1.B 2.D 3C 4B 5D 6C 7D 8 B 9D 10C 11C 12C 13D 14A 15C

16 —> 17this指针 18友元 19构造函数 20delet pa 21不是

22 纯虚函数 纯虚函数 抽象类 23首先派生类，然后基类 24重载多态,强制多态,包含多态,参数多态 25不可以 可以 26虚基类 作用域分辩符 27 static 使用对象名和点操作符

28出代码的控制权，同时附带传递一些信息 异常处理块

29．用一个对象去初始化本类的另一个对象时。

函数的形参是类的对象，在进行形参和实参的结合时。

函数的返回值是类的对象，函数执行完返回时。

30编译多态性也称静态多态性，它是通过函数重载和运算符重载实现的。  
运行多态性也称动态多态性，它是通过继承和虚函数来实现的。

31

1、易维护

采用面向对象思想设计的结构，可读性高，由于继承的存在，即使改变需求，那么维护

也只是在局部模块，所以维护起来是非常方便和较低成本的。

2、质量高

在设计时，可重用现有的，在以前的项目的领域中已被测试过的类使系统满足业务需求

并具有较高的质量。

3、效率高

在软件开发时，根据设计的需要对现实世界的事物进行抽象，产生类。使用这样的方法.

解决问题，接近于日常生活和自然的思考方式，势必提高软件开发的效率和质量。

4、易扩展

由于继承、封装、多态的特性，使得系统更灵活、更容易扩展，而且成本较低。

缺点:

类和继承等特点使得程序会多很多指针操作来定位函数入口和自身要维护虛拟方法表

等额外的工作，程序的处理效率相对要低(但程序开发效率高)。

32 20

20

30

20

33Instance 1 is constructed.

Instance 1 is constructed.

Instance 2 is constructed.

34 i: 10

j：12

35

B' s constructor called.

B s default cons tructor called.

C s constructor called.

5

6

s destructor called.

B' s destructor called.

B' s destructor called.

36第12行改为B(int i, int j, int k, int l) :A(i, j) { x = k; y = l; } //构造函数

37 virtual 3.14\*r\*r

38 //类的定义(Complex.h)

class Complex

{

private:

double m\_real; //复数的实部

double m\_image; //复数的虚部

public:

// 默认构造函数,内联函数

Complex()

{

m\_real = 0; m\_image = 0;

}

// 构造函数

Complex(double r, double i);

//拷贝构造函数

Complex(const Complex& c);

//以下为运算符重载的函数

Complex operator+(const Complex& c) const;//加法运算

Complex operator-(const Complex& c) const; //减法运算

Complex operator\*(const Complex& c) const; //乘法运算

//输出复数的内容

void print();

};

#include <iostream>

using namespace std;

#include "Complex.h"

//有参构造函数

Complex::Complex(double r, double i)

{

m\_real = r;

m\_image = i;

}

//拷贝构造函数

Complex::Complex(const Complex& c)

{

m\_real = c.m\_real;

m\_image = c.m\_image;

}

//加法运算

Complex Complex::operator+(const Complex& c) const

{

Complex rtn;

rtn.m\_real = m\_real + c.m\_real;

rtn.m\_image = m\_image + c.m\_image;

return rtn;

}

Complex Complex::operator-(const Complex& c) const

{

Complex rtn;

rtn.m\_real = m\_real - c.m\_real;

rtn.m\_image = m\_image - c.m\_image;

return rtn;

}

Complex Complex::operator\*(const Complex& c) const

{

Complex rtn;

rtn.m\_real = m\_real \* c.m\_real;

rtn.m\_image = m\_image \* c.m\_image;

return rtn;

}

void main() {

Complex c1(2.0, 3.0);

Complex c2(1.0, 2.0);

cout << c1 + c2 << endl;

cout << c1 - c2 << endl;

cout << c1 \* c2 << endl;

}

39

class Point

{

public:

double x, y;

};

class LineString

{ //由多个点组成的线串类，由 n 个点组成的折线包含 n-1 个线段

public: LineString(Point\* pnts, int num);

//构造函数,通过传入点数组来构造

LineString(const LineString& another); // 复制构造函数

~LineString(); // 析构函数

LineString& operator =(const LineString& rhs); // 赋值函数

Point& operator[](int index);//返回线串中第 index 个点的引用

private: Point\* m\_data;

// 用于保存组成线串的点

int m\_num;

// 线串中点的数量 };

};

LineString::LineString(Point\* pnts, int num) {

m\_data = pnts;

m\_num = num;

}

LineString::LineString(const LineString& another) {

m\_data = another.m\_data;

m\_num = another.m\_num;

}

LineString::~LineString() {

}

LineString& LineString::operator =(const LineString& rhs) {

m\_data = rhs.m\_data;

m\_num = rhs.m\_num;

}

Point& LineString::operator[](int index) {

Point t;

t.x = m\_data[index - 1].x;

t.y = m\_data[index - 1].y;

return t;

}