1.new 、 delete 、 malloc 、 free 关系

delete 会调用对象的析构函数 , 和 new 对应 free 只会释放内存， new 调用构造函数。 malloc 与 free 是 C++/C 语言的标准库函数， new/delete 是 C++ 的运算符。它们都可用于申请动态内存和释放内存。对于非内部数据类型的对象而言，光用 maloc/free 无法满足动态对象的要求。对象在创建的同时要自动执行构造函数，对象在消亡之前要自动执行析构函数。由于 malloc/free 是库函数而不是运算符，不在编译器控制权限之内，不能够把执行构造函数和析构函数的任务强加于 malloc/free 。因此 C++ 语言需要一个能完成动态内存分配和初始化工作的运算符 new ，以及一个能完成清理与释放内存工作的运算符 delete 。注意 new/delete 不是库函数。

总结：new和delete会自动调用对象的构造与析构函数而malloc与free不会；

new和delete式C++运算符，而malloc和free是C/C++标准库函数。  
——————————————————————————————–  
2.delete 与 delete [] 区别

delete 只会调用一次析构函数，而 delete[] 会调用每一个成员的析构函数。在 More Effective C++ 中有更为详细的解释：“当 delete 操作符用于数组时，它为每个数组元素调用析构函数，然后调用 operatordelete 来释放内存。” delete 与 New 配套， delete [] 与 new [] 配套

MemTest\*mTest1=newMemTest[10];

MemTest\*mTest2=newMemTest;

int\*pInt1=newint[10];

int\*pInt2=newint;

delete[]pInt1; //-1-

delete[]pInt2; //-2-

delete[]mTest1;//-3-

delete[]mTest2;//-4-

在 -4- 处报错。

这就说明：对于内建简单数据类型， delete 和 delete[] 功能是相同的。对于自定义的复杂数据类型， delete 和 delete[] 不能互用。 delete[] 删除一个数组， delete 删除一个指针简单来说，用 new 分配的内存用 delete 删除用 new[] 分配的内存用 delete[] 删除 delete[] 会调用数组元素的析构函数。内部数据类型没有析构函数，所以问题不大。如果你在用 delete 时没用括号， delete 就会认为指向的是单个对象，否则，它就会认为指向的是一个数组。

总结： delete 只会调用一次析构函数，而 delete[] 会调用每一个成员的析构函数。  
——————————————————————————————–  
3.C C++ JAVA 共同点，不同之处？

相同点：都是面向对象的语言  
不同点：c/c++是编译型语言，还有一些语言完全是解释型的（如Basie），而java既是编译型的又是解释型的语言  
c/c++存在指针运算，Basie没有显示指针，而java有指针，但取消了指针的运算  
——————————————————————————————–  
4. 继承优缺点。

类继承是在编译时刻静态定义的，且可直接使用，类继承可以较方便地改变父类的实现。但是类继承也有一些不足之处。首先，因为继承在编译时刻就定义了，所以无法在运行时刻改变从父类继承的实现。更糟的是，父类通常至少定义了子类的部分行为，父类的任何改变都可能影响子类的行为。如果继承下来的实现不适合解决新的问题，则父类必须重写或被其他更适合的类替换。这种依赖关系限制了灵活性并最终限制了复用性。  
——————————————————————————————–  
5.C++ 有哪些性质（面向对象特点）  
封装，继承和多态。

在面向对象程序设计语言中，封装是利用可重用成分构造软件系统的特性，它不仅支持系统的可重用性，而且还有利于提高系统的可扩充性；消息传递可以实现发送一个通用的消息而调用不同的方法；封装是实现信息隐蔽的一种技术，其目的是使类的定义和实现分离。  
——————————————————————————————–  
6. 子类析构时要调用父类的析构函数吗？  
析构函数调用的次序是先派生类的析构后基类的析构，也就是说在基类的的析构调用的时候 , 派生类的信息已经全部销毁了定义一个对象时先调用基类的构造函数、然后调用派生类的构造函数；析构的时候恰好相反：先调用派生类的析构函数、然后调用基类的析构函数 JAVA 无析构函数深拷贝和浅拷贝  
——————————————————————————————–  
7. 多态，虚函数，纯虚函数

