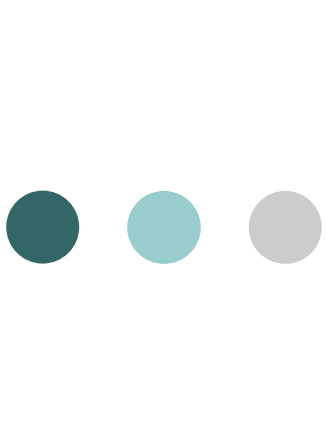


第七章 继承 - 派生类

本章内容

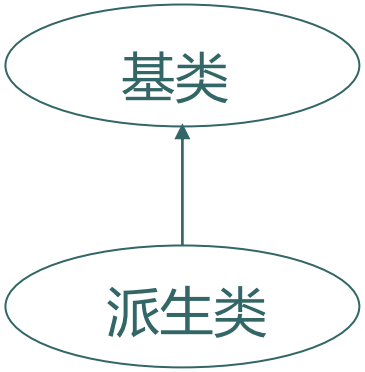
- 继承的基本概念
- 单继承
- 虚函数与动态绑定
- 多继承

继承的概念

- 在开发新软件时，把现有软件或软件的一部分拿过来用称为软件复用。
- 直接复用已有软件比较困难，已有软件与新软件所需要的功能总是有差别的。两种解决方法：
  - 修改已有软件的源代码。存在以下缺点：源码难以获得；需读懂源码；可靠性差、易出错；
  - 继承机制（Inheritance）。在定义一个新的类时，先把一个或多个已有类的功能全部包含进来，然后再给出新功能的定义或对已有类的某些功能重新定义。

基类与派生类

- 在继承关系中存在两个类：基类（或称父类）和派生类（或称子类）。派生类拥有基类的所有特征，并可以定义新的特征或对基类的一些特征进行重定义。



- 继承分为
  - 单继承：一个类最多有一个直接基类。
  - 多继承：一个类可以有多个直接基类。

单继承

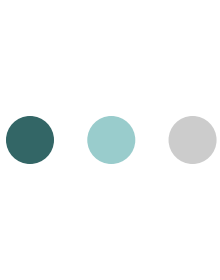
- 派生类只能有一个直接基类，定义格式：

  class <派生类名>：[<继承方式>] <基类名>

  { <成员说明表>

  };

  - <派生类名>为派生类的名字。
  - <基类名>为直接基类的名字。
  - <成员说明表>是在派生类中新定义的成员，其中包括对基类成员的重定义。
  - <继承方式>指出派生类的实例对象以及派生类的派生类对该派生类从基类继承来的成员的访问控制



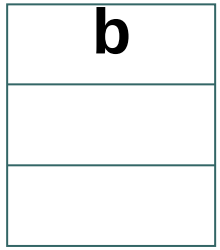
```
class A //基类
{       int x, y;
    public:
        void f();
        void g();
};

class B:  public A //派生类
{       int z; //新成员
    public:
        void h(); //新成员
};
```

关于派生类的一些说明

- 除了拥有新定义的成员外，派生类还拥有基类的大部分成员（基类的构造函数，析构函数和赋值操作符重载函数除外）。

例如： **B b;**




```
b.f();  // A类中的f
b.g();  // A类中的g
b.h();  // B类中的h
```

```
class A //基类
{    int x, y;
  public:
     void f();
     void g();
};
class B: public A //派生类
{     int z; //新成员
   public:
     void h(); //新成员
};
```

- 定义派生类时一定要见到基类的定义。

```
class A;  //声明
class B: public A  //Error
{     int z;
  public:
       void h() { g(); }  //Error，编译程序不知道基类中是否有函
                                  //数g以及函数g的原型。
};
B b; //Error，编译无法确定b所需内存空间的大小。
```

- 友元关系无法继承
  - 如果在派生类中没有显式说明，基类的友元不是派生类的友元；
  - 如果基类是另一个类的友元，而该类没有显示说明，则派生类也不是该类的友元。

在派生类中对基类成员的访问

- 派生类不能直接访问基类的私有成员。

例如：

```
class A
{     int x, y;
  public:
      void f();
      void g() { ... x ... }
 };
class B: public A
{    int z;
  public:
     void h()
     {  ... x ...  //Error，x为基类的私有成员。
        g(); } //OK，通过函数g访问基类的私有成员x。
};
```

