# 一、C++基础

## Q：面向对象的基本特征（封装、继承、多态 ）

      1.封装

　　　　简单来讲: 将现实世界的事物抽象成计算机领域中的对象,对象同时具有属性和行为,这种抽象就是封装.

　　　　封装的一个重要特性: 数据隐藏. 对象只对外提供与其它对象交互的必要接口,而将自身的某些属性和实现细节对外隐藏,通过这种方式，对象对内部数据提供了不同级别的保护，以防止程序中无关的部分意外的改变或错误的使用了对象的私有部分。这样就在确保正常交互的前提下,保证了安全性.

      2.继承

　　　　面向对象的一个重要特性是复用性.继承是实现复用性的一个重要手段.

　　　　可以在不重复编写已实现的功能的前提下,对功能进行复用和拓展.

　　　　继承概念的实现方式有二类：实现继承与接口继承。

　　　　　　\*实现继承是指直接使用基类的属性和方法而无需额外编码的能力

　　　　　　\*接口继承是指仅使用属性和方法的名称、但是子类必须提供实现的能力

      3.多态

　　　　当存在继承关系时,允许将父类对象看成为和它的一个或多个子类对象等同.

　　　　这样,可以根据当前赋给父类对象的子对象的具体特性以不同的方式进行运行.

 　　　　多态的几个前提

　　　　　　\* a:要有继承关系。

　　　　　　\* b:要有方法重写。

　　　　　　\* c:要有父类引用指向子类对象。

        　　多态的好处

　　　　　　\* a:提高了代码的维护性(继承保证)

　　　　　　\* b:提高了代码的扩展性(由多态保证)

　　　　多态的限制

          　　  \* 不能使用子类的特有属性和行为，不能调用虚函数以外的函数。

## Q：面向对象的六大基本原则

A：**单一职责**：一个类只做它该做的事情（高聚合）；

**开放封闭**：软件实体应该对扩展开放、对修改关闭；

**里氏替换原则**：任何时候都可以用子类型替换父类型，子类一定是增加父类的能力而不是减少父类的能力；

**依赖倒置**：高层模块不应该依赖于底层模块，两者应该依赖于其抽象；抽象不应该依赖具体实现，具体实现应该依赖抽象。例如人看历史书，历史书是具体，当需要增加人看地理书时，需要修改人的类；最好的方法是，抽象出一个书类，历史书和地理书都继承于它。

**合成聚合和复用**：优先使用聚合或合成关系复用代码

**接口隔离**：接口要小而专，决不能大而全。把接口分成满足依赖关系的最小接口，类B与类D不需要去实现与他们无关接口方法。

项目中用到的原则：单一职责、开放封闭、依赖倒置、接口隔离；

<https://www.cnblogs.com/qifengshi/p/5709594.html>

## Q：什么是野指针，如何避免？

野指针产生的原因：

1、指针定义时未被初始化：指针在被定义的时候，如果程序不对其进行初始化的话，它会指向随机区域，因为任何指针变量（除了static修饰的指针变量）在被定义的时候是不会被置空的，它的默认值是随机的。

2、指针被释放时没有被置空：我们在用malloc开辟内存空间时，要检查返回值是否为空，如果为空，则开辟失败；如果不为空，则指针指向的是开辟的内存空间的首地址。指针指向的内存空间在用free()或者delete（注意delete只是一个操作符，而free()是一个函数）释放后，如果程序员没有对其置空或者其他的赋值操作，就会使其成为一个野指针。

3、指针操作超越变量作用域：不要返回指向栈内存的指针或引用，因为栈内存在函数结束的时候会被释放，示例（转自高质量C++）：

野指针的危害：

野指针的问题在于，指针指向的内存已经无效了，而指针没有被置空，解引用一个非空的无效指针是一个未被定义的行为，也就是说不一定导致段错误，野指针很难定位到是哪里出现的问题，在哪里这个指针就失效了，不好查找出错的原因。所以调试起来会很麻烦，有时候会需要很长的时间。

规避方法：

初始化指针时将其置为NULL，之后再对其进行操作。

释放指针时将其置为NULL，最好在编写代码时将free()函数封装一下，在调用free()后就将指针置为NULL。

## Q：指针和引用的区别？

A：1指针是一个变量，这个变量指向的是一个地址，引用是一个变量的别名。指针指向一块内存，它的内容是所指内存的地址；而引用则是某块内存的别名。

2 引用必须初始化，定义以后无法改变；指针可以不初始化，可以改变。

3 引用不能用const，引用有类型检查，是类型安全的。

引用可以指向常量，也可以指向变量。例如int &a=b，使引用a指向变量b。而为了让引用指向常量，必须使用常量引用，如const int &a=1; 它代表的是引用a指向一个const int型，这个int型的值不能被改变，而不是引用a的指向不能被改变，因为引用的指向本来就是不可变的，无需加const声明。即指针存在常量指针（指向的那个对象不能被修改）int const \*p和指针常量（指针的指向不能修改）int \*const p，而引用只存在常量引用int const &a，不存在引用常量int& const a。

<http://www.cnblogs.com/kingln/articles/1129114.html>

## Q：extern关键字的作用

A： extern标示变量或者函数的定义在别的文件中，提示编译器遇到此变量和函数时在其他模块中寻找其定义。extern关键字的作用是共享代码。

一个c++程序可以包含多个文件，在一个文件中已定义的变量，如果要在另一个文件中调用，就要先声明一下变量，此时用extern这个关键字来声明。（注意：在一个程序中，变量只能被定义一次，但可以多次声明）区分是定义还是声明关键在于看计算机是否为变量分配了存储空间。

int i;    //这是定义

extern int i;     //这是声明

extern int i = 1; //这也是定义

1、在其他文件中定义过的全局变量，在另一个文件中要调用时，只需在声明语句前加关键字extern。

     //文件1，定义变量

     int i= 1；

     //文件2，声明变量

     extern int i;

2、对于常量，要调用其他文件的常量时，做法如下：

     //文件1,定义常量

     extern const int i = 1;

    //文件2，声明常量

     extern const int i;

**extern "C"**：告诉编译器，按C语言方式编译，请保持函参数类型和函数名连接在一起，以此区分不同函数。由于C++支持多态性与重载，编译时编译器会修饰函数名，C++编译后的函数需要加上参数的类型才能唯一标定重载后的函数，但是C语言不会，因此会造成链接时找不到对应函数的情况，此时C函数就需要用extern “C”进行链接指定，这告诉编译器，请保持我的名称，不要给我生成用于链接的中间函数名。

但是extern不能用于修饰static变量。static修饰的全局变量的作用域只能是本身的编译单元。在其他编译单元使用它时，只是简单的把其值复制给了其他编译单元，其他编译单元会另外开个内存保存它，在其他编译单元对它的修改并不影响本身在定义时的值。

<https://www.cnblogs.com/Dageking/p/3185230.html>

## Q：static和const（注意mutable关键字的使用）的区别？

A：**static：**

static限定符：限定变量或函数为静态类型，存储于静态存储区。具有**文件作用域**。

**C语言中三种用法**

**1. 静态局部变量：**用于函数体内部修饰变量，这种变量的生存期长于该函数。

**2.静态全局变量：**定义在函数体外，用于修饰全局变量，表示该变量只在本文件可见。

**3.静态函数：**静态函数跟静态全局变量的作用类似，表示该变量只在本文件可见：

静态局部变量的特点（括号内为note:2,也就是局部变量的对比）：  
（1）该变量在全局数据区分配内存(局部变量在栈区分配内存);   
（2）静态局部变量在程序执行到该对象的声明处时被首次初始化，即以后的函数调用不再进行初始化(局部变量每次函数调用都会被初始化);  
（3）静态局部变量一般在声明处初始化，如果没有显式初始化，会被程序自动初始化为0(局部变量不会被初始化);  
（4）它始终驻留在全局数据区，直到程序运行结束。但其作用域为局部作用域，也就是不能在函数体外面使用它(局部变量在栈区，在函数结束后立即释放内存);

**C++ 语言的 static 关键字有二种用途：**

**1.静态数据成员**：用于修饰 class 的数据成员，即所谓“静态成员”。这种数据成员的生存期大于 class 的对象（实体 instance）。静态数据成员是每个 class 有一份，普通数据成员是每个 instance 有一份，因此静态数据成员也叫做类变量，而普通数据成员也叫做实例变量。int book::accout = 0;初始化

**2、静态成员函数：**用于修饰 class 的成员函数。

1.静态成员之间可以相互访问，包括静态成员函数访问静态数据成员和访问静态成员函数;  
2.非静态成员函数可以任意地访问静态成员函数和静态数据成员;  
3.静态成员函数不能访问非静态成员函数和非静态数据**成员**; 有可能对象都没创建，但是静态成员函数还是可以用。  
4.调用静态成员函数，可以用成员访问操作符(.)和(->)为一个类的对象或指向类对象的指针调用静态成员函数,也可以用类名::函数名调用(因为他本来就是属于类的，用类名调用很正常)

https://www.cnblogs.com/xudong-bupt/p/3509567.html

**const：**

a、修饰变量，限定变量为常量。对于全局变量，定义时必须初始化，const限定其具有文件作用域，但是通过在定义const变量时加上extern，使可以在多个文件中共享const变量。

