多态-虚函数

复习

- 1.设置虚基类的目的是_____。
  A）简化程序
  B）消除二义性
  C）提高运行效率
  D）减少目标代码

答案：B

- 2.关于多继承二义性的描述中，错误的是_____。

A）一个派生类的基类中都有某个同名成员，在派生类中对这个成员的访问可能出现二义性
B）解决二义性的最常用的方法是对成员名的限定法
C）基类和派生类同时出现的同名函数，也存在二义性问题
D） 一个派生类是从两个基类派生出来的，而这两个基类又有一个共同的基类，对该基类成员进行访问时，可能出现二义性

答案：C

- 下面程序段出错的是：

```
class A
   {
     int i,j;
     public:
        void get(); ------------(A)
   };
class B:A
   {
   int k;        ----------------(B)
   public;
   void make(); ----------------(C)
   };
   void B:: make()
   {
     k=i*j;    ----------------(D)
   }
```

答案：D错误，因为i和j是私有变量，在B类中不可见

面向对象程序设计的多态性

- 在C++中，把类看作类型，把以public方式继承的派生类看作是基类的子类型。这就产生了下面的多态：
  - 对象类型的多态：派生类对象的类型既可以是派生类，也可以是基类，即一个对象可以属于多种类型。
  - 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，即一个对象标识符可以属于多种类型，它可以标识多种对象。
  - 消息的多态：一个可以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解释（处理）。

多态性带来的消息绑定问题

```
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 ？
    ......
}
......
A a;
func1(a);
func2(&a);
B b;
func1(b);
func2(&b);
```

静态消息绑定：编译时刻决定多调用的函数

动态消息绑定：运行时刻确定

C++采用静态绑定，所以以上均调用A::f,实现动态绑定要使用虚函数

消息的动态绑定

- 函数调用与函数体之间的联系在运行时才建立，也就是运行时刻才决定所调用的函数，叫做“动态绑定”，或“动态连接”，“动态联编”。

虚函数

在上述例子中，只要把f声明为虚函数，就可以实现动态绑定：

```
class A
{       int x,y;
   public:
        virtual void f(); //虚函数
};
class B: public A
{       int z;
   public:
        void f(); //基类定义了虚函数后，派生类的f默认为虚函数
        void g();
};
```

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

- 虚函数的动态绑定隐含着：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：派生类中定义的成员函数的名字、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相同，或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(); //返回类型也可为B，对A类中成员f的重定义。
       void f(int); //新定义的成员
       void g(); //新定义的成员。
};
```

- 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。
  - 静态成员函数不能是虚函数。
  - 构造函数不能是虚函数。
  - 析构函数可以（往往）是虚函数。

例：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
```

```
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 *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

- 纯虚函数是指只给出函数声明而没给出实现（包括在类定义的内部和外部）的虚函数，例如：

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

- 包含纯虚函数的类称为抽象类 ，抽象类不能用于创建对象。 例如：

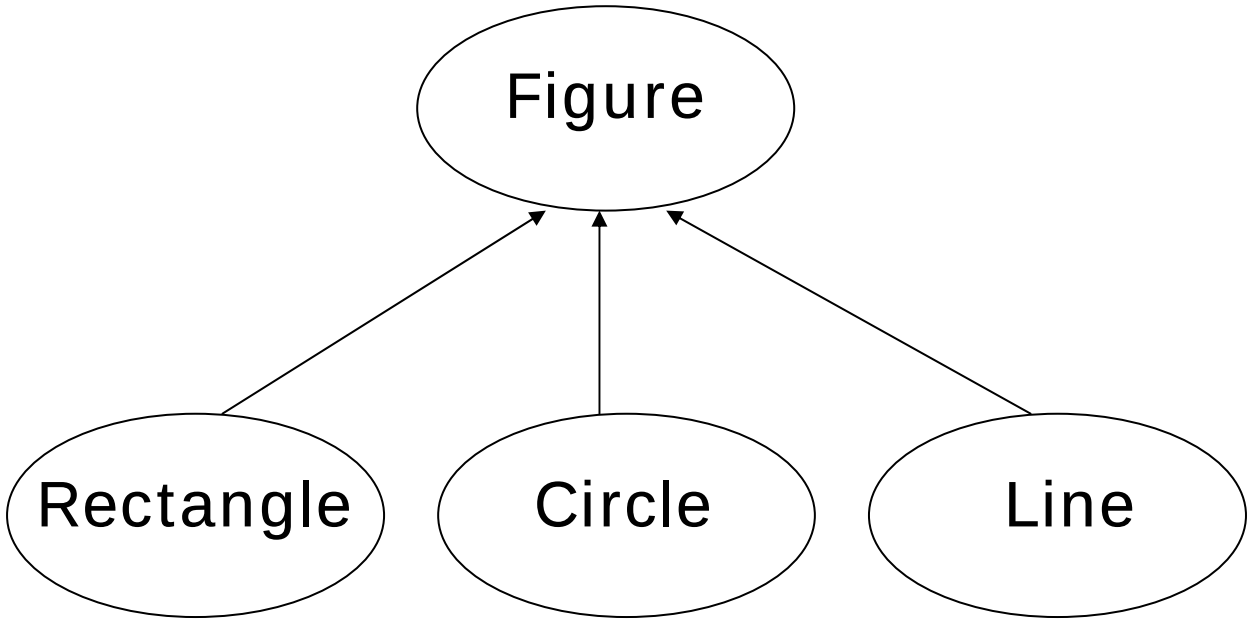
```
class A //抽象类
{ ......
  public:
    virtual int f()=0; //纯虚函数
  ......
};
......
A a;  //Error，A是抽象类
```

- 抽象类的作用是为派生类提供一个基本框架和一个公共的对外接口

抽象类的使用规定

- 抽象类只能用作其他类的基类，不能建立抽象类对象
- 抽象类不能用作参数类型、函数返回类型或显式转换的类型
- 可以声明指向抽象类的指针和引用，此指针可以指向其派生类，从而实现多态性

例：用抽象类和继承解决各种图形的处理问题