- 基类同名成员在派生类的作用域内不可直接访问。访问基类同名成员时要用基类名受限符。

例如：

```
class A //基类
{       int x, y;
    public:
        void f();
        void g();
};
```

```
class B: public A
{   int z;
    public:
        void f();
        void h()
        {  f();  //B类中的f
           A::f();  //A类中的f
        }
};
```

```
B b;
b.f();  //B类中的f。
b.A::f();  //A类中的f
```

- 即使派生类中定义与基类同名但参数不同的成员函数，基类的同名函数在派生类的作用域中也不可直接访问。

例如：

```
class A //基类
{     int x, y;
  public:
      void f();
      void g();
};
```

```
class B: public A
{      int z;
   public:
      void f(int);
      void h()
      {    f(1);  //OK
           f();  //Error
           A::f();  //OK
       }
};
```

```
B b;
b.f(1);  //OK
b.f();  //Error       b.A::f();  //OK
```

- 可以在派生类中使用using声明将基类中的某个函数名对派生类开放

例如：
```
class A //基类
{     int x, y;
  public:
      void f();
      void g();
};
```

```
class B: public A
{       int z;
    public:
        using A::f;
        void f(int);
        void h()
        {    f(1);  //OK
             f();    //OK 等价于A::f()
        }
};
```

```
B b;
b.f(1);  //OK
b.f();    //OK 等价于b.A::f();
```

封装与继承的矛盾

- 在继承机制中，类成员有两种被外界使用的场合：
  - 通过类的对象使用
  - 在派生类中使用
- 在派生类中定义新的成员时，往往需要用到基类的private成员。（矛盾）
- 在C++中，提供了protected访问控制符缓解了封装与继承的矛盾，用它说明的成员不能被对象使用，但可以在派生类中使用。

```
class A //基类
{ protected:
      int x,y;
  public:
      void f();
};
```

```
void f()
{   A a;
    a.f();  //OK
    ... a.x ...  // Error
    ... a.y ...  // Error
 }
```

```
class B: public A
{     ......
      void h()
       {   f();  //OK
           ... x ...  //OK
           ... y ...  //OK
       }
 };
```

继承方式

- 在C++中，派生类拥有基类的大部分成员。问题是基类的成员变成派生类的什么成员呢（public、private或protected）？
- 由继承方式决定。继承方式在定义派生类时指定：

  class <派生类名>：[<继承方式>] <基类名>

  { <成员说明表>

  };

  继承方式可以是：public、private和protected。

  默认的继承方式为：private。

继承方式的含义（P272）

基类 / 派生类 / 继承方式	public	private	protected
public	public	不可直接访问	protected
private	private	不可直接访问	private
protected	protected	不可直接访问	protected

```
class A
{  public:
        void f();
   protected:
        void g();
   private:
        void h();
};

class B: protected A
{  public:
        void q();  //q对A类成员的访问不受B的继承方式影响，
                   //除了h，其它都能访问。
};
```

```
class C: public B
{   public:
        void r()
        {  q();  //OK
           f();   //OK
           g();  //OK
           h();  //Error，h是B的不可直接访问的成员。 }
};
B b;
b.q();  //OK
b.f();  //Error，f是B的protected成员。
b.g();  //Error，g是B的protected成员。
b.h();  //Error，h是B的不可直接访问的成员。
```

访问控制符的调整

- 在任何继承方式中，除了private成员，都可以在派生类中分别调整其访问控制符
  - 格式

    [public: | protected: | private:] <基类名> :: <基类成员名>

```
class A
{ public:
      void  f1();   void  f2();  void  f3();
  protected:
      void  g1();  void  g2();  void  g3();
};
class B: private  A
{ public:
      A::f1;  A::g1;
  protected:
      A::f2;   A::g2;
};
class C: private B
{ public:
      void  h()
      {  f1();  f2();  g1();  g2();  // OK
         f3();  g3();  // Error
      }
}
```

```
B  b;
……
b.f1();  b.g1();  // OK
b.f2();  b.g2();  b.f3();  b.g3(); // Error
```