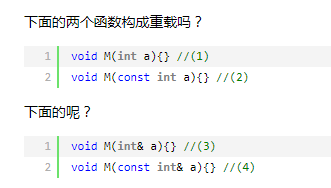
b、对于类内部的成员变量，如果是static const，且变量为整形（int、short、char、enum），可以在定义时初始化；对于非static的const变量，必须在构造函数的成员列表中初始化。

c、修饰类的成员函数，声明函数不会改变类的成员变量，mutable修饰的变量除外，const成员函数内无法调用非const函数，可能造成改变。

d、函数用const修饰可以实现函数的重载，如果该函数只有const版本，则const对象和非const对象都可以调用该函数。

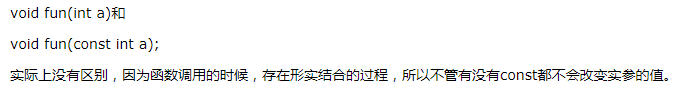
<https://www.cnblogs.com/xudong-bupt/p/3509567.html>

## Q：const重载



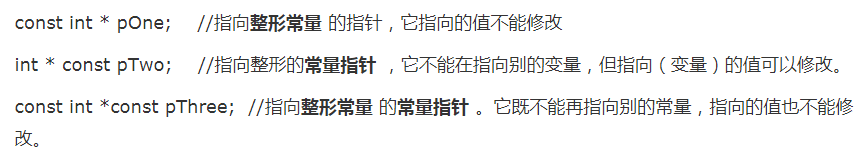
const在函数中的含义是该值在此函数范围内“无法修改”。站在调用者的角度，所有的值传递都是无法修改实参的。所以，(1)/(2)两个函数在调用者看来，是语义相同的，不能构成重载。

(4)与(3)的区别是，(4)无法修改引用指向的对象，而(3)可以。从调用者的角度，两个函数有不同的语义，构成重载。



那 fun(char \*a)和fun(const char \*a)是一样的吗？答案是：不一样。因为char \*a 中a指向的是一个字符串变量，而const char \*a指向的是一个字符串常量，所以当参数为字符串常量时，调用第二个函数，而当函数是字符串变量时，调用第一个函数。

但是char \*a和char \* const a，这两个都是指向字符串变量，不同的是char \*a是指针变量 而char \*const a是指针常量，这就和int i和const int i的关系一样了，所以也会提示重定义。



## Q：static能否与extern关键字一起使用

A： 不能。extern修饰的全局变量和函数，可以在其他文件中使用。

static只有文件作用域，不能在其他文件中使用。

## Q：全局变量、static变量、局部变量？

A：**全局变量：**

非静态全局变量是全作用域，作用于整个源程序；

static全局变量只有文件作用域。

**局部变量：**

普通变量存储于栈区，出函数释放；

static变量存储于静态存储区，出函数仍然保存。

<https://www.cnblogs.com/King-Gentleman/p/5350563.html>

## Q：mutable关键字

A： mutable修饰的数据成员，意思是**该数据成员不算对象的内部状态**，可以在const函数中修改它。mutable通常用于配合const函数使用。

例如，对于一个用来统计某个对象的访问次数的变量。它变成什么显然并不影响对象功用，但编译器并不知道：它仍然会阻止一个声明为const的函数修改这个变量。

把这个计数变量声明为mutable，编译器就明白了：这个变量不算对象内部状态，修改它并不影响const语义，所以就不需要禁止const函数修改它了。

eg：matable int a；

## Q：使用其它文件中的函数和变量的方法

A： 使用头文件（变量和函数需要在头文件中声明）；

使用extern关键字（变量和函数需要在其他.cpp文件中定义，但是不能对static变量和函数使用）

## Q：C++深拷贝、浅拷贝、转移拷贝；

A： 浅拷贝：将数据成员的值进行简单的值拷贝这就导致两个对象的指针域是同一块内存，所以在对象生存周期完毕时，调用析构函数，释放内存的时候出现core down的情况！对象中有指针时，简单拷贝指针的值。例如智能指针中的shared\_ptr。

深拷贝：对象有指针时，指针指向的内容也拷贝一份。

转移拷贝：move；

<https://www.cnblogs.com/gossiplee/p/4612983.html>

## Q：strlen与sizeof

**sizeof()**是运算符，在头文件中typedef为unsigned int，其值在编译时即计算好了，参数可以是数组、指针、类型、对象、函数等。 它的功能是：获得保证能容纳实现所建立的最大对象的字节大小。由于在编译时计算，因此sizeof不能用来返回动态分配的内存空间的大小。实际上，用sizeof来返回类型以及静态分配的对象、结构或数组所占的空间，返回值跟对象、结构、数组所存储的内容没有关系。  
    具体而言，当参数分别如下时，sizeof返回的值表示的含义如下：  
    数组——编译时分配的数组空间大小；   
    指针——存储该指针所用的空间大小（存储该指针的地址的长度，应该为4）；  
    类型——该类型所占的空间大小；   
    对象——对象的实际占用空间大小；   
    函数——函数的返回类型所占的空间大小。函数的返回类型不能是void。

**strlen()**是函数，要在运行时才能计算。参数必须是字符型指针（char\*）。当数组名作为参数传入时，实际上数组就退化成指针了。  
    它的功能是：返回字符串的长度。该字符串可能是自己定义的，也可能是内存中随机的，该函数实际完成的功能是从代表该字符串的第一个地址开始遍历，直到遇到结束符\0。返回的长度大小不包括\0。

https://blog.csdn.net/follow\_blast/article/details/79084726

Q：void \*

任何类型的指针都可以直接赋值给void\*，无需进行强制类型转换：

## Q：strcpy的实现，如何实现内存覆盖，地址越界怎么处理？strcpy和strncpy哪个更安全？

A： 字符串的拷贝函数可以使用strcpy、strncpy、memcpy，memmove等，其中strcpy、strncpy只能用于字符串拷贝。strcpy、strncpy、memcpy函数均未考虑内存重叠时的情况，如果要考虑内存重叠，可以使用memmove。

**考虑内存重叠的strcpy的实现：**[**https://www.cnblogs.com/luoquan/p/5265273.html**](https://www.cnblogs.com/luoquan/p/5265273.html)

char \* strcpy1(char \* destination, const char \* source)

{

char \*result = destination;

if (nullptr == destination || nullptr == source)

return nullptr;

size\_t len = strlen(source);

if (destination < source || source + len < destination)

{

while (\*source != '\0')

\*(destination++) = \*(source++);

}

//内存重叠时，从高地址到低地址复制

else

{

destination += len;

source += len;

while (len--)

\*(destination--) = \*(source--);

}

return result;

}

**strcpy和strncpy比较：**

char \*strncpy(char \*destination, const char \*source, size\_t num);

strcpy无法防止数组越界，strncpy增加了一个大小参数num，可以将num设置为destination数组的大小，以防止地址越界，复制num个字符。

## Q：new/delete和malloc/free的区别；

A： a、C++中的操作符；C中的标准函数，需要加stdlib.h

b、作用不同，malloc/free只负责内存的分配和释放；new/delete：调用opeator new(size\_t )分配内存、调用类的构造函数构造对象。

c、输入参数不同，new：数量、类型、初始值；而malloc：内存块字节数大小

d、返回值的安全性不同，new返回的是对象的指针，malloc返回的是void型指针，需要强制转换。

e、new/delete可以通过重载operator new/operator delete、set\_new\_handler等改变new/delete的行为；malloc/free不能被重载。**delete销毁对象，释放内存**

f、new分配失败时，如果用户重载了类的set\_new\_handler，则按照用户设定的方式提示错误，否则抛出bad\_alloc异常；malloc分配内存失败时返回NULL；

内存调用失败时调用new\_handler

free和delete都只是释放了内存，内存中保存的内容删除了，但是指针仍然指向这个地址。不置为NULL就是一个野指针

<http://blog.csdn.net/nie19940803/article/details/76358673>

<https://www.cnblogs.com/maluning/p/7944231.html>

在C++中，内存区分为5个区

**全局/静态存储区域**：存全局变量，静态变量

**栈：**存放函数的参数值，局部变量，函数执行结束时会被自动释放。栈内存分配运算内置于处理器的指令集中，效率高，但是容量有限。

**堆（动态内存分配）**：通过new和malloc由低到高分配，由delete或free手动释放或者程序结束自动释放

**字符/文字常量区**： 存放常量字符串，程序结束时由系统释放

**程序代码区**： 存放函数体的二进制代码

## Q：new、operator new、placement new区别

A：**new**

C++操作符，在堆上分配内存并构造出对象。先调用operator new分配内存，再调用构造函数初始化内存；返回对象的指针。

**operator new**

用于分配内存，可以被重载。返回void指针。

**placement new**

对operator new重载的一个版本，在已经分配的内存上调用构造函数创建对象，返回void指针。new创建的对象可以直接调用~析构函数析构对象，但并不释放空间，可以继续调用placement new在该位置构造对象。

