# 面向对象

面向对象程序设计（Object-oriented programming，OOP）是种具有对象概念的程序编程典范，同时也是一种程序开发的抽象方针。

面向对象特征

面向对象三大特征 —— 封装、继承、多态

## 封装

数据和代码捆绑在一起，避免外界干扰和不确定性访问。

封装，也就是**把客观事物封装成抽象的类**，并且类可以把自己的数据和方法只让可信的类或者对象操作，对不可信的进行信息隐藏，例如：将公共的数据或方法使用public修饰，而不希望被访问的数据或方法采用private修饰。

把客观事物封装成抽象的类，并且类可以把自己的数据和方法只让可信的类或者对象操作，对不可信的进行信息隐藏。关键字：public, protected, private。不写默认为 private。

* public 成员：可以被任意实体访问
* protected 成员：只允许被子类及本类的成员函数访问
* private 成员:只允许被本类的成员函数、友元类、友元函数访问

## 继承

让某种类型对象获得另一个类型对象的属性和方法。

它可以使用现有类的所有功能，并在无需重新编写原来的类的情况下对这些功能进行扩展

### 常见的继承有三种方式

实现继承：指使用基类的属性和方法而无需额外编码的能力

接口继承：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

可视继承：指子窗体（类）使用基窗体（类）的外观和实现代码的能力（C++里好像不怎么用）

例如，将人定义为一个抽象类，拥有姓名、性别、年龄等公共属性，吃饭、睡觉、走路等公共方法，在定义一个具体的人时，就可以继承这个抽象类，既保留了公共属性和方法，也可以在此基础上扩展跳舞、唱歌等特有方法

### 基类（父类）——> 派生类（子类）

1) 类与类之间的关系

has-A包含关系，用以描述一个类由多个部件类构成，实现has-A关系用类的成员属性表示，即一个类的成员属性是另一个已经定义好的类；

use-A，一个类使用另一个类，通过类之间的成员函数相互联系，定义友元或者通过传递参数的方式来实现；

is-A，继承关系，关系具有传递性；

2) 继承的相关概念

所谓的继承就是一个类继承了另一个类的属性和方法，这个新的类包含了上一个类的属性和方法，被称为子类或者派生类，被继承的类称为父类或者基类；

3) 继承的特点

子类拥有父类的所有属性和方法，子类可以拥有父类没有的属性和方法，子类对象可以当做父类对象使用；

4) 继承中的访问控制

public、protected、private

5) 继承中的构造和析构函数

6) 继承中的兼容性原则

## 多态

同一事物表现出不同事物的能力，即向不同对象发送同一消息，不同的对象在接收时会产生不同的行为**（重载实现编译时多态，虚函数实现运行时多态）**。

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。**简单一句话：允许将子类类型的指针赋值给父类类型的指针**

实现多态有二种方式：覆盖（override），重载（overload）。覆盖：是指子类重新定义父类的虚函数的做法。重载：是指允许存在多个同名函数，而这些函数的参数表不同（或许参数个数不同，或许参数类型不同，或许两者都不同）。例如：基类是一个抽象对象——人，那教师、运动员也是人，而使用这个抽象对象既可以表示教师、也可以表示运动员。

多态，即多种状态（形态）。简单来说，我们可以将多态定义为消息以多种形式显示的能力。

多态是以封装和继承为基础的。

### C++ 多态分类及实现

* 重载多态（Ad-hoc Polymorphism，编译期）：函数重载、运算符重载
* 子类型多态（Subtype Polymorphism，运行期）：虚函数
* 参数多态性（Parametric Polymorphism，编译期）：类模板、函数模板
* 强制多态（Coercion Polymorphism，编译期/运行期）：基本类型转换、自定义类型转换

#### 静态多态（编译期/早绑定）

函数重载

#### 动态多态（运行期期/晚绑定）

虚函数：用 virtual 修饰成员函数，使其成为虚函数

* 普通函数（非类成员函数）不能是虚函数
* 静态函数（static）不能是虚函数
* 构造函数不能是虚函数（因为在调用构造函数时，虚表指针并没有在对象的内存空间中，必须要构造函数调用完成后才会形成虚表指针）
* 内联函数不能是表现多态性时的虚函数。

### 基类的虚函数表存放在内存的什么区

### 虚表指针vptr的初始化时间

首先整理一下虚函数表的特征：

虚函数表是全局共享的元素，即全局仅有一个，在编译时就构造完成

虚函数表类似一个数组，类对象中存储vptr指针，指向虚函数表，即虚函数表不是函数，不是程序代码，不可能存储在代码段。

虚函数表存储虚函数的地址,即虚函数表的元素是指向类成员函数的指针,而类中虚函数的个数在编译时期可以确定，即虚函数表的大小可以确定,即大小是在编译时期确定的，不必动态分配内存空间存储虚函数表，所以不在堆中

根据以上特征，虚函数表类似于类中静态成员变量.静态成员变量也是全局共享，大小确定，因此最有可能存在全局数据区，测试结果显示：

虚函数表vtable在Linux/Unix中存放在可执行文件的只读数据段中(rodata)，这与微软的编译器将虚函数表存放在常量段存在一些差别。

由于虚表指针vptr跟虚函数密不可分，对于有虚函数或者继承于拥有虚函数的基类，对该类进行实例化时，在构造函数执行时会对虚表指针进行初始化，并且存在对象内存布局的最前面。

《虚函数表存放在哪里》：

https://blog.csdn.net/u013270326/article/details/82830656

一般分为五个区域：栈区、堆区、函数区（存放函数体等二进制代码）、全局静态区、常量区

C++中虚函数表位于只读数据段（.rodata），也就是C++内存模型中的常量区；而虚函数则位于代码段（.text），也就是C++内存模型中的代码区。

## 构造函数为什么不能为虚函数

(超重要)构造函数为什么不能为虚函数？析构函数为什么要虚函数？

**1、 从存储空间角度，**虚函数相应一个指向vtable虚函数表的指针，这大家都知道，但是这个指向vtable的指针事实上是存储在对象的内存空间的。

问题出来了，假设构造函数是虚的，就须要通过 vtable来调用，但是对象还没有实例化，也就是内存空间还没有，怎么找vtable呢？所以构造函数不能是虚函数。

**2、 从使用角度，**虚函数主要用于在信息不全的情况下，能使重载的函数得到相应的调用。

构造函数本身就是要初始化实例，那使用虚函数也没有实际意义呀。

所以构造函数没有必要是虚函数。虚函数的作用在于通过父类的指针或者引用来调用它的时候可以变成调用子类的那个成员函数。而构造函数是在创建对象时自己主动调用的，不可能通过父类的指针或者引用去调用，因此也就规定构造函数不能是虚函数。

**3、构造函数不须要是虚函数，也不同意是虚函数，**由于创建一个对象时我们总是要明白指定对象的类型，虽然我们可能通过实验室的基类的指针或引用去訪问它但析构却不一定，我们往往通过基类的指针来销毁对象。这时候假设析构函数不是虚函数，就不能正确识别对象类型从而不能正确调用析构函数。

**4、从实现上看，**vbtl在构造函数调用后才建立，因而构造函数不可能成为虚函数从实际含义上看，在调用构造函数时还不能确定对象的真实类型（由于子类会调父类的构造函数）；并且构造函数的作用是提供初始化，在对象生命期仅仅运行一次，不是对象的动态行为，也没有必要成为虚函数。

**5、当一个构造函数被调用时，它做的首要的事情之中的一个是初始化它的VPTR。**

因此，它仅仅能知道它是“当前”类的，而全然忽视这个对象后面是否还有继承者。当编译器为这个构造函数产生代码时，它是为这个类的构造函数产生代码——既不是为基类，也不是为它的派生类（由于类不知道谁继承它）。所以它使用的VPTR必须是对于这个类的VTABLE。

并且，仅仅要它是最后的构造函数调用，那么在这个对象的生命期内，VPTR将保持被初始化为指向这个VTABLE, 但假设接着另一个更晚派生的构造函数被调用，这个构造函数又将设置VPTR指向它的 VTABLE，等.直到最后的构造函数结束。

VPTR的状态是由被最后调用的构造函数确定的。这就是为什么构造函数调用是从基类到更加派生类顺序的还有一个理由。可是，当这一系列构造函数调用正发生时，每一个构造函数都已经设置VPTR指向它自己的VTABLE。假设函数调用使用虚机制，它将仅仅产生通过它自己的VTABLE的调用，而不是最后的VTABLE（全部构造函数被调用后才会有最后的VTABLE）。

因为构造函数本来就是为了明确初始化对象成员才产生的，然而virtual function主要是为了再不完全了解细节的情况下也能正确处理对象。另外，virtual函数是在不同类型的对象产生不同的动作，现在对象还没有产生，如何使用virtual函数来完成你想完成的动作。

直接的讲，C++中基类采用virtual虚析构函数是**为了防止内存泄漏。**

具体地说，如果派生类中申请了内存空间，并在其析构函数中对这些内存空间进行释放。假设基类中采用的是非虚析构函数，当删除基类指针指向的派生类对象时就不会触发动态绑定，因而只会调用基类的析构函数，而不会调用派生类的析构函数。那么在这种情况下，派生类中申请的空间就得不到释放从而产生内存泄漏。

所以，为了防止这种情况的发生，C++中基类的析构函数应采用virtual虚析构函数。

## 虚析构函数

* 虚析构函数是为了解决基类的指针指向派生类对象，并用基类的指针删除派生类对象。
* 如果有虚析构函数，则delete释放内存时，先调用子类析构函数，再调用基类析构函数，防止内存泄漏。

## 构造函数和析构函数可以调用虚函数吗

简要结论：

从语法上讲，调用完全没有问题。

但是从效果上看，往往不能达到需要的目的。

《Effective C++》的解释是：  
 派生类对象构造期间进入基类的构造函数时，对象类型变成了基类类型，而不是派生类类型。同样，进入基类析构函数时，对象也是基类类型。

#include<iostream>  
using namespace std;  
  
class Base  
{  
public:  
    Base()  
    {  
       Function();  
    }  
  
    virtual void Function()  
    {  
        cout << "Base::Fuction" << endl;  
    }  
    ~Base()  
    {  
        Function();  
    }  
};  
  
class A : public Base  
{  
public:  
    A()  
    {  
      Function();  
    }  
  
    virtual void Function()  
    {  
        cout << "A::Function" << endl;  
    }  
    ~A()  
    {  
        Function();  
    }  
};  
  
int main()  
{  
    Base\* a = new Base;  
    delete a;  
    cout << "-------------------------" <<endl;  
    Base\* b = new A;//语句1  
    delete b;  
}  
//输出结果  
//Base::Fuction  
//Base::Fuction  
//-------------------------  
//Base::Fuction  
//A::Function  
//Base::Fuction

语句1讲道理应该体现多态性，执行类A中的构造和析构函数，从实验结果来看，语句1并没有体现。

执行流程是先构造基类，所以先调用基类的构造函数，构造完成再执行A自己的构造函数，析构时也是调用基类的析构函数，也就是说构造和析构中调用虚函数并不能达到目的，应该避免。

