目录

[拷贝构造与赋值（重载=） 1](#_Toc8019)

[析构函数 2](#_Toc17462)

[Swap函数 2](#_Toc9756)

[移动 2](#_Toc4078)

[Const 3](#_Toc24666)

[函数指针 3](#_Toc20510)

[函数重载 4](#_Toc32172)

[可调用对象与function 4](#_Toc24226)

[类类型转换 4](#_Toc7757)

[函数继承 4](#_Toc6487)

[访问控制 5](#_Toc5387)

[引用 5](#_Toc19113)

[initializer\_list与可变模版参数 6](#_Toc8720)

[组合类 6](#_Toc7108)

[Static\_cast,dynamic\_cast,reinterpret\_cast,const\_cast 6](#_Toc13535)

[Typeid 6](#_Toc1308)

[模版类编程 6](#_Toc30504)

[Volatile 7](#_Toc6250)

[其他内容 8](#_Toc12137)

[尾置返回类型 11](#_Toc5962)

[值初始化 11](#_Toc16762)

[默认初始化 11](#_Toc9252)

[extern与static 11](#_Toc1128)

[字面值常量类constexpr变量和constexpr函数 12](#_Toc1799)

[静态成员的一些用处 12](#_Toc8277)

[Std：：forward 12](#_Toc3735)

[C语言宏定义、宏函数、内置宏与常用宏 12](#_Toc31802)

# 拷贝构造与赋值（重载=）

（调用拷贝构造函数，注意不是拷贝赋值构造函数）

1. 非引用实参
2. 非引用返回值
3. 花括号初始数组或者聚合类
4. =右边为对象或（）里为对象

赋值是一个已经存在的对象赋给另一个对象，而拷贝构造是创建对象

拷贝构造和赋值都是拷贝初始化，相应的有直接初始化

自定义赋值运算符要考虑是自身给自己赋值的情况

拷贝构造与赋值都是拷贝初始化，参数都是一个自身类型的引用，返回的一般是类对象本身引用，而直接初始化调用其他类型参数

不使用=为直接初始化，要求普通函数匹配原则（也包括拷贝构造函数），当使用=时，我们要求编译器将右侧运算对象的值拷贝到正在创建的对象中，有时会进行类型转换

综上，会使用拷贝初始化发生在使用=以及上面4种情况下。

# 析构函数

1. 析构函数不会自动析构普通指针，因为智能指针是类类型，具有自己的析构函数，所以智能指针在析构阶段会自动销毁
2. 对于动态分配的对象，当对指向他的指针使用delete函数时才会被销毁
3. 对于一个类，析构函数自身不会销毁类中其他成员，类中的其他成员是调用他自生的析构函数
4. 通常需要自定义析构函数的类也需要定义相应的构造和赋值函数，因为默认情况下构造和析构是浅拷贝，需自定义以为深拷贝，当自定义析构函数delete动态分配内存时，如果为浅拷贝就会出现意想不到的错误

# Swap函数

标准库为我们提供了函数模板std::swap()。在C++11之前，它被定义在<algorithm>头文件中；而在C++11后，他被移动到了<utility>中。

std::swap()会调用类的swap()成员函数

如：Hasptr类自定义swap,Foo中有HasPtr对像h，编写Foo类swap应该为：

Void swap(Foo &a,Foo &b)

{

Using std::swap;

Swap(a.h,b.h);//调用HasPtr的swap（这个与模版有关，参见642页）

}

# 移动

使用移动构造函数后原来的对象指针内容要置为空

只有当一个类没有任何一个拷贝控制成员时，且类的非static都是可移动的，编译器就会生成移动函数。

如果类既有拷贝构造又有移动构造，则会按照普通函数的匹配原则进行匹配，一般定义了移动操作就要定义拷贝构造函数

写引用时，一般都将&写在后面

可以将右值引用转换到常左值引用，根据函数匹配原则，Foo z (std::move(x))调用拷贝构造函数。如果一个类只有移动构造没有拷贝构造，其对象通过拷贝构造来“移动”。