这么一大堆名词，实际上就围绕一件事展开，就是多态，其他三个名词都是为实现C++的多态机制而提出的一些规则，下面分两部分介绍，第一部分介绍【多态】，第二部分介绍【虚函数，纯虚函数，抽象类】  
一 【多态】  
多态的概念 ：关于多态，好几种说法，好的坏的都有，分别说一下：  
1 指同一个函数的多种形态。  
个人认为这是一种高手中的高手喜欢的说法，对于一般开发人员是一种差的不能再差的概念，简直是对人的误导，然人很容易就靠到函数重载上了。

以下是个人认为解释的比较好的两种说法，意思大体相同：  
2多态是具有表现多种形态的能力的特征，在OO中是指，语言具有根据对象的类型以不同方式处理之，特别是重载方法和继承类这种形式的能力。  
这种说法有点绕，仔细想想，这才是C++要告诉我们的。

3多态性是允许你将父对象设置成为和一个或更多的他的子对象相等的技术，赋值之后，父对象就可以根据当前赋值给它的子对象的特性以不同的方式运作。简单的说，就是一句话：允许将子类类型的指针赋值给父类类型的指针。多态性在Object Pascal和C++中都是通过虚函数（Virtual Function） 实现的。  
这种说法看来是又易懂，又全面的一种，尤其是最后一句，直接点出了虚函数与多态性的关系，如果你还是不太懂，没关系，再把3读两遍，有个印象，往后看吧。  
- – - – - – - – - – - – - – - – - – - – - – - – - – - – - – -  
二 【虚函数，纯虚函数，抽象类】  
多态才说了个概念，有什么用还没说就进入第二部分了？看看概念3的最后一句，虚函数就是为多态而生的，多态的作用的介绍和虚函数简直关系太大了，就放一起说吧。

多态的作用：继承是子类使用父类的方法，而多态则是父类使用子类的方法。这是一句大白话，多态从用法上就是要用父类(确切的说是父类的对象名)去调用子类的方法，例如：  
【例一】  
class A {  
public:  
A() {}  
（virtual） void print() {  
cout << “This is A.” << endl;  
}  
};  
class B : public A {  
public:  
B() {}  
void print() {  
cout << “This is B.” << endl; 　　} 　　}; 　　int main(int argc, char\* argv[]) { 　　 B b; A a; a = b;a.print;—————————————- make1 // A &a = b; a->print();———————————-make2  
//A \*a = new B();a->print();——————————–make3  
return 0;  
}  
这将显示：  
This is B.  
如果把virtual去掉，将显示：  
This is A.  
(make1,2,3分别是对应兼容规则(后面介绍)的三种方式，调用结果是一样的)  
加上virtual ,多态了，B中的print被调用了，也就是可以实现父类使用子类的方法。  
对多态的作用有一个初步的认识了之后，再提出更官方，也是更准确的对多态作用的描述：  
多态性使得能够利用同一类(基类)类型的指针来引用不同类的对象,以及根据所引用对象的不同,以不同的方式执行相同的操作。把不同的子类对象都当作父类来看，可以屏蔽不同子类对象之间的差异，写出通用的代码，做出通用的编程，以适应需求的不断变化。赋值之后，父对象就可以根据当前赋值给它的子对象的特性以不同的方式运作(也就是可以调用子对象中对父对象的相关函数的改进方法)。

那么上面例子中为什么去掉virtual就调用的不是B中的方法了呢，明明把B的对象赋给指针a了啊，是因为C++定义了一组对象赋值的兼容规则，就是指在公有派生的情况下，对于某些场合，一个派生类的对象可以作为基类对象来使用，具体来说，就是下面三种情形：

Class A ;  
class B:public A

1. 派生的对象可以赋给基类的对象  
A a;  
B b;  
a = b;  
2. 派生的对象可以初始化基类的引用  
B b;  
A &a = b;  
3. 派生的对象的地址可以赋给指向基类的指针  
B b;  
A \*a = &b;  
或  
A \*a = new B();  
由上述对象赋值兼容规则可知，一个基类的对象可兼容派生类的对象，一个基类的指针可指向派生类的对象，一个基类的引用可引用派生类的对象，于是对于通过基类的对象指针(或引用)对成员函数的调用，编译时无法确定对象的类，而只是在运行时才能确定并由此确定调用哪个类中的成员函数。  
看看刚才的例子，根据兼容规则，B的对象根本就被当成了A的对象来使用，难怪B的方法不能被调用。  
【例二】  
#include  
using namespace std;  
class A  
{  
public:  
void (virtual) print(){cout << “A print”<