1) 在C++中，提倡不在构造函数和析构函数中调用虚函数；

2) 构造函数和析构函数调用虚函数时都不使用动态联编，如果在构造函数或析构函数中调用虚函数，则运行的是为构造函数或析构函数自身类型定义的版本；

3) 因为父类对象会在子类之前进行构造，此时子类部分的数据成员还未初始化，因此调用子类的虚函数时不安全的，故而C++不会进行动态联编；

4) 析构函数是用来销毁一个对象的，在销毁一个对象时，先调用子类的析构函数，然后再调用基类的析构函数。所以在调用基类的析构函数时，派生类对象的数据成员已经销毁，这个时候再调用子类的虚函数没有任何意义。

## 虚函数、纯虚函数

* 类里如果声明了虚函数，这个函数是实现的，哪怕是空实现，它的作用就是为了能让这个函数在它的子类里面可以被覆盖（override），这样的话，编译器就可以使用后期绑定来达到多态了。纯虚函数只是一个接口，是个函数的声明而已，它要留到子类里去实现。
* 虚函数在子类里面可以不重写；但纯虚函数必须在子类实现才可以实例化子类。虚函数的类用于 “实作继承”，继承接口的同时也继承了父类的实现。
* 纯虚函数关注的是接口的统一性，实现由子类完成。
* 带纯虚函数的类叫抽象类，这种类不能直接生成对象，而只有被继承，并重写其虚函数后，才能使用。抽象类被继承后，子类可以继续是抽象类，也可以是普通类。
* 虚基类是虚继承中的基类，具体见下文虚继承。

### 纯虚函数

* 纯虚函数是一种特殊的虚函数，在基类中不能对虚函数给出有意义的实现，而把它声明为纯虚函数，它的实现留给该基类的派生类去做。
* virtual int A() = 0;

## 虚函数指针、虚函数表

* 虚函数指针：在含有虚函数类的对象中，指向虚函数表，在运行时确定。
* 虚函数表：在程序只读数据段（.rodata section，见：目标文件存储结构），存放虚函数指针，如果派生类实现了基类的某个虚函数，则在虚表中覆盖原本基类的那个虚函数指针，在编译时根据类的声明创建。

## override

当在父类中使用了虚函数时候，你可能需要在某个子类中对这个虚函数进行重写，以下方法都可以：

class A  
{  
    virtual void foo();  
}  
class B : public A  
{  
    void foo(); //OK  
    virtual void foo(); // OK  
    void foo() override; //OK  
}

如果不使用override，当你手一抖，将foo()写成了foo()会怎么样呢？结果是编译器并不会报错，因为它并不知道你的目的是重写虚函数，而是把它当成了新的函数。如果这个虚函数很重要的话，那就会对整个程序不利。所以，override的作用就出来了，它指定了子类的这个虚函数是重写的父类的，如果你名字不小心打错了的话，编译器是不会编译通过的：

class A  
{  
    virtual void foo();  
};  
class B : public A  
{  
    virtual void f00(); //OK，这个函数是B新增的，不是继承的  
    virtual void f0o() override; //Error, 加了override之后，这个函数一定是继承自A的，A找不到就报错  
};

## 虚继承

* 虚继承用于解决多继承条件下的菱形继承问题（浪费存储空间、存在二义性）。
* 底层实现原理与编译器相关，一般通过虚基类指针和虚基类表实现，每个虚继承的子类都有一个虚基类指针（占用一个指针的存储空间，4字节）和虚基类表（不占用类对象的存储空间）（需要强调的是，虚基类依旧会在子类里面存在拷贝，只是仅仅最多存在一份而已，并不是不在子类里面了）；当虚继承的子类被当做父类继承时，虚基类指针也会被继承。
* 实际上，vbptr 指的是虚基类表指针（virtual base table pointer），该指针指向了一个虚基类表（virtual table），虚表中记录了虚基类与本类的偏移地址；通过偏移地址，这样就找到了虚基类成员，而虚继承也不用像普通多继承那样维持着公共基类（虚基类）的两份同样的拷贝，节省了存储空间。

## 虚继承、虚函数

都利用了虚指针（均占用类的存储空间）和虚表（均不占用类的存储空间）

* 虚基类依旧存在继承类中，只占用存储空间
* 虚基类表存储的是虚基类相对直接继承类的偏移
* 虚函数不占用存储空间
* 虚函数表存储的是虚函数地址

## 模板类、成员模板、虚函数

* 模板类中可以使用虚函数
* 一个类（无论是普通类还是类模板）的成员模板（本身是模板的成员函数）不能是虚函数

## 抽象类、接口类、聚合类

### 抽象类

* 含有纯虚函数的类

### 接口类

* 仅含有纯虚函数的抽象类

### 聚合类

用户可以直接访问其成员，并且具有特殊的初始化语法形式。

* 所有成员都是 public
* 没有定义任何构造函数
* 没有类内初始化
* 没有基类，也没有 virtual 函数

# 引用

## 左值、右值

* 概念1：

左值：可以放到等号左边的东西叫左值。

右值：不可以放到等号左边的东西就叫右值。

* 概念2：

左值：**可以取地址并且有名字**的东西就是左值。

右值：不能取地址的没有名字的东西就是右值。

举例：

int a = b + c; //a + b返回一个临时变量

a是左值，有变量名，可以取地址，也可以放到等号左边, 表达式b+c的返回值是右值，没有名字且不能取地址，&(b+c)不能通过编译，而且也不能放到等号左边。

int a = 4; // a是左值，4作为普通字面量是右值

左值一般有：

1. 函数名和变量名
2. 返回左值引用的函数调用
3. 前置自增自减表达式++i、--i
4. 由赋值表达式或赋值运算符连接的表达式(a=b, a += b等)
5. 解引用表达式\*p
6. 字符串字面值"abcd"

### 纯右值、将亡值

纯右值和将亡值都属于右值。

* 纯右值

运算表达式产生的临时变量、不和对象关联的原始字面量、非引用返回的临时变量、lambda表达式等都是纯右值。

举例：

1. 除字符串字面值外的字面值
2. 返回非引用类型的函数调用
3. 后置自增自减表达式i++、i--
4. 算术表达式(a+b, a\*b, a&&b, a==b等)
5. 取地址表达式等(&a)

* 将亡值

将亡值是指C++11新增的和右值引用相关的表达式，通常指将要被移动的对象、T&&函数的返回值、std::move函数的返回值、转换为T&&类型转换函数的返回值， 将亡值可以理解为即将要销毁的值，通过“盗取”其它变量内存空间方式获取的值，在确保其它变量不再被使用或者即将被销毁时，可以避免内存空间的释放和分配，延长变量值的生命周期，常用来完成移动构造或者移动赋值的特殊任务。

举例：

class A { xxx;};

A a;

auto c = std::move(a); // c是将亡值 auto d = static\_cast<A&&>(a);  // d是将亡值

## 左值引用、右值引用

根据名字大概就可以猜到意思，左值引用就是对左值进行引用的类型，右值引用就是对右值进行引用的类型，他们都是引用，都是对象的一个别名，并不拥有所绑定对象的堆存，所以都必须立即初始化。

右值引用可实现转移语义（Move Sementics）和精确传递（Perfect Forwarding），它的主要目的有两个方面：

* 消除两个对象交互时不必要的对象拷贝，节省运算存储资源，提高效率。
* 能够更简洁明确地定义泛型函数。

int a = 5;int &b = a; // b是左值引用b = 4;

int &c = 10; // error，10无法取地址，无法进行引用

const int &d = 10; // ok，因为是常引用，引用常量数字，这个常量数字会存储在内存中，可以取地址

可以得出结论：对于左值引用，等号右边的值必须可以取地址，如果不能取地址，则会编译失败，或者可以使用const引用形式，但这样就只能通过引用来读取输出，不能修改数组，因为是常量引用。

如果使用右值引用，那表达式等号右边的值需要时右值，可以使用std::move函数强制把左值转换为右值。

int a = 4;

int &&b = a; // error, a是左值

int &&c = std::move(a); // ok

## 移动

移动语义，在看来可以理解为转移所有权，之前的拷贝是对于别人的资源，自己重新分配一块内存存储复制过来的资源，而对于移动语义，类似于转让或者资源窃取的意思，对于那块资源，转为自己所拥有，别人不再拥有也不会再使用，通过C++11新增的移动语义可以省去很多拷贝负担，怎么利用移动语义呢，是通过移动构造函数。

如果不使用std::move()，会有很大的拷贝代价，使用移动语义可以避免很多无用的拷贝，提供程序性能，C++所有的STL都实现了移动语义，方便我们使用。例如

std::vector<string> vecs;

...

std::vector<string> vecm = std::move(vecs); // 免去很多拷贝

注意：移动语义仅针对于那些实现了移动构造函数的类的对象，对于那种基本类型int、float等没有任何优化作用，还是会拷贝，因为它们实现没有对应的移动构造函数。

## 完美转发

完美转发指可以写一个接受任意实参的函数模板，并转发到其它函数，目标函数会收到与转发函数完全相同的实参，转发函数实参是左值那目标函数实参也是左值，转发函数实参是右值那目标函数实参也是右值。那如何实现完美转发呢，答案是使用std::forward()。

## 引用折叠

X& &、X& &&、X&& & 可折叠成 X&

X&& && 可折叠成 X&&

## 指针和引用的区别

* 指针是一个变量，存储的是一个地址，引用跟原来的变量实质上是同一个东西，是原变量的别名
* 指针可以有多级，引用只有一级
* 指针可以为空，引用不能为NULL且在定义时必须初始化
* 指针在初始化后可以改变指向，而引用在初始化之后不可再改变
* sizeof指针得到的是本指针的大小，sizeof引用得到的是引用所指向变量的大小
* 当把指针作为参数进行传递时，也是将实参的一个拷贝传递给形参，两者指向的地址相同，但不是同一个变量，在函数中改变这个变量的指向不影响实参，而引用却可以。
* 引用只是别名，不占用具体存储空间，只有声明没有定义；指针是具体变量，需要占用存储空间。
* 引用在声明时必须初始化为另一变量，一旦出现必须为typename refname &varname形式；指针声明和定义可以分开，可以先只声明指针变量而不初始化，等用到时再指向具体变量。
* 引用一旦初始化之后就不可以再改变（变量可以被引用为多次，但引用只能作为一个变量引用）；指针变量可以重新指向别的变量。
* 不存在指向空值的引用，必须有具体实体；但是存在指向空值的指针。