使用引用限定符，在函数定义后加&表示该成员函数只能使用左值对象访问，类似const限定符，const限定符需在引用限定符之前，需要出现在声明和定义处。

和const一样引用限定符也可以区分重载版本，如果要重载，则需要所有的都加上限定符

# Const

const位于\*的左侧，则const就是用来修饰指针所指向的变量，即指针指向为常量； const位于\*的右侧，const就是修饰指针本身，即指针本身是常量。

const修饰返回值

如前所述，这样做主要为了返回值不可作为左值，从而使其被修改。

const成员函数为常成员函数, 它不改变对象的成员变量。

const对象对该类的任何非const成员函数都不能调用，因为任何非const成员函数均会有修改成员变量的企图。而非const对象可以调用const成员函数。

Const可作为重载的标志，同样的函数，后面加一个const与不加，表示重载

底层const指变量本身是常量，顶层const指指针是常量，即指针指向不能变

执行对象拷贝时有限制，常量的底层const不能赋值给非常量的底层const

一个拥有顶层 const 的形参无法和另一个没有顶层 const 的形参区分开：

使用命名的强制类型转换函数const\_cast时，只能改变运算对象的底层const

顶层const不能改变变量本身

默认情况下，const只在本文件（本编译单元中）有效，不同文件中定义的为不同的副本

# 函数指针

当我们把函数名作为值时，函数自动转换为指针，但因为编译器不会将对象的成员函数隐式转换成函数指针

因此，我们可以直接用函数名为函数指针赋值，也可以在前面加上&，我们使用函数指针时，可以不解引用指针，即前面可以不加\*

函数指针作为形参时，形参看起来是函数类型，实际上可以当成指针来使用，函数名作为实参使用，将自动转换为指向该函数的指针

const double (\*pa[3])(const double \*,int)={f1,f2,f3};注意，\*pa[3]要加括号否则\*会和前面的double组合，就不是函数指针了

运算符[]的优先级高于\*，因此\*pa[3]表明pa是一个包含三个指针的数组，(\*pd)[3] 表示指向包含3个元素的数组的指针

C++11中可以使用 auto p=函数名 来定义函数指针

在定义函数指针时，要灵活使用auto以及decltype

函数指针形参看起来是函数类型，实际上可以当成指针使用，可以把函数作为实参使用，他会自动转换为指针

**成员函数指针：**

void (SomeClass::\*)(int, double) const

SomeClass::\*)：指定这是一个成员函数指针，其中 SomeClass 是类名。

# 函数重载

重载<<,>>第一个参数是非常量ostream的引用，我们无法直接复制一个ostream，输入输出会改变ostream，只能为非成员函数，

逻辑运算符和关系运算符通常返回bool，算术运算符返回一个类类型，赋值运算符和复合运算符以及++，---返回对象的一个引用。

要想类可以{}进行初始化，需要重载=且（）参数为initializer\_list<type>,

如果类重载了函数调用运算符即()，就可以像调用函数一样调用类对象，此时称为函数对象

# 可调用对象与function

可调用对象：函数，函数指针，函数对象，bind

# 类类型转换

类型转换运算符：

Operator type() ;前面加expilct时要显示转换，不过当表达式被用作条件时（在if判别式，for语句判别，条件运算符？ ：，逻辑！等），编译器会自动显示转换

最好不要创建多个参数都是算数类型的转换构造函数，或者多个转换目标都是算术类型的类型转换符

还有一种方法是使用转换构造函数

只允许一步类类型转换

# 函数继承

对于函数中static成员，不论继承多少次，在内存中只存在一个实例。Override来说明派生类中的虚函数，减少错误。Final可以阻值继承，这两个关键字都放在最后。

使用虚函数一定要记住对象要是指针或者是引用

在继承中使用using加作用域符使用继承来的成员变量，也可改变其访问控制类型

只要派生类中有与基类相同的函数名，不管有没有加virtual，不适用作用域限定符的话，通过子类对象都不能直接访问，其中情况包括覆盖和重写（实现虚函数）

当多个基类有同名函数时，使用using baseName::funcNmae可以消除派生类中的二义性，当派生类中定义同名但参数不同的函数时，通过上面的方法基类函数不会被隐藏，2个重载函数可以并存在派生类的作用域中。

# 访问控制

Protected只允许派生类以及友元和成员函数访问，类对象无法直接访问，且派生类以及友元和成员函数只能访问派生类中从基类继承下来的成员，不能直接访问基类中的protected成员。

