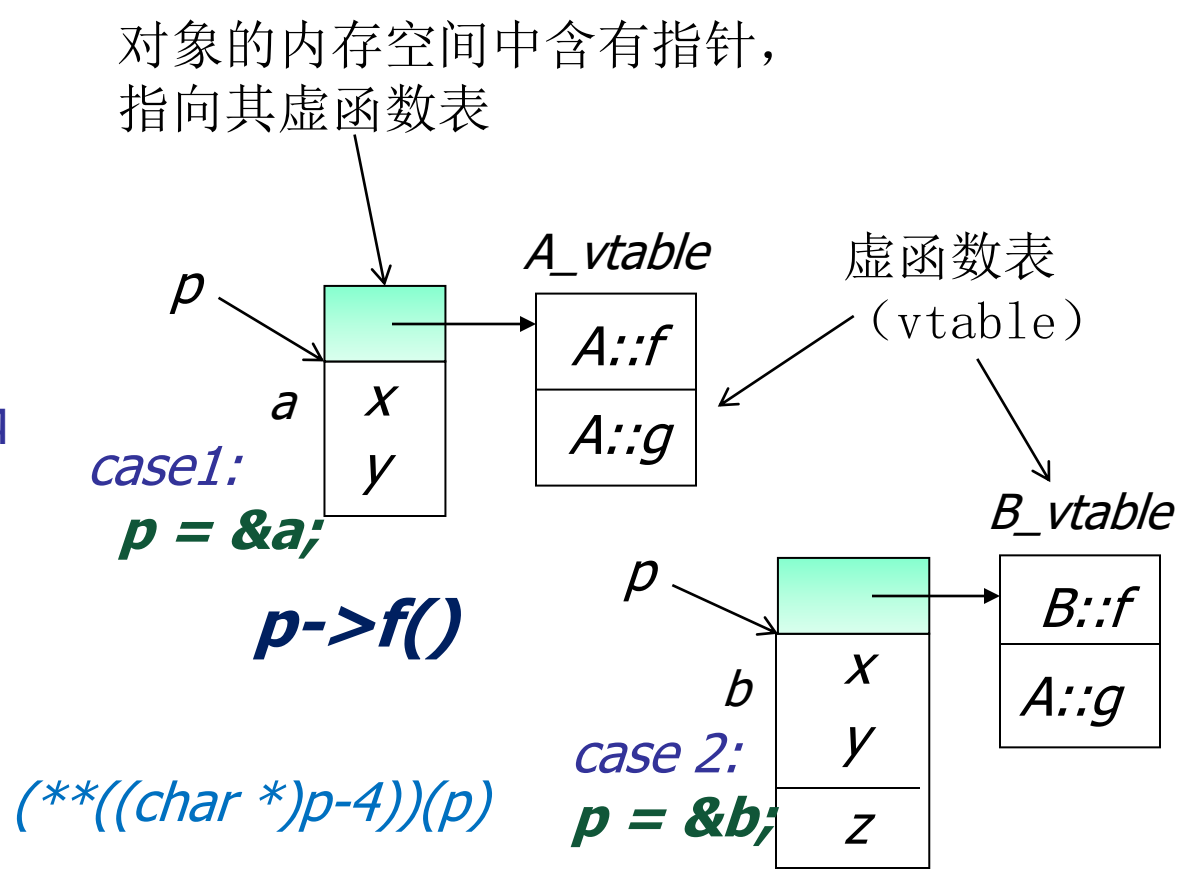
- 如基类中被定义为虚成员函数，则派生类中对其重定义的成员函数均为虚函数

- 限制
  - 类的成员函数才可以是虚函数
  - 静态成员函数不能是虚函数
  - 内联成员函数不能是虚函数
  - 构造函数不能是虚函数
  - 析构函数可以（往往）是虚函数

虚函数

- 后期绑定的实现

```
class A
{     int x,y;
public:
      virtual f();
     virtual g();
     h();
};
class B: public A
{      int z;
public:
        f();
       h();
};
A a; B b;
A *p;
```



虚函数

```
class A
{  public:
       virtual void f( ) ;
       void g() ;
};

class B: public A
{   public:
       void f( ) { g(); }
       void g() ;
};

B b;
A* p = &b;
p->f();   //b.B::g
```

B* const this

this->g();

直到构造函数返回之后，对象方可正常使用

```
class A
{    public:
          A() { f();}
          virtual  void f();
           void g();
          void h() { f(); g(); }
};
class B: public A
{   public:
        void f();
        void g();
};
```

```
...
B b;          // A::A()，A::f, B::B(),
A *p=&b;
p->f();      //B::f
p->g();      //A::g
p->h();      //A::h, B::f, A::g
```