例如：

MyClass \* p1 = new MyClass;//先调用operator new (sizeof(MyClass))，再调用MyClass默认构造函数

MyClass \*p2 = new(p1)MyClass;//先调用operator new（sizeof(MyClass)，p1），再调用构造函数在P1处构造对象。

## Q：智能指针

A：**智能指针的目的？**

管理堆空间，防止内存泄露和二次释放等问题。

**智能指针的实现：**

智能指针是一种RAII技术（资源获取即初始化，利用对象生命周期控制资源获取及释放），它对普通指针进行封装。利用对象在离开作用域时需要调用析构函数的特性，保证堆内存的释放。

**auto\_ptr：**

auto\_ptr的核心思想就是管理权限的转移，通过赋值运算符的重载或拷贝构造时，把p1置空，即将p1开辟的空间的管理权限转交给p2后

a、不允许共享指针，用移动语义实现拷贝。拷贝构造时，将原指针置为null。

b、不能用于保存指向动态分配的数组的指针，不保证全部释放；

c、auto\_ptr对象不能存储在容器中。原因是其拷贝语义。

**shared\_ptr（网络编程中的close就是通过类似shared\_ptr的方式实现）：**

a、shared\_ptr允许多个智能指针指向相同的对象。采用引用计数控制内存释放。

b、shared\_ptr存在环形引用的问题，例如children和parent互相引用，此时其引用计数永远都是1，内存泄露。

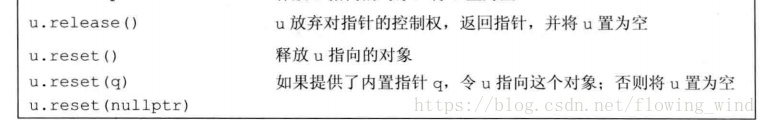
c、shared\_ptr不是多线程安全的，shared\_ptr的引用计数，需要维护两个成员指针，且对两个指针不是原子操作。多个线程同时写shared\_ptr需要加锁。

d、同auto\_ptr相同，shared\_ptr不能用于保存指向动态分配的数组的指针。

<http://www.cppblog.com/Solstice/archive/2013/01/28/197597.html>

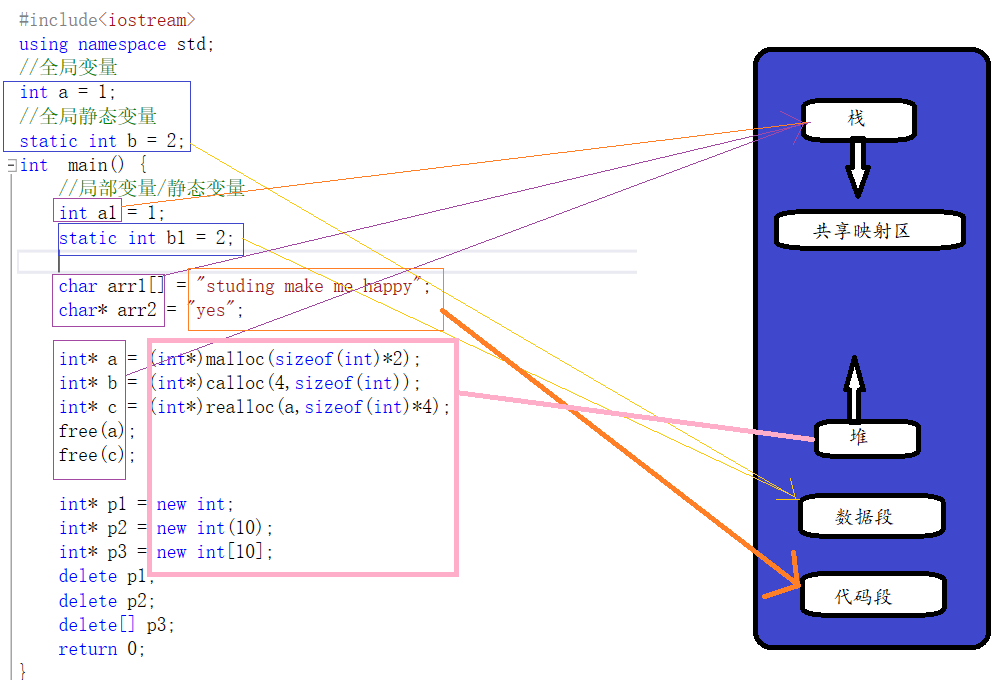
**unique\_ptr：**

同一时刻只能有一个unique\_ptr指向给定对象。禁止拷贝赋值运算符、允许移动赋值运算符。



**weak\_ptr:**

weak\_ptr是一种弱引用，可以用于解决shared\_ptr的环形引用问题。

weak\_ptr需要用shared\_ptr初始化，但是不会导致shared\_ptr的引用计数增加。expired()用于检測所管理的对象是否已经释放；lock()用于获取所管理的对象的强引用指针。 https://blog.csdn.net/leichaowen/article/details/53064294

1）栈又叫堆栈，非静态局部变量/函数参数/返回值等等，栈是向下增长的。

2）内存映射段是高效的I/O映射方式，用于装载一块共享内存，用户可以通过接口创建共享内存，做进程间通信。

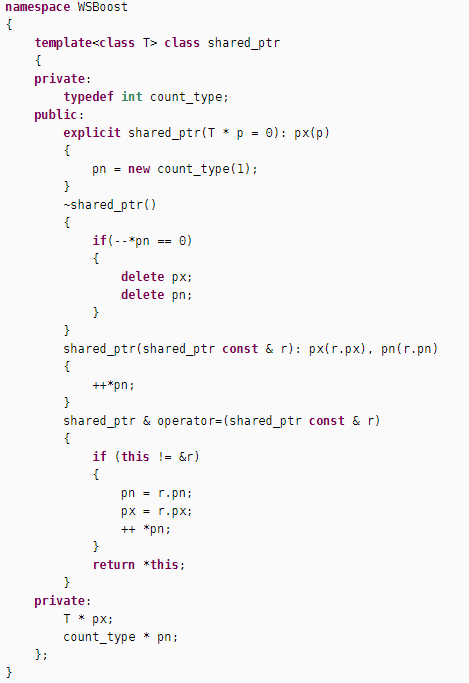
3）堆，用于程序运行是动态内存分配，堆上向上增长的。

4）数据段---存储全局数据和静态数据。

5）代码段---可执行的代码/只读常量。

**Q：手写智能指针的实现**

A：



<http://daojin.iteye.com/blog/900764>

C++中的explicit关键字只能用于修饰只有一个参数的类构造函数, 它的作用是表明该构造函数是显示的, 而非隐式的 https://www.cnblogs.com/rednodel/p/9299251.html

## Q：如何禁止拷贝语义（禁止拷贝对象）

A： a、声明类的拷贝构造函数、拷贝赋值运算符为=delete；

1. NoCopyable(const NoCopyable&) = delete;
2. const NoCopyable& operator = (const NoCopyable&) = delete;

b、拷贝构造函数和拷贝赋值运算符声明为 private， 并不给出实现；一般而言，成员函数和友元函数还是有可能调用它们，在链接期间，这会导致一个链接错误（因为你没有定义函数）；

c、可将错误移到编译阶段。专门定义一个类，类中拷贝构造函数和拷贝赋值运算符被声明为 private。让目标类继承该类，当成员函数和友元函数还是有可能调用目标类的拷贝函数时，编译器便试着生成一个copy构造函数，这些函数会尝试调用基类的拷贝函数，这些调用会被编译器拒绝，并报错。

Q：右值引用、移动语义、完美转**发**

A：**右值引用**

绑定到右值的引用，右值要么是字面常量，要么是表达式求值过程中创建的临时对象。右值引用的目的是支持移动操作；右值引用必须绑定到即将被销毁的对象，且该对象没有其他用户。所以右值引用可以自由接管所引用的对象的资源。

**移动语义**

移动资源而不是拷贝资源。根据右值引用，将临时对象的资源直接用于其他对象，避免不必要的析构和重新构造开销。**move函数接受一个参数，返回一个该参数的右值引用。**

例如：

vector<A>V;

A obj(10);

V.push\_back(std::move(obj)); //调用移动语义，拷贝完成后，obj为空

v.push\_back(A(10)); //STL进行了优化，这样会调用const T&&版本的push

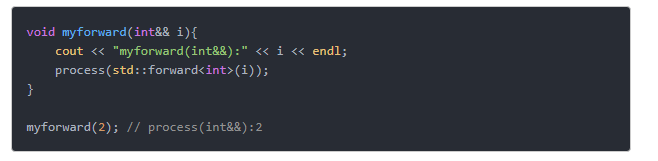
C++11中的swap是通过三个move实现

template <class T> void swap (T& a, T& b) { T c(std::move(a)); a=std::move(b); b=std::move(c); }

