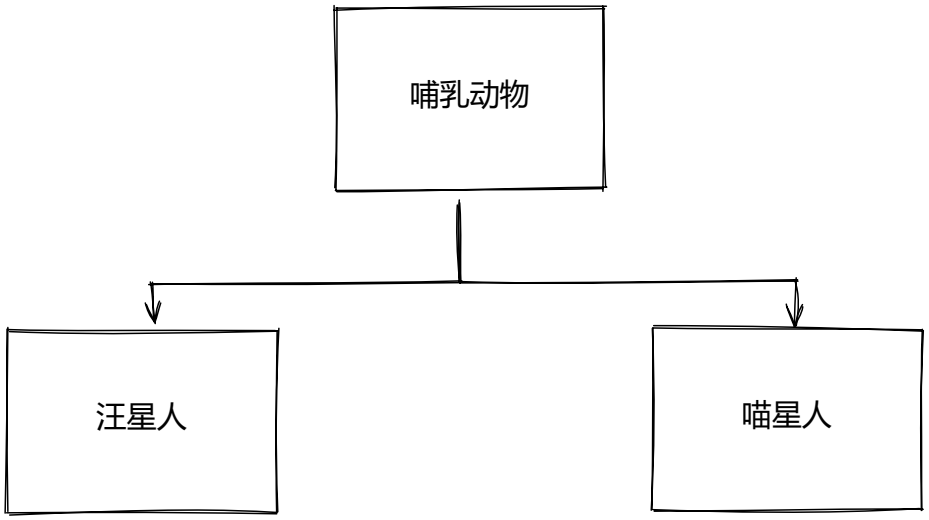
第六讲 继承与派生

继承与派生

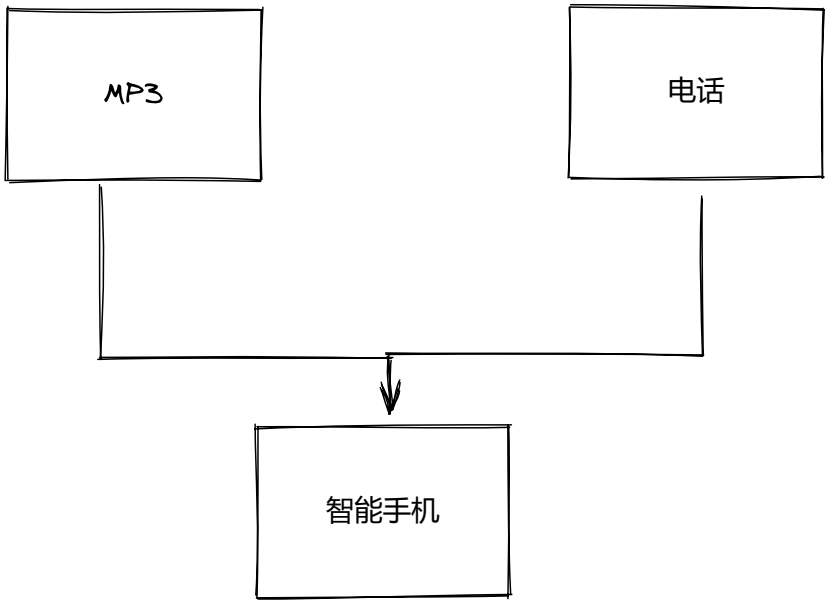
在 C++中，在定义一个新类时，如果该类与某个已有类相似（即新类有已有类的全部特点），则可以让新类继承已有类，从而拥有已有类的属性和行为，并且可以在已有类的基础上修改或扩展新的属性和行为。此时，称已有类为基类（base class）或父类（super class）；称新类为派生类（derived class）或子类（sub class）。
可以说：子类继承了基类，或基类派生出了子类。