派生类对象的初始化

- 派生类对象的初始化由基类和派生类共同完成：
  - 基类的数据成员由基类的构造函数初始化
  - 派生类的数据成员由派生类的构造函数初始化。
- 当创建派生类的对象时，派生类的构造函数在进入其函数体之前会去调用执行基类的构造函数。
- 默认情况下，调用基类的默认构造函数，如果要调用基类的非默认构造函数，则必须在派生类构造函数的成员初始化表中指出。

```
class A
{       int x;
    public:
        A() { x = 0; }
        A(int i) { x = i; }
};
class B: public A
{       int y;
    public:
        B() { y = 0; }
        B(int i) { y = i; }
        B(int i, int j):A(i) { y = j; }
};
......
B b1;  //执行A::A()和B::B()，b1.x等于0，b1.y等于0。
B b2(1);  //执行A::A()和B::B(int)，b2.x等于0，b2.y等于1。
B b3(1,2);  //执行A::A(int)和B::B(int,int)，b3.x等于1，
                    //b3.y等于2。
```

- 如果一个类D既有基类B、又有成员对象类M，则
  - 在创建D类对象时，构造函数的执行次序为B->M->D
  - 当D类的对象消亡时，析构函数的执行次序为D->M->B

- 对于拷贝构造函数：
  - 派生类的隐式拷贝构造函数（由编译程序提供）将会调用基类的拷贝构造函数。
  - 派生类自定义拷贝构造函数在默认情况下则调用基类的默认构造函数。需要时，可在派生类自定义拷贝构造函数的“基类成员初始化表”中显式地指出调用基类的拷贝构造函数。

```
class A
{    int a;
  public:
      A(); { a = 0; }
      A(const A& x);  { a = x.a; }
};
class B: public A
{ public:
      B() { …… }
};
class C: public A
{ public:
      C() { …… }
      C(const C&) { …… }
}
class D: public A
{ public:
      D() { …… }
      D(const D& d): A(d){ …… }
}
```

```
B  b1;       // 调用A()
B  b2(b1); // 调用A(const  A&)
C  c1;       // 调用A()
C  c2(c1);  // 调用A()
D  d1;       // 调用A()
D  d2(d1); // 调用A(const  A&)
```

派生类对象的赋值

- 派生类不从基类继承赋值操作符。如果派生类没有提供赋值操作符重载函数，则系统提供一个隐式的赋值操作符重载函数，其行为是：
  - 对基类成员调用基类的赋值操作符进行赋值，
  - 对派生类的成员按逐个成员赋值。
- 派生类自定义的赋值操作符重载函数不会自动调用基类的赋值操作，需要在派生类自定义的赋值操作符重载函数中显式地调用基类的赋值操作符来实现基类成员的赋值

```
class A
{  ......
};

class B: public A
{       ......
    public:
        B& operator =(const B& b)
        {  if (&b == this)    return *this;  //防止自身赋值。
           *(A*)this = b; //调用基类的赋值操作符对基类成员进行赋值
                          //或 this->A::operator =(b);
            ......
            return *this;
        }
};
B b1,b2;
b1 = b2;
```

代码复用的另一种方式 - 聚集

- 聚集把一个类作为另一个类的成员对象类来使用。

```
例如：class A
      { public:
            void f();
            ......
      };
class B
{     A a;  //定义一个A类的成员对象。
  public:
      void f() {  a.f();  } //复用A的f实现B的f
      ......
};
```

利用一个线性表类实现一个队列类

- 线性表由若干元素构成，元素之间有线性次序关系

```
class LinearList
{    ......
   public:
        bool insert( int x, int pos );
        bool remove( int &x,  int pos );
        int element( int pos ) const;
        int search( int x ) const;
        int length( ) const;
};
```

- 队列（Queue）是一种特殊的线性表，插入操作在一端，删除操作在另一端。又称先进先出表（First In First Out，FIFO）
- Queue的实现1（继承方法）：

```
class Queue: private LinearList
{  public:
       bool en_queue(int x)
       {  return insert(x, length());
       }
       bool de_queue(int &x)
       {  return remove(x, 1);
       }
};
```

