《面向对象程序设计》朋辈课程 · 第十一辑

多态与抽象类

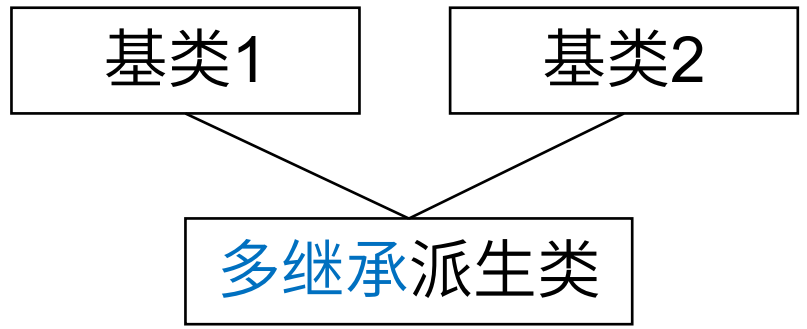
信息学院 赵家宇

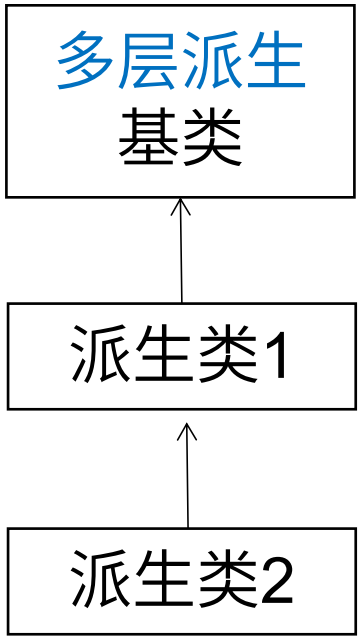
PART 01

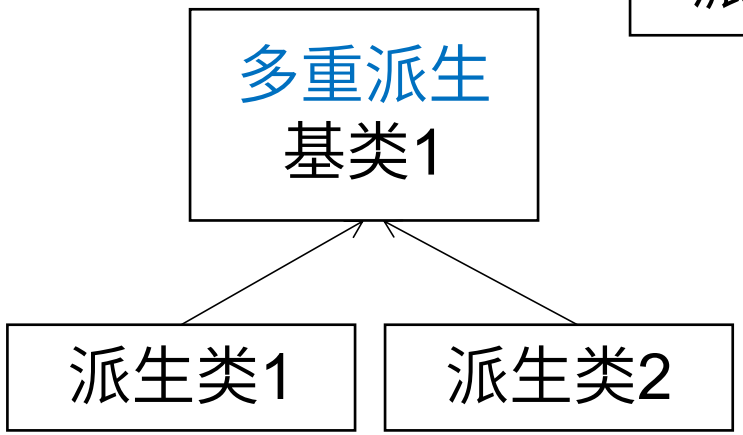
多继承

基类与派生类的对应关系

- 从派生类来看：
  - 单继承：派生类只从一个基类派生
  - 多继承：派生类从多个基类派生
- 从基类来看：
  - 多重派生：由一个基类派生出多个不同的派生类
  - 多层派生：派生类又作为基类，继续派生新的类

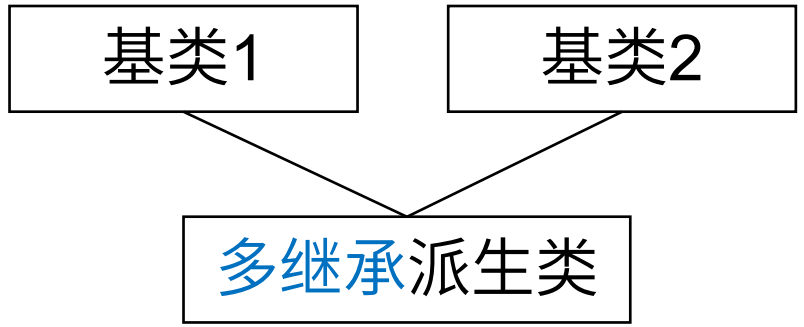






多继承

- 多继承时派生类的声明如下，其中每一个“继承方式”，只用于限制紧随其后的基类继承。




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

多继承

```
class A1 { private: int a1;
    public: A1(int i) { a1=i;
        cout<<"A1 construct "<<endl; }
    void print( ){ cout<<a1<<endl; }
};
class A2{ private: int a2;
    public: A2(int i) { a2=i;
        cout<<"A2 construct "<<endl; }
    void print(){ cout<<a2<<endl; }
};
```

```
class A3 { private: int a3;
    public: A3(int i) { a3=i;
        cout<<"A3 construct "<<endl;}
    void print(){ cout<<a3<<endl; }
};
class B: public A2, public A1 { private: A3 a3;
    public: B(i,j,k): A1(i),A2(j),a3(k) {
        cout<<"B construct "<<endl; }
    void print(){
            A1::print();A2::print();a3.print();}
};
```

```
B b(1,2,3,4);
b.print(); //输出为：A2 construct A1 construct A3 construct B construct 1 2 3 4
```