多重继承

单向继承：



多继承：



一个派生类也可以有两个或两个以上的基类，这样的继承方式称为多继承或多重继承（multiple inheritance,MI）

派生类的定义和构成

派生类的定义

派生类也是类，因此基本的定义格式与构成和类的定义与构成方式是一致的，二者的区别主要在于如何表达派生类与基类的关系。

派生类定义的语法为：

```
class 派生类名：继承方式1 基类名1，继承方式2 基类名2,…
{
private:
    派生类的私有数据和函数
public:
    派生类的公有数据和函数
protected:
    派生类的保护数据和函数
};
```

➢ 单继承时，基类表中只有一个基类；多继承时，基类表中要列出所有的直接基类，基类之间用逗号分隔。

➢ 继承方式指定了派生类成员和外部对象对其从基类继承的成员的访问方式，继承方式有三种，即公有继承（public）、私有继承（private）和保护继承（protected）。

➢ 基类表中每个基类都按其继承方式继承，若不显示标明某基类的继承方式，则默认该基类为私有继承。

派生类的成员构成

- 继承的基类成员
- 覆盖基类得到的新的成员
- 添加的新成员

继承的方式

总结如下表，记住即可

| 基类 | public | private | protected |

| ---------- | --------- | -- | --------- |

| 公有派生类 | public | 不可访问 | protected |

| 私有派生类 | private | 不可访问 | private |

| 保护派生类 | protected | 不可访问 | protected |

派生类的构造与析构

以类外定义格式为例，派生类构造函数定义的一般格式为：

```
派生类名( 参数总表): 基类名1(参数表1),..., 基类名m (参数表m), 成员对象名1(成员对象参数表1),..., 成员对象名n( 成员对象参数表n)
{
    派生类新增成员的初始化;
}
```

派生类的析构

派生类析构的顺序与构造的顺序正好相反，即在构造过程中先构造的，在析构过程中后析构。

单继承的构造

派生类从基类继承了成员，因此先要对基类成员进行初始化，避免发生未初始化就使用的错误。调用基类构造函数时，有以下两种方式：

显式调用

```
#include <iostream>
using namespace std;
class Base
{
public:
    int n;              //基类成员变量
    Base(int i) : n(i) //基类构造函数
    {
        cout << "基类" << n << "构造" << endl;
    }
};
class Derived : public Base
{
public:
    Derived(int i) : Base(i) //派生类构造函数
    {
        cout <<"派生类构造" << endl;
    }
};
// test code
int main()
{
    Derived Obj(1); //派生类生成对象obj，传参数1
    return 0;
}
// 结果:
// 基类1构造
// 派生类构造
```

隐式调用

在派生类的构造函数中，如果不列出基类构造函数，则派生类构造函数会自动调用基类的默认构造函数，即无参的构造函数，但是如果基类显示定义了带参构造函数后，系统不会给其

分配默认的无参构造函数，此时编译器会报错。

当出现成员对象时，该类的构造函数要包含对成员的初始化。如果构造函数的成员初始化列表没有对成员对象初始化，则使用成员对象的默认构造函数。

```
#include <iostream>
using namespace std;

classA
{
private:
    int nTa;
public:
    A(int ta);
};
class C
{
private:
    int nTc;
public:
    C(int tc);
};
class B : public A
{
private:
    int nTb;
    C obj1, obj2;
public:
    B(int ta, int tc1, int tc2, int tb);
};

B::B(int ta, int tc1, int tc2, int tb) : A(ta), obj1(tc1),
                                         obj2(tc2), nTb(tb)
{
    cout << 123 << endl;
}
// 上例中尽管C没有无参构造函数，但是在调用时进行了初始化操作
```

