# 第十三讲

# 继承 与 派生

# 继承/派生 —— 类的继承/派生

- 什么是类的继承/派生
- 怎么定义派生类
- 如何继承父类的成员
- 派生类的构造函数和析构函数
- 类型兼容规则：派生类对象/父类对象
- 多重继承时重复继承问题 — 虚父类

# 什么是继承/派生

**什么是继承/派生：**

—— 在已有的类的基础上定义新的类 ⟶ 类的派生

**为什么继承/派生：**

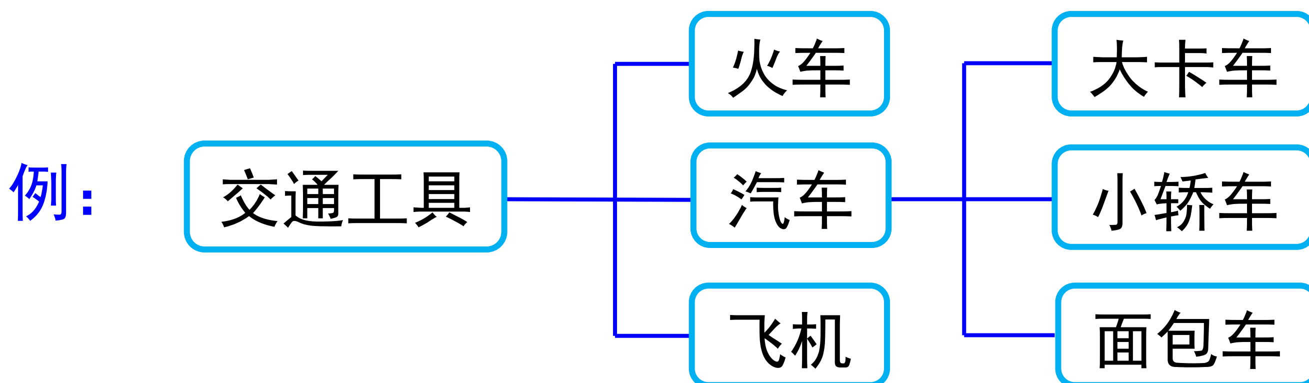
—— 继承可以使得派生类具有父类的各种属性和功能，而不需要再次编写相同的代码。

† 派生类在继承父类的同时，还可以通过重新定义某些属性或改写某些方法，来更新父类的原有属性和功能，或增加新的属性和功能。

# 什么是继承/派生（续）

- 在已有类的基础上产生新类的过程就是类的派生
- 原有类称为父类或基类，新类称为子类或派生类
- 类的继承：派生类继承了父类的特性（数据和函数）
- 派生类可以加入新的特性
- 派生类也可以作为父类，派生新的子类 ⟶ 继承层次结构

例：




† 继承和派生提高了代码的可重用性，有利于软件开发和维护。

# 怎么定义派生类

```
class 派生类名: 继承方式 父类名1, 继承方式 父类名2, ...
{
    派生类成员声明;
};
```

- 一个派生类可以有多个父类（多重继承）
- 单继承：一个派生类只有一个父类
- 一个父类可以派生出多个子类 ⟶ 类族

- ▶ 继承是可传递的：从父类继承的特性可以传递给新的子类
- ▶ 继承方式：规定了如何访问从父类继承的成员
- ▶ 继承方式有三种：public、protected、private
- ▶ 派生类成员：从父类继承的成员+新增加的成员

# 类的派生过程

**派生过程**：吸收父类成员，改造父类成员，添加新成员。

- 吸收父类成员

  派生类包含父类中除构造和析构函数外的所有非静态成员

- 改造父类成员
  - 父类成员的访问控制（通过继承方式实现）
  - 对父类成员的覆盖或屏蔽，即如果新成员与父类成员同名，则只能访问新成员（包括函数，即使形参不一致，也被屏蔽）

- 添加新成员

  根据实际需要，添加新的数据成员或函数成员

† 注意：构造函数、析构函数、静态成员不能被继承!

# 派生类成员的访问控制

**访问控制：** 能否访问/怎样访问从父类继承得来的成员。

- 主要指派生类中新增成员和派生类外部函数访问派生类从父类继承的成员

**继承方式不同，访问控制不同**

**公有继承（public）**

- 父类的公有和保护成员的访问属性保持不变
- 父类的私有成员不可直接访问

† 从父类继承的成员函数对父类成员的访问不受影响！

# 例：公有继承

Example

```
class Point
{
  public:
    void initPoint(float x=0, float y=0);
    void move(float offx, float offy);
    float getx() const {return x;}
    float gety() const {return y;}
  private:
    float x, y;
}
```

Example

```
class Rectangle: public Point
{
  public:
    void initRect(float x,float y,float w,float h)
    { initPoint(x,y);  this->w=w;  this->h=h; }
    float geth() const {return h;}
    float getw() const {return w;}
  private:
    float h, w;  // 新增私有成员
}
```

