16．用指向对象的指针引用对象成员使用操作符 【1】-> 。

17．当一个对象生成以后，系统就为这个对象定义了一个 【2】 this指针 ，它指向这个对象的地址。

18．非成员函数应声明为类的 【3】友元 函数才能访问这个类的 private 成员。

19．对于常量数据成员和引用数据成员的初始化只能通过 【4】初始化列表 来完成。

20．调用语句 int\* pa = new int[10];动态分配内存 pa，释放 pa 的语句为 【5】delete []pa; 。

21．引用 【6】 不是（填是或者不是）一种数据类型，所以没有引用的指针。

22． 【7】 纯虚函数 是一种特别的虚函数, 拥有 【8】纯虚函数 的类就是 【9】抽象类 , 这种类不能实例化。

23．含有继承关系和子对象的类构造对象时，析构函数的调用顺序是 【10】子对象的析构，派生类的数据成员，基类的析构 。

24．C++支持四种类型的多态，分别是强制多态、 【11】参数多态 、 【12】 包含多态 、 【13】重载多态 。

1. 类的构造函数 【14】不可以 (可以/不可以)是虚函数，类的析构函数 【15】可以 (可以/不可以)是虚函数。
2. 多继承时，多个基类中的同名的成员在派生类中由于标识符不唯一而出现二义性。在派生类中可以采

用 【16】类名限定 或 【17】同名隐藏 来消除该问题。

27．为类而非对象定义数据成员的关键字是 【18】 static ，其访问使用 【19】:: 操作符。

28．throw;语句的作用是 【20】 抛出异常 ，catch(…)语句的作用是 【21】 捕获异常 。

29．列举拷贝构造函数调用的时机。

1、对象以值传递的方式传入函数参数

2、对象以值传递的方式从函数返回

3、对象需要通过另外一个对象进行初始化

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．简述面向对象程序设计的优点和不足。

优点：降低程序的复杂性、提高软件的开发效率和改善工作界面。

不足：面向对象设计方法需要一定的软件基础支持才可以应用，另外在大型的MIS开发中如果不经自顶向下的整体划分，而是一开始就自底向上的采用面向对象设计方法开发系统，同样也会造成系统结构不合理、各部分关系失调等问题

1. CDCBB DDBDD DCDBD

二．

三．

结果：32. 20

20

30

20

33.Instance 1 is constructed.

Instance 1 is constructed.

Instance 2 is constructed.

34.10

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.

四．

36.class B:private A改为class B:public A

B(int I,int j,int k, int l):(i,j)改为B(int I,int j,int k, int l):A(i,j)

37.（1）virtual

(2) 3.14\*r\*r

五．38.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

friend Complex operator<<(ostream& out, const Complex& c) ;//输出运算

Complex.cpp

#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;

}//减法运算

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

{

Complex rtn;

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

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

return rtn;

}//乘法运算

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

{

out << c.m\_real << "+" << c.m\_image << "i";

}

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

m\_data = pnts;

m\_num = num;

}

LineString::~LineString() {} // 析构函数

LineString::LineString(const LineString& another) {

m\_data = another.m\_data;

m\_num = another.m\_num;

}// 复制构造函数

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

m\_data = rhs.m\_data;

m\_num = rhs.m\_num;

}// 赋值函数

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

m\_data[index];

}//返回线串中第 index 个点的引用