多继承中的二义性问题

- 在派生类中对基类成员的访问应该是唯一对的。但是，在多继承情况下，可能造成对基类中某个成员的访问出现不唯一情况。这时就称对基类成员的访问产生了二义性。

```
class Base1 {
 public:
  void fun();
};
```

```
class Base2{
 public:
    void fun();
};
```

```
class Derived: public Base1,public Base2{};
Derived obj;
obj.fun();
//无法确定访问的是哪个基类中的函数
```

多继承中的二义性问题

- 解决二义性的基本措施包括：
  - 措施1：通过域运算符明确指出访问的哪个基类的函数（依赖对象）
  - 措施2：在类中定义同名成员（覆盖原则）

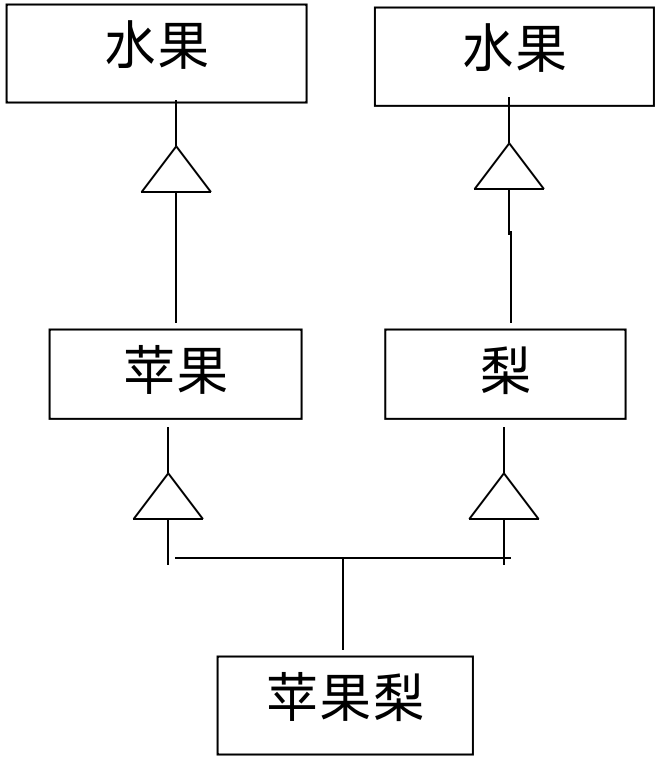
```
class Base1 {
 public:
  void fun();
};
```

```
class Base2{
 public:
   void fun();
};
```

```
class Derived: public Base1,public Base2{
    void fun() { Base1::fun(); }
};
Derived obj;
obj.fun();
obj.Base1::fun();
```

重复继承中的二义性问题

- 多继承时，一个派生类可能有多个基类，若这些基类又有一个共同的基类，这就产生了重复继承问题。在通过重复继承而来的派生类对象访问共同的祖先基类的成员时，可能会产生二义性。
- 如右，苹果和梨均从类水果继承了数据成员和成员函数，因此在苹果和梨中同时存在着同名的数据成员和成员函数；



重复继承中的二义性问题

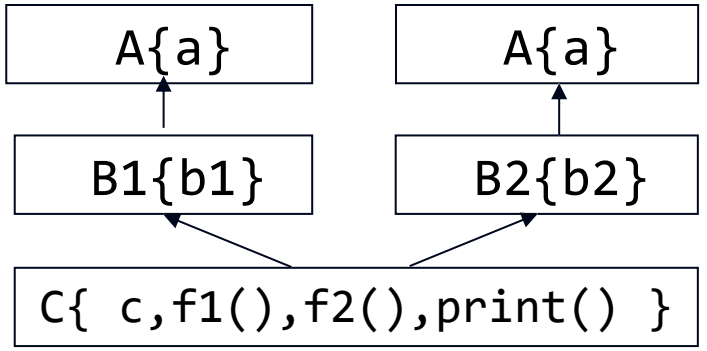
```
class A { protected: int a;
    public: A(int k){ a=k; }
};
class B1:public A { protected: int b1;
    public: B1(int p, q): A(q) { b1=p; }
};
class B2:public A { protected: int b2;
    public: B2(int p, q): A(q) { b2=p; }
};
```

