6 Derived Classes
派生类

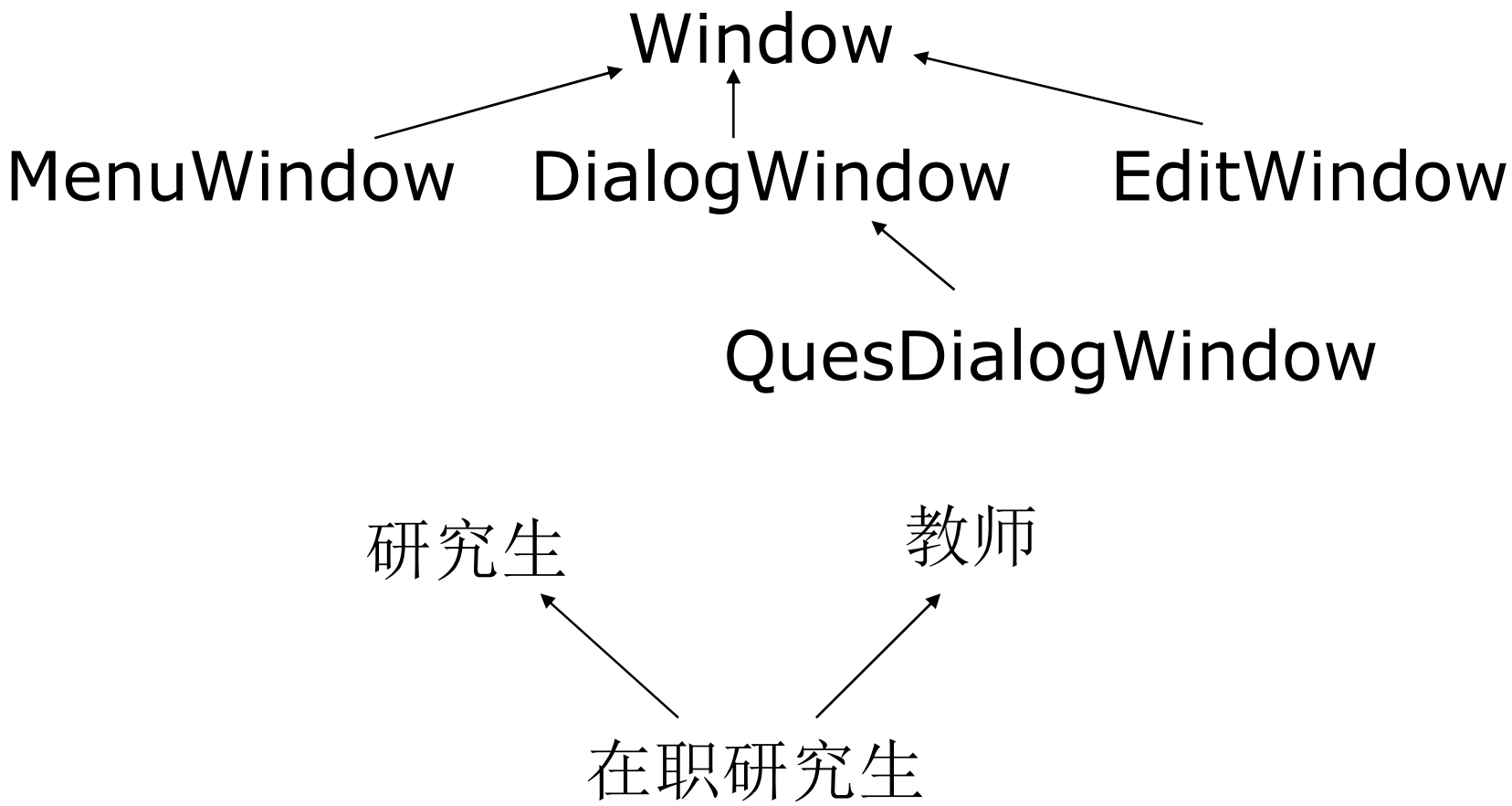
董洪伟

http://hwdong.com

派生（继承）的概念

- 继承(inheritance)是C++软件复用的一种途径。通过从基类(Base class)中定义派生类(Derived class),使派生类在继承基类数据和功能的基础上定义新的数据和功能，使基类的数据和功能在派生类中得到继承。
- 用继承关系对事物（概念）进行层次分类
- 支持软件复用和增量开发

