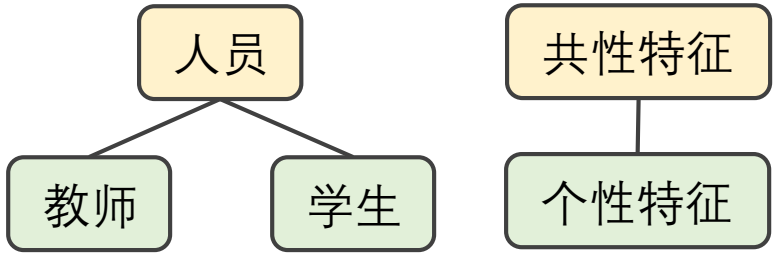
东南大学吴健雄学院
CHIEN-SHIUNG WU COLLEGE OF SEU

C++程序设计

——继承与多态

面向对象程序设计的三大技术特征



- 封装（encapsulation）
- 继承（inheritance）—— 共性到个性分层认识，代码重用，简化设计
- 多态（polymorphism）—— 同一接口实现不同功能，通用编程

内容

继承派生的概念及应用

虚函数与多态

关于通用性编程（讨论）

继承派生的概念与应用

- 本科生
- 研究生
- 在职研究生
- 专职教师
- 行政人员
- 外聘人员

如何构造不同、但有共性的对象？

继承的思想：从已有的对象出发，描述新的、但有共性的对象。

新的对象：继承共性，只描述特性。

【例8.1】人员管理对象的构造。

```
class Person {    //基类：人员基本信息
        string Id;
        string Name;
        char Gender;
public:
        //成员略
};
```

几个名词：

- 类派生（class derivation）技术
- 基类（base class）或超类（superclass）
- 派生类（derived class）或子类（subclass）

```
class Student : public Person {   //派生类：学生信息
        string NoStu;     //学号
        course cs[40];    //40门课成绩
public:
        //成员略
};
```

```
struct course
{       string courseName;
        int grade;
};
```

Student对象中有哪些成员？

派生类成员

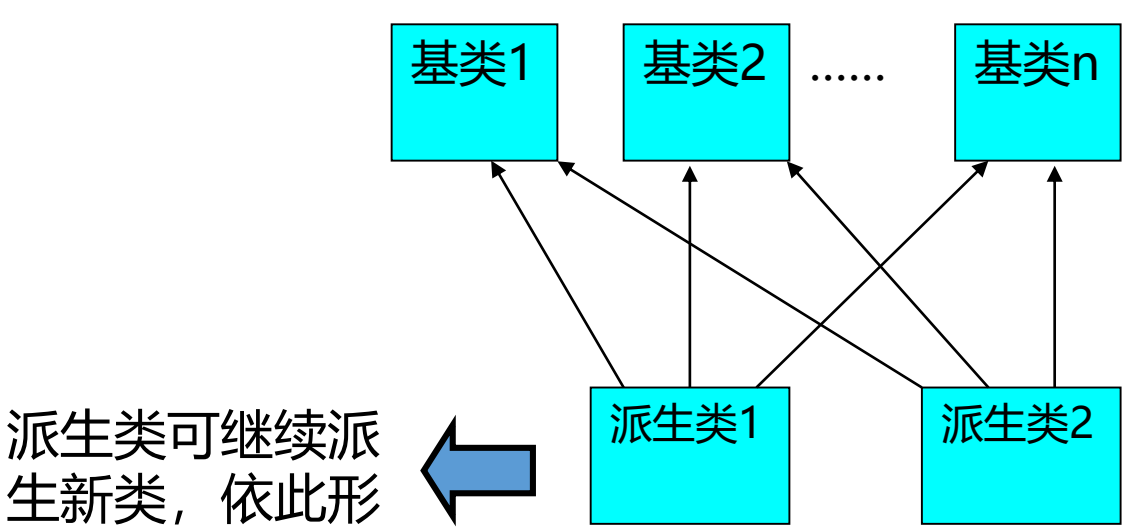
- 继承基类成员 ——除构造析构外，接收基类成员，派生类中不再重复描述
- 改造基类成员 ——可用与基类成员同名的成员重构（override，覆盖）
- 发展新成员 ——定义新成员，使派生类得以具有个性

