作者：Re\_born  
链接：<https://www.nowcoder.com/discuss/62368>  
来源：牛客网

**其实我面试的时候C++基本很少问及，不过还是要有扎实的基础。以下是自己的一份总结，比较适合自己，所以根据你们自己的需求拿走我的笔记，如有错误也请提出我好更改。回馈牛客网，谢谢。**

**1.深拷贝和浅拷贝**

深拷贝(Memberwise copy semantics)是指源对象与拷贝对象互相独立，其中任何一个对象的改动都不会对另外一个对象造成影响。

浅拷贝是指源对象与拷贝对象共用一份实体，仅仅是引用的变量不同（名称不同）。对其中任何一个对象的改动都会影响另外一个对象。

浅拷贝在类里面有指针成员的情况下只会复制指针的地址，会导致两个成员指针指向同一块内存，这样在要是分别delete释放时就会出现问题，因此需要用深拷贝。

2.**delete this**

在类的成员函数中能不能调用delete this？答案是肯定的，能调用，而且很多老一点的库都有这种代码。假设这个成员函数名字叫release，而delete this就在这个release方法中被调用，那么这个对象在调用release方法后，还能进行其他操作，如调用该对象的其他方法么？答案仍然是肯定 的，调用release之后还能调用其他的方法，但是有个前提：被调用的方法不涉及这个对象的数据成员和虚函数。说到这里，相信大家都能明白为什么会这样了。

根本原因在于delete操作符的功能和类对象的内存模型。当一个类对象声明时，系统会为其分配内存空间。在类对象的内存空间中，只有数据成员和虚函数表指针，并不包含代码内容，类的成员函数单独放在代码段中。在调用成员函数时，隐含传递一个this指针，让成员函数知道当前是哪个对象在调用它。当调用delete this时，类对象的内存空间被释放。在delete this之后进行的其他任何函数调用，只要不涉及到this指针的内容，都能够正常运行。一旦涉及到this指针，如操作数据成员，调用虚函数等，就会出现不可预期的问题。

为什么是不可预期的问题？delete this之后不是释放了类对象的内存空间了么，那么这段内存应该已经还给系统，不再属于这个进程。照这个逻辑来看，应该发生指针错误，无访问权限之类的令系统崩溃的问题才对啊？这个问题牵涉到操作系统的内存管理策略。delete this释放了类对象的内存空间，但是内存空间却并不是马上被回收到系统中，可能是缓冲或者其他什么原因，导致这段内存空间暂时并没有被系统收回。此时这段内存是可以访问的，你可以加上100，加上200，但是其中的值却是不确定的。当你获取数据成员，可能得到的是一串很长的未初始化的随机数；访问虚函数表，指针无效的可能性非常高，造成系统崩溃。

大致明白在成员函数中调用delete this会发生什么之后，再来看看另一个问题，如果在类的析构函数中调用delete this，会发生什么？实验告诉我们，会导致堆栈溢出。原因很简单，delete的本质是“为将被释放的内存调用一个或多个析构函数，然后，释放内存”(来自effective c++)。显然，delete this会去调用本对象的析构函数，而析构函数中又调用delete this，形成无限递归，造成堆栈溢出，系统崩溃。

3.**构造函数初始化时必须采用初始化列表一共有三种情况**

1.需要初始化的数据成员是对象(继承时调用基类构造函数)  
2.需要初始化const修饰的类成员  
3.需要初始化引用成员数据

**C/C++ 内存管理**

<http://blog.csdn.net/bizhu12/article/details/6668834>

<http://blog.csdn.net/wdzxl198/article/details/9050587>

4.**C/C++ 分配内存**

1) malloc函数：void \*malloc(unsigned int size)

     在内存的动态分配区域中分配一个长度为size的连续空间，如果分配成功，则返回所分配内存空间的首地址，否则返回NULL，申请的内存不会进行初始化。

2）calloc函数：void \*calloc(unsigned int num, unsigned int size)

     按照所给的数据个数和数据类型所占字节数，分配一个num \* size连续的空间。

    calloc申请内存空间后，会自动初始化内存空间为0，但是malloc不会进行初始化，其内存空间存储的是一些随机数据。   
3）realloc函数：void \*realloc(void \*ptr, unsigned int size)

    动态分配一个长度为size的内存空间，并把内存空间的首地址赋值给ptr，把ptr内存空间调整为size。

申请的内存空间不会进行初始化。  
4）new是动态分配内存的运算符，自动计算需要分配的空间，在分配类类型的内存空间时，同时调用类的构造函数，对内存空间进行初始化，即完成类的初始化工作。动态分配内置类型是否自动初始化取决于变量定义的位置，在函数体外定义的变量都初始化为0，在函数体内定义的内置类型变量都不进行初始化。

<http://blog.csdn.net/hackbuteer1/article/details/6789164>

**处理New分配内存失败情况？**

我们经常会使用new给一个对象分配内存空间，而当内存不够会出现内存不足的情况。C++提供了两种报告方式：

1、抛出bad\_alloc异常来报告分配失败；

2、返回空指针，而不会抛出异常

当operator new 无法满足内存需求时，它会不只一次地调用new\_handler函数（如果new\_handler没有退出程序的话）；它会不断地调用，直到找到足够的内存为止。可以用set\_new\_hander 函数为 new 设置用户自己定义的异常处理函数，也可以让 malloc 享用与 new 相同的异常处理函数。

**当一个类A中没有声命任何成员变量与成员函数这时sizeof(A)的值是多少，如果不是零，请解释一下编译器为什么没有让它为零**

通常是1，用作占位的。**为了确保不同对象有不同的地址**

**new和malloc的10点区别**

|  |  |  |
| --- | --- | --- |
| **特征** | **new/delete** | **malloc/free** |
| 分配内存的位置 | 自由存储区  自由存储区不仅可以是堆，还可以是静态存储区 | 堆 |
| 内存分配成功的返回值 | 完整类型指针  类型严格与对象匹配，无须进行类型转换，故new是符合**类型安全**性的操作符 | void\*  需要通过强制类型转换将void\*指针转换成我们需要的类型 |
| 内存分配失败的返回值 | 默认抛出异常 | 返回NULL |
| 分配内存的大小 | 由编译器根据类型计算得出 | 必须显式指定字节数 |
| 处理数组 | 有处理数组的new版本new[] | 需要用户计算数组的大小后进行内存分配 |
| 已分配内存的扩充 | 无法直观地处理 | 使用realloc简单完成 |
| 是否相互调用 | 可以，看具体的operator new/delete实现 | 不可调用new |
| 分配内存时内存不足 | 客户能够指定处理函数或重新制定分配器 | 无法通过用户代码进行处理 |
| 函数重载 | 允许 | 不允许 |
| 构造函数与析构函数 | 调用 | 不调用 |