**完美转发**

将实参“完美”地传给形参，完美指的是参数的const属性和左右值属性不变。

用右值引用折叠参数的const属性和左右值属性。



template<typename T>

void cover(T&& para)

{ func(forward(forward<T>(para))); //利用右值折叠参数类型，再用forward转成原始类型

}

<http://www.jb51.net/article/91290.htm>

## Q：手写单例模式（包括考虑线程安全和不考虑线程安全两种）

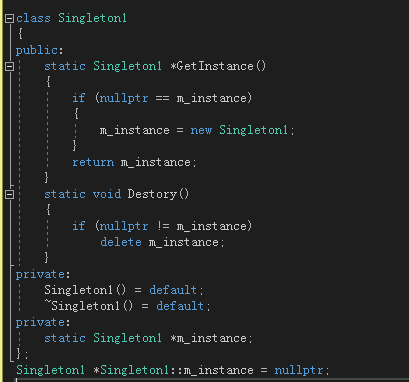
A：　手写时有两点需要注意：

a、class定义的末尾有一个“;”；

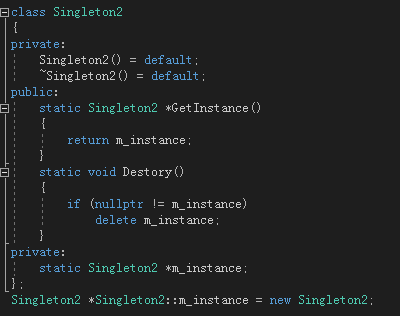
b、class的静态成员在类外定义时，仍然需要声明数据类型；

eg：A \*A::m\_instance = new A;

//线程不安全



//饿汉模式（线程安全）



饿汉模式可以实现多线程安全，但是单例的创建时间不是首次调用instance()函数时，而是在首次用到singleton类的时候，例如，调用singleton中的其它public的静态函数时，就会创建出实例。更好的方法是创建一个单独的类：

class singleton\_instance

{

public:

static singleton \*instance;

}

singleton \*singleton\_instance::instance = new singleton;

对应修改singleton如下：

class singleton

{

protected:

    singleton()

    {}

public:

    static singleton\* initance();

};

singleton\* singleton::initance()

{

    return singleton\_instance::instance;

}

https://www.cnblogs.com/dupengcheng/p/7205527.html

## Q：C++和java的区别？

A： a、java不提供指针，有效防止c/c++语言中指针操作失误（空指针、二次释放）；

b、java不支持操作符重载；

c、java是完全面向对象的语言，所有函数和变量都必须是类的一部分；C++允许定义全局的变量和函数；

d、java提供自动内存管理功能。

## Q：C和C++的区别

A： a、C++是C的超集，兼容C的大部分语法；

b、C++是面向对象语言，C是面向过程的语言；

c、C++采用new/delete动态管理内存，C采用malloc/free；

d、C++支持函数重载；

e、C++支持引用；

f、C++采用iostream类库替代了C中的stdio函数库；

## Q：对于一个空类A，sizeof(A)是多少？

A： 一个字节，每个实例在内存中都有一个独一无二的地址，为了达到这个目的，编译器往往会给一个空类的对象隐含的加一个字节，这样空类在实例化后在内存得到了独一无二的地址。

## Q：形参和实参的区别？

A：实参指的是外部调用的时候传入的参数，形参是函数内部使用的参数。

<http://blog.csdn.net/haojiahj/article/details/11155249>

## Q：预处理、编译、汇编？

A： 预处理、编译、汇编、链接、运行

预处理：替换宏定义、替换头文件；

编译：生成汇编代码；通过词法分析、语法分析，在确认所有的指令都符合语法规则之后，将其翻译成等价的中间代码或汇编代码。在C++中，以下函数如果你没有定义但是被需要时，编译器会产生他们：1. 默认构造函数、拷贝构造函数、析构函数、赋值运算符和一对取址运算符. 2. 替换inline内联函数， 3. 重载函数生成中间函数名编译期，编译器只检测程序语法及函数、变量是否被声明。在链接期，链接器会在所有的目标文件中查找函数的实现，如果找不到，会报未定义的外部符号错误。

汇编：生成目标代码（机器代码）；把汇编语言代码翻译成目标机器指令的过程，目标文件包括代码段，数据段

链接：链接库文件；

<http://blog.csdn.net/weeds085490/article/details/72834487>

## Q：动态链接和静态链接

A：**静态链接**

特点：链接阶段，把 要调用的库函数直接链接到可执行文件，成为可执行文件的一部分。

优点：在程序发布的时候就不需要的依赖库，程序可以独立执行。

缺点：a、程序体积会相对大一些；b、如果静态库有更新的话，所有可执行文件都得重新链接。

**动态链接**

特点：链接时不拷贝函数代码到目标文件中，只记录函数名和参数等信息，程序运行阶段，去动态库中加载函数代码。

优点：多个程序可以共享同一段代码，而不需要存储多个拷贝。**不同的应用程序如果调用相同的库，那么在内存里只需要有一份该共享库的实例**，规避了空间浪费问题

缺点：由于是运行时加载，可能会影响程序的执行性能。

<http://blog.csdn.net/lisongjia123/article/details/49003159>

## Q：动态链接库和静态链接库

A：**静态库**：在vs中新建生成静态库的工程，编译生成成功后，只产生一个.lib文件

**动态库**：在vs中新建生成动态库的工程，编译成功后，产生一个.lib文件和一个.dll文件

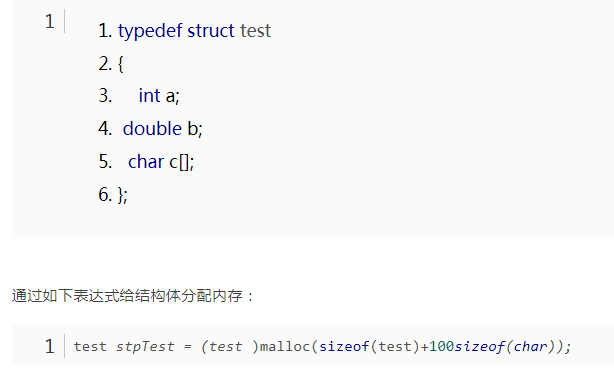
**静态库中的lib**：该LIB包含函数代码本身（即包括函数的索引，也包括实现），在编译时直接将代码加入程序当中

**动态库中的lib**：该LIB包含了函数所在的DLL文件和文件中函数位置的信息（索引），函数实现代码由运行时加载在进程空间中的DLL提供

**总之，lib是编译时用到的，dll是运行时用到的。如果要完成源代码的编译，只需要lib；如果要使动态链接的程序运行起来，只需要dll**

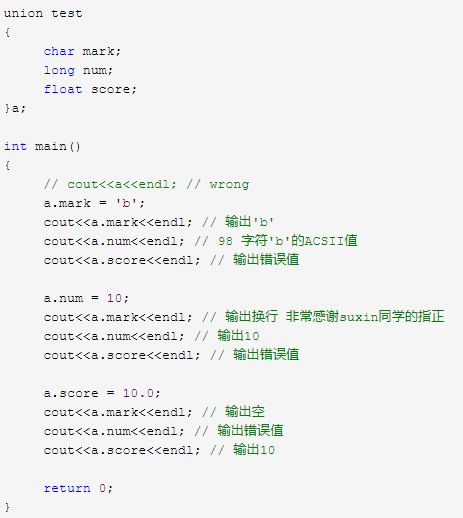
## Q：柔性数组

A： 结构体中，最后一个元素是一个零长数组，该数组不占用结构体空间。该数组中可以动态分配大小，可以很容易创建出缓冲区和数据包，相对于定长数组，使用柔性数组可以节省空间，在通信中节省流量。



## Q：union操作符

A： union是一种数据类型，一个union结构中，允许装入union中定义的任意一种数据，这些数据共用一段内存空间。union变量的长度是各成员的最大长度。使用union类型可以减少类型强制转换的次数。



## Q：windows下和linux下检测内存泄露的方法

A：

windows：

a、包含crtdbg.h头文件，在入口函数中添加\_CrtDumpMemoryLeaks();

b、使用Visual Leak Detector插件（VLD）, 包含入口函数的*CPP*或*C*文件中把*vld.h*头文件包含进来即可

linux：

使用valgrind工具，编译时使用-g选项，检查时，调用以下命令：

*gcc -Wall example.c -g -o example*

valgrind --tool=memcheck --leak-check=full ./test

## Q：VS中常用工具

A： a、VA（番茄）

b、VLD插件，检查内存泄露；

## Q：什么是交叉编译？

A： 一般采用的是本地编译，即在当前平台下，编译出来的程序只能放在当前平台下运行。交叉编译是指在当前编译平台下，编译出在其它体系结构下另一平台上，但在编译平台本身不能运行该程序。比如在X86平台编译能在ARM上运行的程序。