重写构造函数与析构函数（构造析构函数不会被继承）

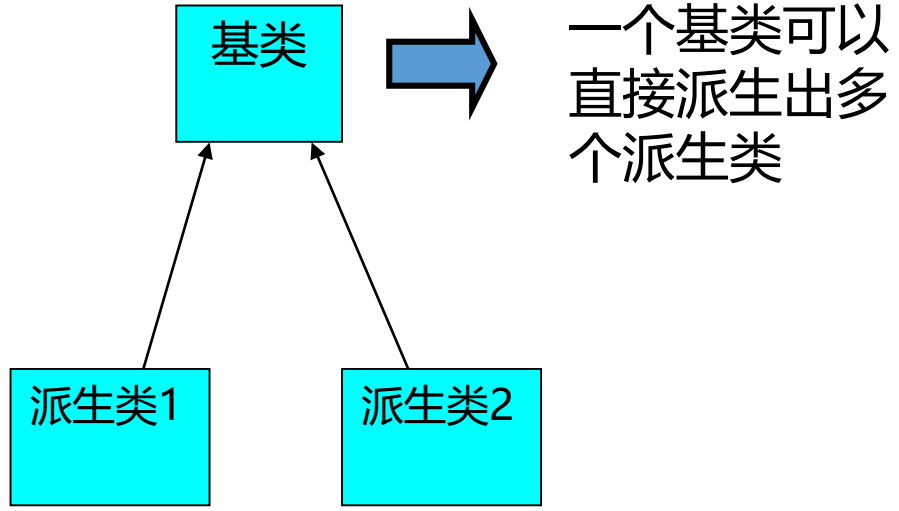
成员重构（override,覆盖）不同于函数重载（overload）

继承派生的方式

多重继承（multiple-inheritance）与单一继承 （single-inheritance）



（a）多重继承



（b）单继承

公有（public）派生和私有（private）派生

派生方式	基类中的访问限定	基类成员在派生类中访问权限	派生类外对基类成员的访问
公有派生	public	public	可以直接访问
	protected	protected	不可直接访问
	private	private	不可直接访问
私有派生	public	private	不可直接访问
	protected	private	不可直接访问
	private	private	不可直接访问

派生类可直接访问基类中的公有及保护成员

派生类的构造函数

派生类对象成员包括：继承成员 + 新增成员。

对于如下派生体系：

```
class A{ };     class B{ };
class C : public A, public B  {   };
```

派生类构造函数形式为：

```
C :: C（C类中所有需初始化的成员对应的参数）：A（参数表1），B(参数表2)
{
      //派生类新增成员的初始化
}
```