private:  
};  
class B : public A  
{  
public:  
void print(){cout << “B print”< private:  
};  
void test(A &tmpClass)  
{  
tmpClass.print();  
}  
int main(void)  
{  
B b;  
test(b);  
getchar();  
return 0;  
}  
这将显示：  
B print  
如果把virtual去掉，将显示：  
A print  
那么，为什么加了一个virtual以后就达到调用的目的了呢，多态了嘛~那么为什么加上virtual就多态了呢，我们还要介绍一个概念：联编  
函数的联编：在编译或运行将函数调用与相应的函数体连接在一起的过程。  
1 先期联编或静态联编：在编译时就能进行函数联编称为先期联编或静态联编。  
2 迟后联编或动态联编：在运行时才能进行的联编称为迟后联编或动态联编。  
那么联编与虚函数有什么关系呢，当然，造成上面例子中的矛盾的原因就是代码的联编过程采用了先期联编，使得编译时系统无法确定究竟应该调用基类中的函数还是应该调用派生类中的函数，要是能够采用上面说的迟后联编就好了，可以在运行时再判断到底是哪个对象，所以，virtual关键字的作用就是提示编译器进行迟后联编，告诉连接过程：“我是个虚的，先不要连接我，等运行时再说吧”。  
那么为什么连接的时候就知道到底是哪个对象了呢，这就引出虚函数的原理了：当编译器遇到virtual后，会为所在的类构造一个表和一个指针，那个表叫做vtbl，每个类都有自己的vtbl，vtbl的作用就是保存自己类中虚函数的地址，我们可以把vtbl形象地看成一个数组，这个数组的每个元素存放的就是虚函数的地址.指针叫做vptr，指向那个表。而这个指针保存在相应的对象当中，也就是说只有创建了对象以后才能找到相应虚函数的地址。  
【注意】  
1为确保运行时的多态定义的基类与派生类的虚函数不仅函数名要相同，其返回值及参数都必须相同，否则即使加上了virtual，系统也不进行迟后联编。  
2 虚函数关系通过继承关系自动传递给基类中同名的函数，也就是上例中如果A中print有virtual,那么 B中的print即使不加virtual,也被自动认为是虚函数。  
\*3 没有继承关系，多态机制没有意义，继承必须是公有继承。  
\*4现实中，远不只我举的这两个例子，但是大的原则都是我前面说到的“如果发现一个函数需要在派生类里有不同的表现，那么它就应该是虚的”。这句话也可以反过来说：“如果你发现基类提供了虚函数，那么你最好override它”。

纯虚函数：  
虚函数的作用是为了实现对基类与派生类中的虚函数成员的迟后联编，而纯虚函数是表明不具体实现的虚函数成员，即纯虚函数无实现代码。其作用仅仅是为其派生类提过一个统一的构架，具体实现在派生类中给出。  
一个函数声明为纯虚后，纯虚函数的意思是：我是一个抽象类！不要把我实例化！纯虚函数用来规范派生类的行为，实际上就是所谓的“接口”。它告诉使用者，我的派生类都会有这个函数。  
抽象类：  
含有一个或多个纯虚函数的类称为抽象类。  
【例三】  
#include

using namespace std;  
class A  
{  
public:  
virtual float print() = 0;  
protected:  
float h,w;  
private:  
};  
class B : public A  
{  
public:  
B(float h0,float w0){h = h0;w = w0;}  
float print(){return h\*w;}  
private:  
};  
class C : public A  
{  
public:  
C(float h0,float w0){h = h0;w = w0;}  
float print(){return h\*w/2;}  
private:  
};