## Q：SVN和git的区别？

A： a、SVN是一个集中式版本控制系统，GIT是一个分布式版本控制系统。

b、GIT在check out代码后会在自己机器上克隆一个自己的版本库。可以在本地进行提交和查看历史版本记录等。

c、GIT把内容按照元数据方式存储，而SVN是按照文件的方式。

d、GIT没有一个完整的全局版本号，而SVN有

e、SVN的每个分支就是一个完整的目录，目录中包含了完整的实际文件。GIT的每个分支只需要获取每个版本的元素，然后只载入主要的分支。

f、SVN有一个中央版本库，当中央版本库出问题时，所有工作成员一起摊还。对于GIT，每个GIT就是一个版本库。如果主要版本库出问题，工作成员仍然可以在本地版本库提交。

## Q：程序在debug版本下运行正常，但在release下运行出错，可能原因：

A： debug和release只是默认的编译选项不同。变量优化、初始化、

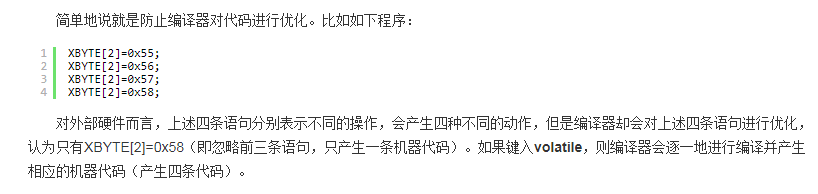
a、release下，常用变量可能会被编译器优化到寄存器中，如果该变量需要在多线程中保证同步时，可能会导致出错。此时应该将这种变量声明为volatile（不可优化的）

<https://blog.csdn.net/sinat_41144773/article/details/90724182>

b、变量优化，bebug版本下的一些变量可能会掩盖数组越界。在release版本下，该变量可能并未被用到，被编译器优化去掉，导致数组出现越界。

c、debug下会用0xCC（烫烫）初始化所有自动变量，0xCD（屯屯）填充堆中未初始化的内存。release下，未初始化的变量是随机的。

<http://blog.csdn.net/likexin/article/details/1809742>



## Q：哪些原因会导致栈溢出？

A： 函数堆栈默认大小是1M。

a、函数出现无限递归；

b、局部数组变量空间太大；

## Q：如何用C实现C++的封装、继承、多态？

A： 可以使用结构体实现，C的结构体本身不能有成员函数，可以通过定义函数指针实现对变量和函数的封装。通过在子类中包含指向基类的指针实现继承。通过在void \*指针的强制转换实现多态。使用GObject框架可以实现。

<http://dongxicheng.org/cpp/ooc/>

## Q：C++异常是直接结束程序吗，可以在异常处理函数中抛出异常吗

A：执行一个函数的过程中发现异常，可以不用在本函数内立即进行处理， 而是抛出该异常，让函数的调用者直接或间接处理这个问题。 直接跳到函数最后，一层层返回，直到被捕捉。 可以，catch中继续throw。

## Q：从汇编的角度看函数调用的过程

A： a、当前栈基地址压栈；（保存当前基地址，重新开始一个新的栈）

b、建立新的栈帧，实参参数从右往左入栈；

c、下一条指令地址进栈；（函数调用返回后从这里继续）

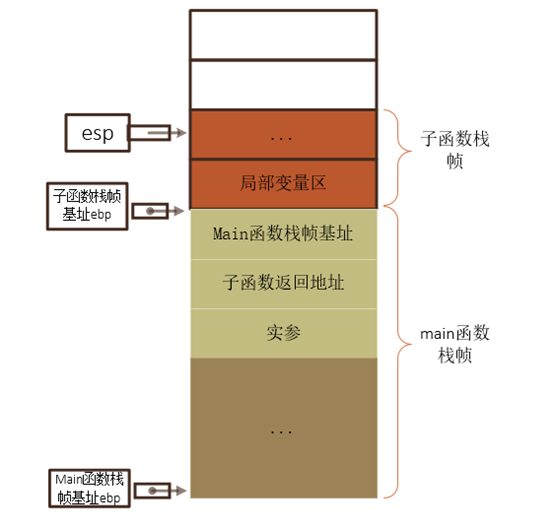
d、修改指令寄存器rip的值，使其指向被调函数执行位置；

d、栈顶指针esp归位，回到子函数栈底指针ebp；（清空当前函数所使用的栈）

e、基地址ebp回退到上一个函数（main）的基地址ebp；

f、eip退回到上一个函数即将要执行的语句上

 ebp（栈帧基址指针）则是指明了当前函数的栈帧基地址，对函数的资源（局部变量、实参等）的访问，都要通过 ebp+offset（偏移量）来进行访问。而 esp 则是栈指针，指示当前栈空间栈顶的位置。



## Q：inline关键字

A：**inline函数的申请**

a、显式设置：在头文件中，函数声明的前面加上inline；

b、隐式设置：类的成员函数默认是inline。

**inline的优点：**

a、编译期以函数本体替换函数调用，免除了执行时的函数调用开销；

b、编译器的最优化机制可以对inline函数执行相关最大优化；

**inline的缺点：**

a、过度声明inline函数会导致代码膨胀；

b、inline函数的存在会导致版本升级时，需要重新编译代码，无法用于动态链接。

c、inline只是一个申请，编译器可以进行忽略；太过复杂的函数（例如带有递归、循环），编译器拒绝inline；

d、通过函数指针调用的函数拒绝inline；

e、拒绝对虚函数inline；因为虚函数是运行时确定，inline是编译期替换；

## Q：C++11的新特性

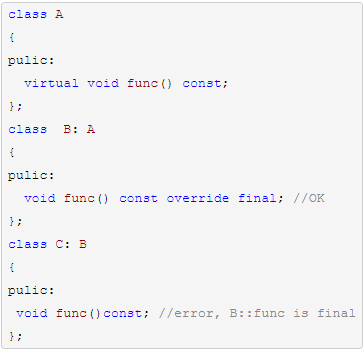
A： 增加了long long类型、auto自动推导类型、lambda表达式、智能指针、右值引用、final（阻止继承）、范围for循环、unordered\_map、unordered\_set。

int 16, long 32, long long 64

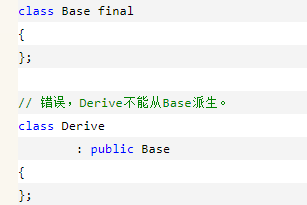
## Q：final关键字

A： a、修饰类，表示该类不能再被派生；

b、修饰虚函数，表示该虚函数不能再被复写；

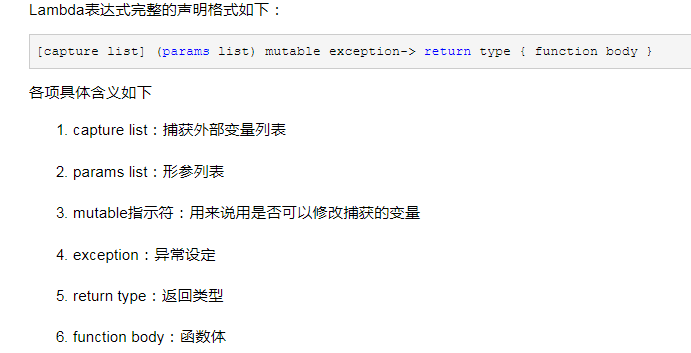


**override明确地表示一个函数是对基类中一个虚函数的重载**



## Q：lambda表达式

A： lambda表达式是函数对象，每个lambda表达式对应一个重载了函数调用操作符的类的一个实例。



sort(lbvec.begin(), lbvec.end(), [](int a, int b) -> bool { return a < b; }); // Lambda表达式

https://www.cnblogs.com/DswCnblog/p/5629165.html

## Q：隐式转换

A： a、算术转换；

b、数组转换成指针, 实参到形参；

c、算术类型、指针类型转换bool型；

d、类类型的转换；对于存在单个参数的构造函数的类来说，会调用该构造函数。

https://www.cnblogs.com/lijiaxin/p/10659157.html

## Q：RAII（Resource Acquisition Is Initialization，资源获取即初始化）

A： 利用对象的生命周期来控制程序资源，STL容器、智能指针都是RAII

RAII（Resource Acquisition Is Initialization）是一种利用对象生命周期来控制程序资源（如内存、文件句柄、网络连接、互斥量等等）的简单技术。   
　　RAII 的一般做法是这样的：在对象构造时获取资源，接着控制对资源的访问使之在对象的生命周期内始终保持有效，最后在**对象析构的时候释放资源**。借此，我们实际上把管理一份资源的责任托管给了一个对象。这种做法有两大好处：

