一、单项选择题

1.B

2.D

3.C

4.B

5.B

6.D

7.D

8.B

9.D

10.D

11.D

12.C

13.D

14.A

15.D

1. 填空与简答题

16.->

17.this指针

18.友元

19.初始值列表

20.delete[]pa;

21.不是

22.纯虚函数 纯虚函数 抽象类

23.先执行派生类析构函数的函数体，然后析构派生类自身的成员，再按照继承链逐个析构基类的成员

24.重载多态 参数多态 包含多态

1. 不可以 可以
2. 虚基类 作用域运算符

27.static 范围解析

28.抛出异常 捕获异常

29.（1）一个对象以值传递的方式传入函数体   
（2） 一个对象以值传递的方式从函数返回   
（3） 一个对象需要通过另外一个对象进行初始化

1. 编译时多态性指在程序编译时确定调用哪个函数，通过函数重载和模板实例化实现；运行时多态性指程序运行时才决定调用的函数，通过继承关系和虚函数实现。
2. 优点：面向对象更能应对复杂类型的程序开发，拥有丰富的特性，其中封装性保护了数据，提高了代码的易用性，抽象性使得代码易于维护，继承及多态性提高了代码的复用性及扩展性

不足：代码不利于理解

1. 看程序写结果
2. 20

20

30

20

1. Instance1is constructed

Instance1is constructed

Instance2is constructed

1. i:10

J:12

1. B’s constructor called.

C’s constructor called.

5

6

C’s destructor called.

B’s destructor called.

1. 综合题
2. 12行 类A无默认构造函数

B(int i,int j,int k,int l):A(i,j){x=k;y=l;}

15行 类A的show函数被类b的show函数屏蔽

void fl(){A::Show();}

1. 程序填空
2. virtual
3. 3.14\*r\*r
4. 编程题

38.

**//Complex.h**

#pragma once

#include<iostream>

using namespace std;

class Complex{ //Complex类

public:

Complex(double x=0, double y=0) {real\_part = x; imaginary\_part= y;}

~Complex(){}

Complex operator+(const Complex& rhs); //重载运算符+

Complex operator-(const Complex& rhs); //重载运算符-

Complex operator\*(const Complex& rhs); //重载运算符\*

friend ostream& operator<<(ostream& os,const Complex& rhs); //重载输出运算符

private:

double real\_part, imaginary\_part;

};

**//Complex.cpp**

#include”Complex.h”

#include<iostream>

using namespace std;

Complex Complex::operator+(const Complex& rhs){ //重载运算符+

this->real\_part += rhs.real\_part;

this->imaginary\_part += rhs.imaginary\_part;

return \*this;

}

Complex Complex::operator-(const Complex& rhs){ //重载运算符-

this->real\_part -= rhs.real\_part;

this->imaginary\_part -= rhs.imaginary\_part;

return \*this;

}

Complex Complex::operator\*(const Complex& rhs){ //重载运算符\*

this->real\_part = this->real\_part \* rhs.real\_part - this->imaginary\_part \* rhs.imaginary\_part;

this->imaginary\_part = this->real\_part \* rhs.imaginary\_part - this->imaginary\_part \* rhs.real\_part;

return \*this;

}

ostream& operator<<(ostream& os, const Complex& rhs){ //重载输出运算符

os << rhs.real\_part << "+" << rhs.imaginary\_part << "i" ;

return os;

}

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

m\_num = num;

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

for (int i = 0; i < m\_num; i++)m\_data[i] = pnts[i];

}

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

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];

}

~LineString() { delete[]m\_data; m\_num = 0; } //析构函数

LineString\* operator=(const LineString& rhs) {//赋值函数

m\_num = rhs.m\_num;

delete[]m\_data;

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

for (int i = 0; i < m\_num; i++)m\_data[i] = rhs.m\_data[i];

return this;

}

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

return m\_data[index];

}

private:

Point\* m\_data;//用于保存组成线串的点

int m\_num; //线串中点的数量

};