# 派生类成员的访问控制（续）

## 私有继承（private）

- 父类的公有和保护成员都成为派生类的私有成员
- 父类的私有成员不可直接访问

## 保护继承（protected）

- 父类的公有和保护成员都成为派生类的保护成员
- 父类的私有成员不可直接访问

† 与私有继承的区别：
父类成员（特别是公有函数）可以在以后的派生中作为保护成员继承下去。

† 私有继承后，父类成员（特别是公有函数）无法在以后的派生类中直接发挥作用，相当于终止了父类功能的继续派生。因此，私有继承较少使用。

# 访问控制小结

- 父类成员函数访问父类成员：正常访问
- 派生类成员函数访问派生类新增成员：正常访问
- 父类成员函数访问派生类新增成员：不能访问
- 派生类成员函数访问父类成员：继承方式+成员本身访问属性
- 非成员函数访问派生类所有成员：只能访问公有成员

- 派生类成员按访问属性可划分为下面四类：
  - 不可访问成员：父类的私有成员
  - 私有成员：父类继承的部分成员 + 新增的私有成员
  - 保护成员：父类继承的部分成员 + 新增的保护成员
  - 公有成员：父类继承的部分成员 + 新增的公有成员

† 如果没有指定继承方式，则缺省为 `private`

# 派生类访问控制：例一

```
class Person     // 父类
{
  public:
    Person(string & str, int age ) : name(str)
    {   this->age = age; }

    void show()
    {
        cout << "Name: " << name << endl;
        cout << "Age: " << age << endl;
    }

  private:
    string name;     // 姓名
    int age;       // 年龄
};
```

# 派生类访问控制：例一（续）

```
class Student : public Person    // 派生类
{
  public:
    Student(string & str, int age, int stuid) : Person(str, age)
    {   this->stuid = stuid; }
    void showStu()
    {   this->show();  // 不能直接访问 name 和 age
        cout << "Stuid: " << stuid << endl;
    }
  private:
    int stuid;   // 学号
};
```

ex13_Inheritance_01.cpp

```
int main()
{
    string str="Xi Jiajia";
    Student stu1(str, 18, 20150108);
    stu1.showStu();
}
```

# 派生类访问控制：例二

```
class Person
{
  public:
    Person(string & str, int age) : name(str)
    { this->age = age; }
    void ShowName() { cout << "Name: " << name << endl; }
    void ShowAge() { cout << "Age: " << age << endl; }

  private:
    string name;

  protected:
    int age;
};
```

# 派生类访问控制：例二（续）

```
class Student : public Person    // 派生类
{
  public:
    Student(string & str, int age, int stuid) : Person(str, age)
    { this->stuid = stuid; }
    void ShowStu()
    {    cout << "Name: " << name << endl; // Error: 不能直接访问
          cout << "Age: " << age << endl;  // OK
          cout << "Id: " << stuid << endl;
    }

  private:
        int stuid;
};
```

# 派生类的构造函数

**派生类对象的初始化：**

- 派生类的构造函数只负责新增数据成员的初始化
- 从父类继承的成员需通过调用父类的构造函数进行初始化

- 派生类对象的数据成员：父类成员 + 新增成员
- 初始化：父类成员初始化 + 新增成员初始化
- 数据成员：基本类型数据成员 + 其它类的对象（即内嵌对象）
- 由于父类成员需要调用父类构造函数，因此在派生类构造函数的参数中，有一些参数是传递给父类的构造函数的

```
派生类名(总参数列表)： 父类1(参数),...,父类n(参数),
                       内嵌对象1(参数), ..., 内嵌对象m(参数)
{ 新增数据成员的初始化（不包括继承的父类成员）； }
```

# 派生类的构造函数（续）

- 派生类构造函数的执行顺序：
  - 调用父类的构造函数，按被继承时声明的顺序执行
  - 对派生类新增内嵌对象初始化，按它们在类中声明的顺序
  - 执行派生类的构造函数体的内容

† 若父类使用缺省（即不带形参）构造函数，则可以不写

† 若成员对象使用缺省（即不带形参）构造函数来初始化，也可以不写

† 构造函数的总参数列表中的参数需要带数据类型（形参），其他不需要

# 派生类构造函数举例

```
class B1// 类B1，构造函数有参数
{   public:
        B1(int i) { cout<<"constructing B1 "<<i<<endl; }
};

class B2// 类B2，构造函数有参数
{   public:
        B2(int j) { cout<<"constructing B2 "<<j<<endl; }
};

class B3// 类B3，构造函数无参数
{   public:
        B3() { cout<<"constructing B3 *"<<endl; }
};
```