* 不需要显式地释放资源。
* 采用这种方式，对象所需的资源在其生命期内始终保持有效。

## Q：C++指针、引用存在的问题

A： 指针就是一块内存的地址，通过指针无法判断出内存中的对象是否有效。这是C/C++指针问题的根源。

## Q：如何保证代码的异常安全性

A： a、以对象管理资源，不直接操作裸指针，防止内存泄露；

b、做异常安全保证：保证程序要么到达函数成功执行后的状态；要么回到函数被调用前的状态

一个函数如果说是“异常安全”的，必须同时满足以下两个条件：1.不泄漏任何资源；2.不允许破坏数据

## Q：声明和定义的区别

A： a、声明：用于向程序表明变量的类型和名字。给一个类型和名称

定义：指建立变量或函数，并且为它们分配内存空间，还可为变量指定初始值。程序中，变量有且仅有一个定义。

b、在编译期，编译器只检测程序语法及函数、变量是否被声明。在链接期，链接器会在所有的目标文件中查找函数的实现，如果找不到，会报未定义的外部符号错误。

<http://www.jb51.net/article/54613.htm> 深度好文

# 二、类、继承

## Q：什么是多态，C++中多态是如何实现的（结合虚函数表、虚表指针，介绍单继承、多继承及菱形继承、虚继承）

<https://blog.csdn.net/ijn842/article/details/80936236> **应用案例**

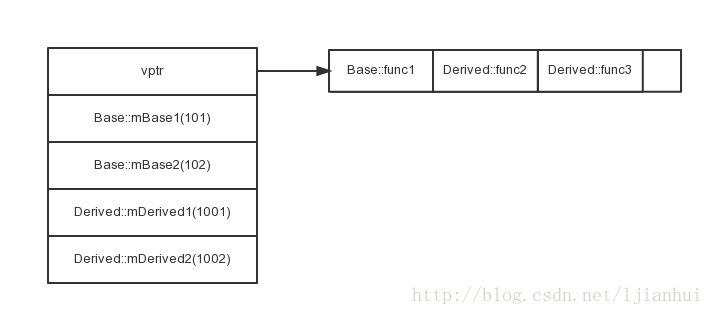
多态：关于多态，简而言之就是用父类型别的指针指向其子类的实例，然后通过父类的指针调用实际子类的成员函数。这种技术可以让父类的指针有“多种形态”，这是一种泛型技术。多态性在C++中是通过虚函数实现的。虚函数就是允许被其子类重新定义的成员函数。而子类重新定义父类虚函数的做法，称为“覆盖”，或者称为“重写”。子类重写父类中虚函数时，即使不用virtual声明，该重载函数也是虚函数。

A： 在C++中是通过虚函数和继承、虚继承实现多态，在派生类的虚函数表中，派生类函数指针会覆盖基类虚函数指针。

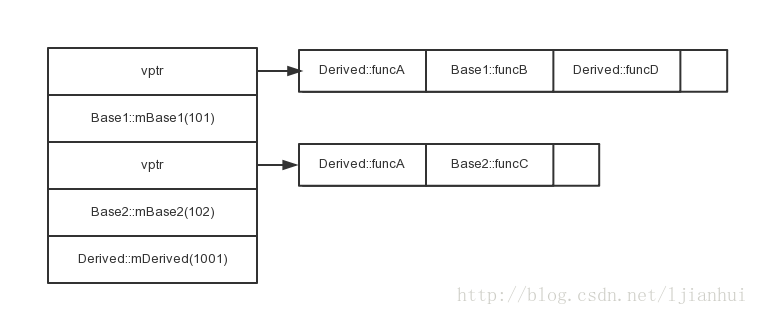
1.      每一个class产生一堆指向虚函数的指针，放在表格之中。这个表格称之为虚函数表（virtual table，vtbl）。

2.      每一个对象被添加了一个指针，指向相关的虚函数表vtbl。通常这个指针被称为虚表指针。vptr的设定（setting）和重置（resetting）都由每一个class的**构造函数**，**析构函数**和**拷贝赋值运算符**自动完成。

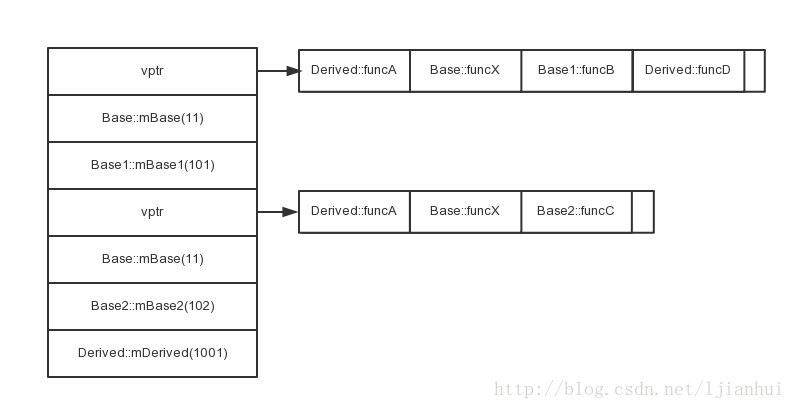
单继承



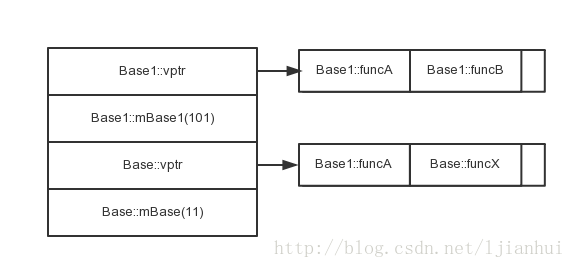
多继承



重复继承：某个父类被间接地重复继承了多次。



单一虚继承



1. class Base { ...... };
2. class Base1 : virtual public Base { ...... };

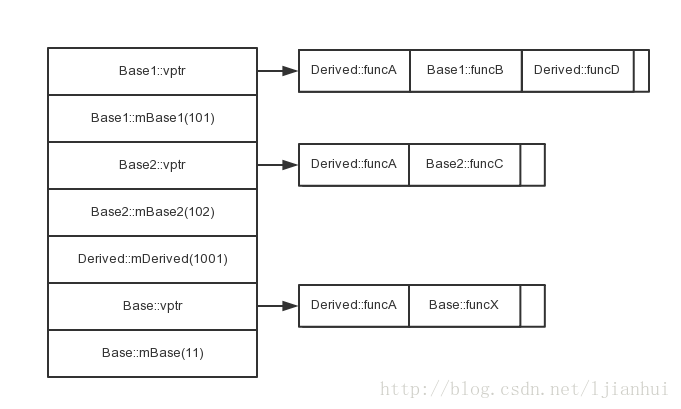
单一虚继承与单一继承的对象的内存布局存在明显的不同。表现为以下的方面：

1）成员的顺序问题。在普通的单一继承中，基类的成员位于派生类的成员之前。而在单一虚继承中，首先是其普通基类的成员，接着是派生类的成员，最后是虚基类的成员。

2）vptr的个数问题。在普通的单一继承中，派生类只有一个虚函数表，所以其对象只有一个vptr。而在单一虚继承中，派生类的虚函数表有n个（n为虚基类的个数）额外的虚数函数表，即总有n+1个虚函数表。

3）派生自虚基类的派生类的虚函数表中，并不含有虚基类中的virtual函数，但是派生类重写的virtual函数会在所有虚函数表中得到更新。如本例中，第一个虚函数表中，并不含有Base::funcX的函数地址。

菱形继承

class Base { ...... };

class Base1 : virtual public Base { ...... };

class Base2 : virtual public Base { ...... };

class Derived : public Base1, public Base2 { ...... };

C++提供虚继承机制，防止类继承关系中成员访问的二义性。

<https://www.cnblogs.com/longcnblogs/archive/2017/10/09/7642951.html> 好文

<https://blog.csdn.net/wue1206/article/details/81272053>虚继承

<https://blog.csdn.net/lyztyycode/article/details/81326699>单继承、多继承

<http://www.cnblogs.com/skynet/p/3343726.html> 超级复杂

<https://www.cnblogs.com/jianyungsun/p/6361670.html> 虚函数

<https://www.cnblogs.com/Rosanna/p/3331578.html>类的继承与派生

## Q：为什么析构函数用虚函数

Base类析构函数不为虚函数的情况下

Base \*basePtr = new Derive();  先调用基类构造，再调用派生类构造

delete basePtr;  只调用基本析构，不会调用派生类析构

在基类中将析构函数标明为虚函数，就表示在使用析构函数时，是采用动态联编的。那么delete basePtr的时候不再是采用静态联编直接在编译的时候确定basePtr指向的析构函数，而是在运行的时候根据 指向的类型来调用析构函数。

https://blog.csdn.net/yue\_jijun/article/details/82290853

## Q：重载、覆盖、隐藏

A： a、**重载**是对同一函数名，可以定义多个不同参数列表、不同实现的函数；编译期间，编译器根据不同参数列表，对函数名称进行修饰，对于编译器而言，这些同名函数就成了不同函数。其它地方调用时，根据参数类型的匹配，决定选用最合适定义。

