第十三讲　继承与派生

潘建瑜@MATH.ECNU

类的继承与派生

- 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
- 同名成员屏蔽规则
- 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 — 虚继承

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

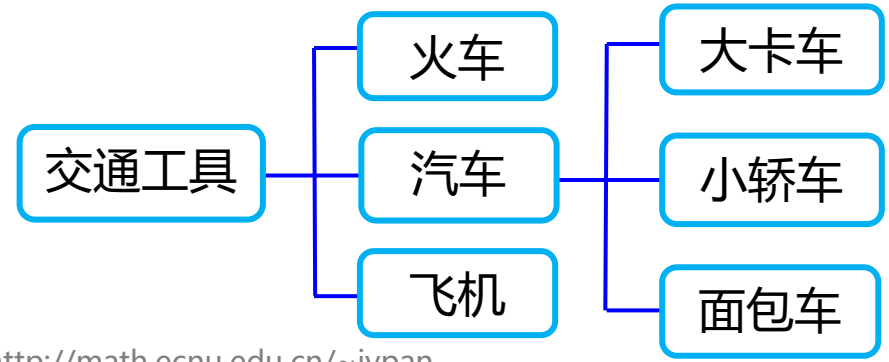
为什么继承/派生

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，提高了代码的可重用性，有利于软件开发和维护。

继承与派生的基本概念

基本概念

- 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



直接基类：如“汽车”是“小轿车”的直接基类

间接基类：如“交通工具”是“小轿车”的间接基类

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，从而形成类族

- 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，如果没有指定，缺省为 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
  - 修改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!

从父类继承的成员的访问属性

访问属性：能否访问/怎样访问 从父类继承得来的成员

即派生类中 新增成员函数 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，访问属性不同

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

注意：从父类继承的成员函数对父类成员的访问不受影响！

举例：公有继承

```
class Point
{  public:
     void initPoint(float x=0, float y=0) { this->x = x; this->y = y; }
     void ShowPoint() { cout << "(" << x << "," << y << ")"; }

   private:
     float x, y;
};
```

ex13_Inheritance_Point01.cpp

```
class Rectangle: public Point
{  public:
     void initRect(float x,float y,float w,float h)
     { initPoint(x,y);  this->w=w;  this->h=h; }  // 不能直接访问父类的私有成员
     void Display()
     { cout << "中心: "; ShowPoint(); cout << ", 宽与高: " << w << ", " << h; }

   private:
     float h, w;  // 新增私有成员
};
```

从父类继承的成员的访问属性

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

- 保护继承与私有继承的区别：父类公有和保护成员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。

举例：保护继承

```
class Point
{  public:
     void initPoint(float x=0, float y=0) { this->x = x; this->y = y; }

   protected:    // 保护型成员
     float x, y;
};
```

ex13_Inheritance_Point02.cpp

```
class Rectangle: protected Point   // 保护方式继承
{  public:
     void initRect(float x,float y,float w,float h)
     {
         this->x = x; this->y = y; // OK, 可以直接访问父类的保护型成员
         this->w = w; this->h = h;
     }

   private:
     float h, w;
};
```

小结

- 父类的成员函数访问父类成员：正常访问
- 派生类的成员函数访问派生类新增成员：正常访问
- 父类的成员函数访问派生类新增成员：不能访问
- 派生类的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

派生类的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

举例：访问属性

```
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
};
```

举例：访问属性

```
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_Person01.cpp

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

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（函数体）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

几点注意

- 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

派生类的构造函数的一般形式

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)
{ 新增基本数据成员的初始化（不包括继承的父类成员和内嵌对象）；}

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）构造函数进行初始化，则可以省略，内嵌对象类似
- 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

示例：派生类构造函数

```
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，构造函数不带参数
{   public:
    B3() { cout<<"constructing B3 *"<<endl; }
};
```

示例：派生类构造函数

ex13_Inheritance_BC01.cpp

```
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初始化，则可以省略
        // 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 *
```


http://math.ecnu.edu.cn/~jypan

示例：派生类构造函数

注意：如果派生类构造函数在类外部定义，则在类内部声明时不要加初始化列表！

```
class C: public B2, public B1, public B3
{
  public:
    C(int a, int b, int c, int d);  // 声明时不要加初始化列表
  . . . . . .
};

C::C(int a, int b, int c, int d) : B1(a), memberB2(d), memberB1(c), B2(b)
{  x = a;  }
```

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

```
class C: public B
{
   C(const C &v);  // 声明时无需加初始化列表
   ... ...
};

C::C(const C &v) : B(v)   // 派生类复制构造函数，定义时加初始化列表
{
   ... ...
}
```

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

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
```

同名成员屏蔽规则

同名成员屏蔽规则

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，则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。