初始化顺序：先调用基类构造函数初始化继承成员，再初始化新增成员。

析构顺序：与构造相反

- 先析构派生类新增成员；
- 再调用基类构造析构函数析构继承成员。

一般派生类不用定义析构函数，除非新增指针成员。

【例8.1】人员类公有派生学生，学生类再派生研究生类。

```
class Person                    //基类
{     string IdPerson ;         //身份证
      string Name;
      char Sex;                 //性别
public:
      Person(string =“00000000”, string =“#”, char =‘#’);
      void SetName(string);
      string GetName() {  return Name;  }
      void SetSex(char sex ) {  Sex = sex;  }
      char GetSex() {   return Sex;  }
      void SetId(string id) {   IdPerson = id;   }
      string GetId() {   return IdPerson;  }
      void print();
};
```

```
class Student : public Person    //派生学生类
{     string StuNo;              //学号
      course cs[40];             //40门课程成绩
      int num;                   //已录入课程门数
public:
      Student(string =“00000000”, string =“#”, char =‘#’, string =“0000”);
      bool SetCourse(string, int);      //修改或录入某门课成绩
      int GetGrade(string);             //读某课程成绩
      void print();
};

Student :: Student(string id, string name, char sex, string stuno) : Person(id, name, sex)
{     StuNo = stuno;
      for(int i = 0; i < 40; i++)
      {     cs[i].coursename =“#”;    cs[i].grade = 0;    }
      num = 0;
}
```

```
int Student :: SetCourse(string cname,  int cgrade)
{    for(int i = 0; i < num; i++)
     {   if (cs[i].coursename == cname)     //找到该课,修改成绩
         {   cs[i].grade = cgrade;
             break;
         }
     }
     if (i >= num && num < 40)  //尚未录入此课程，且成绩表未满
     {     cs[i].coursename = cname;
           cs[i].grade = cgrade;
           num++;                          //增加一门课成绩信息
     }
     if (num >= 40)  cout << "成绩表已满，无法录入！" ;
}
```

```
int Student :: GetGrade(string  cname)
{     for(int i = 0; i  < num; i++)
      {    if (cs[i].coursename  ==  cname)
                 break;
      }
      if (i < m)    return cs[i].grade;     else  return   -1;
}

void Student :: print()
{     Person :: print();                    //调用基类函数打印继承成员
      cout << '\t' << NoStudent;
      for(int i = 0; i < num; i++)
        cout << cs[i].coursename << '\t' << cs[i].grade << endl;
}
```

```
class Gstudent : public Student        //派生研究生类
{     int  TeachGrade;                 //助教成绩
public:
      GStudent(string = “00000000”, string = “#”, char = ‘m’, string = “0000”, int = 0);
      void SetTGrade(int tgrade)  {    TeachGrade = tgrade;   }      //修改助教成绩
      int GetTGrade() {      return TeachGrade;      }                 //读助教成绩
      void print() {    Student :: print();       cout << ‘\t’ << TeachGrade;    }
};

Gstudent :: GStudent(string id, string name, char sex, string stuno, int tgrade)
 : Student (id, name, sex, stuno)       //直接基类
{     TeachGrade = tgrade;     }
```

（多重）派生中的同名问题

```
class A {    public: void  print();     };
class B {    public: void  print();     };
class C : public A, public B
 {     public:  void print();     };
```

C类中“有”3个print，若有：C c;

解决同名冲突的办法——

```
c. A::print();    c .B::print();
c.print();      //调用的是哪个?
```

一种特殊冲突的解决——虚继承(虚基类)*

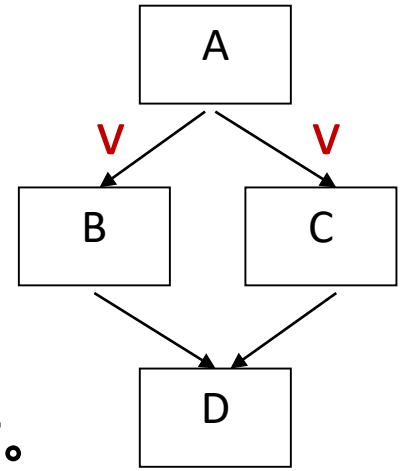
```
class Person
{    string  Name ;
public:
    //成员略
};

class Student : public Person {};        //学生
class Teacher : public Person {};        //教师
class Tstudent : public Teacher, public Student { };     //在职学生
```

Tstudent的成员出现什么问题？ ——Tstudent中有2个独立的Name成员

什么原因？ 这个同名冲突作用域运算符是否能够解决？ ——不能

问题模型——“近亲繁殖”

解决办法——分支时用虚继承（虚基类）（virtual base class）




```
class B : virtual  public A{...};
class C : virtual  public A{...};
```

派生类不是直接继承基类成员，而是产生指向基类成员的指针。

含有虚基类的派生体系，派生类构造函数必须单独调用虚基类构造函数。

```
class D : public B, public C
{     //成员略
public:
      D (……) : B(…), C(…), A(…)  {  …  }          //A是非直接基类
};
```