## 值传递、指针传递、引用传递的区别

### 值传递

有一个形参向函数所属的栈拷贝数据的过程，如果值传递的对象是类对象   或是大的结构体对象，将耗费一定的时间和空间。（传值）

### 指针传递

同样有一个形参向函数所属的栈拷贝数据的过程，但拷贝的数据是一个固定为4字节的地址。（传值，传递的是地址值）

引用传递

同样有上述的数据拷贝过程，但其是针对地址的，相当于为该数据所在的地址起了一个别名。（传地址）

效率上讲，指针传递和引用传递比值传递效率高。一般主张使用引用传递，代码逻辑上更加紧凑、清晰。

## 从汇编层去解释一下引用

9:      int x = 1;  
00401048  mov     dword ptr [ebp-4],1  
10:     int &b = x;  
0040104F   lea     eax,[ebp-4]  
00401052  mov     dword ptr [ebp-8],eax

x的地址为ebp-4，b的地址为ebp-8，因为栈内的变量内存是从高往低进行分配的，所以b的地址比x的低。

lea eax,[ebp-4] 这条语句将x的地址ebp-4放入eax寄存器

mov dword ptr [ebp-8],eax 这条语句将eax的值放入b的地址

ebp-8中上面两条汇编的作用即：将x的地址存入变量b中，这不和将某个变量的地址存入指针变量是一样的吗？所以从汇编层次来看，的确引用是通过指针来实现的。

# 构造析构

## 析构函数的作用，如何起作用？

1)  构造函数只是起初始化值的作用，但实例化一个对象的时候，可以通过实例去传递参数，从主函数传递到其他的函数里面，这样就使其他的函数里面有值了。

规则，只要你一实例化对象，系统自动回调用一个构造函数就是你不写，编译器也自动调用一次。

2)  析构函数与构造函数的作用相反，用于撤销对象的一些特殊任务处理，可以是释放对象分配的内存空间；特点：析构函数与构造函数同名，但该函数前面加~。

析构函数没有参数，也没有返回值，而且不能重载，在一个类中只能有一个析构函数。当撤销对象时，编译器也会自动调用析构函数。

每一个类必须有一个析构函数，用户可以自定义析构函数，也可以是编译器自动生成默认的析构函数。一般析构函数定义为类的公有成员。

## 类什么时候会析构？

1)  对象生命周期结束，被销毁时；

2)  delete指向对象的指针时，或delete指向对象的基类类型指针，而其基类虚构函数是虚函数时；

3)  对象i是对象o的成员，o的析构函数被调用时，对象i的析构函数也被调用。

## 构造函数、析构函数的执行顺序？

### 构造函数顺序

①   基类构造函数。如果有多个基类，则构造函数的调用顺序是某类在类派生表中出现的顺序，而不是它们在成员初始化表中的顺序。

②   成员类对象构造函数。如果有多个成员类对象则构造函数的调用顺序是对象在类中被声明的顺序，而不是它们出现在成员初始化表中的顺序。

③   派生类构造函数。

### 析构函数顺序

①   调用派生类的析构函数；

②   调用成员类对象的析构函数；

③   调用基类的析构函数。

## 构造函数析构函数可否抛出异常

1)   C++只会析构已经完成的对象，对象只有在其构造函数执行完毕才算是完全构造妥当。在构造函数中发生异常，控制权转出构造函数之外。

因此，在对象b的构造函数中发生异常，对象b的析构函数不会被调用。因此会造成内存泄漏。

2)  用auto\_ptr对象来取代指针类成员，便对构造函数做了强化，免除了抛出异常时发生资源泄漏的危机，不再需要在析构函数中手动释放资源；

3)  如果控制权基于异常的因素离开析构函数，而此时正有另一个异常处于作用状态，C++会调用terminate函数让程序结束；

4)  如果异常从析构函数抛出，而且没有在当地进行捕捉，那个析构函数便是执行不全的。如果析构函数执行不全，就是没有完成他应该执行的每一件事情。

## 构造函数的几种关键字

### default

default关键字可以显式要求编译器生成合成构造函数，防止在调用时相关构造函数类型没有定义而报错。

#include <iostream>  
using namespace std;  
  
class CString  
{  
public:  
    CString() = default; //语句1  
    //构造函数  
    CString(const char\* pstr) : \_str(pstr){}  
    void\* operator new() = delete;//这样不允许使用new关键字  
    //析构函数  
    ~CString(){}  
public:  
     string \_str;  
};  
  
  
int main()  
{  
   auto a = new CString(); //语句2  
   cout << "Hello World" <<endl;  
   return 0;  
}  
//运行结果  
//Hello World

如果没有加语句1，语句2会报错，表示找不到参数为空的构造函数，将其设置为default可以解决这个问题。

### delete

delete关键字可以删除构造函数、赋值运算符函数等，这样在使用的时候会得到友善的提示。

#include <iostream>  
using namespace std;  
  
class CString  
{  
public:  
    void\* operator new() = delete;//这样不允许使用new关键字  
    //析构函数  
    ~CString(){}  
};  
  
int main()  
{  
   auto a = new CString(); //语句1  
   cout << "Hello World" <<endl;  
   return 0;  
}

在执行语句1时，会提示new方法已经被删除，如果将new设置为私有方法，则会报惨不忍睹的错误，因此使用delete关键字可以更加人性化的删除一些默认方法。

### =0

将虚函数定义为纯虚函数（纯虚函数无需定义，= 0只能出现在类内部虚函数的声明语句处；当然，也可以为纯虚函数提供定义，不过函数体必须定义在类的外部）。

## 构造函数

对象不存在，没用别的对象初始化，在创建一个新的对象时调用构造函数

## 拷贝构造函数

对象不存在，但是使用别的已经存在的对象来进行初始化

### 拷贝构造函数和赋值运算符重载的区别？

* 拷贝构造函数是函数，赋值运算符是运算符重载。
* 拷贝构造函数会生成新的类对象，赋值运算符不能。
* 拷贝构造函数是直接构造一个新的类对象，所以在初始化对象前不需要检查源对象和新建对象是否相同；赋值运算符需要上述操作并提供两套不同的复制策略，另外赋值运算符中如果原来的对象有内存分配则需要先把内存释放掉。
* 形参传递是调用拷贝构造函数（调用的被赋值对象的拷贝构造函数），但并不是所有出现"="的地方都是使用赋值运算符，如下：

Student s;  
 Student s1 = s;    // 调用拷贝构造函数  
 Student s2;  
 s2 = s;    // 赋值运算符操作

注：类中有指针变量时要重写析构函数、拷贝构造函数和赋值运算符

## 赋值运算符

对象存在，用别的对象给它赋值，这属于重载“=”号运算符的范畴，“=”号两侧的对象都是已存在的。

#include <iostream>  
using namespace std;  
  
class A  
{  
public:  
    A()  
    {  
        cout << "我是构造函数" << endl;  
    }  
    A(const A& a)  
    {  
        cout << "我是拷贝构造函数" << endl;  
    }  
    A& operator = (A& a)  
    {  
        cout << "我是赋值操作符" << endl;  
        return \*this;  
    }  
    ~A() {};  
};  
  
int main()  
{  
    A a1; //调用构造函数  
    A a2 = a1; //调用拷贝构造函数  
    a2 = a1; //调用赋值操作符  
    return 0;  
}  
//输出结果  
//我是构造函数  
//我是拷贝构造函数  
//我是赋值操作符

## 移动构造函数

1)  我们用对象a初始化对象b，后对象a我们就不在使用了，但是对象a的空间还在呀（在析构之前），既然拷贝构造函数，实际上就是把a对象的内容复制一份到b中，那么为什么我们不能直接使用a的空间呢？这样就避免了新的空间的分配，大大降低了构造的成本。这就是移动构造函数设计的初衷；

2)  拷贝构造函数中，对于指针，我们一定要采用深层复制，而移动构造函数中，对于指针，我们采用浅层复制。浅层复制之所以危险，是因为两个指针共同指向一片内存空间，若第一个指针将其释放，另一个指针的指向就不合法了。

所以我们只要避免第一个指针释放空间就可以了。避免的方法就是将第一个指针（比如a->value）置为NULL，这样在调用析构函数的时候，由于有判断是否为NULL的语句，所以析构a的时候并不会回收a->value指向的空间；

 移动构造函数的参数和拷贝构造函数不同，拷贝构造函数的参数是一个左值引用，但是移动构造函数的初值是一个右值引用。

意味着，移动构造函数的参数是一个右值或者将亡值的引用。也就是说，只用用一个右值，或者将亡值初始化另一个对象的时候，才会调用移动构造函数。而那个move语句，就是将一个左值变成一个将亡值。

## 深拷贝与浅拷贝是怎么回事？

### 浅复制

只是拷贝了基本类型的数据，而引用类型数据，复制后也是会发生引用，我们把这种拷贝叫做“（浅复制）浅拷贝”，换句话说，浅复制仅仅是指向被复制的内存地址，如果原地址中对象被改变了，那么浅复制出来的对象也会相应改变。

### 深复制

在计算机中开辟了一块新的内存地址用于存放复制的对象。

在某些状况下，类内成员变量需要动态开辟堆内存，如果实行位拷贝，也就是把对象里的值完全复制给另一个对象，如A=B。这时，如果B中有一个成员变量指针已经申请了内存，那A中的那个成员变量也指向同一块内存。这就出现了问题：当B把内存释放了（如：析构），这时A内的指针就是野指针了，出现运行错误。

### 移动

移动语义，可以理解为转移所有权，之前的拷贝是对于别人的资源，自己重新分配一块内存存储复制过来的资源，而对于移动语义，类似于转让或者资源窃取的意思，对于那块资源，转为自己所拥有，别人不再拥有也不会再使用，通过C++11新增的移动语义可以省去很多拷贝负担，怎么利用移动语义呢，是通过移动构造函数。

## 返回值优化

返回值优化(RVO)是一种C++编译优化技术，当函数需要返回一个对象实例时候，就会创建一个临时对象并通过复制构造函数将目标对象复制到临时对象，这里有复制构造函数和析构函数会被多余的调用到，有代价，而通过返回值优化，C++标准允许省略调用这些复制构造函数。

那什么时候编译器会进行返回值优化呢?

1. return的值类型与函数的返回值类型相同
2. return的是一个局部对象

# 对象复用的了解，零拷贝的了解

## 对象复用（享元模式）

对象复用其本质是一种设计模式：Flyweight享元模式。

通过将对象存储到“对象池”中实现对象的重复利用，这样可以避免多次创建重复对象的开销，节约系统资源。

## 零拷贝(std::move)

零拷贝就是一种避免 CPU 将数据从一块存储拷贝到另外一块存储的技术。

零拷贝技术可以减少数据拷贝和共享总线操作的次数。

在C++中，vector的一个成员函数emplace\_back()很好地体现了零拷贝技术，它跟push\_back()函数一样可以将一个元素插入容器尾部，区别在于：使用push\_back()函数需要调用拷贝构造函数和转移构造函数，而使用emplace\_back()插入的元素原地构造，不需要触发拷贝构造和转移构造，

## 内存池

内存池（Memory Pool） 是一种内存分配方式。通常我们习惯直接使用new、malloc 等申请内存，这样做的缺点在于：由于所申请内存块的大小不定，当频繁使用时会造成大量的内存碎片并进而降低性能。内存池则是在真正使用内存之前，先申请分配一定数量的、大小相等(一般情况下)的内存块留作备用。当有新的内存需求时，就从内存池中分出一部分内存块， 若内存块不够再继续申请新的内存。这样做的一个显著优点是尽量避免了内存碎片，使得内存分配效率得到提升。

这里简单描述一下《STL源码剖析》中的内存池实现机制：

allocate包装malloc,deallocate包装free

一般是一次20\*2个的申请，先用一半，留着一半，为什么也没个说法，侯捷在STL那边书里说好像是C++委员会成员认为20是个比较好的数字，既不大也不小。

首先客户端会调用malloc()配置一定数量的区块（固定大小的内存块，通常为8的倍数），假设40个32bytes的区块，其中20个区块（一半）给程序实际使用，1个区块交出，另外19个处于维护状态。剩余20个（一半）留给内存池，此时一共有（20\*32byte）

客户端之后有有内存需求，想申请（20\*64bytes）的空间，这时内存池只有（20\*32bytes），就先将（10\*64bytes)个区块返回，1个区块交出，另外9个处于维护状态，此时内存池空空如也。

接下来如果客户端还有内存需求，就必须再调用malloc()配置空间，此时新申请的区块数量会增加一个随着配置次数越来越大的附加量，同样一半提供程序使用，另一半留给内存池。申请内存的时候用永远是先看内存池有无剩余，有的话就用上，然后挂在0-15号某一条链表上，要不然就重新申请。