举例：同名成员屏蔽规则

```
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;
};
```

举例：同名成员屏蔽规则

ex13_Inheritance_Person02.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;
}
```

类型兼容规则/多态

类型兼容规则

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，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，这时从不同路径继承来的共同父类的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。如果该父类构造函数不带参数，则可以省略。

```
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);
   ...
};
```

举例：虚继承方式

```
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
};
```

举例：虚继承方式

```
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) {  }  // 初始化列表

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

举例：虚继承方式

```
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
};
```

举例：虚继承方式

ex13_Inheritance_virtual.cpp

```
int main()
{
    Graduate grad1("Xi Jiajia", 24, "assistant", 89.5, 1234.5);
    grad1.show();
    return 0;
}
```

在初始化 Graduate 类的对象时, 只会通过 Graduate 构造函数中 Person(str,a) 来初始化从 Person 类继承的数据成员，Teacher(str,a,tit) 和 Student(str,a,sco) 中对 Person 构造函数的直接调用会被忽略

```
class Graduate: public Teacher, public Student
{ public:
    Graduate(string str, int a, string tit, float sco, float w)
      : Person(str,a), Teacher("AA",2,tit),
        Student("BB",3,sco), wage(w)  { }
           // Teacher 和 Student 中对 Person 构造函数的直接调用被忽略
   ... ...
};
```

虚继承方式

虚继承几点注记

- 一个父类可以在派生某个类时采用虚继承，而在派生另一个类时不采用虚继承
- 为了保证父类成员在派生类中只继承一次，建议在该父类派生其他类时都采用虚继承方式，否则仍然可能会出现对该父类的多重继承

第十三讲上机作业

1、继承与派生

(a) 编写函数 `int gcd(int a, int b)`，利用递归方法（辗转求余法）计算最大公约数。

(b) 设计名为 **Integer** 的类，这个类包括：

- 一个保护型的 `int` 数据成员：`a`
- 一个不带形参的构造函数，用于设置缺省值 `a = 0`
- 一个带形参的构造函数：`Integer(int a)`

(c) 设计名为 **Fraction** 的派生类，代表最简分数，以公有方式继承 `Integer`，这个类包括：

- 一个保护型的 `int` 数据成员：`b`，代表分母，继承的 `a` 代表分子
- 一个不带形参的构造函数，用于设置缺省值 `a = 0, b = 1`
- 一个带形参的构造函数：`Fraction(int a, int b);` (注意分子分母要通分)
- 成员函数 `void Display()`，以 `a/b` 形式输出有理数，如 `2/3, -7/5`
- 以非成员函数方式重载加法“`+`”，实现两个分数的加法（注意分子分母要通分）
  `Fraction operator+(const Fraction & x, const Fraction & y)`

实现这两个类，并在主函数中测试：创建分数 `x=2/3` 和 `y=-1/6`，计算 `z=x+y` 并输出。

（程序取名为 **hw13_01.cpp**）

第十三讲上机作业

2、继承与派生：同名成员屏蔽与类型兼容规则

(a) 设计一个 **Point** 的类，表示平面坐标下的一个点，这个类包括：

- 两个私有型 `double` 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 dist(const Point& p)`，返回当前点（目标对象）与给定点（形参）的距离

(b) 设计一个 **Point3D** 类，表示三维空间的一个点，以公有方式继承 **Point**，这个类包括：

- 一个私有型 `double` 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 dist(const Point3D& p)`，返回当前点与给定点的距离

实现这两个类，并在主函数中测试：

(1) 创建点 `A1(0,0)` 和 `A2(4,5.6)`，并输出它们之间的距离。

(2) 创建点 `B1(0,0,0)` 和 `B2(4,5.6,7.8)`，并输出它们之间的距离。（程序取名为 **hw13_02.cpp**）

第十三讲上机作业

3、派生与继承：虚继承

(a) 设计 **Employee** 类，包括：

- 两个保护型数据成员：`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 `Employee(const string & name, int age)`，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（以虚继承方式）**Saleman**，包括：

- 新增保护型数据成员：`int sales`，表示销售额
- 一个带三个形参的构造函数 `Saleman(const string & name, int age, int sales)`
- 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（以虚继承方式）**Manager**，包括：

- 新增保护型数据成员：`int members`，表示管理的人数
- 一个带三个形参的构造函数 `Manager(const string & name, int age, int members)`
- 成员函数 `void Display()`，输出所有信息（即姓名、年龄和管理人数）

第十三讲上机作业

(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

  `SaleManager(const string & name, int age, int sales, int members)`

- 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）

实现这四个类，并在主函数中测试：创建一个 `SaleManager`，姓名: `Zhang San`, 年龄: `32`, 销售额: `128`, 管理人数: `8`。在屏幕上输出该销售经理的相关信息。（程序取名为 **hw13_03.cpp**）