派生（继承）的概念


Window
MenuWindow
DialogWindow
EditWindow
QuesDialogWindow
研究生
教师
在职研究生

派生（继承）的概念

- 继承是一种is-a关系，从一个基类继承叫做单继承，从多个基类继承叫多继承。
- 另一种表示部分与整体的关系叫has-a，即组合关系。如一个飞机由各种部件组成。学生概念包含姓名，性别等概念等。
- 组合关系表现在部分是作为整体的成员。即类定义中包含多个不同对象。

继承-派生类

class <派生类>: [继承方式] <基类1> , [继承方式] <基类2> ,…
{
};

```
class D:public B
{

};
```

```
class D: public B,public C
{

};
```

例子

```
#include <cstdio>
class B {
  public:
      int ia;
      void print_ia(){ printf("ia = %d\n", ia);}
};

class A: public B {
  public:
      float fb;
      void print_fb(){ printf("fb = %f\n", fb);}
};
```

例子

```
main(){
    A b;
    b.ia = 1;
    b.fb = 2.0;
    b.print_ia();
    b.print_fb();
}
```

派生类-访问控制

- 继承方式有public、protected、private。默认是private继承。
- 继承方式规定了那些成员可以被派生类使用，以及在派生类的访问属性。

```
class B{
    int n;
};
```

```
class D: B{
    int f() {return n;}
};
```

派生类-访问控制

```
class D: B{
    int f() {return n;}
};
class D: public B{
    int f() {return n;}
};
class D: protected B{
    int f() {return n;}
};
```

派生类-访问控制

- public继承：

  基类的public成员，在派生类中仍然为public;

  基类的protected成员，在派生类中仍然为protected;

  基类的private成员，在派生类中不可访问。

不可访问是否等于不存在？-否

派生类-访问控制

- protected继承：

  基类的public成员，在派生类中仍然为protected;

  基类的protected成员，在派生类中仍然为private成员;

  基类的private成员，在派生类中不可访问。

  不可访问是否等于不存在？-否

派生类-访问控制

- Private继承：

  基类的public成员，在派生类中仍然为private;

  基类的protected成员，在派生类中仍然为private;

  基类的private成员，在派生类中不可访问。

不可访问是否等于不存在？-否

派生类-无关系的雇员经理

```
struct Employee {
    string _name;
    Date hiring_date;
    short department;
   // ...
};
```

```
struct Manager {
   Employee emp ;
  set<Employee*> group;
  short level;
  // ...
};
```

- 编译器不知道经理是一个雇员！

派生类-无关系的雇员经理

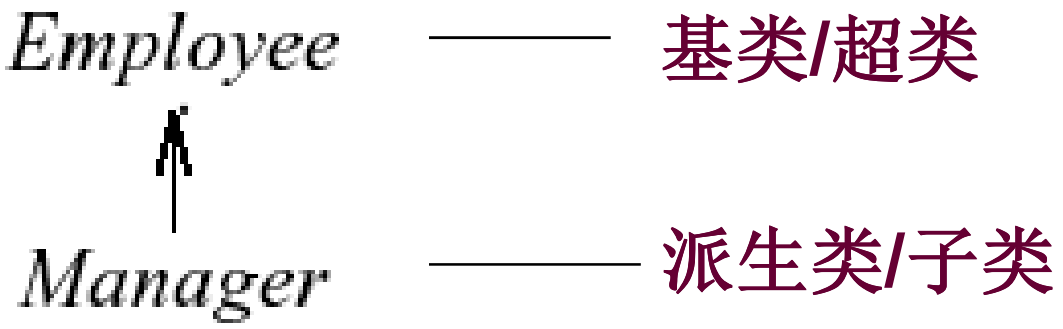
- 如何定义一个包含经理和雇员的数组？

  Employee Epes[10];

