一． 选择题 (含 20 个小题，每小题 2 分，计 40 分)

1—5	C	B	B	B	C
6—10	A	D	B	A	D
11—15	D	C	D	D	C
16—20	B	B	A	C	D

二．填空题 (含 5 个小题，每空 2 分，计 18 分)

21.	.h（**1 分**）　　.cpp（**1 分**）
22.	vector、deque、list、map、multimap、set multiset …… （**各 1 分**）

23.	（1）如果对一个对象操作之后修改了这块空间的内容，则另一个对象将会受到影响。如果不是设计者特意所为，这将是一个隐藏的错误（浅复制）。
	当对象 a1 和 a2 消亡时，将会分别去调用它们的析构函数，这会使得同一块内存区域将被归还两次，从而导致程序运行异常。
	（2 分）
	
	（2）解决上面问题的办法是在类 A 中显式定义一个拷贝构造函数，实现深复制。
	A::A(const A& a)
	{　x = a.x;
	　y = a.y;
	　size = a.size;
	　p = new int[size];
	　for (int i = 0; i < size; i++)
	　　p[i] = a.p[i];
	}
	（2 分）

24.	B::f() A::f() A::g() A::h()、B::f()、A::g() B::B()

	A::~A() **（各 1 分）**

三．简答题 (含 5 个小题，每小题 4 分，计 20 分)

25.	• 虚函数（Virtual Function）：在面向对象的编程中，虚函数是一种特殊的成员函数，它可以在派生类中被重写。通过在基类中声明虚函数，在派生类中重新定义该函数，可以实现多态性，即在运行时根据对象的实际类型调用相应的函数。通过使用虚函数，可以实现动态绑定和运行时多态。 **（1 分）** • 纯虚函数（Pure Virtual Function）：纯虚函数是在基类中声明的没有实际实现的虚函数，它的声明形式为 virtual <成员函数声明> = 0;。纯虚函数的存在意味着基类只提供接口，而不提供默认的实现。派生类必须实现纯虚函数才能被实例化。纯虚函数使得基类成为抽象类。 **（1 分）** • 虚基类（Virtual Base Class）：虚基类用于解决多继承中的菱形继承问题，即派生类通过多个路径继承同一个基类，导致基类在派生类中出现多次。通过将共同基类声明为虚基类，可以确保只有一份基类的实例被派生类共享，避免了数据的冗余和二义性。 **（1 分）** • 抽象类（Abstract Class）：抽象类是包含纯虚函数的类，它不能被实例化。抽象类用于定义接口和基本行为，要求派生类实现纯虚函数才能创建对象。抽象类可以包含非纯虚函数和数据成员，但不能直接创建对象。抽象类提供了一种接口定义的机制，可以通过继承抽象类来实现多态性和共享接口。 **（1 分）** 总结： 虚函数允许在派生类中重新定义基类中的函数，实现多态性和动态绑定。 纯虚函数是没有实际实现的虚函数，使得基类成为抽象类，要求派生类实现纯虚函数。 虚基类用于解决多继承中的菱形继承问题，确保只有一份基类的实例被派生类共享。 抽象类是包含纯虚函数的类，不能直接创建对象，用于定义接口和基本行为。
26.	this 指针是一个隐含于每个非静态成员函数的特殊指针，指向当前对象的地址。下面是三种常见的情况，其中 this 指针会被使用：

	1、在成员函数中访问当前对象的成员变量；**（1 分）** 2、在成员函数中返回当前对象的引用；**（1 分）** 3、在类的构造函数或析构函数中使用 this 指针进行对象的初始化或清理。**（1 分）** 不能使用 this 指针的情况是在静态成员函数中。静态成员函数不属于任何对象，因此没有 this 指针。**（1 分）**
27.	复制构造函数是一种特殊的构造函数，用于创建一个新对象并将其初始化为与现有对象相同的副本。复制构造函数主要在以下三种情况下被调用： 1、利用现有对象初始化同类型对象时；**（2 分）** 2、把对象作为值参数传给函数时；**（1 分）** 3、把对象作为值返回时。**（1 分）**
28.	一种需要显式定义复制构造函数的情况是当类中包含指针成员变量时。在默认情况下，C++ 会提供一个合成的复制构造函数，该构造函数执行浅拷贝，即简单地复制指针的值，而不是创建新的指针和复制指针所指向的内容。这可能会导致问题，因为在多个对象共享同一个指针时，当其中一个对象销毁时，会释放指针所指向的内存，导致其他对象访问无效的内存。 为了避免这种问题，需要显式定义复制构造函数，并执行深拷贝，即创建新的指针并复制指针所指向的内容。 **（4 分）**
29.	操作符重载是一种 C++的特性，允许我们重新定义操作符的行为，使其适用于自定义类型的对象。通过操作符重载，我们可以对自定义类型执行类似于内置类型的操作，使得代码更加简洁、直观和易于理解。操作符重载的作用包括： 1、增加代码的可读性和可维护性：通过重载操作符，我们可以使用类似于内置类型的语法来操作自定义类型的对象，使代码更加自然和易于理解。 2、提供自定义类型的语义一致性：通过重载操作符，我们可以定义自定义类型的操作行为，使其与内置类型的操作行为保持一致，从而提供一致的语义。 3、方便使用和表达：操作符重载可以简化对自定义类型对象的操作，使代码更加简洁和表达力强。 **（2 分）** 操作符重载的两种主要方式： 1、作为一个类的非静态的成员函数方式来实现； 2、用带有类、结构、枚举以及它们的引用类型参数的全局函数来实现。 **（2 分）**