使用虚基类时派生类对象的构造顺序——

调用虚基类构造函数初始化虚拟继承的成员；

按声明顺序调用非虚基类构造函数初始化基类成员；

按定义顺序调用对象成员构造函数初始化对象成员；

初始化派生类特有成员。

派生类与基类的赋值兼容

派生类对象赋值给基类的对象。反之不行。如：

```
Person p;   Student s;   p=s;
```

基类指针指向派生类对象。反之不行。如：

```
Person *pointP;    pointP = &s;
```

派生类对象取基类别名。反之不行。如：

```
Person &refP = s;    //常用于函数的参数传递，如派生类拷贝构造
```

以上赋值后都只能访问到继承成员。

Q&A

回顾上节

- 继承（派生）：利用已有的类构造新的类，代码重用。
- 利用已有的类构造新的类的不同方法：
  - 继承（共性与个性，派生体系）
  - 聚合（整体和部分）
- Protected
- 派生类中实际拥有的成员
- 派生类构造函数写法：关心直接基类

  E : :E （参数总表）：C（参数表），D（参数表）{ E 成员初始化；}
- 赋值兼容的3种情况：在多态中的应用
- 理解虚拟继承（虚基类）的概念，注意虚拟派生的类再共同派生类，构造函数写法（直接基类，共同祖类）

虚函数与多态

多态的思想：同一接口（函数调用）实现不同功能。看一例：

```
class Person            //基类
{
    string Name;
public:
    Person(string s) {   Name=s;   }
    void Print() {    cout << “姓名：“ << Name << '\t';    }
};
```

```
class Student : public Person      //学生
{     string idNo;          //学号
      double score;         //成绩
public:
      Student(string id, string name, double s)
      : Person(name)
     {    idNo = id;   score=s;     }
     void Print()
    {     Person :: Print();
          cout << "学号:“ << idNo << '\t‘
          << "成绩:“ << score << '\t';
    }
};
```

```
class Teacher : public Person  //教师
{     string idNo;          //职编
      double salary;        //工资
public:
      Teacher(string id, string name, double s)
      : Person(name)
      {    idNo=id;      salary=s;    }
      void Print()
      {    Person::Print();
           cout << "工号:“ << idNo << '\t‘
           << "工资:“ << salary << '\t';
      }
};
```

```
int main()
{   Student s("61519101", "Jiang", 100);
    Teacher t("4001", "Wang",10000);
    Person *p;
    p = &s;
    p->Print(); cout << endl;
    p = &t;
    p->Print(); cout << endl;
    return 0;
}
```

或者，定义函数：

```
void printObj1 (Person& p)
{   p.Print();  }

void printObj2 (Person* p)
{   p->Print();  }
```

调用：

```
printObj1(s);     printObj1(t);
printObj2(&s);    printObj2(&t);
```

以上结果是什么？为什么？　　如何调整可以实现打印出不同对象？

——基类使用虚函数

```
virtual void Person :: Print()  {   cout << "姓名：" << Name << '\t';   }
```

虚函数与多态的实现

虚函数是类的成员函数，是实现多态的条件，定义格式：

virtual 成员函数；

语法提示：

- 声明时指出，类外定义时不加virtual。
- 虚函数在所有派生类中均保持虚函数的特征。
- 派生类中对虚函数的超载或覆盖（override）,要求函数头完全相同。

【例8.6修改】由本科生课程派生研究生课程，成员组成相同，只是学时学分的计算方式不同（从合理性角度对代码有修改，与教材不完全相同）。

```
class Ucourse
{    string cname;              //课程名
     int chr;                   //学时
     int credit;                //学分,由学时折算
public:
     UCourse(string name = "#" , int hour = 0) {   cname = name;  chr = hour; credit = calCredit();   }
     virtual double calCredit() {     return chr / 16;   }      //与教材例不同
     void SetCredit() {    credit = calCredit();     }          //计算学分
     void SetCourse(string name, int  hour) {     cname = name;   chr = hour;    }   //修改课名学时
     int GetHour() {   return chr;   }                          //读学时数
     double GetCredit() {     return credit;     }             //读学分数
     void Print() ;                                             //输出课程信息
};
```

```
class GCourse: public Ucourse
{
public:
      GCourse(string name = “#”, int hour = 0) : UCourse(name, hour)  {  setCredit();  }
      double calCredit() {     return GetHour() / 20;     }
};
void main() //测试方法与教材例不同，去掉了用对象调用
{    UCourse s,   *ps;
     GCourse g;
     ps = &s;
     ps->SetCourse("物理", 48);
     ps->SetCredit();
     ps->print();
     ps = &g;
     ps->SetCourse("英语", 48);
     ps->SetCredit();
     ps->print();
}
```