  Manager Mngs[10];

派生类-经理也是一个雇员

```
struct Manager : public Employee {
    set<Employee*> group;
    short level;
    // ...
};
```



派生类-经理也是一个雇员

- 因此，经理可以被当作雇员使用。

  可以用指向基类的指针(或引用)指向(或引用)派生类的对象

```
void f( Manager m1, Employee e1){
    vector<Employee*> eVec;
    eVec.push_front(&m1) ;
    eVec.push_front(&e1) ;
    // ...
}
```

派生类-经理也是一个雇员

- 一个*Manager** 可当作一个 *Employee**

```
void f( Manager m1, Employee e1){
    vector<Employee*> eVec;
    eVec.push_front(&m1) ;
    eVec.push_front(&e1) ;
    // ...
}
```

派生类-经理也是一个雇员

- *一个Manager* 可当作一个 Employee**。*
- 即一个派生类指针可以被当作一个基类指针使用
- 如果要将一个基类指针当作一个派生类指针使用，则需要显式的类型转换。但如果指向的实际是一个基类对象，则这样做的后果无法预料。

派生类-经理也是一个雇员

```
void g(Manager mm, Employee ee){
  Employee* pe = &mm;
                    // ok: every Manager is an Employee
  Manager* pm = &ee;
                  // error: not every Employee is a Manager
  pm->level = 2;    // disaster: ee doesn't have a 'level'
  pm = static_cast<Manager*>(pe) ;
                     // brute force: works because pe points
                      // to the Manager mm
  pm->level = 2;      // fine: pm points to the Manager
                        // mm that has a 'level'
}
```

派生类-经理也是一个雇员

- 基类必须在派生类之前已经定义。

```
class Employee; //声明但不是定义
class Manager : public Employee {
                    // error: Employee not defined
    // ...
};
```

派生类-经理也是一个雇员

- 基类必须在派生类之前已经定义或声明。

```
class Employee; //声明但不是定义
class Manager : public Employee {

    // ...
};
class Employee{ //定义了
};
```

派生类-经理也是一个雇员

- 基类必须在派生类之前已经定义。

```
class Employee{ //定义
};
class Manager : public Employee {

    // ...
};
```

派生类-重定义成员函数

- 覆盖了基类的相同成员函数

```
class Employee{
    string _name;
 public:
   void print() const{ cout<<_name<<endl;}
   string name() const{
      return _name;}
};
class Manager : public Employee{
 public:
   void print() const;
};
```

派生类-重定义成员函数

- 派生类不能访问基类的私有成员

```
void Manager: :print() const {
    cout << " name is " << _name << ´\n´;
    // ...
}
```

如果派生类能够访问基类的私有成员,那么程序员只要简单地从一个类X派生新类,就可以任意访问类X的成员, 访问控制有什么用呢?

派生类-构造函数

- 在定义派生类构造函数时，需要调用基类的某个构造函数；如果基类没有默认构造函数，则派生类必须显式调用其中的一个,基类的参数在派生类中也要显式出现。

```
class Employee{
   string _name;
 public:
  Employee(string s);
};
```

```
class Manager : public Employee
{
    int level;
 public:
  Manager(string s,int l);
};
```

派生类-构造函数

```
Manager:: Manager(string s,int l)
    : Employee(s),level(l){
    //…
}
```

派生类-构造函数

- 派生类的构造函数只能描述它自己的成员和其直接基类的初始式，不能去初始化基类的成员。

```
Manager:: Manager(string s,int l)
   : level(l), _name(s)
{
   //…
}
```