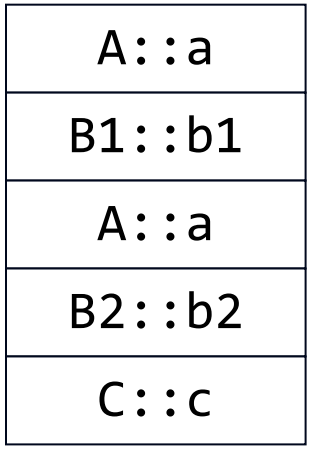
```
class C:public B1,public B2 { private: int c;
    public: C(int p, q, r, s):B1(p,q),B2(r,s){ }
    int f( ){ c=a+b1+b2; } //二义性语句
    int f1( ){ return c=B1::a+b1+b2; }
    int f2( ){ return c=B2::a+b1+b2; }
    void print(){
        cout<<"equal "<<f1()<<"or"<<f2(); }
};
```

```
C obj_c(1,2,3,4);
obj_c.print(); //输出: equal 6 or 8
```

重复继承中的二义性问题

- 在上例中若声明C c1，则对继承的数据成员a的访问c1.a;或c1.A::a都会出现二义性。为避免出现这种二义性，可以用作用域运算符::加以限定。例如c1.B1::a或c1.B2::a;
- 要想从根本上消除这种二义性，应使这个公共基类在间接派生类中只产生一个该公共基类的子对象。这就需要将这个公共基类定义为虚基类。





虚基类

```
class 派生类名: virtual 继承方式 基类名 { … }
```

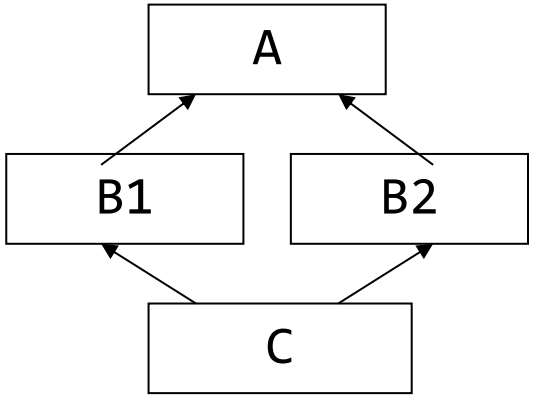
- 虚基类是解决重复继承二义性的有效方法，其与普通基类的区别只有在发生重复继承时才能表现出来：被重复继承的基类被说明为虚基类后，其成员在派生类中只产生唯一副本，从而在根本上解决了二义性问题。
- 虚基类的说明在派生类的定义中进行，虚基类的说明是在定义派生类时，写在派生类名的后面。其中，virtual 是说明虚基类的关键字。

虚基类

- 对于虚基类而言，由于其派生的对象只有一个虚基类子对象，因此该虚基类构造函数必须只被调用一次。规定将在定义对象时所指定的类称为最后派生类，虚基类子对象由最后派生类构造函数通过调用虚基类构造函数进行初始化。
- 如果一个派生类有一个直接或间接的虚基类，那么派生类的构造函数的成员初始化列表中必须列出虚基类构造函数的调用，如果没有列出，则表示使用该虚基类的缺省构造函数来初始化派生类对象中的虚基类子对象。

虚基类

```
class A { protected: int a;
    public: A(int x) { a = x ;
                cout<<"A construct "<<a<<endl; }
};
class B1: virtual public A { protected: int b1;
    public: B1(int x,int y): A(x) { b1 = y;
        cout<<"B1 construct "<<b1<<endl ; }
};
class B2: virtual public A { protected: int b2;
    public: B2(int x,int y): A(x) { b2 = y;
        cout<<"B2 construct "<<b2<<endl; }
};
```




```
//输出
A construct 10
B1 construct 20
B2 construct 30
C construct 40
```

```
class C:public B1, public B2 {
    private: int c;
    public:
    C(int x,y,z,w): B1(x,y),B2(x,z),A(x)
    { c=w;cout<<"C construct "<<c; }
};
C *pc=new C(10,20,30,40);
```