- Queue的实现2（聚集方法）：

```
class Queue
{     LinearList list;
   public:
      bool en_queue(int i)
      {  return list.insert(i, list.length());
      }
      bool de_queue(int &i)
      {  return list.remove(i, 1);
      }
};
```

继承与聚集复用方式的比较

- 都可以实现软件复用。
- 聚集适用于类关系为“部分”与“整体”的情况。
- 继承与封装存在矛盾，聚集则不存在与封装的矛盾。
- 继承更容易实现子类型功能，即派生类的对象可以作为基类的对象来使用。
  - 如果一个类型S是另一个类型T的子类型，则对用T表达的所有程序P，当用S去替换程序P中的T时，程序P的功能不变。

面向对象程序设计的多态性

假设B是A的派生类，f是A类的成员函数，g是B类的成员函数（A中没有g）

下面是合法的：

A a, *p;

B b, *q;

b.f(); //如果B中有f，则调用B的f；否则调用A的f。

a = b; //用b改变a的状态，属于B但不属于A的数据成员被忽略。

p = &b; //A类指针p指向B类对象b。

下面是不合法的：

b = a; //Error，导致b有不确定的成员数据（a没有这些数据）。

q = &a; //Error，导致通过q向a发送它不能处理的消息，如：q->g();

多态与绑定

- 把以public方式继承的派生类看作是基类的子类型。则产生下面的多态：
  - 对象类型的多态。派生类对象既属于派生类，也可以属于基类。
  - 对象标识的多态。基类的指针或引用可以指向或引用基类对象，也可以指向或引用派生类对象。
  - 消息的多态。发送到基类对象的消息，也可以发送到派生类对象，从而可能会得到不同的处理。
- 上面消息的多态性带来了消息的绑定问题：
  - 向基类的指针或引用所指向或引用的对象发送消息，将调用什么成员函数（基类或派生类）来处理这个消息？

消息的静态绑定

```
class A
{       int x,y;
    public:
        void f();
};

class B: public A
{       int z;
    public:
        void f();
        void g();
};
```

```
void func1(A& x)
{   ......
    x.f();  // 调用A::f还是B::f？
    ......
}
void func2(A *p)
{   ......
    p->f(); // 调用A::f还是B::f？
    ......
}
```

答案是：A::f (静态绑定)

答案是：A::f （静态绑定）

静态绑定：在编译时刻，根据x和p的静态类型来决定f属于哪一个类

```
......
A a;
func1(a);  //调用A::f
func2(&a); //调用A::f
B b;
func1(b); //调用A::f
func2(&b); //调用A::f
```

消息的静态绑定

- 在上面的例子中，在编译时刻，根据形参（如引用x和指针p）的静态类型来决定f属于哪一个类。x和p静态类型分别是A&和A*，因此在函数func1和func2，都调用A::f

- C++默认采用的是消息的静态绑定，不管p和x指向和引用的是A类对象还是B类对象

虚函数 - 消息的动态绑定

- 一般情况下，需要在func1（或func2）中根据x（或p）实际引用（或指向）的对象来决定是调用A::f还是B::f。即采用动态绑定，而C++采用虚函数来实现动态绑定。

```
class A
{      int x,y;
   public:
       virtual void f(); //虚函数
};
class B: public A
{      int z;
   public:
       void f();
       void g();  };
```

```
void func1(A& x)
{   ......
    x.f();  // 调用A::f还是B::f？
    ......
}
void func2(A *p)
{   ......
    p->f(); // 调用A::f还是B::f？
    ......
}

......
A a;
func1(a);    //在func1中调用A::f
func2(&a); //在func2中调用A::f
B b;
func1(b);   //在func1中调用B::f
func2(&b); //在func2中调用B::f
```

虚函数 - 消息的动态绑定

- 基类中的一个成员函数如果被定义成虚函数，则在派生类中定义的、与之具有相同型构的成员函数是对基类该成员函数的重定义（或称覆盖，override）。
- 相同的型构是指：派生类中的成员函数的名字、参数类型和个数与基类相应成员函数相同，其返回值类型与基类成员函数返回值类型或者相同，或者是基类成员函数返回值类型的派生类。

```
class A
{  public:
        virtual A f();
        void g();
};

class B: public A
{  public:
        A f();    //对A类中成员f的重定义。返回类型也可为B
        void f(int); //新定义的成员
        void g(); //新定义的成员。
};
```