派生类-构造函数和析构函数

- 类对象的构造程序是：先基类、后成员、最后类对象本身；而析构的过程与之相反。
- 请单步调试看看是否如此？

```
Manager m;
```

派生类-虚函数与多态性

- 用一个指向基类的指针分别指向基类和派生类对象，并2次调用print()函数输出，结果如何？

```
Manager m(“Zhang”);
Employee e(“Li”),*p = &e;
p->print();
p = &m;
p->print();
```

派生类-虚函数与多态性

- 将基类和派生类中的print都定义成虚函数(*virtual function*)。

```
class Employee{
    string _name;
 public:
   virtual void print() const{ cout<<_name<<endl;}
   string name() const{
      return _name;}
};
```

派生类-虚函数与多态性

- 将基类和派生类中的print都定义成虚函数(*virtual function*)。

```
class Manager {
    int level
 public:
   Manager(string s,int l) : level(l), _name(s){}
   virtual  void print() const{
        cout<<level<<" "; Employee ::print();}
};
```

派生类-虚函数与多态性

- 这种在动态运行时，根据指针或引用指向的实际对象类型调用对应虚函数的行为称为多态性。
- 1）只有类成员才能是虚函数
- 2）静态函数不能是虚函数
- 3）构造函数不能是虚函数,构造函数内调用虚函数也是无效的
- 4）析构函数可以(通常)是虚函数

派生类-虚函数与多态性

- 动态绑定：根据运行时指针或引用指向的对象实际类型调用虚函数的过程
- 静态绑定：在编译期间，根据对象（包括指针或引用）的类型确定调用函数的过程。
- 非虚成员函数当然不存在所谓的“动态绑定”，必然是静态绑定。

派生类-纯虚函数与抽象基类

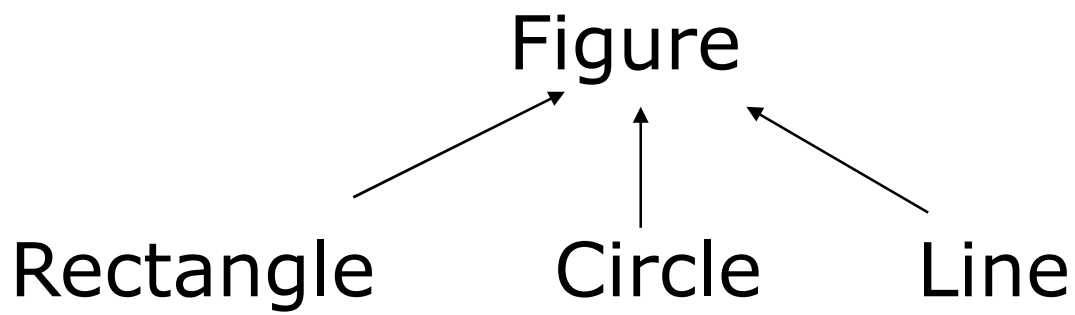
- 纯虚函数是只给出声明而未给出实现的虚函数。
- 纯虚函数在函数声明后跟”=0”

```
class A{…
  public:
    virtual int f()=0;
};
```

- 包含纯虚函数的类称为抽象基类

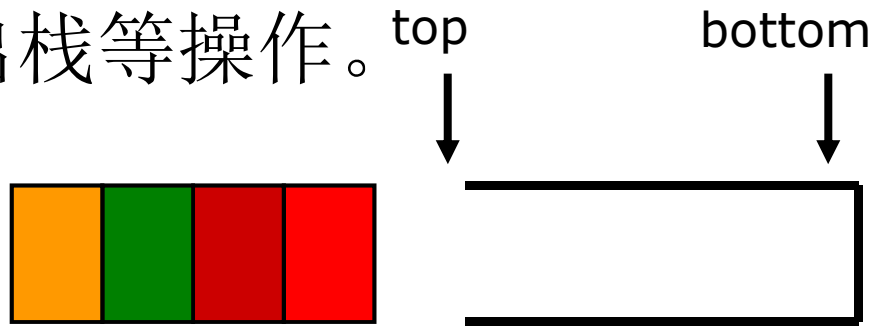
派生类-纯虚函数与抽象基类

- 抽象基类主要作用是为派生类提供一个基本框架和一个公共的接口。派生类必须对抽象基类的每个纯虚函数进行实现。
- 如可定义一个抽象基类Figure描述图形对象，它包含一个draw()的纯虚函数。