虚基类

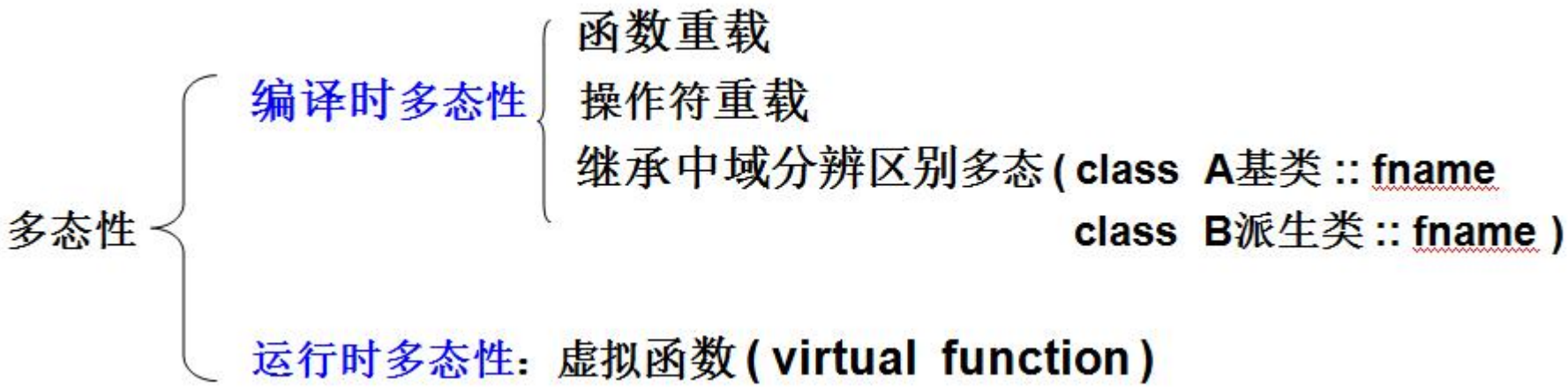
- 在虚基类直接或间接继承的派生类中的构造函数的成员初始化列表中都要列出这个虚基类构造函数的调用。
- 在建立对象时，只有派生类构造函数的成员初始化列表中虚基类构造函数被调用，而基类中对虚基类构造函数的调用将被忽略。而该派生类的基类中所列出的对这个虚基类的构造函数的调用在执行中被忽略，这样便保证了对虚基类的子对象只初始化一次。在一个成员初始化列表中出现对虚基类和非虚基类构造函数的调用，则虚基类的构造函数先于非虚基类构造函数的执行。

PART 02

多态

多态

- 多态性是面向对象编程的一个强大功能，在大多数C++程序中都要用到；
- 多态性是一种现象：同一符号或名字在不同情况下具有不同解释的现象，即是指同一个函数的多种形态。C++支持两种多态性，编译时的多态性和运行时的多态性。



回忆：公有继承下的赋值兼容规则

- 在公有派生的情况下，允许将派生类的对象赋值给基类的对象，但反过来却不行，即不允许将基类的对象赋值给派生类的对象。这是因为一个派生类对象的存储空间总是大于它的基类对象的存储空间。若将基类对象赋值给派生类对象，这个派生类对象中将会出现一些未赋值的不确定成员。
- 多态是以继承为基础的，而“公有继承下的赋值兼容规则”则是多态的理论支撑！

回忆：公有继承下的赋值兼容规则

- 允许将派生类的对象赋值给基类的对象，有以下三种具体作法：
  - 直接将派生类对象赋值给基类对象，假设Derived已定义为Base的派生类：

    ```
    ObjB=objD;  //合法    ObjD=objB;  //非法
    ```

  - 定义派生类对象的基类引用，例如：`Base &b=objD`
  - 用指向基类对象的指针指向它的派生类对象，例如：`Base *pb=&objD;`
- 当派生类对象赋值给基类或基类指针后，只能通过基类名或基类指针访问派生类中从基类继承来的成员，不能访问派生类中的其它成员。

多态

原先：

f(基类) { … }

f(派生类) { …}

现在期望：一个f(基类) { .. }

根据传递的实参，f表现出与传递信息对应的操作程序员不需要单独再写

- 自动适应类型变化的性质，就是多态性；这是一个极为强大的机制，代码人员常不能事先确定要处理哪种类型的对象，即在设计期间或编译期间不能确定类型，只能在运行期间确定。使用多态性可以轻松地解决这个问题！

多态

```
class Base {
  public:
    void Print() {          //大
       cout<<"Base::Print"<<endl; }
};
class Derived: public Base {
  public:
    void Print() {      //小
       cout<<"Derived::Print"<<endl;}
    void fun(const Base &cb) {
        cb.Print(); }
};
    Derived obj;
    fun(obj);
```