<http://www.cnblogs.com/QG-whz/p/5140930.html>

5.**c++如何避免内存泄漏**

a、使用RAII(ResourceAcquisition Is Initialization,资源获取即初始化)技法，以构造函数获取资源(内存),析构函数释放。

b、相比于使用原生指针，更建议使用[**智能**](http://lib.csdn.net/base/aiplanning)指针，尤其是[**C++**](http://lib.csdn.net/base/cplusplus)11标准化后的智能指针。

c、注意delete和delete[]的使用方法。

d、这是很复杂的一种情况，是关于类的copy constructor的。首先先介绍一些概念。

同defaultconstructor一样，标准保证，如果类作者没有为class声明一个copy constructor，那么编译器会在需要的时候产生出来(这也是一个常考点：问道"如果类作者未定义出default/copy constructor，编译器会自动产生一个吗？"答案是否定的)

不过请注意！！这里编译器即使产生出来，也是为满足它的需求，而非类作者的需求！！

而什么时候是编译器"需要"的时候呢？是在当这个class【不表现出】bitwise copy semantics(位逐次拷贝，即浅拷贝)的时候。

在4中情况下class【不表现出】bitwisecopy semantics

(1)、当class内含一个memberobject且该member object声明了一个copy     constructor(无论该copy ctor是类作者自己生明的还是编译器合成的)；

(2)、当class继承自一个baseclass且该base class有一个copy constructor(无论    该copy ctor是类作者自己生明的还是编译器合成的)；

(3)、当class声明了virtual function；

(4)、当class派生自一个继承链，且该链中存在virtual base class时。

言归正传，如果class中仅仅是一些普通资源，那么bitwisecopy semantics是完全够用的；然而，挡在该class中存在了一块动态分配的内存，并且在之后执行了bitwise copy semantics后，将会有一个按位拷贝的对象和原来class中的某个成员指向同一块heap空间，当执行它们的析构函数后，该内存将被释放两次，这是未定义的行为。因此，在必要的时候需要使用Memberwise copy semantics(即深拷贝)，来避免内存泄露。**位拷贝拷贝的是地址，而值拷贝则拷贝的是内容。**

6.**四种情况下编译器会生成默认构造函数**

其实默认构造函数也是分为两类的:有用的、无用的。

所谓**有用的标准也是就默认构造函数会为我们的类做一些初始化操作**。那么无用的就不会做任何工作,从而对我们的类也就没有任何意义。所以,我们通常所说的默认构造函数是指**有用的默认构造函数,其英文名字叫nontrivial default constructor**。

那么到底什么时候编译器会为我们产生nontrivial default constructor呢?有下面四中情况:

①如果一个类里面某个成员对象有nontrivial default constructor,编译器就会为我们的类产生nontrivial default constructor。

那么编译器这样做的理由是什么?

答案是因为类成员对象有nontrivial default constructor,那么编译器就需要**显式的来调用这个类成员对象的**nontrivial default constructor。而编译器想显式的调用类成员对象的nontrivial default constructor,就需要自己来合成一些代码来调用。但是记住,编译器合成的nontrivial default constructor**仅仅调用类成员对象的默认构造函数,而不对我们类里面的其它变量做任何初始化操作。**

也就是说,如果你想初始化类成员变量以外的变量例如一个int、一个String,那么必须**自己定义**默认构造函数来完成这些变量的初始化。而编译器会对你定义的默认构造函数做相应的扩展,从而调用类成员对象的nontrivial default constructor。

②如果一个派生类的**基类有**nontrivial default constructor,那么编译器会为派生类合成一个nontrivial default constructor。

编译器这样的理由是:因为派生类被合成时需要显式调用基类的默认构造函数。

③如何一个类里面隐式的含有**任何virtual function table(或vtbl)、pointer member(或vptr)。**

编译器这样做的理由很简单:因为这些vtbl或vptr需要编译器隐式(implicit)的合成出来,那么编译器就把合成动作放到了默认构造函数里面。所以编译器必须自己产生一个默认构造函数来完成这些操作。

所以如果你的类里带有任何**virtual function,那么编译器会合成一个默认构造函数。**

④如果一个**类虚继承于其它类。**

编译器这样做的理由和③类似:因为虚继承需要维护一个类似指针一样,可以动态的决定内存地址的东西(不同编译器对虚继承的实现不仅相同)。

那么**除了以上四种情况,编译器并不会为我们的类产生默认构造函数。**

7.**c++的const函数特点**

1.不能在const函数中修改所在类的对象的数据，因为const函数中的\*this是常量，同样只能访问const函数；

2.const函数中只能调用其他的const函数，不能调用非const函数，因为对象调用函数是需要传递对象自己，const函数中的\*this是常量，非const函数中的\*this是变量，因此不可以调用(除非去除\*this的const属性)；

Note:使用const\_cast后，可以在const函数中调用非const函数的

3.const对象只能调用const函数 ，但是非const对象可以调用const函数；

4.const函数与同名的非const函数是重载函数。

8.**面向对象的三个基本特征**

[**http://blog.csdn.net/chenyi8888/article/details/5336912**](http://blog.csdn.net/chenyi8888/article/details/5336912)

C++继承体系中，初始化时构造函数的调用顺序如下

(1)任何虚拟基类的构造函数按照他们被继承的顺序构造

(2)任何非虚拟基类的构造函数按照他们被继承的顺序构造

(3)任何成员对象的函数按照他们声明的顺序构造

(4)类自己的构造函数

9.**指针和引用**

1、本质：指针是一个变量，存储内容是一个地址，指向内存的一个存储单元。而引用是原变量的一个别名，实质上和原变量是一个东西，是某块内存的别名。

2、指针的值可以为空，且非const指针可以被重新赋值以指向另一个不同的对象。而引用的值不能为空，并且引用在定义的时候必须初始化，一旦初始化，就和原变量“绑定”，不能更改这个绑定关系。(没有NULL的引用可能会比指针效率更高，因为不用测试有效性)

3、对指针执行sizeof()操作得到的是指针本身的大小（32位系统为4,64位系统为8）。而对引用执行sizeof()操作得到的是所绑定的对象的所占内存大小。

4、指针的自增(++)运算表示对地址的自增，自增大小要看所指向单元的类型。而引用的自增(++)运算表示对值的自增。

5、在作为函数参数进行传递时的区别：指针所以函数传输作为传递时，函数内部的指针形参是指针实参的一个副本，改变指针形参并不能改变指针实参的值，通过解引用\*运算符来更改指针所指向的内存单元里的数据。而引用在作为函数参数进行传递时，实质上传递的是实参本身，即传递进来的不是实参的一个拷贝，因此对形参的修改其实是对实参的修改，所以在用引用进行参数传递时，不仅节约时间，而且可以节约空间。