1. 只有当D公有继承B时，用户代码才能使用派生类向基类到转换；如果D继承B的方式时受保护的或者私有的，则用户代码不能使用该转换。
2. 不论D以什么方式继承B，D的成员函数和友员函数中都能使用派生类向基类的转换；派生类向其直接基类的类型转换对于派生类的成员和友员而言永远是可访问的。
3. 如果D继承B 的方式是公有的或者受保护的，则D的派生类的成员和友员可以使用D向B 的类型转换；反之，如果D继承B 的方式时私有的，则不能使用

# 引用

引用传参的时候，返回参数出了作用域不能被销毁。不要引用局部变量，因为局部变量返回时会重置

引用和指针的区别:

1. 引用在概念上是定义一个变量的别名，指针是存储一个变量的地址
2. 引用在定义时必须初始化，指针没有要求
3. 引用在初始化时引用一个实体后，就不能再引用其他实体，而指针可以在任何时候指向任何一个同类型实体
4. 没有NULL引用，但有NULL指针
5. 在sizeof中含义不同：引用结果为引用类型的大小，但指针始终是地址空间所占字节个数(32位平台下占4个字节)
6. 引用自加即引用的实体增加1，指针自加即指针向后偏移一个类型的大小
7. 有多级指针，但是没有多级引用
8. 访问实体方式不同，指针需要显式解引用，引用编译器自己处理
9. 引用比指针使用起来相对更安全
10. 引用本身不是对象，不能定义引用的引用，而指针本身是一个对象

# initializer\_list与可变模版参数

也就是说initializer\_list对象只能用大括号{}初始化。

拷贝一个initializer\_list对象并不会拷贝里面的元素。其实只是引用而已。而且里面的元素全部都是const的。

# 组合类

函数会隐式调用拷贝构造函数为各个内嵌对象进行初始化。如果组合类编写拷贝构造函数，则需要为内嵌成员对象的拷贝构造函数传递参数。

C::C( C & c1):b(c1.b) {......}

# Static\_cast,dynamic\_cast,reinterpret\_cast,const\_cast

Const\_cast去除底层const

# Typeid

当typeid作用于多态类型时，进行的才是运行时的类型识别，指针类型本身不是多态类型

# 模版类编程

被特化的模版，像非特化的模版和成员函数一样，无论是否被调用，只要是定义了，相关代码都会被生成。因此要写在cpp文件中，如果写在h文件中，在连接时可能会产生2段相同代码，而报错。

在类模版被全特化时，成员函数的类外实现不需要加 template<>。

类中的虚函数不能用类型参数模板。

当模板形参T表示的是模板参数时（常出现于2个模板类，其中一个组合另一个），需要使用template<template<typename X> typename T>

函数名后加<>,表示优先调用函数模版，而不是普通函数,类模版才能进行偏特化

模版可以使用特化，停止递归，我们只能部分特例化类模版而不能部分特例化函数模板。

类模板可以针对某一些特性场景进行部分特化，比如我们针对模板参数是指针进行偏特化：

注意类声明与全特化的不同：（这个例子是偏特化为指针）

template<typename T>

class Stack<T\*> {};

使用：

Stack<int\*> ptrStack; // stack of pointers (special implementation)

ptrStack.push(new int{42});



# Volatile

编译器每次读取 volatile 定义的变量时，都从内存地址处重新取值。编译器有时候会从寄存器处取变量的值，而不是每次都从内存中取。因为编译器认为变量并没有变化，所以认为寄存器里的值是最新的，另外，通常来说，访问寄存器比访问内存要快很多，编译器通常为了效率，可能会读取寄存器中的变量。但是，变量在内存中的值可能会被其它元素修改，比如：硬件或其它线程等。

# 其他内容

C++程序基本都是分离式编译机制，允许我们将程序分割成若干个文件，每个文件可被独立编译。

C++规定，一个指针变量加/减一个整数是将该指针变量的原值(是一个地址)和它指向的变量所占用的内存单元字节数相加或相减。如p+i代表这样的地址计算：p+i\*d，d为p所指向的变量单元所占用的字节数。这样才能保证p+i指向p下面的第i个元素。