如果整个堆的空间都不够了，就会在原先已经分配区块中寻找能满足当前需求的区块数量，能满足就返回，不能满足就向客户端报bad\_alloc异常

《STL源码解析》侯捷 P68

allocator就是用来分配内存的，最重要的两个函数是allocate和deallocate，就是用来申请内存和回收内存的，外部（一般指容器）调用的时候只需要知道这些就够了。内部实现，目前的所有编译器都是直接调用的::operator new()和::operator delete()，说白了就是和直接使用new运算符的效果是一样的，所以老师说它们都没做任何特殊处理。

最开始GC2.9之前：

new和 operator new 的区别：new 是个运算符，编辑器会调用 operator new(0)

operator new()里面有调用malloc的操作，那同样的 operator delete()里面有调用的free的操作

GC2.9的alloc的一个比较好的分配器的实现规则

维护一条0-15号的一共16条链表，其中0表示8 bytes ，1表示 16 bytes,2表示 24bytes。。。。而15 表示 16\* 8 = 128bytes，如果在申请时并不是8的倍数，那就找刚好能满足内存大小的那个位置。比如想申请 12，那就是找16了，想申请 20 ，那就找 24 了

但是现在GC4.9及其之后 也还有，变成\_pool\_alloc这个名字了，不再是默认的了，你需要自己去指定它可以自己指定，比如说vector<string,\_\_gnu\_cxx::pool\_alloc<string> style="margin: 0px; padding: 0px; max-width: 100%; box-sizing: border-box !important; word-wrap: break-word !important; font-size: inherit; color: inherit; line-height: inherit;">vec;这样来使用它，现在用的又回到以前那种对malloc和free的包装形式了。

# 类成员初始化方式

构造函数的执行顺序 ？为什么用成员初始化列表会快一些？

1. 赋值初始化，通过在函数体内进行赋值初始化；
2. 列表初始化，在冒号后使用初始化列表进行初始化。

这两种方式的主要区别在于：

对于在函数体中初始化,是在所有的数据成员被分配内存空间后才进行的。

列表初始化是给数据成员分配内存空间时就进行初始化,就是说分配一个数据成员只要冒号后有此数据成员的赋值表达式(此表达式必须是括号赋值表达式),那么分配了内存空间后在进入函数体之前给数据成员赋值，就是说初始化这个数据成员此时函数体还未执行。

2)  一个派生类构造函数的执行顺序如下：

①   虚拟基类的构造函数（多个虚拟基类则按照继承的顺序执行构造函数）。

②   基类的构造函数（多个普通基类也按照继承的顺序执行构造函数）。

③   类类型的成员对象的构造函数（按照初始化顺序）

④   派生类自己的构造函数。

3)  方法一是在构造函数当中做赋值的操作，而方法二是做纯粹的初始化操作。我们都知道，C++的赋值操作是会产生临时对象的。临时对象的出现会降低程序的效率。

## 拷贝初始化和直接初始化

当用于类类型对象时，初始化的拷贝形式和直接形式有所不同：直接初始化直接调用与实参匹配的构造函数，拷贝初始化总是调用拷贝构造函数。

拷贝初始化首先使用指定构造函数创建一个临时对象，然后用拷贝构造函数将那个临时对象拷贝到正在创建的对象。举例如下。

string str1("I am a string"); //语句1 直接初始化

string str2(str1);

//语句2 直接初始化，str1是已经存在的对象，直接调用构造函数对str2进行初始化

string str3 = "I am a string"; //语句3 拷贝初始化，先为字符串”I am a string“创建临时对象，再把临时对象作为参数，使用拷贝构造函数构造str3

string str4 = str1;

//语句4 拷贝初始化，这里相当于隐式调用拷贝构造函数，而不是调用赋值运算符函数

为了提高效率，允许编译器跳过创建临时对象这一步，直接调用构造函数构造要创建的对象，这样就完全等价于直接初始化了（语句1和语句3等价）。但是需要辨别两种情况。

当拷贝构造函数为private时：语句3和语句4在编译时会报错。

使用explicit修饰构造函数时：如果构造函数存在隐式转换，编译时会报错。

## 初始化和赋值的区别

对于简单类型来说，初始化和赋值没什么区别。

对于类和复杂数据类型来说，这两者的区别就大了。

class A{  
public:  
    int num1;int num2;  
public:  
    A(int a=0, int b=0):num1(a),num2(b){};  
    A(const A& a){};  
    //重载 = 号操作符函数  
    A& operator=(const A& a){  
        num1 = a.num1 + 1;  
        num2 = a.num2 + 1;  
        return \*this; };};  
int main(){  
    A a(1,1);  
    A a1 = a; //拷贝初始化操作，调用拷贝构造函数  
    A b;  
    b = a;//赋值操作，对象a中，num1 = 1，num2 = 1；对象b中，num1 = 2，num2 = 2  
    return 0;}

## 成员初始化列表

### 成员初始化列表的概念

在类的构造函数中，不在函数体内对成员变量赋值，而是在构造函数的花括号前面使用冒号和初始化列表赋值

### 效率

用初始化列表会快一些的原因是，对于类型，它少了一次调用构造函数的过程，而在函数体中赋值则会多一次调用。而对于内置数据类型则没有差别。举个例子：

#include <iostream>  
using namespace std;  
class A  
{  
public:  
    A()  
    {  
        cout << "默认构造函数A()" << endl;  
    }  
    A(int a)  
    {  
        value = a;  
        cout << "A(int "<<value<<")" << endl;  
    }  
    A(const A& a)  
    {  
        value = a.value;  
        cout << "拷贝构造函数A(A& a):  "<<value << endl;  
    }  
    int value;  
};  
  
class B  
{  
public:  
    B() : a(1)  
    {  
        b = A(2);  
    }  
    A a;  
    A b;  
};  
int main()  
{  
    B b;  
}  
  
//输出结果：  
//A(int 1)  
//默认构造函数A()  
//A(int 2)

从代码运行结果可以看出，在构造函数体内部初始化的对象b多了一次构造函数的调用过程，而对象a则没有。

由于对象成员变量的初始化动作发生在进入构造函数之前，对于内置类型没什么影响，但如果有些成员是类，那么在进入构造函数之前，会先调用一次默认构造函数，进入构造函数后所做的事其实是一次赋值操作(对象已存在)，所以如果是在构造函数体内进行赋值的话，等于是一次默认构造加一次赋值，而初始化列表只做一次赋值操作。

### 好处

* 更高效：少了一次调用默认构造函数的过程。

### 有些场合必须要用初始化列表

* 初始化一个const成员。常量成员，因为常量只能初始化不能赋值，所以必须放在初始化列表里面
* 初始化一个reference成员。引用类型，引用必须在定义的时候初始化，并且不能重新赋值，所以也要写在初始化列表里面
* 没有默认构造函数的类类型，因为使用初始化列表可以不必调用默认构造函数来初始化
* 调用一个基类的构造函数，而该函数有一组参数。
* 调用一个数据成员对象的构造函数，而该函数有一组参数。

### 必须使用成员初始化的四种情况

①    当初始化一个引用成员时；

②    当初始化一个常量成员时；

③    当调用一个基类的构造函数，而它拥有一组参数时；

④    当调用一个成员类的构造函数，而它拥有一组参数时；

成员初始化列表做了什么

①    编译器会一一操作初始化列表，以适当的顺序在构造函数之内安插初始化操作，并且在任何显示用户代码之前；

②    list中的项目顺序是由类中的成员**声明顺序**决定的，不是由初始化列表的顺序决定的；

### initializer\_list 列表初始化

* 用花括号初始化器列表初始化一个对象，其中对应构造函数接受一个 std::initializer\_list 参数。
* [Day1\_基础\Day1\initializer\_list列表初始化.h](Day1_基础/Day1/initializer_list列表初始化.h)

# 定义只在堆(栈)生成对象类

## 只能在堆上

* 方法：将析构函数设置为私有
* 原因：C++ 是静态绑定语言，编译器管理栈上对象的生命周期，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。若析构函数不可访问，则不能在栈上创建对象。

## 只能在栈上

* 方法：将 new 和 delete 重载为私有
* 原因：在堆上生成对象，使用 new 关键词操作，其过程分为两阶段：第一阶段，使用 new 在堆上寻找可用内存，分配给对象；第二阶段，调用构造函数生成对象。将 new 操作设置为私有，那么第一阶段就无法完成，就不能够在堆上生成对象。

## 不能被继承

### final关键字

当不希望某个类被继承，或不希望某个虚函数被重写，可以在类名和虚函数后添加final关键字，添加final关键字后被继承或重写，编译器会报错。

class Base final{

}

class Base  
{  
    virtual void foo();  
};  
  
class A : public Base  
{  
    void foo() final;

 // foo 被override并且是最后一个override，在其子类中不可以重写  
};  
  
class B final : A  // 指明B是不可以被继承的  
{  
    void foo() override;  // Error: 在A中已经被final了  
};  
  
class C : B  // Error: B is final  
{  
};

### 构造函数private

### 构造函数delete

# 强制类型转换运算符

## static\_cast

static\_cast < type-id > (expression)

该运算符把expression转换为type-id类型，但没有运行时类型检查来保证转换的安全性。它主要有如下几种用法：

* 用于类层次结构中基类（父类）和派生类（子类）之间指针或引用引用的转换
* 进行上行转换（把派生类的指针或引用转换成基类表示）是安全的。
* 进行下行转换（把基类指针或引用转换成派生类表示）时，由于没有动态类型检查，所以是不安全的（转换安全性不如 dynamic\_cast）。编译时类型检查（安全性强于reinterpret\_cast）。
* 用于基本数据类型之间的转换，如把int转换成char，把int转换成enum。这种转换的安全性也要开发人员来保证。
* 把空指针转换成目标类型的空指针
* 把任何类型的表达式转换成void类型

注意：static\_cast不能转换掉expression的const、volatile、或者\_\_unaligned属性。

### 使用场景

* 通常用于转换数值数据类型（如 float -> int）
* 可以在整个类层次结构中移动指针，子类转化为父类安全（向上转换），父类转化为子类不安全（因为子类可能有不在父类的字段或方法）
* void\*转event\*

struct event \*ev = static\_cast<struct event\*>(arg);

## dynamic\_cast

有类型检查，基类向派生类转换比较安全，但是派生类向基类转换则不太安全

dynamic\_cast(expression)

该运算符把expression转换成type-id类型的对象。type-id 必须是类的指针、类的引用或者void\*

如果 type-id 是类指针类型，那么expression也必须是一个指针，如果 type-id 是一个引用，那么 expression 也必须是一个引用

dynamic\_cast运算符可以在执行期决定真正的类型，也就是说expression必须是多态类型。如果下行转换是安全的（也就说，如果基类指针或者引用确实指向一个派生类对象）这个运算符会传回适当转型过的指针。如果 如果下行转换不安全，这个运算符会传回空指针（也就是说，基类指针或者引用没有指向一个派生类对象）

* dynamic\_cast主要用于类层次间的上行转换和下行转换，还可以用于类之间的交叉转换。
* 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。

### 使用场景

* 用于多态类型的转换
* 执行行运行时类型检查
* 只适用于指针或引用
* 对不明确的指针的转换将失败（返回 nullptr），但不引发异常
* 可以在整个类层次结构中移动指针，包括向上转换、向下转换