多态的要点提示

- 实现条件：派生类体系、虚函数、基类指针或引用，三者缺一不可。
- 多态实现通用性的代价是执行速度慢。
- 不存在“虚构造函数”概念，为什么？
- *在派生类体系中，一般应定义虚析构函数。否则，如果派生类对象使用到动态内存空间，当用基类指针指向派生类对象并用基类指针撤销派生类对象时，因调用不了派生类析构函数可能造成内存泄漏。

Q&A

纯虚函数与抽象类

纯虚函数（pure virtual）

—— 虚函数的算法无法实现时，可定义为纯虚函数： virtual 成员函数头 = 0

纯虚函数不能被调用，有待于在派生类中重构（override）。

抽象类（abstract class）——含有纯虚函数的类。

抽象类不能创建对象，是用来定义框架、进行派生的，是多态基类。

【例8.9】用虚函数实现通用定积分。

```
class Integrate
{     double a, b, value;      //上下限、积分结果
      int n;                   //等分数
public:
      virtual double fun(double x)  = 0;          //被积函数
      Integrate (double ra = 0,double rb = 0,int rn = 1000)   //构造
      {     a = ra;   b = rb;   n = rn;    }
      void calculate() {    value = …;       }      //矩形法、梯形法等
      void print();             //代码略
      double getValue() {    return value;    }
};
```

```
class Integ1 : public Integrate {
public:
    A(double ra = 0, double rb = 0, int rn = 1000) : Integrate(ra, rb, rn) { }
    double fun(double x)  {    return sin(x) ;    }        //重构被积函数
};
class Integ2 : public Integrate {
public:
    B(double ra = 0,double rb = 0,int rn = 1000) : integrate(ra, rb, rn) { }
    double fun(double x)  {    return x * x ;    }        //重构被积函数
};
void main()
{   Integ1 integ1 (0, 3.14159/2);   Integ2 integ2(0, 3, 2000);
    Integrate* p;
    p=&integ1;   p->calculate();   p->print();
    p=&integ2;   p->calculate();  p->print();
}
```