派生类-纯虚函数与抽象基类

- 堆栈类型：入栈、出栈等操作。



ADT Stack{

线形关系的一组数据

操作:

bool Push(e); //入栈

bool Pop(&e); //出栈

bool Top(&e); //取栈顶

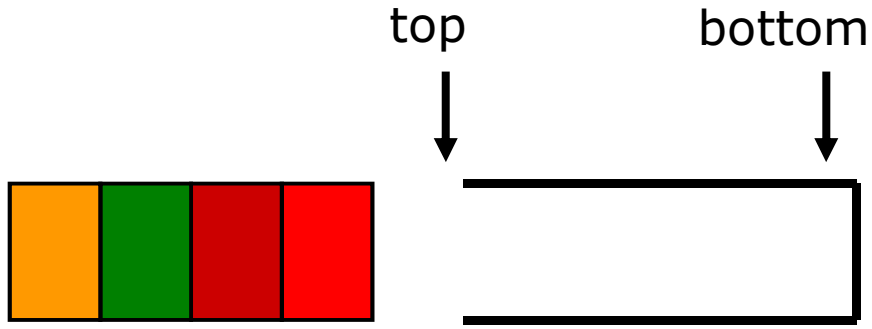
bool IsEmpty(); //空吗?

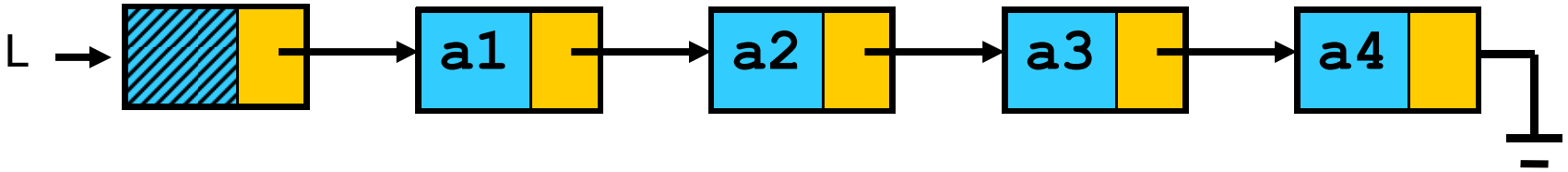
bool Clear(); //清空

}

派生类-纯虚函数与抽象基类

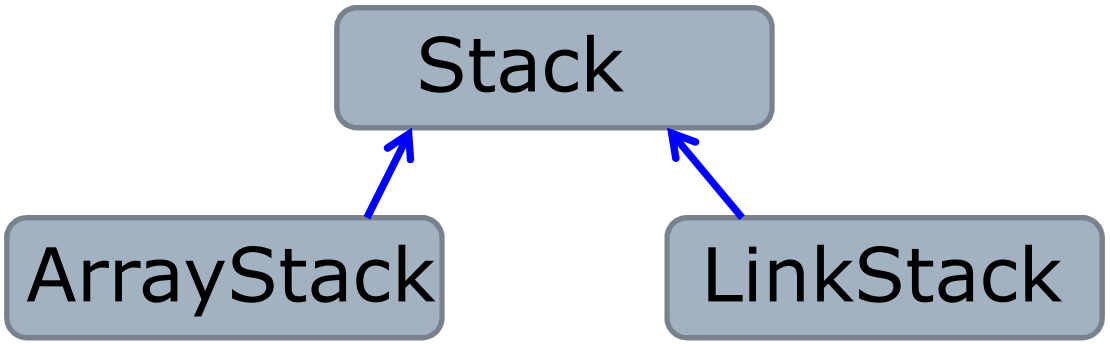
- 堆栈实现方式：数组、链表。





派生类-纯虚函数与抽象基类

- 数组表示的栈和链表表示的栈内部表示和实现不同，但具有共同抽象的性质：入栈、出栈、栈空吗？
- 因此可以定义抽象栈类Stack,作为共同的接口，从Stack派生出具体的链式栈LinkStack和数组栈ArrayStack