10.**构造函数和析构函数能否重载**

函数重载就是同一函数名的不同实现，并且能在编译时能与一具体形式匹配，这样参数列表必须不一样。由于重载函数与普通函数的差别是没有返回值，而返回值不能确定函数重载，所以构造函数可以重载；析构函数的特点是参数列表为空，并且无返回值，从而不能重载。

11.**析构函数什么情况下定义为虚函数**

一般情况下,这样的删除只能够删除基类对象,而不能删除子类对象,形成了删除一半形象,造成内存泄漏。

如果不需要基类对派生类及对象进行操作,则不能定义虚函数,因为这样会增加内存开销。当类里面有定义虚函数的时候,编译器会给类添加一个虚函数表,里面来存放虚函数指针,这样会增加类的存储空间。所以只有当一个类被用来作为基类的时候,才把析构函数写成虚函数。

如果基类析构函数不为虚的话，在释放派生类对象的时候就不会调用派生类的析构函数，有可能造成内存泄露。

12.**拷贝构造函数的参数类型必须是引用**

如果拷贝构造函数中的参数不是一个引用，即形如CClass(const CClass c\_class)，那么就相当于采用了传值的方式(pass-by-value)，而传值的方式会调用该类的拷贝构造函数，从而造成无穷递归地调用拷贝构造函数。因此拷贝构造函数的参数必须是一个引用。  
需要澄清的是，传指针其实也是传值，如果上面的拷贝构造函数写成CClass(const CClass\* c\_class)，也是不行的。事实上，只有传引用不是传值外，其他所有的传递方式都是传值。

**为什么内联函数，构造函数，静态成员函数不能为virtual函数？**

1> 内联函数

内联函数是在编译时期展开,而虚函数的特性是运行时才动态联编,所以两者矛盾,不能定义内联函数为虚函数。

2> 构造函数

构造函数用来创建一个新的对象,而虚函数的运行是建立在对象的基础上,在构造函数执行时,对象尚未形成,所以不能将构造函数定义为虚函数

3> 静态成员函数

静态成员函数属于一个类而非某一对象,没有this指针,它无法进行对象的判别。

4>  友元函数

C++不支持友元函数的继承，对于没有继承性的函数没有虚函数

这个可以从两个角度去理解：

1。virtual意味着在执行时期进行绑定，所以在编译时刻需确定信息的不能为virtual

构造函数需在编译时刻，因为需构造出个对象，才能执行动作。  
静态成员函数不属于任何一个对象，编译时刻确定不存在执行的时候选择执行哪个的情形。  
内联函数，由于属于编译器的建议机制，所以其实可以virtual。

2。virtual意味着派生类可以改写其动作  
派生类的构造函数会先执行基类的构造函数而不是取代基类构造函数，也就是说基类的构造函数可以看作派生类构造函数的组成，所以并不能改写这个函数。  
静态成员函数不属于任何一个对象，所以更不能改写其动作了。

inline和virtual不会同时起作用。带virtual的函数在不需要动态绑定调用的时候，就可以inline。

**构造函数和析构函数为什么没有返回值？**

构造函数和析构函数是两个非常特殊的函数：它们没有返回值。这与返回值为void的函数显然不同，后者虽然也不返回任何值，但还可以让它做点别的事情，而构造函数和析构函数则不允许。在程序中创建和消除一个对象的行为非常特殊，就像出生和死亡，而且总是由编译器来调用这些函数以确保它们被执行。如果它们有返回值，要么编译器必须知道如何处理返回值，要么就只能由客户程序员自己来显式的调用构造函数与析构函数，这样一来，安全性就被人破坏了。另外，析构函数不带任何参数，因为析构不需任何选项。

**C++异常机制**

异常事件在C++中表示为**异常对象**。异常事件发生时，程序使用throw关键字抛出异常表达式，抛出点称为异常出现点，由操作系统为程序设置当前异常对象，然后执行程序的当前异常处理代码块，在包含了异常出现点的最内层的**try**块，依次匹配catch语句中的异常对象（只进行类型匹配，catch参数有时在catch语句中并不会使用到）。若匹配成功，则执行catch块内的异常处理语句，然后接着执行**try...catch...**块之后的代码。如果在当前的try...catch...块内找不到**匹配**该异常对象的catch语句,则由更外层的try...catch...块来处理该异常；如果当前函数内所有的try...catch...块都不能匹配该异常，则递归回退到调用栈的上一层去处理该异常。如果一直退到主函数main()都不能处理该异常，则调用系统函数terminate()终止程序。

<http://www.cnblogs.com/QG-whz/p/5136883.html>

**析构函数可以抛出异常吗？为什么不能抛出异常？除了资源泄露，还有其他需考虑的因素吗？**

（1）C++中析构函数的执行不应该抛出异常；

（2）如果析构函数抛出异常，则**异常点之后的程序不会执行**，如果析构函数在异常点之后执行了某些必要的动作比如释放某些资源，则这些动作不会执行，会造成诸如资源泄漏的问题。

（3）通常异常发生时，c++的机制会调用已经构造对象的析构函数来释放资源，此时若析构函数本身也抛出异常，则前一个异常尚未处理，又有新的异常，会造成程序崩溃的问题。析构函数中抛出异常导致程序不明原因的崩溃是许多系统的致命内伤

**解决办法：**那就是**把异常完全封装在析构函数内部，决不让异常抛出函数之外。如使用**

**Try**

**{  }**

**Catch**

**{**/这里可以什么都不做，只是**保证catch块的程序抛出的异常不会被扔出析构函数之外**。**}**

[**http://www.cnblogs.com/fly1988happy/archive/2012/04/11/2442765.html**](http://www.cnblogs.com/fly1988happy/archive/2012/04/11/2442765.html)

**在构造函数和析构函数中抛出异常会发生什么？什么是栈展开？**

构造函数中可以抛出异常，**构造抛异常之前必须把已经申请的资源释放掉**。**这样，就算你的对象是new出来的，也不会造成内存泄漏。**  
因为析构函数不会被调用，所以抛出异常后，你没机会释放资源。