int main(void)  
{  
A \*a1,\*a2;  
B b(1,2);  
C c(1,2);  
a1 = &b;  
a2 = &c;  
cout << a1->print()<print()<print;a2->print就不能用了)，给多态性造成不便，这里要强调的是，我们是希望用基类的指针调用派生类的方法，希望用到多态机制，如果读者并不想用基类指针，认为用b，c指针直接调用更好，那纯虚函数就没有意义了，多态也就没有意义了，了解一下多态的好处，再决定是否用纯虚函数吧。  
【注意】  
1 抽象类并不能直接定义对象，只可以如上例那样声明指针，用来指向基类派生的子类的对象，上例中的A \*a1,\*a2;该为 A a1,a2;是错误的。  
2 从一个抽象类派生的类必须提供纯虚函数的代码实现或依旧指明其为派生类，否则是错误的。  
3 当一个类打算被用作其它类的基类时，它的析构函数必须是虚的。  
【例三】  
class A  
{  
public:  
A() { ptra\_ = new char[10];}  
~A() { delete[] ptra\_;} // 非虚析构函数  
private:  
char \* ptra\_;  
};

class B: public A  
{  
public:  
B() { ptrb\_ = new char[20];}  
~B() { delete[] ptrb\_;}  
private:  
char \* ptrb\_;  
};

void foo()  
{  
A \* a = new B;  
delete a;  
}  
在这个例子中，程序也许不会象你想象的那样运行，在执行delete a的时候，实际上只有A::~A()被调用了，而B类的析构函数并没有被调用！这是否有点儿可怕？ 如果将上面A::~A()改为virtual，就可以保证B::~B()也在delete a的时候被调用了。因此基类的析构函数都必须是virtual的。纯虚的析构函数并没有什么作用，是虚的就够了。通常只有在希望将一个类变成抽象类（不能实例化的类），而这个类又没有合适的函数可以被纯虚化的时候，可以使用纯虚的析构函数来达到目的。

最后通过一个例子说明一下抽象类，纯虚函数以及多态的妙用吧：  
我们希望通过一个方法得到不同图形面积的和的方式：  
#include  
using namespace std;  
class A //定义一个抽象类，用来求图形面积  
{  
public:  
virtual float area() = 0;//定义一个计算面积的纯虚函数，图形没确定，当  
//不能确定具体实现  
protected:  
float h,w; //这里假设所有图形的面积都可以用h和w两个元素计算得出  
//就假设为高和长吧  
private:  
};  
class B : public A //定义一个求长方形面积的类  
{  
public:  
B(float h0,float w0){h = h0;w = w0;}  
float area (){return h\*w;}//基类纯虚函数的具体实现  
private:  
};  
class C : public A //定义一个求三角形面积的类  
{  
public:  
C(float h0,float w0){h = h0;w = w0;}  
float area (){return h\*w/2;}//基类纯虚函数的具体实现  
private:  
};

float getTotal(A \*s[],int n)//通过一个数组传递所有的图形对象  
//多态的好处出来了吧，不是多态，不能用基类A调用  
//参数类型怎么写，要是有100个不同的图形，怎么传递  
{  
float sum = 0;  
for(int i = 0;i < n; i++) sum = sum + s[i]->area();  
return sum;  
}  
int main(void)  
{  
float totalArea;  
A \*a[2];  
a[0] = new B(1,2); //一个长方形对象  
a[1] = new C(1,2);//一个三角形对象  
totalArea = getTotal(a , 2);//求出两个对象的面积和  
getchar();  
return 0;  
}  
——————————————————————————————–

8. 求下面函数的返回值（微软）  
int func(x)  
{  
int countx = 0;  
while(x)  
{  
countx ++;  
x = x&(x-1);  
}  
return countx;  
}

假定 x = 9999 。 答案： 8

思路：将 x 转化为 2 进制，看含有的 1 的个数。  
——————————————————————————————–  
9. 什么是 “ 引用 ” ？申明和使用 “ 引用 ” 要注意哪些问题？  
答：引用就是某个目标变量的 “ 别名 ”(alias) ，对应用的操作与对变量直接操作效果完全相同。申明一个引用的时候，切记要对其进行初始化。引用声明完毕后，相当于目标变量名有两个名称，即该目标原名称和引用名，不能再把该引用名作为其他变量名的别名。声明一个引用，不是新定义了一个变量，它只表示该引用名是目标变量名的一个别名，它本身不是一种数据类型，因此引用本身不占存储单元，系统也不给引用分配存储单元。不能建立数组的引用。  
——————————————————————————————–  
10. 将 “ 引用 ” 作为函数参数有哪些特点？  
（ 1 ）传递引用给函数与传递指针的效果是一样的。这时，被调函数的形参就成为原来主调函数中的实参变量或对象的一个别名来使用，所以在被调函数中对形参变量的操作就是对其相应的目标对象（在主调函数中）的操作。