单继承的构造函数调用顺序

➢ 调用基类构造函数

➢ 调用成员对象的构造函数，并且应该按照其在当前派生类中的定义顺序调用

➢ 调用派生类自己的构造函数

➢ 若继承（派生）层次有多层，则派生类构造函数的调用顺序将从最上层的基类开始，由上而下到最下层的派生类，依次按上面的一般调用顺序进行调用

多继承

多继承（multiple inheritance，MI）是指派生类具有两个或两个以上的直接基类（direct class）

➢ 调用各基类构造函数；各基类构造函数调用顺序按照基类被继承时声明的顺序，从左向右依次进行。

➢ 调用内嵌成员对象的构造函数；成员对象的构造函数调用顺序按照它们在类中定义的顺序依次进行。

➢ 调用派生类的构造函数；

二义性问题

对多继承来说，由于继承自不同的基类，而被继承的基类之间原本没有关联，因此其内部定义时自成体系，没有考虑过是否同名的问题，这时，如果作为派生类用同一个成员标识符去调用，很可能会遇到几个基类都有同名成员可以被调用的情况，这就造成了编译器无法确定要调用到底是哪个成员。这种由于多继承引起的对某类的某成员进行访问时，不能唯一确定的情况，就称为二义性问题。

```
#include <iostream>
using namespace std;
class Car
{
public:
    Car(int p, int s)
    {
        power = p;
        seat = s;
    }
    void Show()
    {
        cout << "汽车动力:" << power << ", 座位数: " << seat << endl;
    }
```

```
private:
    int power; //马力
    int seat;  //座位
};
class Boat
{
public:
    Boat(int c, int s)
    {
        capacity = c;
        seat = s;
    }
    void Show()
    {
        cout << "船载重量：" << capacity << ", 座位数：" << seat <<
endl;
    }
private:
    int capacity; //载重量
    int seat;     //座位
};
//水陆两栖车
class AAmobile : public Car, public Boat
{
public:
    AAmobile(int power, int cseat, int capacity, int bseat) :
Car(power, cseat),

Boat(capacity, bseat) {}
    void ShowAA()
    {
        cout << "水陆两用车：" << endl;
        Car::Show();
        Boat::Show();
    }
};
int main()
{
    Car NormalCar(200, 2);
    Boat NormalBoat(1000, 2);
    AAmobile CoolCar(500, 5, 2000, 4);
    NormalCar.Show();
```

```
    NormalBoat.Show();
    // CoolCar.Show(); //二义性 ❌
    CoolCar.ShowAA();
    return 0;
}
// 结果:
// 汽车动力:200, 座位数: 2
// 船载重量: 1000, 座位数: 2
// 水陆两用车:
// 汽车动力:500, 座位数: 5
// 船载重量: 2000, 座位数: 4
```

类型兼容

类型兼容是指在公有派生的情况下，一个派生类对象可以作为基类的对象来使用的情况。类型兼容又称为类型赋值兼容或类型适应

在 C++中，类型兼容主要指以下三种情况:

➢ 派生类对象可以赋值给基类对象

➢ 派生类对象可以初始化基类的引用，或者说基类引用可以直接引用派生类对象

➢ 派生类对象的地址可以赋给基类指针，或者说基类指针可以指向派生类对象

```
#include <iostream>
using namespace std;
class A
{
private:
    int na;

public:
    A(int na)
    {
        this->na = na;
    }
    void Show()
    {
        cout << na << endl;
    }
};
class B : public A
```

```
{
private:
    int nb;

public:
    B(int na, int nb) : A(na)
    {
        this->nb = nb;
    }
    void Show()
    {
        cout << nb << endl;
    }
};
void Display(A aobj)
{
    aobj.Show();
}
int main()
{
    A a(1);
    B b(200, 200);
    Display(a); // 1
    Display(b); // 200
    return 0;
}
```