构造函数中抛出异常时概括性总结  
（1） C++中通知对象构造失败的唯一方法那就是在构造函数中抛出异常；

（2）  构造函数中抛出异常将导致对象的析构函数不被执行；

（3） 当对象发生部分构造时，已经构造完毕的子对象将会逆序地被析构；

栈展开：抛出异常时，将暂停当前函数的执行，开始查找匹配的catch子句。首先检查throw本身是否在try块内部，如果是，检查与该try相关的catch子句，看是否可以处理该异常。如果不能处理，就退出当前函数，并且释放当前函数的内存并销毁局部对象，继续到上层的调用函数中查找，直到找到一个可以处理该异常的catch。这个过程称为栈展开（stack unwinding）

**C++保护和私有构造函数与析构函数**

**如何定义一个只能在堆上（栈上）生成对象的类?**

构造函数定义为protected后，就意味着你不能在类的外部构造对象了，而只能在外部构造该类的子类的对象

构造函数定义为private后，意味着不仅仅不能在类的外部构造对象了，而且也不能在外部构造该类的子类的对象了，只能通过类的static静态函数来访问类的内部定义的对象，单件singleton模式就是私有构造函数的典型实例

对于堆中的对象，通常都是用new/delete来创建/销毁，当调用new时，它会自动调用相应类的构造函数，当调用delete时，它会自动调用相应类的析构函数。而在栈中产生对象时，对象的创建/销毁是自动完成的，也就是在创建时自动调用构造函数，在销毁时自动调用析构函数，即不需要显示调用new/delete，但有个前提是类的构造/析构函数都必须是public的。  
    析构函数无论是protected还是priavte，其共同作用都是禁止在栈中产生对象，因为无法自动完成析构函数的调用，自然就不能在栈中创建对象了；当然如果在堆上创建对象时，也不能直接delete对象了，因为这样也会在外部析构该对象，但是可以间接完成堆对象的析构

私有和保护析构函数区别在于私有的析构函数不仅禁止了栈中产生对象，而且同时也禁止了继承

来源：牛客网

在C++中，类的对象建立分为两种，一种是静态建立，如A a；另一种是动态建立，如A\* ptr=new A；这两种方式是有区别的。

静态建立一个类对象，是由编译器为对象在栈空间中分配内存，是通过直接移动栈顶指针，挪出适当的空间，然后在这片内存空间上调用构造函数形成一个栈对象。使用这种方法，直接调用类的构造函数。

动态建立类对象，是使用new运算符将对象建立在堆空间中。这个过程分为两步，第一步是执行operator new()函数，在堆空间中搜索合适的内存并进行分配；第二步是调用构造函数构造对象，初始化这片内存空间。这种方法，间接调用类的构造函数。

那么如何限制类对象只能在堆或者栈上建立呢？下面分别进行讨论。

1、只能建立在堆上

类对象只能建立在堆上，就是不能静态建立类对象，即不能直接调用类的构造函数。

容易想到将构造函数设为私有。在构造函数私有之后，无法在类外部调用构造函数来构造类对象，只能使用new运算符来建立对象。然而，前面已经说过，new运算符的执行过程分为两步，C++提供new运算符的重载，其实是只允许重载operator new()函数，而operator()函数用于分配内存，无法提供构造功能。因此，这种方法不可以。

当对象建立在栈上面时，是由编译器分配内存空间的，调用构造函数来构造栈对象。当对象使用完后，编译器会调用析构函数来释放栈对象所占的空间。编译器管理了对象的整个生命周期。如果编译器无法调用类的析构函数，情况会是怎样的呢？比如，类的析构函数是私有的，编译器无法调用析构函数来释放内存。所以，编译器在为类对象分配栈空间时，会先检查类的析构函数的访问性，其实不光是析构函数，只要是非静态的函数，编译器都会进行检查。如果类的析构函数是私有的，则编译器不会在栈空间上为类对象分配内存。

因此，将析构函数设为私有，类对象就无法建立在栈上了。代码如下：

1. **class**  A

2. {

3. **public** :

4.     A(){}

5.     **void**  destory(){ **delete**   **this** ;}

6. **private** :

7.     ~A(){}

8. };

试着使用A a;来建立对象，编译报错，提示析构函数无法访问。这样就只能使用new操作符来建立对象，构造函数是公有的，可以直接调用。类中必须提供一个destory函数，来进行内存空间的释放。类对象使用完成后，必须调用destory函数。

上述方法的一个缺点就是，无法解决继承问题。如果A作为其它类的基类，则析构函数通常要设为virtual，然后在子类重写，以实现多态。因此析构函数不能设为private。还好C++提供了第三种访问控制，protected。将析构函数设为protected可以有效解决这个问题，类外无法访问protected成员，子类则可以访问。

另一个问题是，类的使用很不方便，使用new建立对象，却使用destory函数释放对象，而不是使用delete。（使用delete会报错，因为delete对象的指针，会调用对象的析构函数，而析构函数类外不可访问）这种使用方式比较怪异。为了统一，可以将构造函数设为protected，然后提供一个public的static函数来完成构造，这样不使用new，而是使用一个函数来构造，使用一个函数来析构。代码如下，类似于单例模式：

1. **class**  A

2. {

3. **protected** :

4.     A(){}

5.     ~A(){}

6. **public** :

7.     **static**  A\* create()

8.     {

9.         **return**   **new**  A();

10.     }

11.     **void**  destory()

12.     {

13.         **delete**   **this** ;

14.     }

15. };

这样，调用create()函数在堆上创建类A对象，调用destory()函数释放内存。

2、只能建立在栈上

只有使用new运算符，对象才会建立在堆上，因此，只要禁用new运算符就可以实现类对象只能建立在栈上。将operator new()设为私有即可。代码如下：

1. **class**  A

2. {

3. **private** :

4.     **void** \* operator  **new** ( **size\_t**  t){}      // 注意函数的第一个参数和返回值都是固定的

5.     **void**  operator  **delete** ( **void** \* ptr){}  // 重载了new就需要重载delete

6. **public** :

7.     A(){}

8.     ~A(){}

9. };

[**内联函数和宏定义的区别**](http://blog.csdn.net/gao675597253/article/details/7397373)

内联函数用来降低程序的运行时间。当内联函数收到编译器的指示时，即可发生内联：编译器将使用函数的定义体来替代函数调用语句，这种替代行为发生在编译阶段而非程序运行阶段。内联函数仅仅是对编译器的内联建议，编译器是否觉得采取你的建议取决于函数是否符合内联的有利条件。如何函数体非常大，那么编译器将忽略函数的内联声明，而将内联函数作为普通函数处理。