派生类-纯虚函数与抽象基类

```
class Stack{
 public:
    virtual bool push(int i)=0;
    virtual bool pop()=0;
    virtual int top()=0;
    virtual bool empty()=0;
};
```

派生类-纯虚函数与抽象基类

```
class ArrayStack: public Stack{
    int* data; int top,
 public:
    virtual bool push(int i);
    virtual bool pop();
    virtual int top();
    virtual bool empty();
};
```

派生类-纯虚函数与抽象基类

```
class LinkStack : public Stack{
    struct LNode{
        int data;     LNode*next;
    }* first;
 public:
    virtual bool push(int i);
    virtual bool pop();
    virtual int top();
    virtual bool empty();
};
```

派生类-纯虚函数与抽象基类

- 假设实现好了ArrayStack和LinkStack，现在测试一下。

```
void main(){
    ArrayStack aS; LinkStack lS; int i;
     while(cin>>i){
       aS.push(4); lS.push(4);
     }
     while(!aS.empty()){
       i = aS.top(4); aS.pop();   cout<<i<<" ";
       i = aS.top(4); aS.pop();   cout<<i<<"\n";
   }
}
```

派生类-多继承

- 从多个基类定义派生类时，会出现名字冲突问题。加入A、B类中都含有函数f

```
class C:public A,public B{
 void fun(){…;f();…}
};
```

- 解决方法是采用名字限定。

```
class C:public A,public B{
 void fun(){…;A::f();…}
};
```

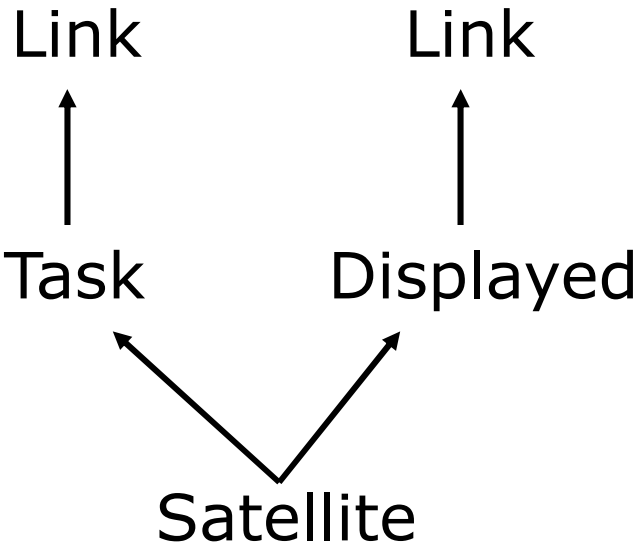
重复基类

```
struct Link{
    Link *next;
};
class Task:public Link{
    //这个Link用于维护Task链表
};
class Displayed:public Link{
    //这个Link用于维护Displayed链表
};
class Satellite:public Task,public Displayed {
};
```