四．设计题 (含 2 个小题，每小题 11 分，计 22 分)

30.

第一小题1分
第二小题1分
第三小题1分
第四小题，前置++重载2分，后置++重载2分
第五小题2分
Main函数1分

```
#include <iostream>
using namespace std;

class Clock {
private:
    int hour, minute, second;
public:
    Clock(int h=0, int m=0, int s=0) : hour(h), minute(m), second(s) {}
    Clock(const Clock &c) {
        hour = c.hour;
        minute = c.minute;
        second = c.second;
    }
    void setTime(int new_h, int new_m, int new_s) {
        hour = new_h;
        minute = new_m;
        second = new_s;
    }

    Clock & operator ++() {
        second++;
        if (second >= 60) {
            second -= 60;
            minute++;
            if (minute >= 60) {
                minute -= 60;
                hour = (hour + 1) % 24;
            }
        }
        return *this;
    }

    Clock operator ++ (int) {
```

```
        Clock old = *this;
        ++(*this);
        return old;
    }

    friend ostream& operator << (ostream& out, const Clock &c) {
        out << c.hour << ":" << c.minute << ":" << c.second << endl;
        return out;
    }
};

int main() {
    Clock myClock(23, 59, 59);     // 自定义
    cout << "First time output: " << endl;
    cout << myClock;

    cout << "Show myClock++: " << endl;
    cout << (myClock++);

    cout << "Show ++myClock: " << endl;
    cout << (++myClock);


    return 0;
}
```

32.

定义基类Shape 1分
定义派生类Cuboid 1分
定义派生类Cylinder 1分
定义派生类Sphere 1分
正确使用基类对象指针指向派生类对象 2分
正确使用基类对象指针实例化vector并存放派生类对象 3分
正确使用STL的排序功能 3分（未使用STL而使用其他排序方法的，1分）

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

const double PI = 3.14159;

class Shape {
public:
```

```
    virtual double volume() const { return 0.0; }
    virtual void display() const { cout << "Shape of volume: " << volume()
<< endl; }
};

class Cuboid : public Shape {
private:
    double l, w, h;
public:
    Cuboid(double new_l, double new_w, double new_h) : l(new_l),
w(new_w), h(new_h) {}
    double volume() const { return l*w*h; }
    void display() const { cout << "Cuboid of volume: " << volume() <<
endl; }
};

class Cylinder : public Shape {
private:
    double r, h;
public:
    Cylinder(double new_r, double new_h) : r(new_r), h(new_h) {}
    double volume() const { return PI*r*r*h; }
    void display() const { cout << "Cylinder of volume: " << volume() <<
endl; }
};

class Sphere : public Shape {
private:
    double r;
public:
    Sphere(double new_r) : r(new_r) {}
    double volume() const { return 4.0/3.0*PI*r*r*r; }
    void display() const { cout << "Sphere of volume: " << volume() <<
endl; }
};

bool compareVolume(const Shape *s1, const Shape *s2) {
    return s1->volume() < s2->volume();
}

int main() {

    Shape *p1, *p2, *p3;
```

```
    Cuboid cu1(2.0, 2.0, 2.0);     // 自定义
    Cylinder cy1(2.0, 2.0);        // 自定义
    Sphere sp1(1.0);               // 自定义

    p1 = &cu1;
    p2 = &cy1;
    p3 = &sp1;

   //注意：vector<Shape> s无法实现动态绑定，是错误的！
    vector<Shape*> s;
    s.push_back(p1);
    s.push_back(p2);
    s.push_back(p3);

    sort(s.begin(), s.end(), compareVolume);

    s[0]->display();
    s[1]->display();
    s[2]->display();


    return 0;
}
```