### bad\_cast

* 由于强制转换为引用类型失败，dynamic\_cast 运算符引发 bad\_cast 异常。

## const\_cast

const\_cast(expression)

该运算符用来修改类型的const或volatile属性。除了const 或volatile修饰之外， type\_id和expression的类型是一样的。用法如下：

* 常量指针被转化成非常量的指针，并且仍然指向原来的对象
* 常量引用被转换成非常量的引用，并且仍然指向原来的对象
* const\_cast一般用于修改底指针。如const char \*p形式。

用于删除 const、volatile 和 \_\_unaligned 特性（如将 const int 类型转换为 int 类型 ）

## reinterpret\_cast

reinterpret\_cast(expression)

type-id 必须是一个指针、引用、算术类型、函数指针或者成员指针。它可以用于类型之间进行强制转换。

* 用于位的简单重新解释
* 滥用 reinterpret\_cast 运算符可能很容易带来风险。除非所需转换本身是低级别的，否则应使用其他强制转换运算符之一。
* 允许将任何指针转换为任何其他指针类型（如 char\* 到 int\* 或 One\_class\* 到 Unrelated\_class\* 之类的转换，但其本身并不安全）
* 也允许将任何整数类型转换为任何指针类型以及反向转换。
* reinterpret\_cast 运算符不能丢掉 const、volatile 或 \_\_unaligned 特性。
* reinterpret\_cast 的一个实际用途是在哈希函数中，即，通过让两个不同的值几乎不以相同的索引结尾的方式将值映射到索引。
* sockaddr转sockaddr\_in

sockaddr\_in\* SocketAddress::GetAsSockAddrIn()

{

return reinterpret\_cast<sockaddr\_in\*>(&mSockAddr);

}

## 总结

在进行下行转换时，dynamic\_cast安全的，如果下行转换不安全的话其会返回空指针，这样在进行操作的时候可以预先判断。

而使用static\_cast下行转换存在不安全的情况也可以转换成功，但是直接使用转换后的对象进行操作容易造成错误。

# this指针

this 指针是一个隐含于每一个非静态成员函数中的特殊指针。它指向调用该成员函数的那个对象。this指针是类的指针，指向对象的首地址。

this指针只能在成员函数中使用，在全局函数、静态成员函数中都不能用this。

this指针只有在成员函数中才有定义，且存储位置会因编译器不同有不同存储位置。

当对象为空时，若函数不调用this指针内存，程序不会报错。

* 当对一个对象调用成员函数时，编译程序先将对象的地址赋给 this 指针，然后调用成员函数，每次成员函数存取数据成员时，都隐式使用 this 指针。
* 当一个成员函数被调用时，自动向它传递一个隐含的参数，该参数是一个指向这个成员函数所在的对象的指针。
* this 指针被隐含地声明为: ClassName const this，这意味着不能给 this 指针赋值；在 ClassName 类的 const 成员函数中，this 指针的类型为：const ClassName const，这说明不能对 this 指针所指向的这种对象是不可修改的（即不能对这种对象的数据成员进行赋值操作）；
* this 并不是一个常规变量，而是个右值，所以不能取得 this 的地址（不能 &this）。

## this指针的用处

一个对象的this指针并不是对象本身的一部分，不会影响sizeof(对象)的结果。this作用域是在类内部，当在类的非静态成员函数中访问类的非静态成员的时候（全局函数，静态函数中不能使用this指针），编译器会自动将对象本身的地址作为一个隐含参数传递给函数。也就是说，即使你没有写上this指针，编译器在编译的时候也是加上this的，它作为非静态成员函数的隐含形参，对各成员的访问均通过this进行。

## this指针的使用

一种情况就是，在类的非静态成员函数中返回类对象本身的时候，直接使用 return \*this；

另外一种情况是当形参数与成员变量名相同时用于区分，如this->n = n （不能写成n = n）

## 类的this指针有以下特点

(1）this只能在成员函数中使用，全局函数、静态函数都不能使用this。实际上，成员函数默认第一个参数为T \* const this

如：

class A{  
public:  
    int func(int p){}  
};

其中，func的原型在编译器看来应该是：

int func(A \* const this,int p);

（2）由此可见，this在成员函数的开始前构造，在成员函数的结束后清除。这个生命周期同任何一个函数的参数是一样的，没有任何区别。当调用一个类的成员函数时，编译器将类的指针作为函数的this参数传递进去。如：

A a;  
a.func(10);  
//此处，编译器将会编译成：  
A::func(&a,10);

看起来和静态函数没差别，对吗？不过，区别还是有的。编译器通常会对this指针做一些优化，因此，this指针的传递效率比较高，例如VC通常是通过ecx（计数寄存器）传递this参数的。

## 需要显式引用 this 指针

在以下场景中，经常需要显式引用 this 指针：

* 为实现对象的链式引用；
* 为避免对同一对象进行赋值操作；
* 在实现一些数据结构时，如 list。

## 一些问题

### A. this指针是什么时候创建的？

this在成员函数的开始执行前构造，在成员的执行结束后清除。

但是如果class或者struct里面没有方法的话，它们是没有构造函数的，只能当做C的struct使用。

采用TYPE xx的方式定义的话，在栈里分配内存，这时候this指针的值就是这块内存的地址。采用new的方式创建对象的话，在堆里分配内存，new操作符通过eax（累加寄存器）返回分配的地址，然后设置给指针变量。之后去调用构造函数（如果有构造函数的话），这时将这个内存块的地址传给ecx，之后构造函数里面怎么处理请看上面的回答。

### this指针存放在何处？

this指针存放在何处？堆、栈、全局变量，还是其他？

this指针会因编译器不同而有不同的放置位置。可能是栈，也可能是寄存器，甚至全局变量。在汇编级别里面，一个值只会以3种形式出现：立即数、寄存器值和内存变量值。不是存放在寄存器就是存放在内存中，它们并不是和高级语言变量对应的。

### C. this指针是如何传递类中的函数的

C. this指针是如何传递类中的函数的？绑定？还是在函数参数的首参数就是this指针？那么，this指针又是如何找到“类实例后函数的”？

大多数编译器通过ecx（寄数寄存器）寄存器传递this指针。事实上，这也是一个潜规则。一般来说，不同编译器都会遵从一致的传参规则，否则不同编译器产生的obj就无法匹配了。

在call之前，编译器会把对应的对象地址放到eax中。this是通过函数参数的首参来传递的。this指针在调用之前生成，至于“类实例后函数”，没有这个说法。类在实例化时，只分配类中的变量空间，并没有为函数分配空间。自从类的函数定义完成后，它就在那儿，不会跑的

### D. this指针是如何访问类中的变量的？

如果不是类，而是结构体的话，那么，如何通过结构指针来访问结构中的变量呢？如果你明白这一点的话，就很容易理解这个问题了。

在C++中，类和结构是只有一个区别的：类的成员默认是private，而结构是public。

this是类的指针，如果换成结构体，那this就是结构的指针了。

E.我们只有获得一个对象后，才能通过对象使用this指针。如果我们知道一个对象this指针的位置，可以直接使用吗？

this指针只有在成员函数中才有定义。因此，你获得一个对象后，也不能通过对象使用this指针。所以，我们无法知道一个对象的this指针的位置（只有在成员函数里才有this指针的位置）。当然，在成员函数里，你是可以知道this指针的位置的（可以通过&this获得），也可以直接使用它。

### F.每个类编译后，是否创建一个类中函数表保存函数指针，以便用来调用函数？

普通的类函数（不论是成员函数，还是静态函数）都不会创建一个函数表来保存函数指针。只有虚函数才会被放到函数表中。但是，即使是虚函数，如果编译期就能明确知道调用的是哪个函数，编译器就不会通过函数表中的指针来间接调用，而是会直接调用该函数。正是由于this指针的存在，用来指向不同的对象，从而确保不同对象之间调用相同的函数可以互不干扰。

# 一些关键字

## explicit(显式)关键字

* explicit 修饰构造函数时，可以防止隐式转换和复制初始化
* explicit 修饰转换函数时，可以防止隐式转换，但按语境转换除外

## auto

C++11新标准引入了auto类型说明符，用它就能让编译器替我们去分析表达式所属的类型。和原来那些只对应某种特定的类型说明符(例如 int)不同，

auto 让编译器通过初始值来进行类型推演。从而获得定义变量的类型，所以说 auto 定义的变量必须有初始值。

auto并没有让C++成为弱类型语言，也没有弱化变量什么，只是使用auto的时候，编译器根据上下文情况，确定auto变量的真正类型。

auto在C++14中可以作为函数的返回值，因此auto AddTest(int a, int b)的定义是没问题的。

auto作为函数返回值时，只能用于定义函数，不能用于声明函数(编译无法通过)。但如果把实现写在头文件中，可以编译通过，因为编译器可以根据函数实现的返回值确定auto的真实类型。如果读者用过inline类成员函数，这个应该很容易明白，此特性与inline类成员函数类似。

## decltype

* decltype 关键字用于检查实体的声明类型或表达式的类型及值分类。
* decltype ( expression )

有的时候我们还会遇到这种情况，我们希望从表达式中推断出要定义变量的类型，但却不想用表达式的值去初始化变量。还有可能是函数的返回类型为某表达式的值类型。在这些时候auto显得就无力了，所以C++11又引入了第二种类型说明符decltype

它的作用是选择并返回操作数的数据类型。在此过程中，编译器只是分析表达式并得到它的类型，却不进行实际的计算表达式的值。

## friend友元类和友元函数

* 能访问私有成员
* 破坏封装性
* 友元关系不可传递
* 友元关系的单向性
* 友元声明的形式及数量不受限制

## Inline内联函数

### 特征

* 相当于把内联函数里面的内容写在调用内联函数处；
* 相当于不用执行进入函数的步骤，直接执行函数体；
* 相当于宏，却比宏多了类型检查，真正具有函数特性；
* 编译器一般不内联包含循环、递归、switch 等复杂操作的内联函数；
* 在类声明中定义的函数，除了虚函数的其他函数都会自动隐式地当成内联函数。
* [Day1\_基础\Day1\inline内联.h](Day1_基础/Day1/inline内联.h)

### 编译器对 inline 函数处理步骤

* 将 inline 函数体复制到 inline 函数调用点处；
* 为所用 inline 函数中的局部变量分配内存空间；
* 将 inline 函数的的输入参数和返回值映射到调用方法的局部变量空间中；
* 如果 inline 函数有多个返回点，将其转变为 inline 函数代码块末尾的分支（使用 GOTO）。

### 优缺点

* 内联函数同宏函数一样将在被调用处进行代码展开，省去了参数压栈、栈帧开辟与回收，结果返回等，从而提高程序运行速度。
* 内联函数相比宏函数来说，在代码展开时，会做安全检查或自动类型转换（同普通函数），而宏定义则不会。
* 在类中声明同时定义的成员函数，自动转化为内联函数，因此内联函数可以访问类的成员变量，宏定义则不能。
* 内联函数在运行时可调试，而宏定义不可以。

### 虚函数

虚函数（virtual）可以是内联函数（inline）吗？

* 虚函数可以是内联函数，内联是可以修饰虚函数的，但是当虚函数表现多态性的时候不能内联。
* 内联是在编译器建议编译器内联，而虚函数的多态性在运行期，编译器无法知道运行期调用哪个代码，因此虚函数表现为多态性时（运行期）不可以内联。
* inline virtual 唯一可以内联的时候是：编译器知道所调用的对象是哪个类（如 Base::who()），这只有在编译器具有实际对象而不是对象的指针或引用时才会发生。