优点：

1.它通过避免函数调用所带来的开销来提高你程序的运行速度。

2.当函数调用发生时，它节省了变量弹栈、压栈的开销。

3.它避免了一个函数执行完返回原现场的开销。

4.通过将函数声明为内联，你可以把函数定义放在头文件内。

缺点：

1.内联是以代码膨胀（复制）为代价，内联函数增大了可执行程序的体积，导致内存消耗代价较高。

2.C++内联函数的展开是中编译阶段，这就意味着如果你的内联函数发生了改动，那么就需要重新编译代码。

1.内联函数在运行时可调试，而宏定义不可以  
2.编译器会对内联函数的参数类型做安全检查或自动类型转换（同普通函数），而宏定义则不会   
3.内联函数可以访问类的成员变量，宏定义则不能   
4.在类中声明同时定义的成员函数，自动转化为内联函数。

[**http://blog.csdn.net/gao675597253/article/details/7397373**](http://blog.csdn.net/gao675597253/article/details/7397373)

虚函数

**C++多态性 !!!!!**

[**http://blog.csdn.net/hackbuteer1/article/details/7475622**](http://blog.csdn.net/hackbuteer1/article/details/7475622)

**C++对象模型 !!!!!**

[**http://www.cnblogs.com/QG-whz/p/4909359.html#\_label3**](#_label3)

[**http://blog.csdn.net/ljianhui/article/details/46408645**](http://blog.csdn.net/ljianhui/article/details/46408645)

[**http://blog.csdn.net/haoel/article/details/1948051**](http://blog.csdn.net/haoel/article/details/1948051)

[**http://www.cnblogs.com/taoxu0903/archive/2008/02/04/1064234.html**](http://www.cnblogs.com/taoxu0903/archive/2008/02/04/1064234.html)

[**http://www.cnblogs.com/skynet/p/3343726.html**](http://www.cnblogs.com/skynet/p/3343726.html)

**当把一个派生类对象指针赋值给其基类指针时会发生什么样的行为**

当使用基类的指针指向一个派生类的对象时，编译器会安插相应的代码，调整指针的指向，使基类的指针指向派生类对象中其对应的基类子对象的起始处。

这些指针都指向了对应的类型的子对象，且其都包括一个vptr，所以就可以通过虚函数表中的第-1项的type\_info对象的地址来获取type\_info对象，从而获得类型信息。而这些地址值都是相同的，即指向同一个type\_info对象，且该type\_info对象显示该对象的类型为Derived，也就能正确地输出其类型信息。

**1.类是怎么通过虚函数实现多态的？**

多态性是“一个接口，多种方法”，多态性分为两类: 静态多态性和动态多态性。以前学过的函数重载和运算符重载实现的多态性属于静态多态性，动态多态性是通过虚函数(virtual function)实现的。静态多态性是指：在程序编译时系统就能决定调用的是哪个函数，因此静态多态性又称编译时的多态性。动态多态性是在程序运行过程中才动态地确定操作所针对的对象。它又称运行时的多态性。类中有虚函数存在，所以编译器就会为他做添加一个vptr指针，并为他们分别创建一个表vtbl,vptr指向那个表，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。，只要vptr不同，指向的vtbl就不同，而不同的vtbl里装着对应类的虚函数地址，所以这样虚函数就可以完成它的任务。子类重写的虚函数的地址直接替换了父类虚函数在虚表中的位置，因此当访问虚函数时，该虚表中的函数是谁的就访问谁。

**注意：存在虚函数的类都有一个一维的虚函数表叫做虚表。类的对象有一个指向虚表开始的虚指针。虚表是和类对应的，虚表指针是和对象对应的。**

**对于虚函数调用来说，每一个对象内部都有一个虚表指针，该虚表指针被初始化为本类的虚表。所以在程序中，不管你的对象类型如何转换，但该对象内部的虚表指针是固定的，所以呢，才能实现动态的对象函数调用，这就是C++多态性实现的原理**

**单继承与多继承：单继承所有的虚函数都包含在虚函数表中，多重继承有多个虚函数表，当子类对父类的虚函数有重写时，子类的函数覆盖父类的函数在对应的虚函数位置，当子类有新的虚函数时，这些虚函数被加在第一个虚函数表的后面**

**虚继承：使公共的基类在子类中只有一份，我们看到虚继承在多重继承的基础上多了vbtable来存储到公共基类的偏移**

**虚函数工作原理与内存占用大小**

[**http://blog.csdn.net/hackbuteer1/article/details/7883531**](http://blog.csdn.net/hackbuteer1/article/details/7883531)

**虚继承**

[**http://blog.csdn.net/crystal\_avast/article/details/7678704**](http://blog.csdn.net/crystal_avast/article/details/7678704)

**2.override重写和overload重载的区别**

成员函数被**重载**的特征：（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。

函数重载不能靠返回值来进行区分

**重写**是指派生类函数重写基类函数，是C++的多态的表现，特征是：  
（1）不同的范围（分别位于派生类与基类）；  
（2）函数名字相同；  
（3）参数相同；

（4）返回值（即函数原型）都要与基类的函数相同  
（5）基类函数必须有virtual关键字。

重写函数的访问修饰符可以不同，尽管virtual函数是private的，在派生类中重写的函数可以是public或protect的

**“隐藏**”是指派生类的函数屏蔽了与其同名的基类函数，调用的函数取决于指向它的指针所声明的类型，规则如下：

（1）如果派生类的函数与基类的函数同名，但是参数不同。此时，不论有无virtual关键字，基类的函数将被隐藏。  
（2）如果派生类的函数与基类的函数同名，并且参数也相同，但是基类函数没有virtual关键字。此时，基类的函数被隐藏。

另一个关于虚函数很微妙的错误情况：参数相同，但是基类的函数是const的，派生类的函数却不是。

（1）一个函数在基类申明一个virtual，那么在所有的派生类都是是virtual的。  
（2）一个函数在基类为普通函数，在派生类定义为virtual的函数称为越位，函数行为依赖于指针/引用的类型，而不是实际对象的类型。

可以提及隐藏的问题

[**http://www.cnblogs.com/luxiaoxun/archive/2012/08/09/2630751.html**](http://www.cnblogs.com/luxiaoxun/archive/2012/08/09/2630751.html)

**虚函数与纯虚函数区别？**

1、为了方便使用多态特性，我们常常需要在基类中定义虚拟函数。  
2、在很多情况下，基类本身生成对象是不合情理的。例如，动物作为一个基类可以派生出老虎、孔雀等子类，但动物本身生成对象明显不合常理。  
为了解决上述问题，引入了纯虚函数的概念，将函数定义为纯虚函数（方法：virtual ReturnType Function()= 0;），则编译器要求在派生类中必须予以重写以实现多态性。同时含有纯虚拟函数的类称为抽象类，它不能生成对象。