标准库allocator类定义在头文件memory中，它帮助我们将内存分配和对象构造分离开来。它提供一种类型感知的内存分配方法，它分配的内存是原始的、未构造的。类似vector，allocator是一个模板。为了定义一个allocator对象，我们必须指明这个allocator可以分配的对象类型。标准库还为allocator类定义了两个伴随算法，可以在未初始化内存中创建对象。它们都定义在头文件memory中

在定义了虚析构函数的情况下，如果用delete运算符删除单个对象时，编译器总是能保证析构函数的正确调用。但是如果用delete[]删除数组对象，并且定义的数组指针是基类的指针时则析构函数不一定能够被正确调用

Lambada表达式的[]是捕获值列表能捕获的值是所有在此作用域可以访问的值，包括这个作用域里面的临时变量，类的可访问成员，全局变量。

不能直接返回函数中定义的数组，会被释放掉,因为它是局部变量，存放在栈中，可以用new动态开辟，或者用一个数组作为参数传入

默认形参只能定义一次，且是静态绑定的，因此继承中基类的默认形参不会给派生类，当使用继承的构造函数时，有默认形参的构造函数会在派生类中产生多个构造函数

类中的静态变量和常量都应该在类外定义，虽然c++允许int和枚举型在类内定义，但当程序中要对其进行取地址操作时，要在类外定义，为其分配空间

使用static表示变量只可以本文件使用，与extern作用相反

Void\*可以显示转换为其他任何类型的指针

使用stable\_sort可以保持在同等情况下相对顺序不变

Sort自定义排序：

1. 实现比较类：重载（）: bool operator() (const Student& s1, const Student& s2)，并在sort的第三个参数中使用函数对象：即cmp()；
2. 实现比较函数，返回为bool，参数为2个类型数据
3. 重载比较运算符<

使用枚举类型，编译阶段就能推导出值。

不允许拷贝数组，使用数组时会将其转换成指针，decltype（数组）并不会将数组转换为指针

对于只有声明，没有定义的类，称为不完全类型，不完全类型不能定义，只能使用其指针或或者引用，类的静态成员可以使用不完全类型进行定义。

带有默认函数时，默认函数只能在声明处，不能再次出现在定义处

const的内置类型变量和引用是无法默认初始化的，因此如果类内有这些成员，则合成的拷贝运算符是删除的

静态成员属于类作用域，但不属于类对象，它的生命周期和普通的静态变量一样，程序运行时进行分配内存和初始化，程序结束时则被释放。所以不能在类的构造函数中进行初始化。

noexcept：将移动构造函数和移动赋值运算符设置为noexcept是必须的，它通知标准库我们的构造函数不会抛出任何异常，由于移动操作“窃取”资源，它通常不分配任何资源，因此通常不会抛出任何异常，但我们应该将此事通知标准库，否则它会认为移动我们的类对象时可能会抛出异常，并为处理这种可能性而做一些额外的工作，我们在一个函数参数列表后制定noexcept，在一个构造函数中，noexcept出现在参数列表和初始化列表开始的冒号之间。我们必须在类头文件的声明中和定义中(如果定义在类外的话)都指定noexcept。

make\_move\_iterator移动迭代器：

new是一个关键字，和sizeof一样，我们无法修改其具体功能。new主要做三件事：分配空间、初始化对象、返回指针。调用operator new分配空间。

operator new重载可以放在全局中，也可以放到类内部。当编译器发现有new关键字，就会现在类和其基类中寻找operator new，找不到就在全局中找，再找不到就用默认的。perator new的主要作用就是分配空间，初始化对象的工作是new关键字的。虽然operator delete重载和operator new相同，但一般不会重载operator delete。

根本原因是重载后的delete不可手动调用。

虽然operator delete重载和operator new相同，但一般不会重载operator delete。

根本原因是重载后的delete不可手动调用。

空悬指针

定义于任何函数体之外的类变量会先进行零初始化再执行默认初始化，定义在函数体内部的类变量会直接执行默认初始化。

如果定义数组时提供了初始值列表，那么未定义的元素若是内置类型或者有合成的默认构造则会先进行零初始化，如果元素是类类型，再执行默认构造函数

如果定义数组时未提供初始化列表，则每个元素执行默认初始化

使用::a,表示使用的是全局变量a

如果你非要返回一个局部变量的地址，加上static，

Char\* s=”hello”,对于返回s，我们返回的是这个字面值的首地址，它在只读区，new的数据在函数退出时也不会自动释放，因为它在堆里，不在栈里

如果类中值有初始值，则默认构造函数使用初始值进行构造

引用类型或const的类型,不管是不是int还是结构体或者类,都必须要显式初始化.