b、**覆盖（重写）：**覆盖往往与继承的类中有virtual修饰的函数有关。virtual修饰的函数为虚函数。要构成覆盖就得满足基类中有virtual修饰的函数，而在派生类中有与基类中的虚函数同名且同参数列表的函数，那么派生类中的的该函数就会将基类中的函数覆盖

c、**隐藏:** 派生类将继承的基类的同名的成员变量和成员方法隐藏起来，通过派生类只能访问到自己的成员变量和成员方法。若想要访问基类的成员变量和成员方法需加上基类的作用域。

https://blog.csdn.net/magic\_world\_wow/article/details/82152839

## Q：public、protected、private继承及基类成员在子类中的可见性？

**A：**在修饰类的成员时，

public：在类内、类外、子类、友元中都能访问。

protected：在类内、子类、友元中访问。

private：只能在类内、友元中访问。



## Q：构造函数、析构函数的调用顺序

A：构造函数：基类、成员对象、自身

析构函数：自身、成员对象、基类

## Q：友元函数、友元类

## Q：构造函数、析构函数做了哪些事情

A： 构造函数：初始化基类变量、初始化成员变量、设置vptr虚表指针（虚函数）、设置指向虚基类的指针（虚继承）；

析构函数：析构成员对象、重设vptr虚表指针（虚函数）、调用基类析构函数、调用虚基类析构函数（虚继承）。

## Q：析构函数中可以抛出异常吗，如果抛出了怎么办？

A： 不可以，如果析构函数抛出异常，可能产生半析构的对象，再次使用可能会出错。

如果析构函数中可能抛出异常，需要用try catch将异常在析构函数内处理，不能抛出析构函数之外。

## Q：构造函数中能不能抛出异常？

A： 可以抛出异常

a、因为构造函数没有返回值，所以在构造函数中抛出异常是C++中通知对象构造失败的唯一方法。

b、构造函数中抛出异常将导致对象的析构函数不被执行；因为构造不完全，C++拒绝为没有完成构造操作的对象调用析构函数。同时，由于构造函数本身也是一个函数，在函数体内抛出异常将导致当前函数运行的结束，并释放已经构造的成员对象，当然包括其基类的成员，即要执行直接基类和成员对象的析构函数。成员对象和析构函数要构造析构一遍，但是本对象的不会。

c、当对象发生部分构造时，已经构造完毕的子对象（非动态分配）将会逆序地被析构。

<http://blog.csdn.net/lisonglisonglisong/article/details/44276343>

https://blog.csdn.net/ywcpig/article/details/52544147

## Q：构造函数能否写成虚函数

A： 不可以，虚函数的调用依赖于指向类的虚表指针，而虚表指针是在构造函数中初始化。如果构造函数是虚函数，则调用构造函数时，虚表指针并未初始化，此时会引起错误。

但是可以采用工厂方法模式，为每个类声明一个工厂类，工厂类中声明一个create的虚函数，返回对应的产品对象。通过工厂类的基类调用create函数时，可以实现虚构造函数的效果。

## Q：default和delete

A：**default**

当使用=default修饰类的成员函数时，指示编译器生成合成版本，类外可以不用显示定义函数内容；自定义默认构造函数效率不高，不定义则无法使用默认构造。C++ 的类有四类特殊的成员函数，分别为：默认构造函数，析构函数，拷贝函数以及拷贝赋值函数。如果程序没有显式地为一个类定义某个特殊成员函数，而又需要用到该特殊成员函数时，编译器会隐式地为这个类生成一个默认的特殊成员函数。如果你已经定义了一个有参的构造函数，那么默认构造函数编译器就不再生成了，你可以通过default关键字让默认构造函数恢复。相对于 =default，还有一个=delete，顾名思义就是禁止生成。

**delete**

使用=delete修饰类的成员函数时，通知编译器，我们声明了他，但不希望以任何方式定义或者使用他。编译器也不会生成默认的这类函数；另外，对于删除析构函数的类，不能生成其对象。例如：iostream类阻止拷贝。

https://www.cnblogs.com/lsgxeva/p/7787438.html

## Q：拷贝构造函数、拷贝函数

A： //拷贝构造函数

class A(const class A &a) //如果不加const，则该函数不能用于const型的拷贝构造；如果不加引用，将产生无限递归；

//拷贝函数

class A & operator=(const class A &a)

{

if(this == &a)

return \*this;

{

//考虑线程安全

class A tmp = a;

swap(&tmp, this);

}

return \*this;

}

## Q：拷贝构造函数和拷贝函数的输入必须是const型引用吗？

A：**为什么必须传引用**

对于拷贝构造函数来说，如果输入参数不加引用，那么参数在从实参传给形参的过程中，又会进行一次拷贝构造过程，这样会导致函数的无限递归。

对于拷贝函数，如果不加引用，同样会调用拷贝构造函数，如果拷贝构造函数加了引用，那么拷贝函数的拷贝功能可以进行，只是从效率上来说，这样开销更大。

**为什么必须是const**

首先从语义上来说，拷贝的过程不该修改输入变量，所以需要有一个const保证。

另外，如果一定不加const，对于A a， A b = a；这样的拷贝没有问题。但是，对于const A a， A b = a；这样的过程，编译器则会报无法找到对应的函数的错误。

<https://www.cnblogs.com/raichen/p/4752025.html>

// 把c3的数据成员的值赋值给c1

12 // 由于c1已经事先被创建，故此处不会调用任何构造函数

13 // 只会调用 = 号运算符重载函数

14 c1 = c3;

// 调用拷贝构造函数( 有下面两种调用方式)

21 Complex c5(c2);

22 Complex c4 = c2; // 注意和 = 运算符重载区分,这里等号左边的对象不是事先已经创建，故需要调用拷贝构造函数，参数为c2

## Q：RTTI（运行时类型识别）

A： 在C++环境中，头文件含有类定义，也就是包含有关类的结构资料，但这些信息只在编译期间使用，在运行期间无法提供信息，例如类的名称、数据成员等。RTTI提供了运行时确定类对象类型的方法。RTTI提供了两个操作符：typeid、dynamic\_cast操作符

a、typeid：返回指针和应用的实际类型，即type\_info；

b、dynamic\_cast：将基类指针、引用安全转成派生类指针和引用；

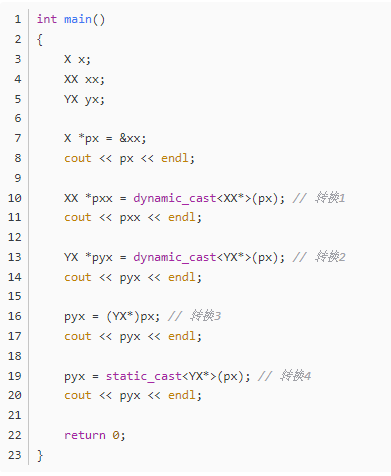
指针类型都是指向基类的

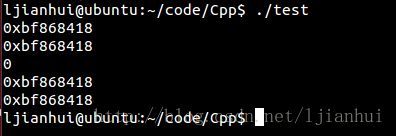
**RTTI的实现**

VS中虚函数表的-1位置存放了指向type\_info的指针。对于存在虚函数的类型，typeid和dynamic\_cast都会去查询type\_info.

<https://blog.csdn.net/gogokongyin/article/details/51178303>

<https://blog.csdn.net/ljianhui/article/details/46487951> 写的很好





px是一个基类（X）的指针，但是它指向了派生类XX的一个对象。在转换1中，转换成功，因为px指向的对象确实为XX的对象。在转换2中，转换失败，因为px指向的对象并不是一个YX对象，此时dymanic\_cast返回NULL。转换3为C风格的类型转换而转换4使用的是C++中的静态类型转换，它们均能成功转换，但是这个对象实际上并不是一个YX的对象，所以在转换3和转换4中，若继续通过指针使用该对象必然会导致错误，所以这个转换是不安全的。Q：类型转换

A：**static\_cast**

static\_cast： 1）完成基础数据类型，2）同一个继承体系中类型的转换 3）任意类型与空指针类型void\*之间的转换。

dynamic\_cast：使用多态的场景，增加了一层对真实调用对象类型的检查

用法：static\_cast < type-id > ( expression )

该运算符把expression转换为type-id类型，在编译时使用类型信息执行转换，在转换执行必要的检测（指针越界，类型检查），其操作数相对是安全的。

但没有运行时类型检查来保证转换的安全性。

**dynamic\_cast**

1.dynamic\_cast是在运行时检查的，用于在集成体系中进行安全的向下转换downcast(当然也可以向上转换，但没必要，因为可以用虚函数实现)

   即：基类指针/引用 -> 派生类指针/引用

   如果源和目标没有继承/被继承关系，编译器会报错！  
2.dynamic\_cast是4个转换中唯一的RTTI操作符，提供运行时类型检查。  
3.dynamic\_cast不是强制转换，而是带有某种”咨询“性质的，如果不能转换，返回NULL。这是强制转换做不到的。

4.源类中必须要有虚函数，保证多态，才能使用dynamic\_cast<source>(expression)