纯虚函数最显著的特征是：它们必须在继承类中重新声明函数（不要后面的＝0，否则该派生类也不能实例化），而且它们在抽象类中往往没有定义。  
定义纯虚函数的目的在于，使派生类仅仅只是继承函数的接口。

定义一个函数为虚函数，不代表函数为不被实现的函数。  
定义他为虚函数是为了允许用基类的指针来调用子类的这个函数。  
定义一个函数为纯虚函数，才代表函数没有被实现。  
定义纯虚函数是为了实现一个接口，起到一个规范的作用，规范继承这个类的程序员必须实现这个函数。纯虚函数最显著的特征是：它们必须在继承类中重新声明函数（不要后面的＝0，否则该派生类也不能实例化），而且它们在抽象类中往往没有定义。

1.  虚函数和纯虚函数可以定义在同一个类(class)中，含有纯虚函数的类被称为抽象类(abstract class)，而只含有虚函数的类(class)不能被称为抽象类(abstract class)。

2.  虚函数可以被直接使用，也可以被子类(sub class)重载以后以多态的形式调用，而纯虚函数必须在子类(sub class)中实现该函数才可以使用，因为纯虚函数在基类(base class)只有声明而没有定义。

3.  虚函数和纯虚函数都可以在子类(sub class)中被重载，以多态的形式被调用

[**http://blog.csdn.net/hackbuteer1/article/details/7558868**](http://blog.csdn.net/hackbuteer1/article/details/7558868)

[**http://www.cnblogs.com/bluestorm/archive/2012/08/29/2662350.html**](http://www.cnblogs.com/bluestorm/archive/2012/08/29/2662350.html)

类型

**旧式C转型方式问题**

1.允许任何类型转为任何其他类型。C++每次转型能够更加精确指明意图 如const\_cast

2.难以辨别。旧式转型（标识符）组成     C++很多地方都是小括号+对象

**四种类型转换(cast)的关键字详解及代码**

1.**static\_cast**

最常用的类型转换符，在正常状况下的类型转换，如把int转换为float，如：int i；float f； f=（float）i；或者f=static\_cast<float>(i)

2.**const\_cast**

用于取出const属性，把const类型的指针变为非const类型的指针，如：const int \*fun(int x,int y){}　　int \*ptr=const\_cast<int \*>(fun(2.3))

3.**dynamic\_cast**

（1）其他三种都是编译时完成的，dynamic\_cast是运行时处理的，运行时要进行类型检查。  
（2）不能用于内置的基本数据类型的强制转换。  
（3）dynamic\_cast转换如果成功的话返回的是指向类的指针或引用，指针转换失败的话则会返回NULL，引用转换失败抛出异常。  
（4）使用dynamic\_cast进行转换的，基类中一定要有虚函数，否则编译不通过。

该操作符用于运行时检查该转换是否类型安全，但只在多态类型时合法，即该类至少具有一个虚函数。dynamic\_cast与static\_cast具有相同的基本语法，dynamic\_cast主要用于类层次间的安全的上行转换和下行转换或跨系转型。在类层次间进行上行转换时，dynamic\_cast和static\_cast的效果是一样的；在进行下行转换时，dynamic\_cast具有类型检查的功能，比static\_cast更安全。

4.**reinterpret\_cast**

这个操作符与编译平台息息相关，不具备移植性

常用于转换函数指针类型。假设有一个数组存储是函数指针，有特定类型。把其他类型的函数指针放入这个数组。  避免使用

interpret是解释的意思，reinterpret即为重新解释，此标识符的意思即为数据的二进制形式重新解释，但是不改变其值。如：int i; char \*ptr="hello freind!"; i=reinterpret\_cast<int>(ptr);这个转换方式很少使用。

[**http://www.cnblogs.com/BeyondAnyTime/archive/2012/08/23/2652696.html**](http://www.cnblogs.com/BeyondAnyTime/archive/2012/08/23/2652696.html)

[**http://www.cnblogs.com/carsonzhu/p/5251012.html**](http://www.cnblogs.com/carsonzhu/p/5251012.html)

[**c++ RTTI（运行时类型识别）**](http://www.cnblogs.com/zhuyf87/archive/2013/03/15/2960899.html)

[**http://www.cnblogs.com/zhuyf87/archive/2013/03/15/2960899.html**](http://www.cnblogs.com/zhuyf87/archive/2013/03/15/2960899.html)

[**http://www.cnblogs.com/nliao/archive/2012/12/10/2811401.html**](http://www.cnblogs.com/nliao/archive/2012/12/10/2811401.html)

[**http://blog.csdn.net/ljianhui/article/details/46487951**](http://blog.csdn.net/ljianhui/article/details/46487951)

**当把一个派生类对象指针赋值给其基类指针时会发生什么样的行为**

当使用基类的指针指向一个派生类的对象时，编译器会安插相应的代码，调整指针的指向，使基类的指针指向派生类对象中其对应的基类子对象的起始处。

这些指针都指向了对应的类型的子对象，且其都包括一个vptr，所以就可以通过虚函数表中的第-1项的type\_info对象的地址来获取type\_info对象，从而获得类型信息。而这些地址值都是相同的，即指向同一个type\_info对象，且该type\_info对象显示该对象的类型为Derived，也就能正确地输出其类型信息。

**操作符Operator**

[**c++隐**](http://www.cnblogs.com/zhuyf87/archive/2013/03/15/2960899.html)**式类型转换**

**1.可用单个形参来调用的构造函数定义了从形参类型到该类类型的一个隐式转换。**

**可通过explicit声明来抑制这种转换，explicit关键字只能用于类内部的构造函数声明上**

2.**隐式类型转换操作符  关键词operator之后加一个类型名称**

**operator double();  //将Rational转为double**

**最好不要提供任何类型转换函数，根本问题在于未预期的情况下，此类函数可能被调用，而其结果可能不正确、不直观的程序行为。很难调试。解决方法是提供一个对等的函数取代类型转换操作符。**

[**http://www.cnblogs.com/QG-whz/p/4472566.html**](http://www.cnblogs.com/QG-whz/p/4472566.html)

[**区**](http://www.cnblogs.com/zhuyf87/archive/2013/03/15/2960899.html)**别 前置/后置（形式）  ++/-- 操作符**

区别前置后置，在后置式有一个int自变量作为形参，并且被调用时，编译器默认为int指定一个0值。

前置返回引用后置必须产生一个临时对象作为返回值，否则i++++;变为合法动作。同时对后置式返回一个const对象禁止以上的动作行为。

**千万不要重载&&  || 操作符**

C++对于 真假值表达式 采用骤死式 评估方式。意思是一旦该表达式的真假值确定，即使表达式中还有部分尚未检验，整个评估工作仍告结束。

例如

char \*p;

if((p!=0) && (strlen(p)>10) ...