（ 2 ）使用引用传递函数的参数，在内存中并没有产生实参的副本，它是直接对实参操作；而使用一般变量传递函数的参数，当发生函数调用时，需要给形参分配存储单元，形参变量是实参变量的副本；如果传递的是对象，还将调用拷贝构造函数。因此，当参数传递的数据较大时，用引用比用一般变量传递参数的效率和所占空间都好。

（ 3 ）使用指针作为函数的参数虽然也能达到与使用引用的效果，但是，在被调函数中同样要给形参分配存储单元，且需要重复使用 “\* 指针变量名 ” 的形式进行运算，这很容易产生错误且程序的阅读性较差；另一方面，在主调函数的调用点处，必须用变量的地址作为实参。而引用更容易使用，更清晰。  
——————————————————————————————–  
11. 在什么时候需要使用 “ 常引用 ” ？  
如果既要利用引用提高程序的效率，又要保护传递给函数的数据不在函数中被改变，就应使用常引用。常引用声明方式： const 类型标识符 & 引用名 = 目标变量名；

例 1

int a ;  
const int &ra=a;  
ra=1; // 错误  
a=1; // 正确

例 2

string foo( );  
void bar(string & s);

那么下面的表达式将是非法的：

bar(foo( ));  
bar(“hello world”);

原因在于 foo( ) 和 “hello world” 串都会产生一个临时对象，而在 C++ 中，这些临时对象都是 const 类型的。因此上面的表达式就是试图将一个 const 类型的对象转换为非 const 类型，这是非法的。引用型参数应该在能被定义为 const 的情况下，尽量定义为 const 。  
——————————————————————————————–  
12. 将 “ 引用 ” 作为函数返回值类型的格式、好处和需要遵守的规则 ?  
格式：类型标识符 & 函数名（形参列表及类型说明） { // 函数体 }

好处：在内存中不产生被返回值的副本；（注意：正是因为这点原因，所以返回一个局部变量的引用是不可取的。因为随着该局部变量生存期的结束，相应的引用也会失效，产生 runtime error! 注意事项：

（ 1 ）不能返回局部变量的引用。这条可以参照 Effective C++[1] 的 Item 31 。主要原因是局部变量会在函数返回后被销毁，因此被返回的引用就成为了 ” 无所指 ” 的引用，程序会进入未知状态。

（ 2 ）不能返回函数内部 new 分配的内存的引用。这条可以参照 Effective C++[1] 的 Item 31 。虽然不存在局部变量的被动销毁问题，可对于这种情况（返回函数内部 new 分配内存的引用），又面临其它尴尬局面。例如，被函数返回的引用只是作为一个临时变量出现，而没有被赋予一个实际的变量，那么这个引用所指向的空间（由 new 分配）就无法释放，造成 memory leak 。

（ 3 ）可以返回类成员的引用，但最好是 const 。这条原则可以参照 Effective C++[1] 的 Item 30 。主要原因是当对象的属性是与某种业务规则（ business rule ）相关联的时候，其赋值常常与某些其它属性或者对象的状态有关，因此有必要将赋值操作封装在一个业务规则当中。如果其它对象可以获得该属性的非常量引用（或指针），那么对该属性的单纯赋值就会破坏业务规则的完整性。

（ 4 ）流操作符重载返回值申明为 “ 引用 ” 的作用：

流操作符 << 和 >> ，这两个操作符常常希望被连续使用，例如： cout << “hello” << endl; 　因此这两个操作符的返回值应该是一个仍然支持这两个操作符的流引用。可选的其它方案包括：返回一个流对象和返回一个流对象指针。但是对于返回一个流对象，程序必须重新（拷贝）构造一个新的流对象，也就是说，连续的两个 << 操作符实际上是针对不同对象的！这无法让人接受。对于返回一个流指针则不能连续使用 << 操作符。因此，返回一个流对象引用是惟一选择。这个唯一选择很关键，它说明了引用的重要性以及无可替代性，也许这就是 C++ 语言中引入引用这个概念的原因吧。 赋值操作符 = 。这个操作符象流操作符一样，是可以连续使用的，例如： x = j = 10; 或者 (x=10)=100; 赋值操作符的返回值必须是一个左值，以便可以被继续赋值。因此引用成了这个操作符的惟一返回值选择。