- 输出：Base::Print
- obj虽然是派生类对象，但在赋值给基类对象操作之后被同化了，丧失了基类的本性。

多态

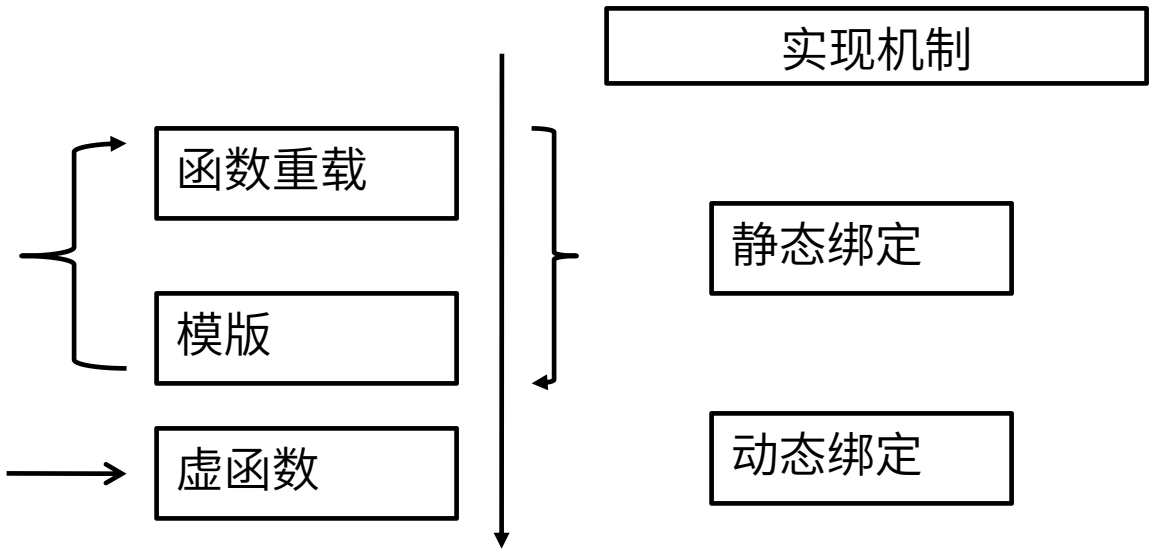
```
class Member{
  void answer(){cout<<"member. ";} };
class  Teacher: public Member {
  void answer(){cout<<"teacher. ";} };
class Student: public Member{
  void answer(){cout<<"student. ";} };
```

```
Member aMember;Teacher aTeacher;
Student aStudent;
Member *Who;
Who=&aMember; Who->answer();
Who=&aTeacher; Who->answer();
Who=&aStudent; Who->answer();
//输出: member. member. member.
```

- 没有出现程序设计者期待的随着赋值不同而展现不同的结果；这是由于函数answer()不是虚函数，所有调用语句都是静态绑定的，即三个Who->answer();调用执行的是同样的代码。

多态

- 一个面向对象的系统常常要求一组具有基本语义的方法能在同一个接口（如函数）下为不同的对象服务（解释为不同不同意义），这就是多态性。
- 多态是一种能力，即同样的消息被不同对象接收时，产生完全不同的行为（调用同名不同功能的函数）
- 多态分类：
  - 编译时多态
  - 运行时多态



虚函数与多态类

- 定义虚函数是为了在程序运行时自动选择各派生类中的重定义版本，所以多态基类一定要定义一个以上的派生类才有意义。
- 在派生类中重新定义虚函数时，其函数原型（包括返回值类型、函数名、参数个数、参数类型及顺序）必须与基类中的原型完全相同。否则编译时会出错或被当作函数重载处理。
- 一个指向基类的指针可以指向它的公有继承的派生类。定义虚函数的目的就是想统一用一个基类对象指针去访问不同派生类中虚函数的重定义代码。

虚函数与多态类

- 虚函数是动态绑定的基础，是非静态的成员函数。在类的声明中，在函数原型之前写virtual，其用来说明类声明中的原型。基类中声明了虚函数，派生类中无论是否说明，同原型函数都自动为虚函数。
- 其本质不是重载声明而是覆盖。通过基类指针或引用，执行时会根据指针指向的对象的类，决定调用哪个函数。

虚函数与多态类

```
class Poultry {
  virtual void can_fly() {
    cout<<"Yes! I can."<<endl; }
};
class Cock : public Poultry {
  void can_fly () {
    cout<<"Yes! But I can't fly high.\n";}
};
class Duck : public Poultry {
  void can_fly ()  {
    cout<<"No! But I can swim.\n"; }
};
```

```
Poultry anyPoultry, *ptr;
Cock aCock; Duck aDuck;
ptr =&anyPoultry;
ptr->can_fly();
ptr =&aCock;
ptr->can_fly();
ptr =&aDuck;
ptr->can_fly();
//程序输出:
Yes! I can.
Yes! But I can't fly high.
No! But I can swimming.
```