# 派生类构造函数举例（续）

```
class C: public B2, public B1, public B3  // 派生类，注意父类名的顺序
{
  public:
    C(int a, int b, int c, int d) :
        B1(a), memberB2(d), memberB1(c), B2(b) { x=a; }
        // 初始化列表，如果父类/内嵌对象使用不带形参的构造函数，则可以不写
        // 注意父类/内嵌对象的出现顺序与构造函数调用顺序无关

  private:
    B1 memberB1;
    B2 memberB2;
    B3 memberB3;
    int x;
};

int main()
{
    C obj(1,2,3,4);
}
```

屏幕输出结果：

```
constructing B2 2
constructing B1 1
constructing B3 *
constructing B1 3
constructing B2 4
constructing B3 *
```

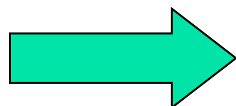
ex13_Inheritance_02.cpp

# 派生类成员的标识与访问

派生类的成员：父类的成员 + 新增成员

派生类成员的标识问题：

如果派生类新增成员与父类成员同名，如何处理？



作用域分辨符 ::

—— 限定要访问的成员所在的类

```
类名::成员名          // 数据成员
类名::成员名(参数)    // 函数成员
```

# 屏蔽规则

如果存在两个或多个具有包含关系的作用域，外层作用域声明的标识符在内层作用域可见，但如果在内层作用域声明了同名标识符，则外层标识符在内层不可见。

- 父类是外层，派生类是内层
- 若在派生类中声明了与父类同名的新函数，即使函数参数表不同，从父类继承的同名函数的所有重载形式都会被屏蔽
- 如何访问被屏蔽的成员：类名+作用域分辨符
- 若派生类有多个父类，且这些父类中有同名标识符，则必须使用作用域分辨符来指定使用哪个父类的标识符！

† 利用作用域分辨符可明确标识从父类继承的成员，从而解决了成员同名问题

# 屏蔽规则举例

```
class Person      // 父类
{
  public:
    Person(string str, int a) : name(str) { age=a; }
    void set(const string& str) { name=str; }
    void set(const int& a) { age=a; }
  protected:
    string name;   int age;
};
```

```
class Student: public Person  // 派生类
{
  public:
    Student(string str, int a, float x): Person(str,a) {score=x}
    void set(const float& x) { score=x; }

  protected:
    float score;
};
```

# 屏蔽规则举例（续）

ex13_Inheritance_03.cpp

```
int main()
{
    Student stu1("Xi Jiajia", 18,  85.5);

    stu1.show();
    cout << endl;

    stu1.set(20);
      // 虽然参数是整型，但父类函数被屏蔽，所以调用的是派生类成员函数
    stu1.show();
    cout << endl;

    stu1.set("Shu Fen");  // Error:  父类函数被屏蔽
    stu1.show();

    return 0;
}
```

# 派生类的复制构造函数

## 派生类复制构造函数的作用机制：

调用父类的复制构造函数完成父类部分的复制，然后再复制派生类部分

Example

```
class C: public B
{
   C(const C &v) : B(v);
   ... ...
};

C::C(const C &v) : B(v)   // 派生类复制构造函数
{
   ... ...
}
```

† 在定义派生类的复制构造函数时，需要为父类的复制构造函数传递参数

# 派生类析构函数

- 派生类的析构函数只负责新增非对象成员的清理工作
- 父类成员的清理工作由父类的析构函数负责
- 新增内嵌对象的清理工作由对象所在类的析构函数负责
- 析构函数的执行顺序（与构造函数相反）：
  - 执行派生类析构函数
  - 执行派生类内嵌对象的析构函数
  - 执行父类的析构函数

# 类型兼容规则/多态

在需要父类对象出现的地方，可以使用派生类（以公有方式继承）的对象来替代。

通俗解释：以公有方式继承的派生类具备了父类的所有功能，凡是父类能解决的问题，公有派生类都可以解决。

类型兼容规则中的替代包括以下情况：

- 派生类的对象可以隐式转化为父类对象
- 派生类的对象可以初始化父类的引用
- 派生类的指针可以隐式转化为父类的指针

† 用派生类对象替代父类对象后，只能使用从父类继承的成员，即派生类只能发挥父类的作用。

# 虚继承

## 为什么虚继承：

在多重继承时，如果派生类的部分或全部父类是从另一个共同父类派生而来，则在最终的派生类中会保留该间接共同父类数据成员的多份同名成员。这时不仅会存在标识符同名问题，还会占用额外的存储空间，同时也增加了访问这些成员的困难，且容易出错。事实上，在很多情况下，我们只需要一个这样的成员副本（特别是函数成员）

## 虚继承：

当某个类的部分或全部父类是从另一个共同父类派生而来时，可以采用虚继承方式，这时从不同路径继承来的同名数据成员在内存中只存放一个副本，同一个函数名也只有一个映射。

# 虚继承方式

```
class 派生类名: virtual 继承方式 父类名
{
    ... ...
};
```

```
class A
{ ... };

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

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

class D : public B, public C
{ ... };
```