### ****构造函数，析构函数****

构造函数和析构函数声明为内联函数是没有意义的。

《Effective C++》中所阐述的是：将构造函数和析构函数声明为inline是没有什么意义的，即编译器并不真正对声明为inline的构造和析构函数进行内联操作，因为编译器会在构造和析构函数中添加额外的操作（申请/释放内存，构造/析构对象等），致使构造函数/析构函数并不像看上去的那么精简。其次，class中的函数默认是inline型的，编译器也只是有选择性的inline，将构造函数和析构函数声明为内联函数是没有什么意义的。

## nullptr关键字

### C++中NULL和nullptr区别

NULL来自C语言，一般由宏定义实现，而 nullptr 则是C++11的新增关键字。在C语言中，NULL被定义为(void\*)0,而在C++语言中，NULL则被定义为整数0。编译器一般对其实际定义如下：

#### NULL

#ifdef \_\_cplusplus  
#define NULL 0  
#else  
#define NULL ((void \*)0)  
#endif

在C++中指针必须有明确的类型定义。但是将NULL定义为0带来的另一个问题是无法与整数的0区分。

#### nullptr

const class nullptr\_t{  
public:  
    template<class T>  inline operator T\*() const{ return 0; }  
    template<class C, class T> inline operator T C::\*() const { return 0; }  
private:  
    void operator&() const;  
} nullptr = {};

以上通过模板类和运算符重载的方式来对不同类型的指针进行实例化从而解决了(void\*)指针带来参数类型不明的问题，另外由于nullptr是明确的指针类型，所以不会与整形变量相混淆。

### 区别

//示例代码1.0 http://www.cnblogs.com/feng-sc/p/5710724.html

class Test

{

public:

void TestWork(int index)

{

std::cout << "TestWork 1" << std::endl;

}

void TestWork(int \* index)

{

std::cout << "TestWork 2" << std::endl;

}

};

int main()

{

Test test;

test.TestWork(NULL);

test.TestWork(nullptr);

}

NULL在c++里表示空指针，看到问题了吧，我们调用test.TestWork(NULL)，其实期望是调用的是void TestWork(int \* index)，但结果调用了void TestWork(int index)。但使用nullptr的时候，我们能调用到正确的函数。

## volatile让编译器不要优化

* volatile int i = 10;
* volatile 关键字是一种类型修饰符，用它声明的类型变量表示可以被某些编译器未知的因素（操作系统、硬件、其它线程等）更改。所以使用 volatile 告诉编译器不应对这样的对象进行优化。
* volatile 关键字声明的变量，每次访问时都必须从内存中取出值（没有被 volatile 修饰的变量，可能由于编译器的优化，从 CPU 寄存器中取值）
* const 可以是 volatile （如只读的状态寄存器）
* 指针可以是 volatile

## register请求放到寄存器

这个关键字请求编译器尽可能的将变量存在CPU内部寄存器中，而不是通过内存寻址访问，以提高效率。

## using

### using 声明

* using namespace\_name::name;
* 一条 using 声明 语句一次只引入命名空间的一个成员。它使得我们可以清楚知道程序中所引用的到底是哪个名字。

### 构造函数的 using 声明

class Derived : Base {  
public:  
using Base::Base;};

* 在 C++11 中，派生类能够重用其直接基类定义的构造函数。
* 如上 using 声明，对于基类的每个构造函数，编译器都生成一个与之对应（形参列表完全相同）的派生类构造函数。
* 生成如下类型构造函数：Derived(parms) : Base(args) { }

### using 指示

* using namespace\_name name;
* using 指示 使得某个特定命名空间中所有名字都可见，这样我们就无需再为它们添加任何前缀限定符了。
* 尽量少使用 using 指示 污染命名空间

## const

### 作用

* 修饰变量

说明该变量不可以被改变；在定义时必须初始化，之后无法更改。

* 修饰指针

分为指向常量的指针（pointer to const）和自身是常量的指针（常量指针，const pointer）；

* 修饰引用

指向常量的引用（reference to const），用于形参类型，即避免了拷贝，又避免了函数对值的修改；

* const成员函数

说明该成员函数内不能修改成员变量。const对象不可以调用非const成员函数；非const对象都可以调用；不可以改变非mutable（用该关键字声明的变量可以在const成员函数中被修改）数据的值。

* const成员变量

不能在类定义外部初始化，只能通过构造函数初始化列表进行初始化，并且必须有构造函数；不同类对其const数据成员的值可以不同，所以不能在类中声明时初始化

1)  阻止一个变量被改变，可以使用const关键字。在定义该const变量时，通常需要对它进行初始化，因为以后就没有机会再去改变它了；

2)  对指针来说，可以指定指针本身为const，也可以指定指针所指的数据为const，或二者同时指定为const；

3)  在一个函数声明中，const可以修饰形参，表明它是一个输入参数，在函数内部不能改变其值；

4)  对于类的成员函数，若指定其为const类型，则表明其是一个常函数，不能修改类的成员变量，类的常对象只能访问类的常成员函数；

5)  对于类的成员函数，有时候必须指定其返回值为const类型，以使得其返回值不为“左值”。

6)  const成员函数可以访问非const对象的非const数据成员、const数据成员，也可以访问const对象内的所有数据成员；

7)  非const成员函数可以访问非const对象的非const数据成员、const数据成员，但不可以访问const对象的任意数据成员；

8)  一个没有明确声明为const的成员函数被看作是将要修改对象中数据成员的函数，而且编译器不允许它为一个const对象所调用。因此const对象只能调用const成员函数。

9)  const类型变量可以通过类型转换符const\_cast将const类型转换为非const类型；

10) const类型变量必须定义的时候进行初始化，因此也导致如果类的成员变量有const类型的变量，那么该变量必须在类的初始化列表中进行初始化；

11) 对于函数值传递的情况，因为参数传递是通过复制实参创建一个临时变量传递进函数的，函数内只能改变临时变量，但无法改变实参。则这个时候无论加不加const对实参不会产生任何影响。但是在引用或指针传递函数调用中，因为传进去的是一个引用或指针，这样函数内部可以改变引用或指针所指向的变量，这时const 才是实实在在地保护了实参所指向的变量。因为在编译阶段编译器对调用函数的选择是根据实参进行的，所以，只有引用传递和指针传递可以用是否加const来重载。一个拥有顶层const的形参无法和另一个没有顶层const的形参区分开来。

### const 的指针与引用

int const a和const int a均表示定义常量类型a。

const int \*a，其中a为指向int型变量的指针，const在 \* 左侧，表示a指向不可变常量。(看成const (\*a)，对引用加const)。

int \*const a，依旧是指针类型，表示a为指向整型数据的常指针。(看成const(a)，对指针const)。

1)  当const修饰指针时，由于const的位置不同，它的修饰对象会有所不同。

2)  int \*const p2中const修饰p2的值,所以理解为p2的值不可以改变，

即p2只能**指向固定的一个变量地址**，但可以通过\*p2读写这个变量的值。顶层指针表示指针本身是一个常量。

3)  int const \*p1或者const int \*p1两种情况中const修饰\*p1，所以理解为\*p1的值不可以改变，即**不可以给\*p1赋值改变p1指向变量的值**，但可以通过给p赋值不同的地址改变这个指针指向。

底层指针表示指针所指向的变量是一个常量。

#### 指针

常量指针和指针常量区别？

##### 底层const（指针常量）p

const在\*后，指针指向不可改变。

* 指针常量是一个不能给改变指向的指针。指针是个常量，不能中途改变指向，如int \*const p。
* 自身是常量的指针（常量指针const pointer）指向不变

char \* const s2，const修饰s2。s2是个指针。

##### 顶层const（常量指针）\*p

const在\*前，指向的内存不可写。int main(int argc , **const char\* argv**)

* 常量指针是一个指针，读成常量的指针，指向一个只读变量。如int const \*p或const int \*p。

指的是const修饰的变量本身是一个常量，无法修改，指的是指针，就是 \* 号的右边。

* 指向常量的指针（pointer to const）值不变

const char\* s1，const修饰\*s1。s1是个变量。

#### 引用

* 指向常量的引用（reference to const）
* 没有 const reference，因为引用本身就是 const pointer
* （为了方便记忆可以想成）被 const 修饰（在 const 后面）的值不可改变，如下文使用例子中的 p2、p3。

# C++模板是什么，你知道底层怎么实现的？

1)  编译器并不是把函数模板处理成能够处理任意类的函数。

编译器从函数模板通过具体类型产生不同的函数；编译器会对函数模板进行两次编译：在声明的地方对模板代码本身进行编译，在调用的地方对参数替换后的代码进行编译。

2)  这是因为函数模板要被实例化后才能成为真正的函数。

在使用函数模板的源文件中包含函数模板的头文件，如果该头文件中只有声明，没有定义，那编译器无法实例化该模板，最终导致链接错误。

## 函数模板机制结论

编译器并不是把函数模板处理成能够处理任意类的函数。

编译器从函数模板通过具体类型产生不同的函数。

编译器会对函数模板进行两次编译。

在声明的地方对模板代码本身进行编译；在调用的地方对参数替换后的代码进行编译。

# C++容器

|  |  |  |
| --- | --- | --- |
| 数据结构 | 描述 | 实现头文件 |
| 向量(vector) | 连续存储的元素 | <vector> |
| 列表(list) | 由节点组成的双向链表，每个结点包含着一个元素 | <list> |
| 双队列(deque) | 连续存储的指向不同元素的指针所组成的数组 | <deque> |
| 集合(set) | 由节点组成的红黑树，每个节点都包含着一个元素，节点之间以某种作用于元素对的谓词排列，没有两个不同的元素能够拥有相同的次序 | <set> |
| 多重集合(multiset) | 允许存在两个次序相等的元素的集合 | <set> |
| 栈(stack) | 后进先出的值的排列 | <stack> |
| 队列(queue) | 先进先出的执的排列 | <queue> |
| 优先队列(priority\_queue) | 元素的次序是由作用于所存储的值对上的某种谓词决定的的一种队列 | <queue> |
| 映射(map) | 由{键，值}对组成的集合，以某种作用于键对上的谓词排列 | <map> |
| 多重映射(multimap) | 允许键对有相等的次序的映射 | <map> |

## std:array

std::array跟数组并没有太大区别，相对于数组，增加了迭代器等函数.

