**一、C++基础**

**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。

项目中用到的原则：

单一职责、开放封闭、依赖倒置、接口隔离；

**Q：指针和引用的区别？**

A： 变量的地址、变量的别名；引用必须初始化，定义后不可变；引用不能用const；引用有类型检查，是类型安全的。

<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可以用于修数原始名称，不要给我生成用于链接的中间函数名。

**extern "C"**：告诉编译器，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，编译时，编译器会修饰全局变量、全局const变量，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。

**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。

**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。

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。

**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通常用于配合const使用。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。

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；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：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：**

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。

**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。

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；

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

**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，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：**

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。多个进程同时读写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。

std::unique\_ptr<int> foo;
std::unique\_ptr<int> bar;
foo = std::unique\_ptr<int>(new int (101)); // rvalue
bar = std::move(foo); // using std::move

**weak\_ptr:**

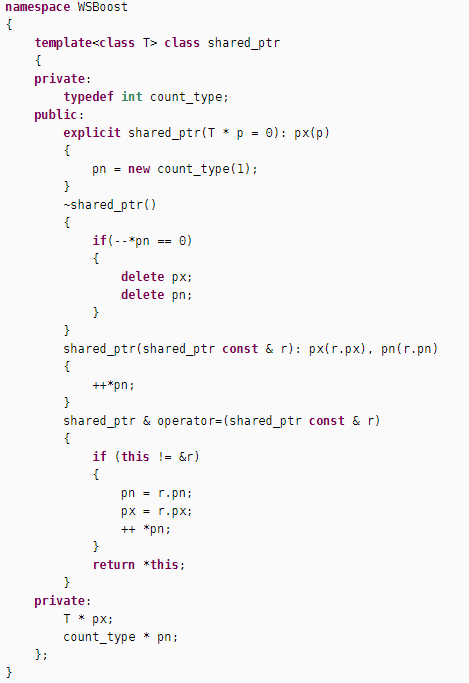
weak\_ptr是一种弱引用，可以用于解决shared\_ptr的环形引用问题。

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。weak\_ptr提供一个lock函数，用于判断引用的shared\_ptr是否有效。

<https://blog.csdn.net/leichaowen/article/details/53064294>

**Q：手写智能指针的实现**

A：



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

**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；

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**

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。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。

move函数接受一个参数，返回一个该参数的右值引用。

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。

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

class Singleton1

{

public:

static Singleton1 \*GetInstance()

{

if(nullptr == m\_instance)

{

m\_instance = new Singleton1;

}

return m\_instance;

}

static void Destory()

{

if(nullptr != m\_instance)

delete m\_instance;

}

private:

Singleton1() = default;

~Singleton1() = default;

private:

static Singleton1 \*m\_instance;

};

Singleton1 \*Singleton1::m\_instance = nullptr;

//线程安全

class Singleton2

{

private:

Singleton2() = default;

~Singleton2() = default;

public:

static Singleton2 \*GetInstance()

{

return m\_instance;

}

static void Destory()

{

if (nullptr != m\_instance)

delete m\_instance;

}

private:

static Singleton2 \*m\_instance;

};

Singleton2 \*Singleton2::m\_instance = new Singleton2;

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;

}

**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：C++中从代码到可执行文件经历了哪些过程？**

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；

汇编：生成机器代码；

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。

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： 编译生成静态链接库时，会生成静态lib。静态链接库将导出声明和实现都放在lib中，编译后所有代码都嵌入到宿主程序。

动态lib相当于一个h文件，是对实现部分（.dll文件）的导出部分的声明。编译后只是将导出声明部分编译到宿主程序中，运行时候需要相应的dll文件支持。

**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）

linux：使用valgrind工具，编译时使用-g选项，检查时，调用以下命令：

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。

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。

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： 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，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，回到本函数ebp；（清空当前函数所使用的栈）

e、基地址回退到上一个函数的基地址；

f、eip退回到上一个函数即将要执行的语句上

**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。

**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；

**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。

<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；

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。

**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

**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

**Q：声明和定义的区别**

A： a、声明是将一个符号引入到一个作用域。而定义提供了一个实体在程序中的唯一描述。

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>

**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）**

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。

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

**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、多态是通过重写虚函数表中的函数指针实现。

c、隐藏分两种：如果派生类和基类的函数名相同，参数列表不同，此时基类函数被隐藏；如果子类父类名称和参数列表相同，但是父类没有virtual，父类函数被隐藏。

**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：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。

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>

**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；

**delete**

使用=delete修饰类的成员函数时，通知编译器，不希望定义这些成员函数。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。