动态绑定

- 只要把派生类对象的地址传递给基类指针，就可以直接使用基类指针调用派生类中虚函数的重定义版本，与通过派生类对象名调用该函数有相同的执行结果。虚函数的引入，使函数调用语句可以动态执行，这就是动态绑定。

```
ptr =&anyPoultry;
ptr->can_fly();
ptr =&aCock;
ptr->can_fly();
ptr =&aDuck;
ptr->can_fly();
//程序输出:
Yes! I can.
Yes! But I can't fly high.
No! But I can swimming.
```

虚函数与重载函数的关系

- 在派生类中重新定义基类虚函数是一种特殊的函数重载。普通的函数重载时，其函数的参数或参数类型必须有所不同，函数的返回类型也可以不同。
- 当重载虚函数时，函数名、返回类型、参数个数、参数的类型和顺序与基类原型应完全相同。如果仅仅返回类型不同，其余均相同，系统会给出错误信息；若仅仅函数名相同，而参数的个数、类型或顺序不同，系统将它作为普通的函数重载，这时将丢失虚函数的特性。

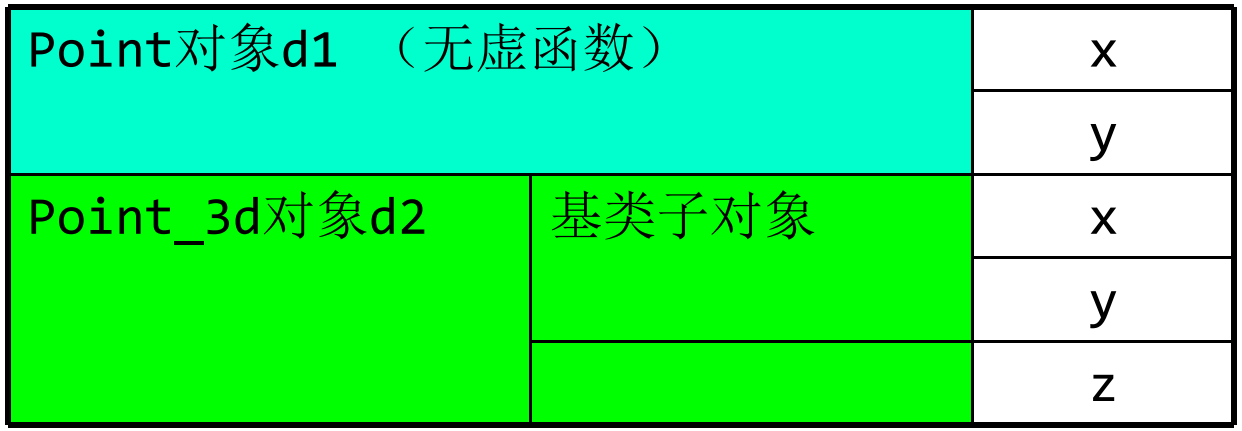
动态绑定

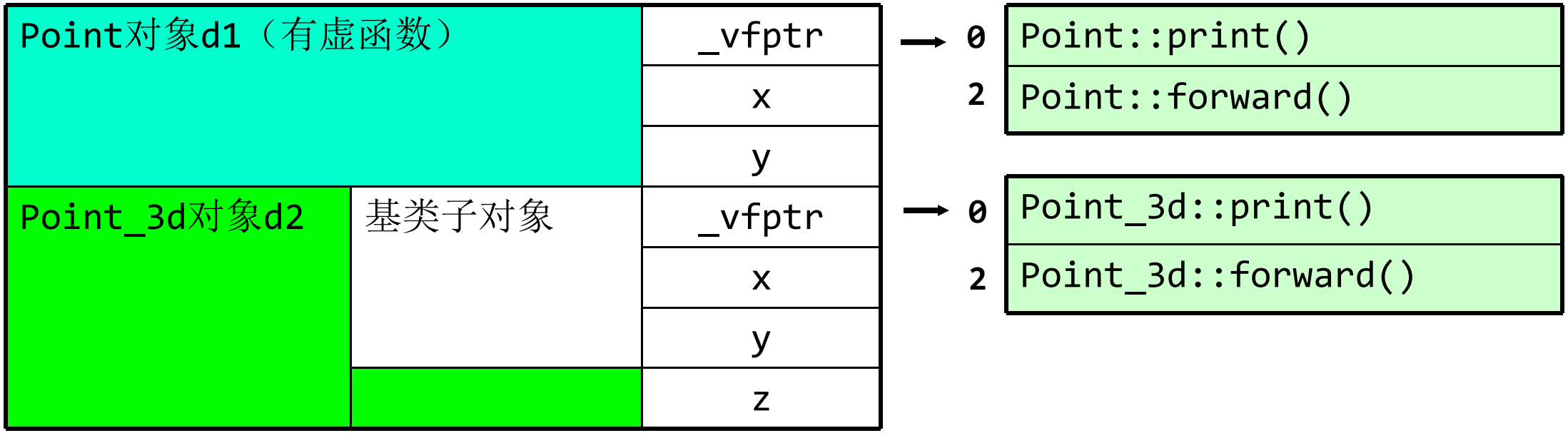
- 所谓绑定（binding）是指编译程序将源程序中的函数调用语句与该函数的执行代码联系在一起的过程。换句话说，绑定就是让函数调用语句找到它要执行的代码。在非面向对象程序设计语言中，绑定过程都是静态的，也就是说，所有的绑定都是在编译期完成的。静态绑定的函数调用语句有唯一确定的语义，不可能具有多态性。