## std:forward\_list

线性表，与list（双向链表）区别在于它是单向链表。我们在学习数据结构的时候都知道，链表在对数据进行插入和删除是比顺序存储的线性表有优势，因此在插入和删除操作频繁的应用场景中，使用list和forward\_list比使用array、vector和deque效率要高很多。

## std::unordered\_map

std::unordered\_map与std::map用法基本差不多，但STL在内部实现上有很大不同，std::map使用的数据结构为二叉树，而std::unordered\_map内部是哈希表的实现方式，哈希map理论上查找效率为O(1)。但在存储效率上，哈希map需要增加哈希表的内存开销。

## std::unordered\_set

std::unordered\_set的数据存储结构也是哈希表的方式结构，除此之外，std::unordered\_set在插入时不会自动排序，这都是std::set表现不同的地方。

## 迭代器失效的情况

以vector为例：

### 插入元素

1、尾后插入：size < capacity时，首迭代器不失效尾迭代失效（未重新分配空间），size == capacity时，所有迭代器均失效（需要重新分配空间）。

2、中间插入：中间插入：size < capacity时，首迭代器不失效但插入元素之后所有迭代器失效，size == capacity时，所有迭代器均失效。

### 删除元素

尾后删除：只有尾迭代失效。

中间删除：删除位置之后所有迭代失效。

# 智能指针

在内存管理方面，C++11的std::auto\_ptr基础上，移植了boost库中的智能指针的部分实现，如std::shared\_ptr、std::weak\_ptr等，当然，想boost::thread一样，C++11也修复了boost::make\_shared中构造参数的限制问题。

智能指针只是用对象去管理一个资源指针，同时用一个计数器计算当前指针引用对象的个数，当管理指针的对象增加或减少时，计数器也相应加1或减1，当最后一个指针管理对象销毁时，计数器为1，此时在销毁指针管理对象的同时，也把指针管理对象所管理的指针进行delete操作。

## 原理

智能指针是一个类，用来存储指向动态分配对象的指针，负责自动释放动态分配的对象，防止堆内存泄漏。动态分配的资源，交给一个类对象去管理，当类对象声明周期结束时，自动调用析构函数释放资源。

C++ 标准库（STL）中，头文件：#include <memory>

## 智能指针的作用

1)  C++11中引入了智能指针的概念，方便管理堆内存。使用普通指针，容易造成堆内存泄露（忘记释放），二次释放，程序发生异常时内存泄露等问题等，使用智能指针能更好的管理堆内存。

2)  智能指针在C++11版本之后提供，包含在头文件中，shared\_ptr、unique\_ptr、weak\_ptr。shared\_ptr多个指针指向相同的对象。shared\_ptr使用引用计数，每一个shared\_ptr的拷贝都指向相同的内存。每使用他一次，内部的引用计数加1，每析构一次，内部的引用计数减1，减为0时，自动删除所指向的堆内存。shared\_ptr内部的引用计数是线程安全的，但是对象的读取需要加锁。

3)  初始化。智能指针是个模板类，可以指定类型，传入指针通过构造函数初始化。也可以使用make\_shared函数初始化。不能将指针直接赋值给一个智能指针，一个是类，一个是指针。例如std::shared\_ptrp4 = new int(1);的写法是错误的

拷贝和赋值。拷贝使得对象的引用计数增加1，赋值使得原对象引用计数减1，当计数为0时，自动释放内存。后来指向的对象引用计数加1，指向后来的对象

4)  unique\_ptr“唯一”拥有其所指对象，同一时刻只能有一个unique\_ptr指向给定对象（通过禁止拷贝语义、只有移动语义来实现）。相比与原始指针unique\_ptr用于其RAII的特性，使得在出现异常的情况下，动态资源能得到释放。unique\_ptr指针本身的生命周期：从unique\_ptr指针创建时开始，直到离开作用域。离开作用域时，若其指向对象，则将其所指对象销毁(默认使用delete操作符，用户可指定其他操作)。unique\_ptr指针与其所指对象的关系：在智能指针生命周期内，可以改变智能指针所指对象，如创建智能指针时通过构造函数指定、通过reset方法重新指定、通过release方法释放所有权、通过移动语义转移所有权。

5)  智能指针类将一个计数器与类指向的对象相关联，引用计数跟踪该类有多少个对象共享同一指针。每次创建类的新对象时，初始化指针并将引用计数置为1；当对象作为另一对象的副本而创建时，拷贝构造函数拷贝指针并增加与之相应的引用计数；对一个对象进行赋值时，赋值操作符减少左操作数所指对象的引用计数（如果引用计数为减至0，则删除对象），并增加右操作数所指对象的引用计数；调用析构函数时，构造函数减少引用计数（如果引用计数减至0，则删除基础对象）。

6)  weak\_ptr 是一种不控制对象生命周期的智能指针, 它指向一个 shared\_ptr 管理的对象. 进行该对象的内存管理的是那个强引用的 shared\_ptr. weak\_ptr只是提供了对管理对象的一个访问手段。weak\_ptr 设计的目的是为配合 shared\_ptr 而引入的一种智能指针来协助 shared\_ptr 工作, 它只可以从一个 shared\_ptr 或另一个 weak\_ptr 对象构造, 它的构造和析构不会引起引用记数的增加或减少。

## C++ 98

std::auto\_ptr<std::string> ps (new std::string(str))；

## shared\_ptr

std::shared\_ptr包装了new操作符动态分别的内存，可以自由拷贝复制，基本上是使用最多的一个智能指针类型。

std::make\_shared封装了new方法，boost::make\_shared之前的原则是既然释放资源delete由智能指针负责，那么应该把new封装起来，否则会让人觉得自己调用了new，但没有调用delete，似乎与谁申请，谁释放的原则不符。C++也沿用了这一做法。

随着引用对象的增加std::shared\_ptr p2 = p1，指针的引用计数有1变为2，当p2退出作用域后，p1的引用计数变回1，当main函数退出后，p1离开main函数的作用域，此时p1被销毁，当p1销毁时，检测到引用计数已经为1，就会在p1的析构函数中调用delete之前std::make\_shared创建的指针。

* Class shared\_ptr 实现共享式拥有（shared ownership）概念。多个智能指针指向相同对象，该对象和其相关资源会在 “最后一个 reference 被销毁” 时被释放。为了在结构较复杂的情景中执行上述工作，标准库提供 weak\_ptr、bad\_weak\_ptr 和 enable\_shared\_from\_this 等辅助类。
* 多个智能指针可以共享同一个对象，对象的最末一个拥有着有责任销毁对象，并清理与该对象相关的所有资源。
* 支持定制型删除器（custom deleter），可防范 Cross-DLL 问题（对象在动态链接库（DLL）中被 new 创建，却在另一个 DLL 内被 delete 销毁）、自动解除互斥锁

### 实现原理

采用引用计数器的方法，允许多个智能指针指向同一个对象，每当多一个指针指向该对象时，指向该对象的所有智能指针内部的引用计数加1，每当减少一个智能指针指向对象时，引用计数会减1，当计数为0的时候会自动的释放动态分配的资源。

* 智能指针将一个计数器与类指向的对象相关联，引用计数器跟踪共有多少个类对象共享同一指针。
* 每次创建类的新对象时，初始化指针并将引用计数置为1
* 当对象作为另一对象的副本而创建时，拷贝构造函数拷贝指针并增加与之相应的引用计数。
* 对一个对象进行赋值时，赋值操作符减少左操作数所指对象的引用计数（如果引用计数为减至0，则删除对象），并增加右操作数所指对象的引用计数
* 调用析构函数时，构造函数减少引用计数（如果引用计数减至0，则删除基础对象）

### 智能指针的循环引用

循环引用是指使用多个智能指针share\_ptr时，出现了指针之间相互指向，从而形成环的情况，有点类似于死锁的情况，这种情况下，智能指针往往不能正常调用对象的析构函数，从而造成内存泄漏。举个例子：

#include <iostream>  
using namespace std;  
  
template <typename T>  
class Node  
{  
public:  
    Node(const T& value)  
        :\_pPre(NULL)  
        , \_pNext(NULL)  
        , \_value(value)  
    {  
        cout << "Node()" << endl;  
    }  
    ~Node()  
    {  
        cout << "~Node()" << endl;  
        cout << "this:" << this << endl;  
    }  
  
    shared\_ptr<Node<T>> \_pPre;  
    shared\_ptr<Node<T>> \_pNext;  
    T \_value;  
};  
  
void Funtest()  
{  
    shared\_ptr<Node<int>> sp1(new Node<int>(1));  
    shared\_ptr<Node<int>> sp2(new Node<int>(2));  
  
    cout << "sp1.use\_count:" << sp1.use\_count() << endl;  
    cout << "sp2.use\_count:" << sp2.use\_count() << endl;  
  
    sp1->\_pNext = sp2; //sp1的引用+1  
    sp2->\_pPre = sp1; //sp2的引用+1  
  
    cout << "sp1.use\_count:" << sp1.use\_count() << endl;  
    cout << "sp2.use\_count:" << sp2.use\_count() << endl;  
}  
int main()  
{  
    Funtest();  
    system("pause");  
    return 0;  
}  
//输出结果  
//Node()  
//Node()  
//sp1.use\_count:1  
//sp2.use\_count:1  
//sp1.use\_count:2  
//sp2.use\_count:2

从上面shared\_ptr的实现中我们知道了只有当引用计数减减之后等于0，析构时才会释放对象，而上述情况造成了一个僵局，那就是析构对象时先析构sp2,可是由于sp2的空间sp1还在使用中，所以sp2.use\_count减减之后为1，不释放，sp1也是相同的道理，由于sp1的空间sp2还在使用中，所以sp1.use\_count减减之后为1，也不释放。

sp1等着sp2先释放，sp2等着sp1先释放,二者互不相让，导致最终都没能释放，内存泄漏。

在实际编程过程中，应该尽量避免出现智能指针之间相互指向的情况，如果不可避免，可以使用弱指针—weak\_ptr，它不增加引用计数，只要出了作用域就会自动析构。

## unique\_ptr

unique\_ptr采用的是独享所有权语义，一个非空的unique\_ptr总是拥有它所指向的资源。

转移一个unique\_ptr将会把所有权全部从源指针转移给目标指针，源指针被置空；所以unique\_ptr不支持普通的拷贝和赋值操作，不能用在STL标准容器中；局部变量的返回值除外（因为编译器知道要返回的对象将要被销毁）。

如果你拷贝一个unique\_ptr，那么拷贝结束后，这两个unique\_ptr都会指向相同的资源，造成在结束时对同一内存指针多次释放而导致程序崩溃。

* class unique\_ptr 实现独占式拥有（exclusive ownership）或严格拥有（strict ownership）概念，保证同一时间内只有一个智能指针可以指向该对象。你可以移交拥有权。它对于避免内存泄漏（resource leak）——如 new 后忘记 delete ——特别有用。
* unique\_ptr 是 C++11 才开始提供的类型，是一种在异常时可以帮助避免资源泄漏的智能指针。采用独占式拥有，意味着可以确保一个对象和其相应的资源同一时间只被一个 pointer 拥有。一旦拥有着被销毁或编程 empty，或开始拥有另一个对象，先前拥有的那个对象就会被销毁，其任何相应资源亦会被释放。

## weak\_ptr

为了解决std::shared\_ptr在相互引用的情况下出现的问题而存在的，与std::shared\_ptr最大的差别是在赋值时，不会引起智能指针计数增加。

弱引用。引用计数有一个问题就是互相引用形成环（环形引用），这样两个指针指向的内存都无法释放。需要使用weak\_ptr打破环形引用。

weak\_ptr是一个弱引用，它是为了配合shared\_ptr而引入的一种智能指针，它指向一个由shared\_ptr管理的对象而不影响所指对象的生命周期，也就是说，它只引用，不计数。

如果一块内存被shared\_ptr和weak\_ptr同时引用，当所有shared\_ptr析构了之后，不管还有没有weak\_ptr引用该内存，内存也会被释放。

所以weak\_ptr不保证它指向的内存一定是有效的，在使用之前使用函数lock()检查weak\_ptr是否为空指针。

* weak\_ptr 允许你共享但不拥有某对象，一旦最末一个拥有该对象的智能指针失去了所有权，任何 weak\_ptr 都会自动成空（empty）。因此，在 default 和 copy 构造函数之外，weak\_ptr 只提供 “接受一个 shared\_ptr” 的构造函数。
* 可打破环状引用（cycles of references，两个其实已经没有被使用的对象彼此互指，使之看似还在 “被使用” 的状态）的问题