✓ C++提供的类型兼容规则使编译器可以自动在派生类和基类之间进行隐式的类型转换，使派生类对象也可以调用成功，无需手动增加重载代码

✓ 可以隐式地将基类和派生类的对象、指针、引用进行类型转换，使得基类的指针（对象\引用）可以访问不同的派生类对象

✓ 只能访问派生类中从基类继承到的成员，不能访问派生类的新增成员

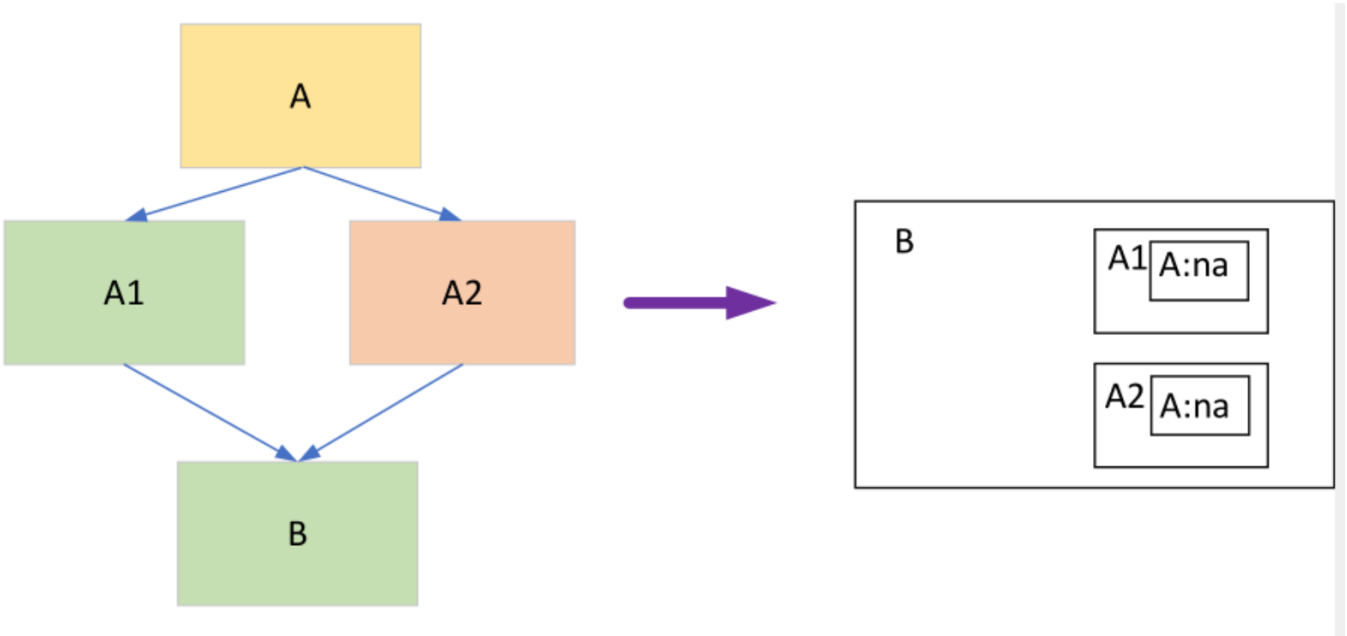
虚基类

如下代码：

```
#include <iostream>
using namespace std;
```

```
//基类A
class A
{
public:
    int na;
    A(int a) { na = a; }
};
//一级派生类
class A1 : public A
{
public:
    A1(int a) : A(a){};
};
class A2 : public A
{
public:
    A2(int a) : A(a){};
};
//二级派生类
class B : public A1, public A2
{
public:
    B(int a) : A1(a), A2(a){};
};
```