重复基类

- Satellite将包含两个相互独立的Link

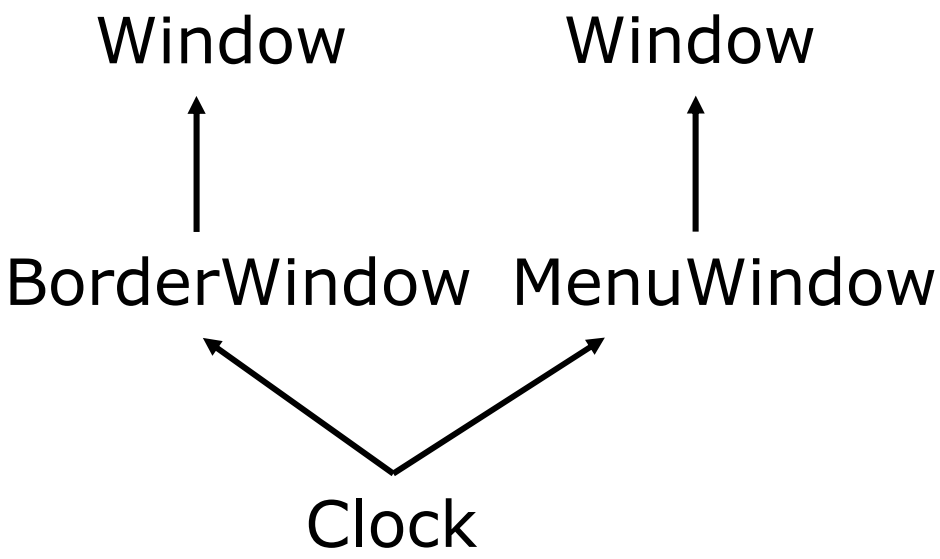
```
void f(Satellite *p){
    p->next = 0; //错
    p->Link::next = 0; //错
    p->Task::next = 0; //ok
    p->Displayed::next = 0; //ok
}
```



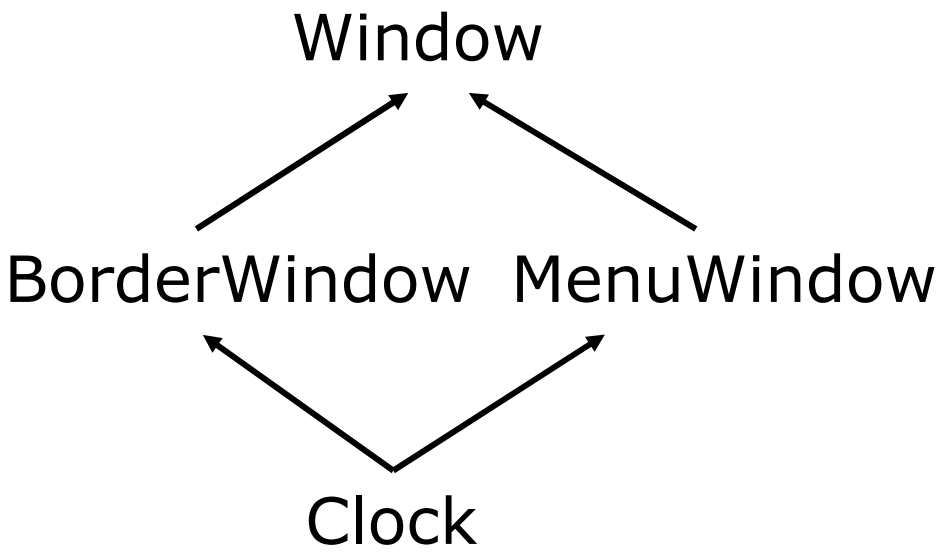
虚基类

- 有时我们不希望公共基类在派生类的对象中产生多个独立的基类对象拷贝。
- 比如从类A派生出类B,C,而从B,C派生出类D,则类D的对象将包含两个A类对象。如上面的重复基类的例子一样。而我们有时不希望如此。

虚基类



虚基类



虚基类

```
struct Window{
    Link *next;
};
class BorderWindow :public virtual Window{
    //这个Link用于维护Task链表
};
class MenuWindow :public virtual Window{
    //这个Link用于维护Displayed链表
};
class Clock:public BorderWindow, public MenuWindow {
};
```

类作为模块化

- 在面向对象设计中，系统是由相互通讯的对象完成的，而对象是类的实例。因此面向对象设计系统的方法是定义系统中有哪些类？类之间构成什么样的派生（继承）或组合关系？类的通讯接口是什么？...
- 面向对象设计中，可能包含许多类，可以有条理组织，如一个类或多个类构成一个相对独立的模块，类的声明和定义放在一个头文件（.h）中，而类的实现则放在源文件中（.cpp,.cxx等）

练习

- 实现雇员、秘书、经理的信息管理程序. 秘书、经理从雇员派生，它们都有一个虚拟函数print输出自身的信息。程序从键盘读入雇员、秘书或经理信息，将它们存储在一个vector或list容器中，并在屏幕上按照输入的程序输出各个员工的信息.