并且编译器不会为具有这些类型的class,struct,合成[默认构造函数](https://www.baidu.com/s?wd=%E9%BB%98%E8%AE%A4%E6%9E%84%E9%80%A0%E5%87%BD%E6%95%B0&tn=SE_PcZhidaonwhc_ngpagmjz&rsv_dl=gh_pc_zhidao).

合成默认构造函数总是不会初始化类的内置类型及复合类型的数据成员（有一个初始化就行）。

 分清楚默认构造函数被程序需要与被编译器需要，只有被编译器需要的默认构造函数，编译器才会合成它。

当一个类派生自一个含有默认构造函数的基类时，该类也符合编译器需要合成默认构造函数的条件。编译器合成的默认构造函数将根据基类声明顺序调用上层的基类默认构造函数。同样的道理，如果设计者定义了多个构造函数，编译器将不会重新定义一个合成默认构造函数，而是把合成默认构造函数的内容插入到每一个构造函数中去。

隐式销毁内置指针不会delete它所指的对象（类中有其他类的指针）

当指向一个对象的引用或者指针离开作用域时，析构函数不会执行

自定义析构函数的类也需要自定义拷贝构造和拷贝赋值函数

如果类中自定义了new出来指针类型的数据成员，需要在析构函数中使用delete删除

自定义拷贝构造和赋值基本上是同时出现的

对于使用拷贝赋值，通常要先清除掉自己之前的数据，记住要判断赋值左右两边是否是同一个对象

我们要使用移动的另一个原因是有些类包含一些不能被共享的资源，不能拷贝只能移动

拷贝/移动构造函数，除了第一个类类型的引用外，所有的额外参数都需要默认实参

如果一个类定义了一个移动构造函数/移动[运算符](https://so.csdn.net/so/search?q=%E8%BF%90%E7%AE%97%E7%AC%A6&spm=1001.2101.3001.7020" \t "https://blog.csdn.net/zhoushiqi_uestc/article/details/_blank)，则该类的合成拷贝构造函数和拷贝赋值运算符会被定义为删除的，也就是说定义了一个移动操作的类也要定义自己的拷贝操作，否则拷贝操作会被默认定义为删除的。

三种引用（左值引用 T & ； const引用 const T& ; 右值引用 T&&）

引用限定符

拷贝构造函数，任何额外的参数都要有默认的实参

对于使用移动构造和移动赋值，原对象指针类型数据都要置为空

如果类定义了拷贝构造、拷贝赋值，析构，将不会合成移动构造、移动赋值

对于拷贝构造，拷贝赋值，移动构造，我们只考虑非static成员能不能进行相应地操作。

如果一个类没有定义任意一个拷贝控制函数且非static成员都可以进行移动，才会合成移动构造或移动赋值运算符

析构函数被定义为删除或者不可访问，合成移动构造函数/拷贝构造，赋值构造也是删除的

   在函数调用时，用指针或者引用做参数，表示把变量的地址传递给子函数，但是子函数只能修改指针所指变量的值，并不能修改指针的指向。如果想用修改指针的指向，就要用指针的指针，或者指针的引用。

只能在发生异常时也能正确释放内存，如果是在new和delete之间发生异常，内存不能被释放

Auto会忽略顶层const，如果要指定顶层const，要在前面加const

Decltype不会忽略顶层这些内容，decltype变量加上（），编译器会把他当成表达式，变量是可以作为左值的，这样得到的就是引用类型，decltype（（variable））结果永远是引用，而decltype（variable）就是器本来的东西。