的继承关系图为：



如果生成一个 B 的对象 b,通过 b 访问 na 时将会发生间接二义性问题

虚基类的定义

虚基类并不是完全有别于普通类而独立存在的一种类的定义形式，它是在派生类的定义中，用关键字 virtual 对继承方式进行修饰得到的，定义格式如下：

`class 派生类名: virtual 继承方式 基类名`

➢ 在某基类的继承方式前使用 virtual 关键字时，说明该基类为此派生类的虚基类。

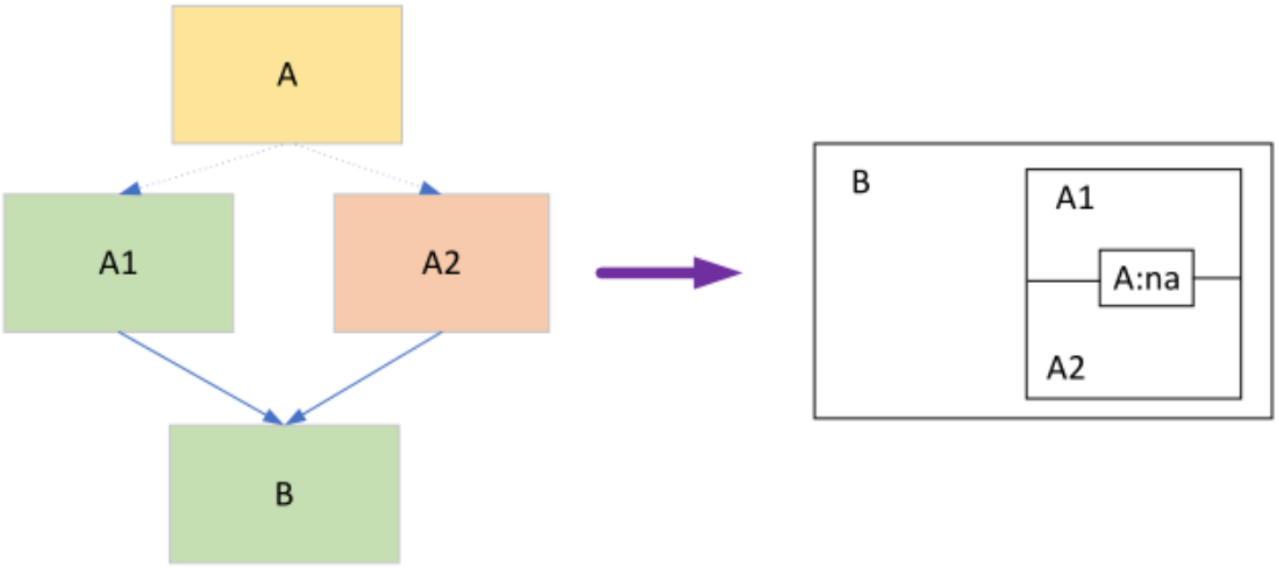
➢ virtual 关键字只限定紧跟其后的一个类。

➢ 派生类声明虚基类后，其效果对后续的派生一直有效。

将代码修改为：

```
// 基类A
class A
{
public:
    int na;
    A(int a = 0) { na = a; }
};
//一级派生类
class A1 : virtual public A
{
public:
    A1(int a) : A(a) {}
};
class A2 : virtual public A
{
public:
    A2(int a) : A(a) {}
};
//二级派生类
class B : public A1, public A2
{
public:
    B(int a) : A1(a), A2(a) {}
};
```

则结构变为：



此时再访问 na，则不会存在二义性问题。

虚基类继承时基类必须有默认构造函数，如果没有则编译出错，因而例10中基类构造函数要设置参数默认值

虚基类的构造

存在虚基类时，对象构造函数的调用顺序与一般情况有所不同，一般主要遵循以下规律：

➢ 虚基类的构造函数在非虚基类构造函数之前调用

➢ 若同一个继承层次中包含多个虚基类，则按照他们的声明次序依次调用虚基类的构造函数。

➢ 若虚基类是有非虚基类派生的，则遵守先调用基类构造函数，再调用派生类构造函数的规则。

```
class A1 {...};
class A2 {...};
class B1 : public A2,virtual public A1 {...};
class B2 : public A2, virtual public A1 {...};
class C : public B1, virtual public B2 {...};
```

C 实例化一个对象，构造函数调用顺序为？

1. 因为C继承虚基类B2，故优先构造B2
    1. 因为B2继承虚基类A1，故优先构造 A1
    2. 因为B2公有继承A2，故构造 A2
    3. B2的基类构造完毕，构造B2本身

2. 因为C继承基类B1，故优先构造B1
    1. 因为B1继承虚基类A1，故构造A1，但A1虚基类构造完毕，故略过构造
    2. 因为B1公有继承A2，故构造A2
    3. B1的基类构造完毕，构造B1本身
3. C的基类构造完毕，构造C本身

    综上所述，顺序为：

    ```
    A1 --> A2 --> B2 --> A2 --> B1 --> C
    ```

    析构顺序与上述顺序正好相反。