如果p为空，strlen绝不会调用。否则对于一个null指针调用strlen结果是不可预期。

如果重载&& || 操作符，就没法提供程序员预期的某种行为模式。

STL

1.**STL中的vector：增减元素对迭代器的影响？**

这个问题主要是针对连续内存容器和非连续内存容器。

a、对于连续内存容器，如vector、deque等，增减元素均会使得当前之后的所有迭代器失效。因此，以删除元素为例：由于erase()总是**指向被删除元素的下一个元素的有效迭代器**，因此，可以利用该连续内存容器的成员erase()函数的返回值。常见的编程写法为：

**[cpp]** [view plain](http://blog.csdn.net/u012286517/article/details/50129617) [copy](http://blog.csdn.net/u012286517/article/details/50129617)

1. **for**(auto iter = myvec.begin(); iter != myvec.end())  //另外注意这里用 "!=" 而非 "<"

2. {

3.     **if**(**delete** iter)

4.         iter = myvec.erase(iter);

5.     **else** ++iter;

6. }

还有两种极端的情况是：

(1)、vector插入元素时位置过于靠前，导致需要后移的元素太多，因此vector增加元素建议使用push\_back而非insert；

2)、当增加元素后整个vector的大小超过了预设，这时会导致vector重新分分配内存，效率极低。因此习惯的编程方法为：在声明了一个vector后，立即调用reserve函数，令vector可以动态扩容。通常vector是按照之前大小的2倍来增长的。

b、对于非连续内存容器，如set、map等。增减元素只会**使得当前迭代器无效**。仍以删除元素为例，由于删除元素后，erase()返回的迭代器将是无效的迭代器。因此，需要在调用erase()之前，就使得迭代器指向删除元素的下一个元素。常见的编程写法为：

**[cpp]** [view plain](http://blog.csdn.net/u012286517/article/details/50129617) [copy](http://blog.csdn.net/u012286517/article/details/50129617)

1. **for**(autoiter = myset.begin(); iter != myset.end()) //另外注意这里用 "!=" 而非 "<"

2. {

3.          **if**(deleteiter)

4.                  myset.erase(iter++);  //使用一个后置自增就OK了

5.          **else**++iter;

6. }

**STL中排序**[**算法**](http://lib.csdn.net/base/datastructure)**的实现是什么？**

解答：STL中的sort()，在数据量大时，采用quicksort，分段递归排序；一旦分段后的数量小于某个门限值，改用Insertion sort，避免quicksort深度递归带来的过大的额外负担，如果递归层次过深，还会改用heapsort。

sort采用的是成熟的"快速排序算法"(目前大部分STL版本已经不是采用简单的快速排序，而是结合内插排序算法) 可以保证很好的平均性能、复杂度，stable\_sort采用的是"归并排序"，分派足够内存是，其算法复杂度为n\*log(n), 否则其复杂度为n\*log(n)\*log(n)，其优点是会保持相等

**++iterator 和iterator++效率问题？**

两种方式iterator遍历的次数是相同的，但在STL中效率不同，前++返回引用，后++返回一个临时对象，因为iterator是类模板，使用 it++这种形式要返回一个无用的临时对象，it++是[函数](http://cpro.baidu.com/cpro/ui/uijs.php?adclass=0&app_id=0&c=news&cf=1001&ch=0&di=128&fv=17&is_app=0&jk=9f2b4e334921bfe5&k=%BA%AF%CA%FD&k0=%BA%AF%CA%FD&kdi0=0&luki=1&n=10&p=baidu&q=65035100_cpr&rb=0&rs=1&seller_id=1&sid=e5bf2149334e2b9f&ssp2=1&stid=0&t=tpclicked3_hc&td=1836545&tu=u1836545&u=http://www.bubuko.com/infodetail-302566.html&urlid=0)重载，所以[编译器](http://cpro.baidu.com/cpro/ui/uijs.php?adclass=0&app_id=0&c=news&cf=1001&ch=0&di=128&fv=17&is_app=0&jk=9f2b4e334921bfe5&k=%B1%E0%D2%EB%C6%F7&k0=%B1%E0%D2%EB%C6%F7&kdi0=0&luki=3&n=10&p=baidu&q=65035100_cpr&rb=0&rs=1&seller_id=1&sid=e5bf2149334e2b9f&ssp2=1&stid=0&t=tpclicked3_hc&td=1836545&tu=u1836545&u=http://www.bubuko.com/infodetail-302566.html&urlid=0)无法对其进行优化，所以每遍历一个元素，你就创建并销毁了一个无用的临时对象。  
除了特殊需要和对内置类型外使用++it来进行元素遍历的。

[**hashmap与map hashmap与hashtable**](http://blog.csdn.net/cws1214/article/details/9842679)

[**http://blog.csdn.net/cws1214/article/details/9842679**](http://blog.csdn.net/cws1214/article/details/9842679)

[**STL map与Boost unordered\_map**](http://blog.csdn.net/orzlzro/article/details/7099231)

[**http://blog.csdn.net/orzlzro/article/details/7099231**](http://blog.csdn.net/orzlzro/article/details/7099231)

|  |  |
| --- | --- |
|  |  |

|  |
| --- |
|  |
|  |  |

|  |
| --- |
|  |
|  |  |

模板

模板是C++支持**参数化**多态的工具，使用模板可以使用户为类或者函数声明一种一般模式，使得类中的某些数据成员或者成员函数的参数、返回值取得任意类型。

模板是一种对**类型**进行**参数化**的工具；

通常有两种形式：**函数模板**和**类模板**；

函数模板针对仅**参数类型**不同的**函数**；

类模板针对仅**数据成员**和**成员函数类型**不同的类。

**使用模板的目的就是能够让程序员编写与类型无关的代码。**

**注意：模板的声明或定义只能在全局，命名空间或类范围内进行。即不能在局部范围，函数内进行，比如不能在main函数中声明或定义一个模板。**

**一、函数模板通式**

**1、函数模板的格式：**

**template** **<class** **形参名，class** **形参名，......>** **返回类型函数名(参数列表)**

**{**

**函数体**

**}**

其中**template**和**class**是关见字，**class**可以用**typename** 关键字代替，在这里**typename 和class没区别**，<>括号中的参数叫**模板形参**，模板形参和函数形参很相像，**模板形参不能为空。一但声明了模板函数就可以用模板函数的形参名声明类中的成员变量和成员函数，即可以在该函数中使用内置类型的地方都可以使用模板形参名。**模板形参需要调用该模板函数时提供的模板实参来初始化模板形参，一旦编译器确定了实际的模板实参类型就称他实例化了函数模板的一个实例。比如**swap**的模板函数形式为