例 3

＃ i nclude  
int &put(int n);  
int vals[10];  
int error=-1;  
void main()  
{  
put(0)=10; // 以 put(0) 函数值作为左值，等价于 vals[0]=10;  
put(9)=20; // 以 put(9) 函数值作为左值，等价于 vals[9]=20;  
cout< cout<=0 && n else { cout< 6″) : puts(” return 0;  
}  
——————————————————————————————–  
114. 交换两个数的宏定义

交换两个参数值的 宏定义 为： . #define SWAP (a,b) (a)=(a)+(b);(b)=(a)-(b);(a)=(a)-(b);  
——————————————————————————————–  
115.Itearator 各指针的区别

游标和指针

我说过游标是指针，但不仅仅是指针。游标和指针很像，功能很像指针，但是实际上，游标是通过重载一元的 ”\*” 和 ”->” 来从容器中间接地返回一个值。将这些值存储在容器中并不是一个好主意，因为每当一个新值添加到容器中或者有一个值从容器中删除，这些值就会失效。在某种程度上，游标可以看作是句柄（ handle ）。通常情况下游标（ iterator ）的类型可以有所变化，这样容器也会有几种不同方式的转变：

iterator—— 对于除了 vector 以外的其他任何容器，你可以通过这种游标在一次操作中在容器中朝向前的方向走一步。这意味着对于这种游标你只能使用 “++” 操作符。而不能使用 “–” 或 “+=” 操作符。而对于 vector 这一种容器，你可以使用 “+=” 、 “—” 、 “++” 、 “-=” 中的任何一种操作符和 “” 、 “>=” 、 “==” 、 “!=” 等比较运算符。  
——————————————————————————————–  
116. C++ 中的 class 和 struct 的区别  
从语法上，在 C++ 中（只讨论 C++ 中）。 class 和 struct 做类型定义时只有两点区别：  
（一）默认继承权限。如果不明确指定，来自 class 的继承按照 private 继承处理，来自 struct 的继承按照 public 继承处理；  
（二）成员的默认访问权限。 class 的成员默认是 private 权限， struct 默认是 public 权限。  
除了这两点， class 和 struct 基本就是一个东西。语法上没有任何其它区别。

不能因为学过 C 就总觉得连 C++ 中 struct 和 class 都区别很大，下面列举的说明可能比较无聊，因为 struct 和 class 本来就是基本一样的东西，无需多说。但这些说明可能有助于澄清一些常见的关于 struct 和 class 的错误认识：  
（ 1 ）都可以有成员函数；包括各类构造函数，析构函数，重载的运算符，友元类，友元结构，友元函数，虚函数，纯虚函数，静态函数；  
（ 2 ）都可以有一大堆 public/private/protected 修饰符在里边；  
（ 3 ）虽然这种风格不再被提倡，但语法上二者都可以使用大括号的方式初始化：

A a = {1, 2, 3}; 不管 A 是个 struct 还是个 class ，前提是这个类 / 结构足够简单，比如所有的成员都是 public 的，所有的成员都是简单类型，没有显式声明的构造函数。  
（ 4 ）都可以进行复杂的继承甚至多重继承，一个 struct 可以继承自一个 class ，反之亦可；一个 struct 可以同时继承 5 个 class 和 5 个 struct ，虽然这样做不太好。  
（ 5 ）如果说 class 的设计需要注意 OO 的原则和风格，那么没任何理由说设计 struct 就不需要注意。  
（ 6 ）再次说明，以上所有说法都是指在 C++ 语言中，至于在 C 里的情况， C 里是根本没有 “class” ，而 C 的 struct 从根本上也只是个包装数据的语法机制。  
—————————————————————

最后，作为语言的两个关键字，除去定义类型时有上述区别之外，另外还有一点点： “class” 这个关键字还用于定义模板参数，就像 “typename” 。但关键字 “struct” 不用于定义模板参数。

在模版中，类型参数前面可以使用class或typename，如果使用struct，则含义不同，struct后面跟的是“non-type template parameter”，而class或typename后面跟的是类型参数。  
template  
void f(X x)  
{  
}  
//出错信息：d:codecpptestcpptestcpptest.cpp(33) : error C2065: ‘X’ : undeclared identifier