对比虚函数和同名函数

- 基类中的一个成员函数如果被定义成虚函数，派生类中与之具有相同型构的成员函数是对基类该成员函数的重定义（覆盖）。
- 如果基类的成员函数不是虚函数，派生类中与基类同名的成员函数不是重定义，而是新定义的成员函数。

```
class A
{  public:
        A f();
        void g();
};

class B: public A
{  public:
        A f();    //新定义的成员函数
        void f(int); //新定义的成员函数
};
```

什么时候需要使用虚函数（动态绑定）

- 基类中虽然给出了某些成员函数的实现，但是实现方法可能不是最好的，今后可能还会有更好的实现方法
- 基类根本无法给出某些成员函数的实现，必须由不同的派生类根据实际情况给出具体实现，这时需要把基类的这些成员函数声明为特殊的虚函数——纯虚函数

- 对虚函数有下面几点限制：
  - 只有类的成员函数才可以是虚函数。
  - 构造函数不能是虚函数，析构函数可以（往往）是虚函数。
  - 只要在基类中说明了虚函数，在派生类、派生类的派生类…中，同型构的成员函数都是虚函数，且virtual可以不写
  - 只有通过指针或引用访问类的虚构函数时才进行动态绑定
  - 构造函数和析构函数对虚函数的调用不进行动态绑定

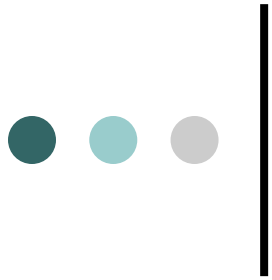
消息动态绑定的各种情况

```
class A
{   public:
        A() { f(); }
        ~A();
        virtual void f();
        void g();
        void h() { f(); g(); }
};
class B: public A
{   public:
        ~B();
        void f();
        void g();
};
```

```
.....

A a;  //调用A::A()和A::f
a.f();  //调用A::f
a.g();  //调用A::g
a.h();  //调用A::h、A::f和A::g

B b;  //调用B::B(), A::A()和A::f
b.f();  //调用B::f
b.g();  //调用B::g
b.h();  //调用A::h、B::f和A::g
```



A *p;
p = &a;
p->f(); //调用A::f
p->g(); //调用A::g
p->h(); //调用A::h, A::f和A::g
p = &b;
p->f(); //调用B::f
p->A::f(); //调用A::f
p->g(); //调用A::g，对非虚函数的调用采用静态绑定。
p->h(); //调用A::h, B::f和A::g
p = new B; //调用B::B(), A::A()和A::f
.....
delete p; //调用A::~A()，因为没有把A的析构函数定义为虚函数。

纯虚函数和抽象类

- 纯虚函数是指函数体为空（=0）的虚函数，只给出声明而不给实现

  例如：

```
class A
{  ......
   public:
       virtual int f()=0; //纯虚函数
   .......
 };
```