虚函数与多态应用的总结

如何确定用不用虚函数？哪些函数定义为虚函数？

——在类派生体系中，基类与派生类中，那些函数功能相同（同样函数头）、但针对不同对象实现方法不同（函数体描述不同的成员函数，应定义为虚函数。

——如果基类的虚函数甚至无法描述出算法，则定义为纯虚函数，相应基类就成为抽象类。

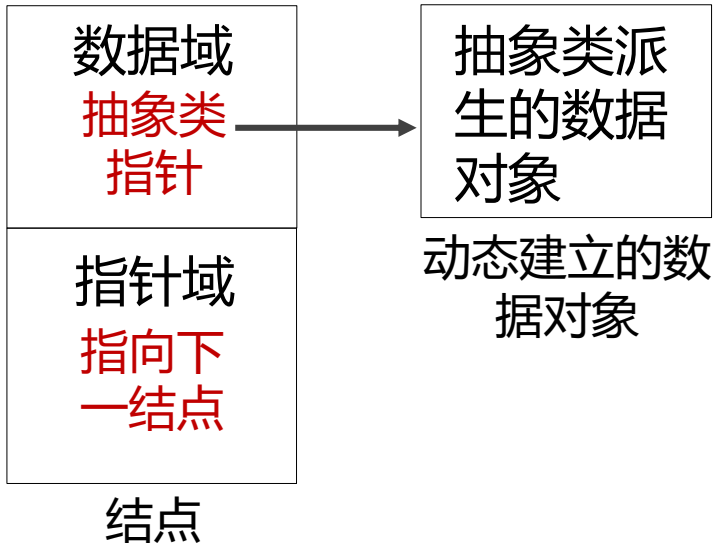
——抽象类的价值就在于定义出类派生体系中的共性操作，为实现多态做好准备。

Q&A

关于通用编程*

多态实现通用链表类——【例8.10】*，基本思路如下：

1. 定义抽象的数据类 —— 重载运算：数据的比较、输出和析构。
2. 定义结点类
3. 定义通用链表类
4. 派生数据类



```
class Object {          //抽象数据类
public:
    Object(){}
    virtual int operator!=(Object &) = 0;     //纯虚函数
    virtual int operator>(Object &) = 0;
    virtual void Print() = 0;
    virtual ~Object() { }
};

class List;
class Node              //节点类
{   Object* pinfo;      //数据域用指针指向数据类对象
    Node* next;         //指针域
public:
    Node() {  pinfo = 0;   next = 0;  }   //生成空结点的构造函数
    void InsertAfter(Node* p);            //在本结点后插入一个结点p
    Node* RemoveAfter();                  //删除本结点的后继结点，返回删除结点
    void Linkinfo(Object* obj) {   pinfo = obj;  }     //类似set函数
    Object* GetInfo() {    return pinfo;  }           //读数据
    friend class List;
};
```

```
class List    //通用链表类
{    Node *head, *tail;
public:
    List() {   head = tail = 0;  }      //空链表无虚结点
    ~List() {   MakeEmpty();  }      //析构函数
    void MakeEmpty();                //清空链表
    Node* Find(Object & obj);    //查找
    int Length();                    //计算单链表长度
    void PrintList();                //打印链表
    void InsertFront(Node* p);  //可生成倒向链表
    void InsertRear(Node* p);   //可生成正向链表
    void InsertOrder(Node* p); //按升序生成链表
    Node* DeleteNode(Node* p);    //删除指定结点
};
```

```
Node* List::Find(Object & obj)        //查找
{    Node* p = head;
     while(p && *(p->GetInfo()) != obj)  p = p->next;
     return p;
}
void List::InsertRear(Node* p)   //生成正向链表
 {    if(head == 0)  head = p;
       else tail->nex t= p;
       tail = p;
}
void MakeEmpty()                          //清空链表
{    Node *p=head;
     head=head->next;
     delete p;
}
```

```
class StringObject : public Object    //派生字符串类
{    string sptr;
public:
     StringObject() {    sptr = “”;    }
     StringObject(string s) {    sptr = s;    }
     int operator != (Object &obj)
    {    StringObject &temp = (StringObject &) obj;
          return (sptr != temp.sptr);
     }
     int operator >(Object &obj);    //代码略
     void Print();
};
```

```
class IntObject:public Object    //派生整数类
{    int data;
public:
     IntObject() {    data = 0;    }
     IntObject(int d) {    data = d;    }
     ~IntObject() {}
     int operator != (Object &obj)
     {    IntObject &temp = (IntObject &) obj;
          return (data != temp.data);
     }
     int operator >(Object &obj); //代码略
     void Print();
};
```

```
void main()
{     Node * p1;
      IntObject* p;
      List list1;
      int a[6] = {3, 2, 6, 4, 5, 7};
      int i;
      for(i = 0; i < 6; i++)
      {     p = new IntObject(a[i]);  //建立数据对象
            p1 =  new Node();         //建立结点
            p1->Linkinfo(p);          //p1节点指向动态数据p，即设置p1节点的数据
            list1.InsertRear(p1);     //正向生成list1
       }
       int x;   cin >> x;
       p = new IntObject(x);          //建立数据对象
       p1 = new Node;
       p1 = Linkinfo(p);              //p1节点指向动态数据p，即设置p1节点的数据

       Node *q = list1.Find(p1);
       if(!q)  cout << "无此元素。"  << endl;
}
```

动态联编——基类指针或引用指向派生类对象，并用该指针调用虚函数。

静态联编——对象名.成员的方式调用虚函数。

Q&A