**面向对象程序设计检测卷**

姓名：赵文鹏 学号：20191001981

1. 单项选择题

BDCBB ADBDD BCDAD

1. 填空题与简答题

16、->

17、this指针

18、友元函数

19、成员初始化列表

20、delete [ ]pa;

21、不是

22、纯虚函数 纯虚函数 抽象类

23、先派生类，再基类

24、参数多态、包含多态、重载多态

25、不可以 可以

26、虚基类或作用域分辨符

27、static 使用对象名加点操作符访问或：：

28、拋出异常 捕获异常

29、列举列举拷贝构造函数调用的时机。

1.当用类的一个对象去初始化该类的另一个对象（或引用）时系统自动调用拷贝构造函数实现拷贝赋值。

2.若函数的形参为类对象，调用函数时，实参赋值给形参，系统自动调用拷贝构造函数。

3.当函数的返回值是类对象时，系统自动调用拷贝构造函数。

30．编译时的多态性与运行时的多态性有何不同？分别怎样实现？

编译时的多态是度通过静态连编来实现的；运行时的多态是用动态连编来实现的。

编译时多态性是通过函数重载（包括运算符重载）和模板实现的。利用函数重载机制，在调用同名的函数时，编译系统可根据实参的具体情况确立所要调用的是哪个函数。

运行时多态性是通过虚函数来实现的。

31．简述面向对象程序设计的优点和不足。

优点：

1、维护简单。面向对象程序设计的一个特征就是模块化。实体可以被表示为类百以及同一名字空间中具有相同功能的类，可以在名字空间中添加一个类而不影响该名字空间的其他成员。度这种特征为程序的维护提供了便捷性。

2、可扩充性。如果有一个具有某一种功能的类，就可以扩充这个类，创建一个具有扩充功能的类。

3、代码重用功能是被封装在类中的，类是作为一个独立实体而存在的，因此可以很简单的提供类库，使代码得以重复使用。

不足：

初学者不易接受、难学。

（1）没有准确的定义：很难提供一个准确的定义来说明面向对象DBMS应建成什么样，这是因为该名称已经应用到很多不同的产品和原型中，而这些产品和原型考虑的方面可能不一样。  
（2）维护困难：随着组织信息需求的改变，对象的定义也要求改变并且需移植现有数据库，以完成新对象的定义。当改变对象的定义和移植数据库时，它可能面临真正的挑战。  
（3）不适合所有的应用：面向对象数据模型用于需要管理数据对象之间存在的复杂关系的应用，它们特别适合于特定的应用，例如工程、电子商务、医疗等，但并不适合所有应用。当用于普通应用时，其性能会降低并要求很高的处理能力。

1. 看程序写结果

32、

20

20

30

20

33、

Instance 1 is constructed.

Instance 1 is constructed.

Instance 2 is constructed.

34、

i:10

j:12

35、

B's constructor called.

B's default constructor called.

C's constructor called.

5

6

C's destructor called.

B's destructor called.

B's destructor called.

1. 综合题

36、

12行（i，j）前面缺少基类的名称。类A不存在相应的构造函数。

修改为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

1. 编程题

38、

//complex.h

#include<iostream>

using namespace std;

class Complex {

private:

double real;

double image;

public:

//构造函数

Complex() {

real = 0.0;

image = 0.0;

};

Complex(double a, double b){

real = a;

image = b;

}

Complex(const Complex& c); //拷贝构造函数

//输出重载

friend ostream& operator<<(ostream & out, const Complex& c);

//运算符重载

friend Complex operator+(const Complex& c1, const Complex& c2);

friend Complex operator-(const Complex& c1, const Complex& c2);

friend Complex operator\*(const Complex& c1, const Complex& c2);

};

//complex.cpp

#include"complex.h"

//复制构造函数

Complex::Complex(const Complex& c)

{ real = c.real; image = c.image; }

//重载输出符号

ostream& operator<<(ostream& out, const Complex& c)

{ if (c.image > 0) out << c.real << '+' << c.image << 'i';

if (c.image < 0) out << c.real << c.image << 'i';

if (c.image == 0) out << c.real;

return out; }

//+

Complex operator + (const Complex & c1, const Complex & c2)

{

Complex t;

t.real =c1. real + c2.real;

t.image = c1.image + c2.image;

return t;

}

//-

Complex operator - (const Complex & c1, const Complex & c2)

{

Complex t;

t.real = c1.real - c2.real;

t.image = c1.image - c2.image;

return t;

}

//\*

Complex operator\* (const Complex& c1, const Complex& c2)

{

Complex t;

t.real = c1.real \* c2.real - c1.image \* c2.image;

t.image = c1.image \* c2.real + c1.real \* c2.image;

return t;

}

void main() {

Complex c1(2.0, 3.0);

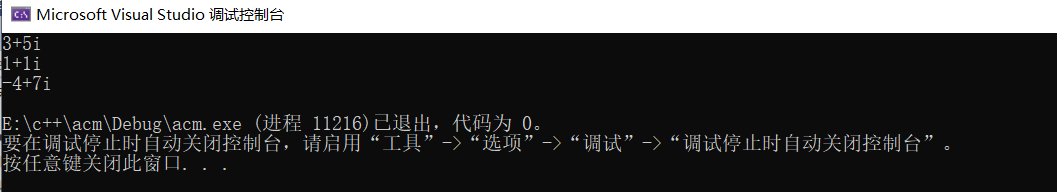
Complex c2(1.0, 2.0);

cout << c1 + c2 << endl;

cout << c1 - c2 << endl;

cout << c1 \* c2 << endl;

}



39、

//构造函数

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

{

m\_data = new Point[num];

for (int i = 0; i < num; i++)

{

m\_data[i] = pnts[i];

}

m\_num = num;

}

//复制构造函数

LineString::LineString(const LineString& another)

{

m\_num = another.m\_num;

m\_data = new Point[m\_num];

for (int i = 0; i < m\_num; i++)

{

m\_data[i] = another.m\_data[i];

}

}

//析构函数

LineString::~LineString()

{

delete[]m\_data;

}

// 赋值函数

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

{

delete[]this->m\_data;

this->m\_num = rhs.m\_num;

m\_data = new Point[m\_num];

for (int i = 0; i < m\_num; i++)

{

m\_data[i] = rhs.m\_data[i];

}

}

//返回线串中第 index 个点的引用

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

{

return m\_data[index - 1];

}