**Q：拷贝构造函数、拷贝函数**

A： //拷贝构造函数

classA(const classA &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

classA & operator=(const classA &a)

{

if(this == &a)

return \*this;

{

//考虑线程安全

classA 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。

**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；

**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>

**Q：类型转换**

A： **static\_cast**

可以用于基类和派生类之间的转换，也可以用于数值类型之间的转换；

**dynamic\_cast**

用于基类和派生类之间的转换；在深层继承的情况下，dynamic\_cast效率很低。（解决dynamic\_cast效率低的一个思路是，要做转换，是因为需要调用子类中新定义的函数，可以在基类中声明子类的函数，但是可以不予实现，那么就能够避免类型转换，利用多态实现函数调用）

**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。

**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**

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**

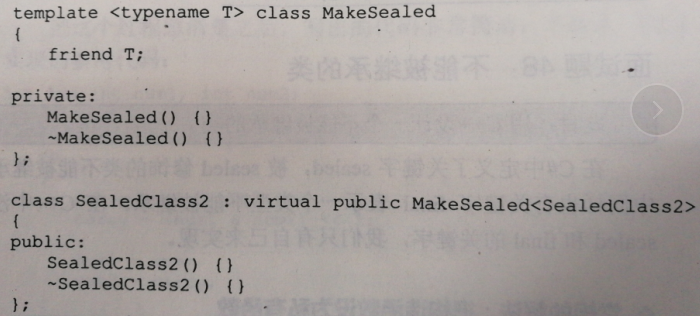
A：输入迭代器、输出迭代器、正向迭代器、双向迭代器、随机访问迭代器

**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。

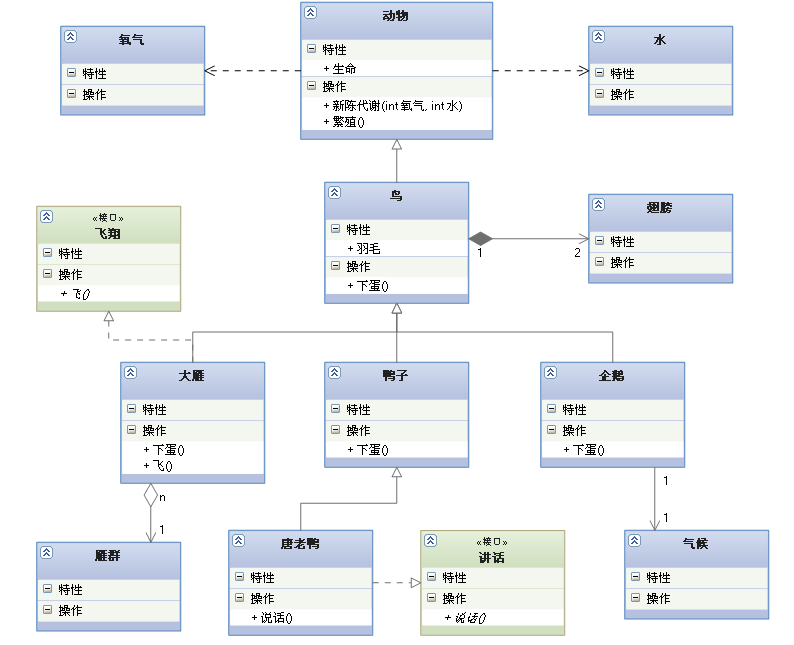
<https://blog.csdn.net/zr1076311296/article/details/52448279>

**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：



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

**Q：POD类型**

A： 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，大端情况下，从低地址到高地址依次为（01，00，00，00）；小端情况下为（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

**Q：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);

**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：