```
class Figure
{  public:
     virtual void draw() const=0;
     virtual void input_data()=0;
};
class Rectangle: public Figure
{    double left,top,right,bottom;
   public:
     void draw() const
     {    ..... //画矩形
     }
     void input_data()
     {    cout << "请输入矩形的左上角和右下角坐标 (x1,y1,x2,y2) : ";
          cin >> left >> top >> right >> bottom;
     }
     double area() const
     { return (bottom-top)*(right-left); }
};
```

```
const double PI=3.1416;
class Circle: public Figure
{   double x,y,r;
  public:
    void draw() const
    {   ..... //画圆
    }
    void input_data()
    {   cout << "请输入圆的圆心坐标和半径 (x,y,r) : ";
        cin >> x >> y >> r;
    }
    double area() const { return r*r*PI; }
};
```

```
class Line: public Figure
{   double x1,y1,x2,y2;
  public:
    void draw() const
    {   ..... //画线
    }
    void input_data()
    {   cout << "请输入线段的起点和终点坐标 (x1,y1,x2,y2) : ";
        cin >> x1 >> y1 >> x2 >> y2;
    }
};
.....
const int MAX_NUM_OF_FIGURES=100;
Figure *figures[MAX_NUM_OF_FIGURES];
int count=0;
```

- 图形数据的输入：

```
for (count=0; count<MAX_NUM_OF_FIGURES; count++)
{  int shape;
   do
   {  cout << "请输入图形的种类(0：线段，1：矩形，2：圆，-1：结束)：";
      cin >> shape;
   } while (shape < -1 || shape > 2);
   if (shape == -1) break;
   switch (shape)
   {  case 0: //线
         figures[count] = new Line; break;
      case 1: //矩形
         figures[count] = new Rectangle; break;
      case 2: //圆
         figures[count] = new Circle; break;
   }
   figures[count]->input_data(); //动态绑定到相应类的input_data
}
```

- 图形的输出：

```
for (int i=0; i<count; i++)
  figures[i]->draw();
      //通过动态绑定调用相应类的draw。
```

例：用抽象类为栈的两个不同实现提供一个公共接口

- 用数组实现的栈类和用链表实现的栈，在C++中式不同的类型
- 用户希望使用栈这一个抽象数据类型，不希望关心具体用数组还是链表来实现
- 如何解决？

```
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
   ......
}
```

小结

- 多态性的概念
- 静态绑定和动态绑定
- 消息动态绑定的各种情况
- 虚函数
- 抽象类