顶层const形参不能区分函数重载，底层可以

编译时，名字查找优于类型查找

内联函数需要再头文件中声明和定义。普通函数是整个工程可见的，可以跨文件链接，而 inline 函数仅仅当前文件可见，不可以跨文件链接。

当我们把函数作为一个值时，函数会自动转换为指针，在使用函数指针调用函数时不需要解引用

静态成员可以不完全类型

类中动态分配的成员要在析构函数中delete，否则他不会自动删除

自定义析构函数一般也需要自定义拷贝赋值和拷贝构造函数

拷贝赋值要注意考虑自己给自己赋值的情况

有const或引用成员，默认赋值是删除的

有const或者引用成员，且未使用类内初始化，默认构造是删除的

使用移动操作后要注意原来的对象是无害的，对于指针来说，一般是把指针置为空，集合清空。

一般不使用const T&&（常右值）

标准库定义的函数对象

Bind，functional

对于类类型，编译器还会查找类所属空间，形参是类类型，查找函数时也是这样

typeid type\_info

Using声明和指示

因为 std::cout 和 std::ofstream以及std::stringstream 都是 std::ostream 类型的对象，而且都支持相同的输出操作符 <<。

stock1 = Stock("wuhu", 10, 50.0);

上述代码右边指的是构造函数产生了一个新的临时的对象，然后将该临时对象的内容赋值给了stock1

因此如果可以初始化尽量不要赋值，提高效率。因此我们就有c++11的列表初始化。

# 尾置返回类型

Auto func(arglist)->int(\*)[10];

# 值初始化

顾名思义，就是用数值初始化变量。如果没有给定一个初始值，就会根据变量或类对象的类型提供一个初始值。对于int类型其值初始化后的值为0。

# 默认初始化

如果定义变量时没有指定初值，则变量被默认初始化。其初始值和变量的类型以及变量定义的位置相关。默认初始化类对象和默认初始化内置类型变量有所不同。

extern与static

C++程序基本都是分离式编译机制，允许我们将程序分割成若干个文件，每个文件可被独立编译。但是这样需要共享代码时，就有点麻烦。因为各个文件中的全局变量是互相不透明的，可见域限制在文件内部。而extern可以解决这个问题

在声明变量的前面加extern，相当于各个文件可见的全局变量，在内存中只有一份副本（在定义时分配内存），static虽然在内存中也只有一份副本，但是，static只能是本对象使用它。如果在头文件中定义变量，头文件又同时多个单元包含，则会出错，同时我们可以使用extern表示要使用别的模块的变量。我们可以在extern声明时，进行定义，例如extern const int buf=1024;//让const变量可以被其他单元可见,实现const变量共享，别的单元使用时，使用extern const int buf, cout<<buf;

如果函数的声明中带有关键字extern，仅仅是暗示这个函数可能在别的源文件里定义，没有其它作用，c++11 非const变量默认为extern

具体参见：<https://cnblogs.com/yc_sunniwell/archive/2010/07/14/1777431.html>

　如果是static修饰的全局变量，且实现的函数写在头文件（h）中，在其他文件也可以访问将static全局变量写在头文件中，所有文件的头文件的操作都会共享这个变量。

​ 　　但如果是在源文件（cpp）中去操作这个静态全局变量，则这个静态全局变量只能在当前文件有效，但是在另外一个文件访问此静态变量，会是该变量初始的默认值，不会是其他文件中修改的值，虽然它们有相同的初始内容，但是存储的物理地址并不一样。

静态全局变量在声明它的整个文件都是可见的，而在文件之外是不可见的；？？？

# 字面值常量类constexpr变量和constexpr函数

一个字面值常量类的构造函数必须时constexpr,由于constexpr函数必须有且只有一条return，所以构造函数的函数体内为空，constexpr函数是指指能用于常量表达式的函数。该函数要遵循规定：函数的返回类型及所有形参的类型都得是字面值类型（声明constexpr变量时用到的类型），并且函数体中必须有且只有一条return语句。

# 静态成员的一些用处

静态成员可以成为默认参数，静态成员可以是不完全类型的类类型，而非静态是需要指针或者是引用

# Std：：forward

# C语言宏定义、宏函数、内置宏与常用宏

# C++11 继承构造

我们通过 using Base::Base 把基类构造函数继承到派生类中，不再需要书写多个派生类构造函数来完成基类的初始化。

为避免继承构造函数冲突，可以通过显示定义来阻止隐式生成的继承构造函数。

# 委托构造