# 虚继承时派生类的构造函数

采用虚继承时，在直接或间接继承父类的所有派生类中，都必须在构造函数的初始化列表中列出对父类的初始化。

```
class A
{
  public:
    A(int x);
   ...
};
class B : virtual public A
{
  public:
    B(int x) : A(x);
   ...
};
class C : public B
{
  public:
    C(int x) : A(x), B(x);
   ...
};
```

# 虚继承举例

```
class Person  // 公共父类 Person
{
  public:
    Person(string str, int a)
    { name=str; age=a;}

  protected:  // 保护成员
    string name;
    int age;
};
```

```
class Teacher: virtual public Person  // 虚继承
{
  public:
    Teacher(string str, int a, string tit):Person(str,a)
    { title=tit; }

  protected:
    string title;  // 职称
};
```

# 虚继承举例（续）

```
class Student: virtual public Person  // 虚继承
{
  public:
    Student(string str, int a, float sco)  // 构造函数
      : Person(str,a), score(sco){ }  // 初始化表

  protected:
    float score;  // 成绩
};
```

# 虚继承举例（续）

ex13_Inheritance03.cpp

```
class Graduate: public Teacher, public Student
{
  public:
    Graduate(string str, int a, string tit, float sco, float w)
      : Person(str,a), Teacher(str,a,tit),
        Student(str,a,sco), wage(w){} // 初始化表
   void show()
   {
      cout << "name:" << name << endl;
      cout << "age:" << age << endl;
      cout << "score:" << score << endl;
      cout << "title:" << title << endl;
      cout << "wage:" << wage << endl;
   }

  private:
    float wage;  // 工资
};
```

# 虚继承几点注记

- 一个父类可以在生成某个派生类时采用虚继承，而在生成另一个派生类时不采用虚继承。
- 为了保证父类成员在派生类中只继承一次，建议在该父类在生成所有直接派生类时采用虚继承方式。否则仍然可能会出现对该父类的多重继承。

# 上机作业

1、设计一个名为 **Point** 的类，表示平面坐标下的一个点，这个类包括：

- 两个 **double** 型数据成员：`x, y`，分别表示横坐标和纵坐标
- 一个不带形参的构造函数：`Point()`，用于创建原点 `(0,0)`
- 一个带形参的构造函数：`Point(double x, double y)`
- 成员函数 `double getx()`，返回横坐标
- 成员函数 `double gety()`，返回纵坐标
- 成员函数 `double dist(const Point& p)`，返回当前点与给定点的距离

实现这个类，并在主函数中测试这个类：创建点 `A(0,0)` 和 `B(4,5.6)`，并输出它们之间的距离。程序取名为 **hw13_01.cpp**

2、在 **Point** 类的基础上定义派生类 **Point3D**，表示三维空间的一个点，包括：

- 一个 **double** 型数据成员：`z`，表示 z-坐标
- 一个不带形参的构造函数：`Point3D()`，用于创建原点 `(0,0,0)`
- 一个带形参的构造函数：`Point3D(double x, double y, double z)`
- 成员函数 `double getz()`，返回 z-坐标
- 成员函数 `double dist(const Point3D& p)`，返回当前点与给定点的距离

实现这个类，并在主函数中测试这个类：创建点 `A(0,0,0)` 和 `B(4,5.6,7.8)`，并输出它们之间的距离。程序取名为 **hw13_02.cpp**

# 上机作业

3、派生与继承：虚父类

(a) 设计 **Employee** 类，包括：

- 两个数据成员：`string name` 和 `int age`，分别表示姓名和年龄
- 一个带两个形参的构造函数，用于初始化数据成员
- 成员函数 `void Display()`，输出所有信息（即姓名和年龄）

(b) 在 **Employee** 基础上定义派生类（以虚父类方式）**Saleman**，包括：

- 新增数据成员：`int sales`，表示销售额
- 一个带三个形参的构造函数，用于初始化数据成员
- 成员函数 `void Display()`，输出所有信息（即姓名、年龄和销售额）

(c) 在 **Employee** 基础上定义派生类（以虚父类方式）**Manager**，包括：

- 新增数据成员：`int members`，表示管理的人数
- 一个带三个形参的构造函数，用于初始化数据成员
- 成员函数 `void Display()`，输出所有信息（即姓名、年龄和管理人数）

(d) 在 **Saleman** 和 **Manager** 基础上定义派生类 **SaleManager**，包括：

- 一个带四个形参的构造函数，用于初始化数据成员
- 成员函数 `void Display()`，输出所有信息（即姓名、年龄、管理人数和销售额）

实现这个类，并在主函数中测试这个类。程序取名为 **hw13_03.cpp**