动态绑定

- 动态绑定是通过虚函数表实现的。
- 对于含有虚函数的多态类，编译器为每个对象生成一个虚表指针。即在每个对象的内存映像中增加了一个_vfptr指针，它指向一个虚函数表vtable。在基类的虚函数表中列出基类所有虚函数的入口地址，在派生类的虚函数表中列出派生类的所有虚函数的入口地址。

有无虚函数时对象内存映象对比





虚函数动态绑定的调用步骤

- 首先为多态类的基类声明一个指针变量，然后让这个指针变量指向此多态类继承树中某一个类的对象。由于基类指针指向对象内存映像的首地址，它直接访问的是该对象的虚表指针，下一步由具体对象的虚表指针就可以访问到该对象所在的类中为虚函数定义的那一份代码。

虚函数的应用场景

1．“一个接口，多种实现”

- 虚函数同派生类的结合可使C++支持运行时的多态性，实现了在基类定义派生类所拥有的通用接口，而在派生类定义具体的实现方法；类的公开成员函数可以看作类封装体向外界提供的接口，虚函数的引入使得基类和它的派生类有了一个共同的接口。这种“一个接口，多种实现”概念所体现的动态多分支选择机制为面向对象系统提供了控制更大复杂性的能力。

虚函数的应用场景

```
class  Figure{
    protected: double a, b;
    public:
        Figure(double x, double y=0){ a=x; b=y; }
        virtual void area(){
                cout<<"Can't define area"<<endl; }
};
class Circle: public Figure {
    public:
        Circle (double x):Figure(x){}
        virtual void area(){
                cout<<"Circle‘s area: "<<PI*a*a<<endl; }
};
```

虚函数的应用场景

```
class Triangle: public Figure {
    public:
        Triangle(double x, y):Figure(x, y) {}
        virtual void area(){
        cout<<"Triangle‘s area: "<<a*b/2<<endl; }
};
class Rectangle: public Figure {
    public:
        Rectangle(double x, y):Figure(x, y) {}
        virtual void area(){
        cout<<"Rectangle‘s area: "<<a*b<<endl; }
};
```

```
Figure *p;
Circle obj_c(10);
Triangle obj_t(10, 5);
Rectangle obj_r(12, 6);
p=&obj_c; p->area();
p=&obj_t; p->area();
p=&obj_r; p->area();

//输出:
Circle‘s area: 314.16
Triangle‘s area: 25
Rectangle‘s area: 72
```

虚函数的应用场景

2．多态数据结构

- 堆栈、队列、链表等数据结构中的数据通常都是单一类型的。利用类的多态性，可以构造异质的数据结构，即数据单元由不同类的对象组成的结构。例如，一个可以压入不同（长度）对象的堆栈。多态数据结构是面向对象的数据库、多媒体数据库的数据存储基础。

虚函数的应用场景

```
class Phone {
    friend class List;
protected:
    char name[20];
    char cityNo[5];
    char phoneNo[10];
    static Phone *ptr;
    Phone *next;
public:
    Phone(char*,char*,char*);
    virtual void insert();
    virtual void print();
};
```

```
class BP_user: public Phone{
    char server[10], call[10];
public: void print();   //重新定义
    void insert();  //重新定义
};
class Fax_user: public Phone{
    char fax[10];
public: void print();     //重新定义
    void insert();  //重新定义
};
class Mobile_user: public Phone{
     char mobileNo[12];
public: void print();     //重新定义
      void insert();  //重新定义
};
```