**const\_cast**

从类中移除const、volatile等特性；

**reinterpret\_cast**

将任何类型指针转换为任何其它类型指针。

**static\_cast和dynamic\_cast的比较：**

在上行转换（子类到父类）时，两者都是安全的。下行转换（父类到子类）时，dynamic\_cast会进行类型检查，如果无法转换（该类就是一个父类，不是子类），返回nullptr；但是static\_cast不会进行类型检查，依然返回一个出错的子类指针。

<https://www.cnblogs.com/carsonzhu/p/5251012.html>

## Q：const vector<int> 和vector <const int>的区别

A：const vector<int>中，const修饰的是vector，vector中的元素定义后，vector的大小及vector中的内容都不能改变。

vector<const int>中const不起作用，和vector<int>使用一致。猜测，vector<class T>没有偏特化为const的版本。

## Q：C++类成员函数内可以调用delete this吗

1. 在调用delete的时候，C++会帮我们调用这个对象的析构函数，因此member在析构函数中被置为了0

2. 调用delete只是告诉系统我们不需要这个对象的内存空间了，请求释放它，但并不会主动帮我们把这个指针置为null，它依然指向原来的内存地址

3在delete this之后，会释放掉类的对象的内存空间，因此如果不在delete this之后调用成员函数，那就没有任何问题，如果调用了成员函数，那就会出错，因为已经没有这个对象的成员了。

A： 从编译的角度，可以调用。但有以下注意事项：

a、析构函数中如果调用delete this，则进入无限递归调用，最后栈溢出；

b、在其他成员函数中调用时，delete后，不能调用虚函数，因为虚函数表已经乱了，会导致函数地址出错。另外，delete后，调用成员变量时，得到的大多是随机值。

## Q：成员函数里memset(this,0,sizeof(\*this))会发生什么

A： 类中的所有数据全部被设置为0。

a、如果有虚函数，会破坏虚函数表，后续调用出错；

b、如果类中包含new的指针，这样做会导致内存泄露；

## Q：STL迭代器型别

A：输入迭代器、输出迭代器、正向迭代器、双向迭代器、随机访问迭代器

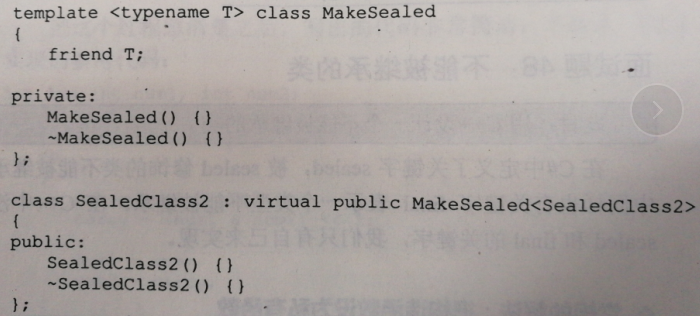
https://www.cnblogs.com/lhuan/p/5706654.html

## Q：如何设置一个类不可以被继承

A： a、使用final关键字；

b、设置构造函数私有；（类似单例模式）

c、虚拟继承，利用友元关系的不可继承性。如下图，SealedClass2是MakeSealed<SealedClass2>的友元，SealedClass2和普通类无区别。但是当SealedClass3想要继承SealedClass2时，由于友元关系不可继承，其不能初始化MakeSealed<SealedClass2>，编译出错。



<https://blog.csdn.net/lazy_tiger/article/details/2224899>

## Q：如何设置一个类只可以在堆上或者栈上创建对象？

A：**只可以在堆上：**

a、设置类的析构函数为private，设置一个public的destroy函数在堆上销毁对象。此时在栈上创建对象时，会检查析构函数的可调用性，如果不可调用，拒绝在栈上分配。这样操作有两个问题：无法继承、使用new创建对象，但是不能直接调用delete销毁对象，改成通过对象调用destroy函数，不符合常规操作习惯。

b、析构函数设置为protected（保证继承），并通过public的静态create和destroy函数实现创建和销毁实例。

**只能在栈上：**

将类的opeartor new()设置为私有函数。

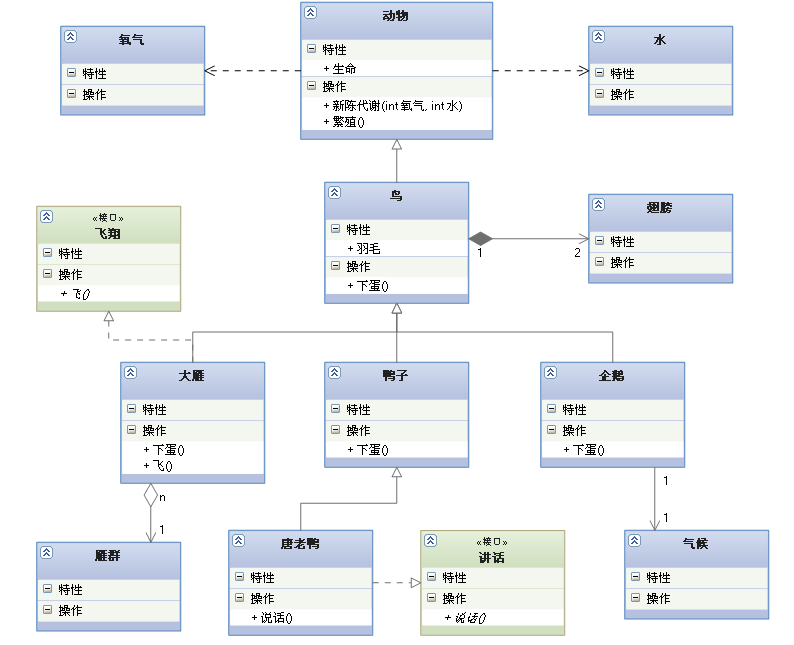
<https://blog.csdn.net/zr1076311296/article/details/52448279>

## Q：C++中成员变量的初始化顺序

A： 初始化顺序只与其在类中的声明顺序一致，与初始化列表中的顺序无关。

## Q：UML类图

A：**泛化 = 实现 > 组合 > 聚合 > 关联 > 依赖**



<http://www.uml.org.cn/oobject/201609062.asp>

## Q：POD类型

A： POD全称Plain Old Data。通俗的讲，一个类或结构体通过二进制拷贝后还能保持其数据不变，那么它就是一个POD类型。当class/struct是极简的(trivial)、属于标准布局(standard-layout)，以及他的所有非静态(non-static)成员都是POD时，会被视为POD。

对于POD类型，可以直接用memcpy的方式进行对象拷贝。

<https://blog.csdn.net/aqtata/article/details/35618709>

## Q：字节对齐

A：**原因：**

字节对齐是为了提高CPU的效率。CPU存取内存的粒度不是一个字节，而是多个字节。如果变量的内存地址和CPU读取的粒度保持一致的话，那么可以做到一次读取就将数据读出来；如果不一致，可能需要多次读取，并还需要拼凑之后才能得到变量。

**规则：**

a、自身对齐值：对于基本数据类型，自身对齐值就是其size大小；结构体或者类，自身对齐值取数据成员的对齐值的最大值。

b、指定对齐值：可以通过#pragma pack (value)设置指定对齐值；

c、类和结构体的有效对齐值取自身对齐值和指定对齐值的小值。

d、对于类和结构体中的成员，第一个位于偏移为0的位置，以后每个数据成员起始位置都必须是有效对齐值的整数倍；

e、类和结构体的最后，需要使用填充字符填充，保证整个结构体大小为有效对齐值的整数倍。

## Q：如何判断操作系统是大端还是小端？

A： 大端和小端的概念：

大端就是存储数据的时候，尾端放在高地址空间，首端放在低地址空间。

如何判断大端还是小端：

以常数1为例，32位系统值，小端情况下，从低地址到高地址依次为（01，00，00，00）；大端情况下为（00，00，00，01）；所以只需通过强制类型转换，得到1的前一个字节存的是01还是00即可，代码如下：

int num = 1; // \*((char\*)&num)获得num的最低字节，为0x00,说明是大端 为0x01,说明是小端 return \*((char\*)&num)?1:0; // 本机返回1：为大端

char类型为8位

## Q：二维数组int (\*p)[3]函数指针和指针函数

A： 函数指针是指向函数的指针变量；

eg：int (\*fun)(int x);

指针函数是返回一个指针类型的函数；

eg：int \*fun(int x);

而int (\*p)[3]是定义了一个指向有三个元素的int类型数组的指针

## Q：C++匿名命名空间

A： 一般命名空间都具有extern连接属性，即本命名空间中定义的内容可以在其它文件中访问到。为了限定变量或函数的作用域在本文件中，可以采用static关键字或者匿名命名空间。匿名命名就是定义命名空间时，不指定命名空间名称。编译器内部将给这样的命名空间生成一个唯一的名字。但是这个名字在其它文件中是无法得到的，所以就限定了命名空间的作用域在本文件内。

<https://www.cnblogs.com/youxin/p/4308364.html>

## Q：将tab设置为四个空格

A：