关于使用大括号初始化

class 和 struct 如果定义了构造函数的话，都不能用大括号进行初始化

如果没有定义构造函数， struct 可以用大括号初始化。

如果没有定义构造函数，且所有成员变量全是 public 的话，可以用大括号初始化。

关于默认访问权限

class 中默认的成员访问权限是 private 的，而 struct 中则是 public 的。

关于继承方式

class 继承默认是 private 继承，而 struct 继承默认是 public 继承。

关于模版

在模版中，类型参数前面可以使用 class 或 typename ，如果使用 struct ，则含义不同， struct 后面跟的是 “non-type template parameter” ，而 class 或 typename 后面跟的是类型参数。

class 中有个默认的 this 指针， struct 没有  
不同点：构造函数，析构函数 this 指针  
——————————————————————————————–  
117. 有关重载函数

返回值类型不同构不成重载  
参数参数顺序不同能构成重载

c++ 函数同名不同返回值不算重载！函数重载是忽略返回值类型的。

成员函数被重载的特征有：  
1) 相同的范围（在同一个类中）；  
2) 函数名字相同；  
3) 参数不同；  
4) virtual 关键字可有可无。

5) 成员函数中 有无 const ( 函数后面 ) 也可判断是否重载  
——————————————————————————————–  
118. 数据库与 T-SQL 语言

关系数据库是表的集合，它是由一个或多个关系模式定义。 SQL 语言中的数据定义功能包括对数据库、基本表、视图、索引的定义。  
——————————————————————————————–  
119. 关系模型的基本概念

关系数据库以关系模型为基础，它有以下三部分组成：  
● 数据结构 —— 模型所操作的对象、类型的集合  
● 完整性规则 —— 保证数据有效、正确的约束条件  
● 数据操作 —— 对模型对象所允许执行的操作方式  
关系（ Relation ）是一个由行和列组成的二维表格，表中的每一行是一条记录（ Record ），每一列是记录的一个字段（ Field ）。表中的每一条记录必须是互斥的，字段的值必须具有原子性。

——————————————————————————————–  
120.SQL 语言概述

SQL （结构化查询语言）是关系数据库语言的一种国际标准，它是一种非过程化的语言。通过编写 SQL ，我们可以实现对关系数据库的全部操作。  
● 数据定义语言（ DDL ） —— 建立和管理数据库对象  
● 数据操纵语言（ DML ） —— 用来查询与更新数据  
● 数据控制语言（ DCL ） —— 控制数据的安全性

起来是一个很简单的问题，每一个使用过 RDBMS 的人都会有一个概念。

事务处理系统的典型特点是具备 ACID 特征。 ACID 指的是 Atomic （原子的）、 Consistent （一致的）、 Isolated （隔离的）以及 Durable （持续的），它们代表着事务处理应该具备的四个特征：

原子性：组成事务处理的语句形成了一个逻辑单元，不能只执行其中的一部分

一致性：在事务处理执行之前和之后，数据是一致的。

隔离性：一个事务处理对另一个事务处理没有影响。

持续性：当事务处理成功执行到结束的时候，其效果在数据库中被永久纪录下来。  
——————————————————————————————–  
121.C 语言中结构化程序设计的三种基本控制结构

顺序结构  
选择结构  
循环结构  
——————————————————————————————–  
122.CVS 是什么

cvs （ Concurrent Version System ） 是一个版本控制系统。使用它，可以记录下你的源文件的历史。

例如，修改软件时可能会不知不觉混进一些 bug ，而且可能过了很久你才会察觉到它们的存在。有了 cvs ，你可以很容易地恢复旧版本，并从中看出到底是哪个修改导致了这个 bug 。有时这是很有用的。

CVS 服务器端对每个文件维护着一个修订号 , 每次对文件的更新，都会使得文件的修订号加 1 。在客户端中也对每个文件维护着一个修订号 ,CVS 通过这两个修订号的关系，来进行 Update,Commit 和发现冲突等操作操作  
——————————————————————————————–  
123. 三种基本的数据模型

按照数据结构类型的不同，将数据模型划分为层次模型、网状模型和关系模型。