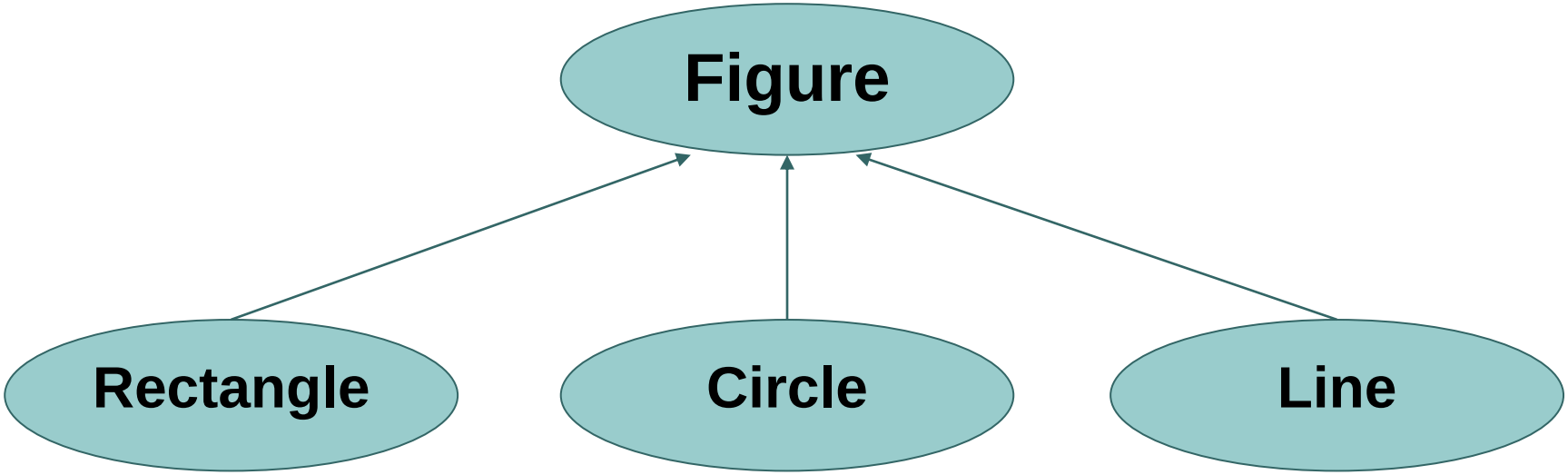
- 包含纯虚函数的类为抽象类，抽象类不用于创建对象。

例如：class A //抽象类

```
{   ......
    public:
        virtual int f()=0; //纯虚函数
    ......
};
......
A a;  //Error，A是抽象类
```

- 抽象类的作用是为派生类提供一个基本框架和公共对外接口，并且真正实现类的数据隐藏。

例：用抽象类为各种图形提供一个基本框架




```
class Figure
{   public:
        virtual void draw() const =0;
        virtual void input_data() = 0;
};
```

```
class Rectangle: public Figure
{      double left, right, top, bottom;
public:
        void draw() const {…}
        void input_data() {cin>>left>>right>>top>>bottom;}
};
class Circle: public Figure
{     double x, y, r;
public:
    void draw() const {…}
    void input_data() {cin>>x>>y>>r;}
};
class Line: public Figure
{     double x1, y1, x2, y2;
public:
    void draw() const {…}
    void input_data() {cin>>x1>>y1>>x2>>y2;}
};
```

```
Enum FigureShape {LINE, RECTANGLE, CIRCLE};
Figure *figures[100];

figures[0] = new Line;
figures[0]->input_data();
figures[0]->draw();

figures[1] = new Circle;
figures[1]->input_data();
figures[1]->draw();

figures[2] = new Rectangle;
figures[2]->input_data();
figures[2]->draw();
```

例：用抽象类为栈的两个不同实现提供一个公共接口

```
class Stack
{   public:
        virtual bool push(int i)=0;
        virtual bool pop(int& i)=0;
};
```

```
class ArrayStack: public Stack
{     int elements[100],top;
  public:
      ArrayStack() { top = -1; }
      bool push(int i) { ......}
      bool pop(int& i) { ...... }
};

class LinkedStack: public Stack
{     struct Node
      {  int content;
         Node *next;
      } *first;
  public:
      LinkedStack() { first = NULL; }
      bool push(int i) { ......}
      bool pop(int& i) { ...... }
};
```

```
void f(Stack *p)
{   ......
    p->push(...);  //将根据p所指向的对象类来确定push的归属
    ......
    p->pop(...);  //将根据p所指向的对象类来确定pop的归属
    ......
}
int  main()
{   ArrayStack st1;
    LinkedStack st2;
    f(&st1);  //OK
    f(&st2);  //OK
    ......
}
```

多继承

- 对于下面的两个类A和B：

```
class A
{       int m;
   public:
        void fa();
};
class B
{       int n;
   public:
        void fb();
};
```

- 如何定义一个类C，它包含A和B的所有成员，另外还拥有新的数据成员r和成员函数fc？

- 用单继承实现：

```
class C: public A
{       int n, r;
   public:
        void fb();
        void fc();
};  或者
class C: public B
{       int m, r;
   public:
        void fa();
        void fc();
};
```