练习

- 实现一个下列类型图书的图书信息管理程序

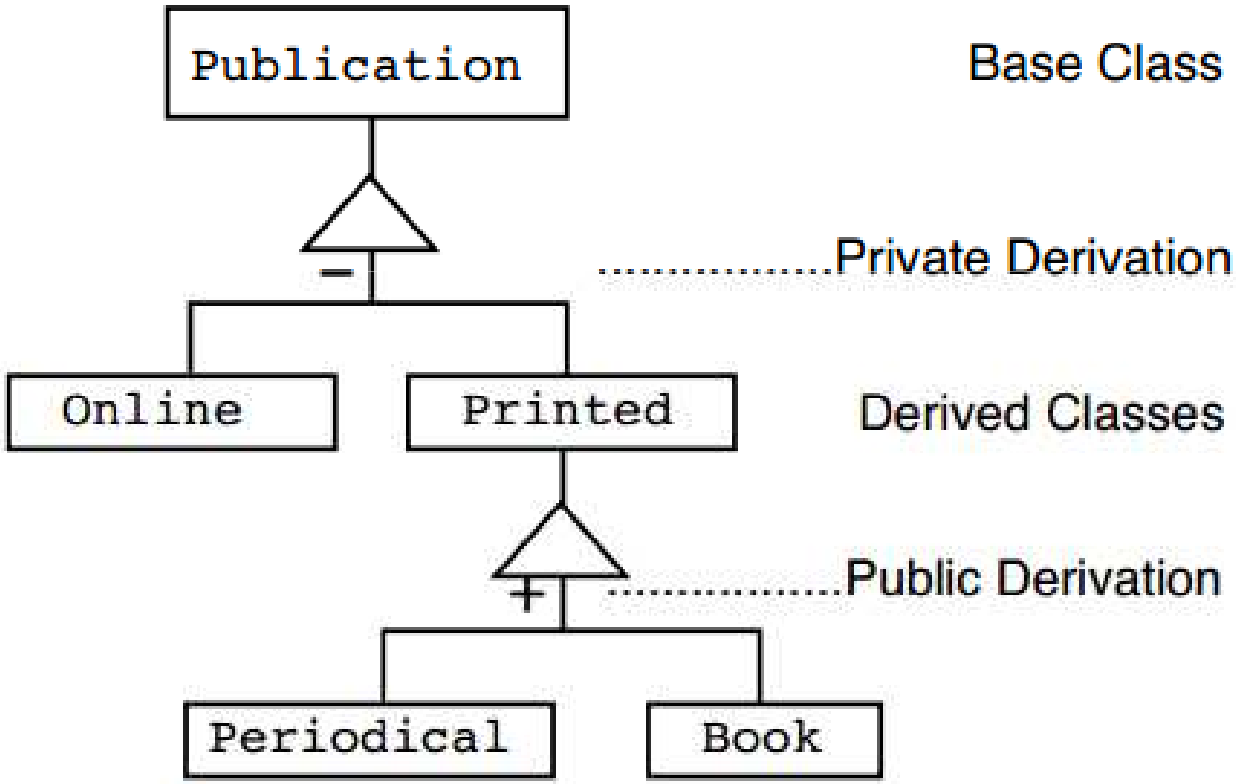


Figure: UML diagram for private and public inheritance.

练习

- 距离说明定义派生类的下列用处

1. To add functions to those defined by an existing class.
2. To extend and specialize the actions of a function defined in an existing class.
3. To mask a function in an existing class and prevent further access to it.
4. To create a different interface for an existing class.
5. To add data members to those included by an existing class.
6. To further restrict the protection level of data members that belong to an existing class.