虚函数的应用场景

```
class List{ Phone *head;
public:
   List(){head=0;}
   void insert_node(Phone *node);
   void remove(char *name);
    void print_list();
};
```

```
//以下是Phone及其派生类的实现部分
Phone::Phone(char*name,*cityNo,*phoneNo){
       strcpy(Phone::name, name);
       strcpy(Phone::cityNo, cityNo);
       strcpy(Phone::phoneNo, phoneNo);
       next=NULL;
}
```

```
BP_user::BP_user(char *name, *cityNo,
*phoneNo, *server, *call):
Phone(name,cityNo,phoneNo){
    strcpy(BP_user::server, server);
    strcpy(BP_user::call, call);
}
Fax_user::Fax_user(char *name, *cityNo,
*phoneNo,*fax):Phone(name,cityNo,phoneNo) {
    strcpy(    Fax_user::fax, fax);
}
Mobile_user::Mobile_user(char* name,*cityNo,
*phoneNo, *mobileNo):
Phone(name,cityNo,phoneNo) {
    strcpy(Mobile_user::mobileNo, mobileNo);
}
```

虚函数的应用场景

```
void Phone::insert(){
    ptr=new Phone(name, cityNo, phoneNo);
}
void BP_user::insert(){
    ptr=new BP_user(name, cityNo, phoneNo,
server, call);
}
void Fax_user::insert(){
    ptr=new Fax_user(name, cityNo, phoneNo,
fax);
}
void Mobile_user::insert(){
    ptr=new Mobile_user(name, cityNo,
phoneNo, mobileNo);
}
```

```
void Phone::print(){
    cout<<endl<<"Name: "<<name<<" Phone: "
<<cityNo<<"-"<<phoneNo<<endl;
}
void BP_user::print(){
    Phone::print();
    cout<<"BP: "<<server<<"-"<<call<<endl;
}
void Fax_user::print(){
    Phone::print();
    cout<<"Fax: "<<fax<<endl;
}
void Mobile_user::print(){
    Phone::print();
    cout<<"Mobile number: "<<mobileNo<<endl;
}
```

纯虚函数与抽象类

- 纯虚函数是在基类中只有说明而没有实现定义的虚函数，它的任何派生类都必须定义自己的实现版本。纯虚函数的定义形式：

```
virtual 类型 函数名（参数表）=0;
```

```
class Figure{
    protected: double a, b;
    public:
        Figure(double x, double y=0){ a=x; b=y; }
        virtual void area() = 0;
};
```

纯虚函数与抽象类

- 通过将虚函数声明为纯虚函数，类的设计者强迫它的所有派生类都必须定义自己的方法实现版本。如果某一派生类没有给予出自己的实现版本而又企图创建它的对象，则将发生编译错误。

```
class Figure{
    virtual void area() = 0;
};
class Circle: public Figure {
    virtual void area(){ cout<<"Circle‘s area: "<<PI*a*a<<endl; }
};
```

纯虚函数与抽象类

- 含有纯虚函数的类称为抽象类，抽象类是表示一组具有某些共性的具体类的公共特征的类。相对于具体类，它表示更高层次的抽象概念。抽象类中除了纯虚函数外还可以包括其它函数。
- 抽象类只能用来作为派生其它类的基类，而不能创建抽象类的对象。即抽象类只能通过它的派生类来实例化。某个抽象类的派生类又称为它的实现类。

纯虚函数与抽象类

- 抽象类不能用来作为参数类型、函数返回类型、显式转换类型。可以声明指向抽象类的指针和引用，指向抽象类的指针可以指向它的派生类以支持运行时的多态性。

```
class Figure {
    virtual void area()=0;
};
```

```
Figure a;  //错，不能创建抽象类的对象
Figure *ptr;  //对，可以声明指向抽象类的指针
Figure function1();  //错，抽象类不能作为函数返回类型
void function2(Figure);
    //错，抽象类不能作为函数参数类型
Figure& function3(Figure &);
    //对，可以声明指向抽象类的引用
```

纯虚函数与抽象类

- 从语法上看，抽象类是一种特殊的多态类，具有动态的多态性。比起多态类来，它更侧重于表达类的抽象层次。
- 抽象类和它的实现类的关系虽然也是一种继承关系，但这种继承与之前讨论的继承有一种质的区别。非抽象类的继承着眼点在于代码重用，称为类继承，抽象类的继承着眼点在于为一组具有某些共性的具体类提供统一的访问接口，称为接口继承。接口继承的目的是为复杂对象提供构造基础。

感谢倾听

给个好评！