- 不足之处：
  - 概念混乱：导致A和B之间增加了层次关系。
  - 易造成不一致：A（或B）中的m、fa（或n、fb）与C中的m、fa（或n、fb）

- 用聚集实现：

```
class C
{     A a;
      B b;
      int r;
   public:
      void fa() { a.fa(); }
      void fb() { b.fb(); }
      void fc();  };
```

- 不足之处：

  不能实现子类型：程序中的A或B不能用C替代。

- 用多继承实现：

```
class C: public A, public B
{       int r;
   public:
        void fc();
};
```

- 多继承是指派生类可以有多个直接基类。定义格式为：

  class <派生类名>： [<继承方式>] <基类名1>，

  [<继承方式>] <基类名2>，…

  { <成员说明表>

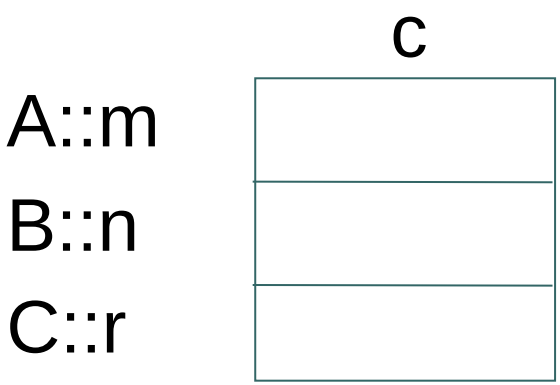
  };

- 继承方式及访问控制的规定同单继承。
- 派生类拥有所有基类的所有成员。
- 基类的声明次序决定：
  - 对基类构造函数/析构函数的调用次序
  - 对基类数据成员的存储安排。

```
class A
{       int  m;
    public:
        void  fa();
};
class B
{       int  n;
    public:
        void  fb();
};
```

```
class C:  public A,  public B
{       int  r;
    public:
        void  fc();
};
......
C c;
```

对象c的内存空间布局是：



- 构造函数的执行次序是：A()、B()、C()
- 下面的操作是合法的：

  c.fa();

  c.fb();

  c.fc();

多继承中的名冲突

```
class A
{       ......
    public:
        void f();
        void g();
};
```

```
class B
{       ......
    public:
        void f();
        void h();
};
```

```
class C: public A, public B
{       ......
    public:
        void func()
        {   f(); //Error，是A的f，还是B的f?
        }
};
......
C c;
c.f();  //Error，是A的f，还是B的f?
```

- 解决名冲突的办法是：基类名受限

```
class C: public A, public B
{       ......
   public:
        void func()
        {       A::f(); //OK，调用A的f。
                B::f(); //OK，调用B的f。
        }
};
.......
C c;
c.A::f(); //OK，调用A的f。
c.B::f(); //OK，调用B的f。
```

重复继承 - 虚基类

- 下面的类D从类A继承两次，称为重复继承：

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

- 上面的类D将包含两个x成员： B::x和C::x。

- 如果需要类D只有一个x，则把A定义为B和C的虚基类

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

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

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

- 对于包含虚基类的类：
  - 虚基类的构造函数由最新派生出的类的构造函数直接调用。
  - 虚基类的构造函数优先非虚基类的构造函数执行。

```
class A
{     int x;
  public:
      A(int i) { x = i; }
};
class B: virtual public A
{     int y;
  public:
       B(int i): A(1) { y = i; }
};
class C: virtual public A
{     int z;
  public:
      C(int i): A(2) { z = i; }
};
```

```
class D: public B, public C
{      int m;
  public:
       D(int i, int j, int k): B(i), C(j), A(3) { m = k; }
};
class E: public D
{      int n;
  public:
       E(int i, int j, int k, int l): D(i,j,k), A(4) { n = l; }
};
......
D d(1,2,3);  //A的构造函数由D调用，d.x初始化为3。
             //构造函数的执行次序是：A(3)、B(1)、C(2)、D(1,2,3)
E e(1,2,3,4);  //A的构造函数由E调用，e.x初始化为4。
               //构造函数的执行次序是：A(4)、B(1)、C(2)、D(1,2,3)、
               //E(1,2,3,4)
```