**template <class T> void swap(T& a, T& b){}，**

当调用这样的模板函数时类型T就会被被调用时的类型所代替，比如**swap(a,b)**其中**a**和**b**是**int** 型，这时模板函数swap中的形参**T**就会被**int**所代替，模板函数就变为**swap(int &a, int &b)**。而当**swap(c,d)**其中**c**和**d**是**double**类型时，模板函数会被替换为**swap(double &a, double &b)**，这样就实现了函数的实现与类型无关的代码。

2、注意：对于函数模板而言不存在 **h(int,int)** 这样的调用，**不能在函数调用的参数中指定模板形参的类型，对函数模板的调用应使用实参推演来进行**，即只能进行 **h(2,3)** 这样的调用，或者**int a, b; h(a,b)**。

**二、类模板通式**

1、类模板的格式为：

**template<class** **形参名，class** **形参名，…>** **class** **类名**

**{ ... };**

类模板和函数模板都是以**template**开始后接模板形参列表组成，模板形参不能为空，**一但声明了类模板就可以用类模板的形参名声明类中的成员变量和成员函数，即可以在类中使用内置类型的地方都可以使用模板形参名来声明。**比如

**template<class T> class A{public: T a; T b; T hy(T c, T &d);};**

在类**A**中声明了两个类型为**T**的成员变量**a**和**b**，还声明了一个返回类型为**T**带两个参数类型为**T**的函数**hy**。

2、类模板对象的创建：比如一个模板类**A**，则使用类模板创建对象的方法为**A<int> m;**在类**A**后面跟上一个**<>**尖括号并在里面填上相应的类型，这样的话类**A**中凡是用到模板形参的地方都会被**int** 所代替。当类模板有两个模板形参时创建对象的方法为**A<int, double> m;**类型之间用逗号隔开。

3、对于类模板，模板形参的类型必须在类名后的尖括号中明确指定。比如**A<2> m;**用这种方法把模板形参设置为**int**是错误的（**编译错误：error C2079: 'a' uses undefined class 'A<int>'**），**类模板形参不存在实参推演的问题。**也就是说不能把整型值**2**推演为**int** 型传递给模板形参。要把类模板形参调置为**int** 型必须这样指定**A<int> m**。

4、在类模板外部定义成员函数的方法为：

**template<模板形参列表> 函数返回类型 类名<模板形参名>::函数名(参数列表){函数体}**

比如有两个模板形参**T1**，**T2**的类**A**中含有一个**void h()**函数，则定义该函数的语法为：

**template<class T1,class T2> void A<T1,T2>::h(){}。**

注意：当在类外面定义类的成员时**template**后面的模板形参应与要定义的类的模板形参一致。

5、再次提醒注意：模板的声明或定义只能在全局，命名空间或类范围内进行。即不能在局部范围，函数内进行，比如不能在**main**函数中声明或定义一个模板。

[**http://www.cnblogs.com/gw811/archive/2012/10/25/2738929.html**](http://www.cnblogs.com/gw811/archive/2012/10/25/2738929.html)

**模板代码膨胀如何消除？**

把C++模板中与参数无关的代码分离出来。也就是让与参数无关的代码只有一份拷贝。

**（1）模板生成多个类和多个函数，所以任何模板代码都不该与某个造成膨胀的模板参数产生相依关系。**

**（2）因非类型模板参数而造成的代码膨胀，往往可消除，做法是以函数或类成员变量替换template参数。**

**（3）因类型参数而造成的代码膨胀，往往可降低，做法是让带有完全相同的二进制表述的具现类型共享实现码。**

[**函数模板与类模板的区别**](http://blog.csdn.net/csw_100/article/details/5844615)**?**

函数模板的实例化是由编译程序在处理函数调用时自动完成的，而类模板的实例化必须由程序员在程序中显式地指定。即函数模板允许隐式调用和显式调用而类模板只能显示调用。

C++11

move语义使得你可以用廉价的move赋值替代昂贵的copy赋值，完美转发使得你可以将传来的任意参数转发给其他函数，而右值引用使得move语义和完美转发成为可能unique\_ptr

auto功能变为类型推断，通知编译器去根据初始化代码推断所声明变量的真实类型。

for range循环

防止对象拷贝，要想禁止拷贝，用=deleted 声明一下两个关键的成员函数

智能指针

Atomic

Thread

杂

**extern**

[**http://www.cnblogs.com/yc\_sunniwell/archive/2010/0**](http://www.cnblogs.com/yc_sunniwell/archive/2010/0)

[**http://blog.csdn.net/jiqiren007/article/details/5933599**](http://blog.csdn.net/jiqiren007/article/details/5933599)

**友元函数**

[**http://www.cnblogs.com/BeyondAnyTime/archive/2012/06/04/2535305.html**](http://www.cnblogs.com/BeyondAnyTime/archive/2012/06/04/2535305.html)

1.继承机制中对象之间是如何转换的？

2.继承机制中引用和指针之间如何转换？

3.继承机制中父类指针转换为子类指针发生了什么？

4.继承机制中子类指针转换为父类指针发生了什么？

[**c++中的左值与右值**](http://www.cnblogs.com/catch/p/3500678.html)[**move 与 forward**](http://www.cnblogs.com/catch/p/3507883.html)

[**http://www.cnblogs.com/catch/p/3500678.html**](http://www.cnblogs.com/catch/p/3500678.html)

[**http://www.cnblogs.com/catch/p/3507883.html**](http://www.cnblogs.com/catch/p/3507883.html)

**Mutable**

[**http://blog.csdn.net/tht2009/article/details/6920511**](http://blog.csdn.net/tht2009/article/details/6920511)

**内存对齐**

32位编译器：32位系统下指针占用4字节

**char** ：1个字节

**char**\*（即指针变量）: 4个字节（32位的寻址空间是2^32, 即32个bit，也就是4个字节。同理64位编译器）

**short** **int** : 2个字节

**int**：  4个字节

      unsigned **int** : 4个字节

**float**:  4个字节

**double**:   8个字节

**long**:   4个字节

**long** **long**:  8个字节

      unsigned **long**:  4个字节

64位编译器：64位系统下指针占用8字节

**char** ：1个字节

**char**\*(即指针变量): 8个字节

**short** **int** : 2个字节

**int**：  4个字节

      unsigned **int** : 4个字节

**float**:  4个字节

**double**:   8个字节

**long**:   8个字节

**long** **long**:  8个字节

      unsigned **long**:  8个字节