### 举例

std::shared\_ptr相互引用的问题示例：  
 创建了一个TestA和一个TestB的对象，但在整个main函数都运行完后，都没看到两个对象被析构。代码中智能指针ptr\_a中引用了ptr\_b，同样ptr\_b中也引用了ptr\_a，在main函数退出前，ptr\_a和ptr\_b的引用计数均为2，退出main函数后，引用计数均变为1，也就是相互引用。相互引用导致的问题就是释放条件的冲突，最终也可能导致内存泄漏。

std::weak\_ptr如何解决相互引用的问题:  
 ptr\_a 和ptr\_b在main函数中退出前，引用计数均为1，也就是说，在TestA和TestB中对std::weak\_ptr的相互引用，不会导致计数的增加。在TestB析构函数中，调用std::shared\_ptr tmp = m\_TestA\_Ptr.lock()，把std::weak\_ptr类型转换成std::shared\_ptr类型，然后对TestA对象进行调用。

## auto\_ptr（被 C++11 弃用）

主要是为了解决“有异常抛出时发生内存泄漏”的问题 。因为发生异常而无法正常释放内存。

auto\_ptr有拷贝语义，拷贝后源对象变得无效，这可能引发很严重的问题；而unique\_ptr则无拷贝语义，但提供了移动语义，这样的错误不再可能发生，因为很明显必须使用std::move()进行转移。

auto\_ptr不支持拷贝和赋值操作，不能用在STL标准容器中。STL容器中的元素经常要支持拷贝、赋值操作，在这过程中auto\_ptr会传递所有权，所以不能在STL中使用。

* unique\_ptr 用于取代 auto\_ptr
* 被 c++11 弃用，原因是缺乏语言特性如 “针对构造和赋值” 的 std::move 语义，以及其他瑕疵。

### 作用

1)  auto\_ptr的出现，主要是为了解决“有异常抛出时发生内存泄漏”的问题；抛出异常，将导致指针p所指向的空间得不到释放而导致内存泄漏；

2)  auto\_ptr构造时取得某个对象的控制权，在析构时释放该对象。我们实际上是创建一个auto\_ptr类型的局部对象，该局部对象析构时，会将自身所拥有的指针空间释放，所以不会有内存泄漏；

3)  auto\_ptr的构造函数是explicit，阻止了一般指针隐式转换为 auto\_ptr的构造，所以不能直接将一般类型的指针赋值给auto\_ptr类型的对象，必须用auto\_ptr的构造函数创建对象；

4)  由于auto\_ptr对象析构时会删除它所拥有的指针，所以使用时避免多个auto\_ptr对象管理同一个指针；

5)  Auto\_ptr内部实现，析构函数中删除对象用的是delete而不是delete[]，所以auto\_ptr不能管理数组；

6)  auto\_ptr支持所拥有的指针类型之间的隐式类型转换。

7)  可以通过\*和->运算符对auto\_ptr所有用的指针进行提领操作；

8)  T\* get(),获得auto\_ptr所拥有的指针；T\* release()，释放auto\_ptr的所有权，并将所有用的指针返回。

## auto\_ptr 与 unique\_ptr 比较

* auto\_ptr 可以赋值拷贝，复制拷贝后所有权转移；unqiue\_ptr 无拷贝赋值语义，但实现了move 语义；
* auto\_ptr 对象不能管理数组（析构调用 delete），unique\_ptr 可以管理数组（析构调用 delete[] ）；

# 多线程

## std::thread

std::thread为C++11的线程类，使用方法和boost接口一样，非常方便，同时，C++11的std::thread解决了boost::thread中构成参数限制的问题。

## std::atomic

std::atomic为C++11分装的原子数据类型。从功能上看，简单地说，原子数据类型不会发生数据竞争，能直接用在多线程中而不必我们用户对其进行添加互斥资源锁的类型。从实现上，大家可以理解为这些原子类型内部自己加了锁。

可以把原子操作理解成一种：不需要用到互斥量加锁（无锁）技术的多线程并发编程方式。  
原子操作：在多线程中不会被打断的程序执行片段。  
从效率上来说，原子操作要比互斥量的方式效率要高。

### 优点

* 执行效率比互斥量更高
* std::atomic来代表原子操作，是个类模板。其实std::atomic是用来封装某个类型的值的。

### 局限性

* 一般只能用于变量的一些简单操作，如++、–、+=、-=、&=、|=、^=，其他操作不一定可以执行。
* 通常在一些计数场景用的多。
* 互斥量的加锁一般是针对一个代码段，而原子操作针对的一般都是一个变量。

## std::mutex

类似mutex\_t，互斥锁。

## std::shared\_lock

读写锁。

## std::condition\_variable

C++11中的std::condition\_variable就像Linux下使用pthread\_cond\_wait和pthread\_cond\_signal一样，可以让线程休眠，直到被唤醒，再重新执行。线程等待在多线程编程中使用非常频繁，经常需要等待一些异步执行的条件的返回结果。

# 代码规范

## 服务器代码基本规范

整型一般使用int8\_t int16\_t int32\_t int64\_t uint8\_t uint16\_t uint32\_t uint64\_t，原则上不使用short int long

不使用Windows或Linux平台特有类型、特性、接口，常见的Windows下可编译通过而Linux下编译失败的情况：

使用“枚举名称::枚举值”的用法（应当直接使用枚举值或者类名::枚举值）

include路径使用了\或大小写错误（应当使用/和准确的大小写）

原则上不使用全局变量，基础代码除外。如需使用全局变量，尽量直接用全局变量，不用全局指针。全局变量命名前加g或者g\_

需要详细了解并掌握服务器的异常系统

不允许创建线程，如果有需求，提出公议后可以使用

## C++语言特性使用规范

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 类别 | 特性 | 年代 | 使用规范 | 说明 |
|  | 多重继承 | C++98 | 不可用 |  |
| C99兼容 | C99预定义宏 | C++11 | 不可用 | 无应用场景 |
| \_\_func\_\_ | C++11 | 可用 |  |
| \_Pragma | C++11 | 可用 |  |
| 变长参数与\_\_VA\_ARGS\_\_ | C++11 | 不可用 | 非类型安全 |
| 宽窄字符串连接 | C++11 | 不可用 | 项目不使用宽字符 |
|  | long long | C++11 | 不可用 | 使用项目自定义的int64类型 |
|  | 整型标准扩展 | C++11 | 非显式特性 |  |
|  | \_\_cplusplus | C++11 | 可用 |  |
|  | static\_assert | C++11 | 可用 |  |
|  | noexcept | C++11 | 不可用 | 使用不当可能导致宕机 |
|  | sizeof非静态成员 | C++11 | 可用 |  |
|  | friend语法扩展 | C++11 | 可用 |  |
|  | final/override | C++11 | 可用 |  |
|  | 模板函数默认模板参数 | C++11 | 可用 |  |
|  | extern template | C++11 | 可用 |  |
|  | 局部和匿名类型做模板实参 | C++11 | 可用 |  |
| 对象初始化 | 成员变量就地初始化 | C++11 | 可用 |  |
| 继承构造函数 | C++11 | 可用 |  |
| 委派构造函数 | C++11 | 可用 |  |
|  | 右值引用 | C++11 | 可用 |  |
|  | 移动语义move | C++11 | 可用 |  |
|  | 完美转发 | C++11 | 可用 |  |
|  | 显式转换操作符explicit | C++11 | 可用 |  |
|  | 初始化列表 | C++11 | 可用 |  |
|  | POD | C++11 | 可用 |  |
|  | 非受限联合体 | C++11 | 可用 |  |
|  | 强类型枚举 | C++11 | 可用 |  |
|  | 自定义字面量 | C++11 | 可用 |  |
|  | 内联名字空间 | C++11 | 可用 |  |
|  | using别名 | C++11 | 可用 |  |
|  | 一般化SFINAE | C++11 | 非显式特性 |  |
|  | 右尖括号改进 | C++11 | 可用 |  |
|  | auto | C++11 | 受限使用 | 以下情况可以使用：（可扩充） stl迭代器类型 lambda函数类型 |
|  | decltype | C++11 | 受限使用 |  |
|  | 追踪返回类型 | C++11 | 受限使用 |  |
|  | 范围for循环 | C++11 | 可用 |  |
|  | 智能指针 | C++11 | 可用 |  |
|  | 垃圾回收 | C++11 | 不可用 |  |
|  | 常量表达式constexpr | C++11 | 可用 |  |
|  | 变长模板参数 | C++11 | 可用 |  |
| 线程 | std::thread |  |  |  |
| std::mutex |  |  |  |
| std::atomic |  |  |  |
| std::condition\_variable |  |  |  |
| std::future |  |  |  |
| 内存模型 | C++11 | 受限使用 | 仅底层可以使用 |
| 线程局部存储 | C++11 | 受限使用 | 仅底层可以使用 |
|  | 快速退出 | C++11 | 不可用 |  |
|  | nullptr | C++11 | 可用 |  |
|  | 默认函数default/delete | C++11 | 可用 |  |
|  | lambda函数 | C++11 | 可用 |  |
|  | 数据对齐 | C++11 | 不可用 |  |
|  | 通用属性 | C++11 | 不可用 |  |
|  | Unicode | C++11 | 不可用 |  |
|  | 原生字符串字面量 | C++11 | 可用 |  |
|  | std::tuple |  |  |  |
|  | std::unordered\_map |  |  |  |
|  | std::unordered\_set |  |  |  |
|  | std::regex |  |  |  |
|  | std::random |  |  |  |
|  | std::ref |  |  |  |
|  | std::function |  |  |  |
|  | 用于元编程的类别属性 |  |  |  |
|  | std::array |  |  |  |
|  | std::chrono |  |  |  |
|  | std::ratio |  |  |  |

## 基本思路

正确性是第一位的，重要性高于运行效率，不要过早优化。

## 内存和指针使用规范

除底层逻辑之外，不动态分配内存。

只在必要时，才使用指针，尽量少使用指针。

可以使用引用时，尽量使用引用。

防止引用已经释放的内存或者无效的内存，使用指针前必须判空。

指针在被delete之后，必须置为nullptr。

使用new[]分配的指针，必须用delete[]释放，使用new分配的指针，必须使用delete释放。

禁止指针与其他类型互转。

不得返回局部变量或者其他临时变量的指针或引用。

不重载operator new delete。

不使用std::auto\_ptr。

## 数组与使用规范

使用solar::array代替原生数组，除用于格式化输出char[]外，不使用原生数组。

使用solar::vector代替std::vector，不使用std::vector。

访问数组或者vector之前，必须判断下标。

## 异常体系

异常体系非常重要，必须数量掌握

SOL\_ASSERT在判定失败后会打印一条异常日志并且抛出异常

SOL\_VERIFY在判定失败后会打印一条异常日志但不会抛出异常

仔细理解一下宏定义：

SOL\_ASSERT、SOL\_VERIFY

\_\_SOL\_TRACE、SOL\_TRACE\_

\_\_SOL\_GUARD、SOL\_GUARD\_

\_\_SOL\_GUARD\_EX、SOL\_GUARD\_CATCHED、SOL\_GUARD\_EX\_

要求在熟悉以上宏定义之后，能够确切回答以下问题：

阅读下面代码并且回答以下问题：

Return true会被执行吗？return false会被执行吗？

将SOL\_ASSERT换成SOL\_VERIFY后呢？return true会被执行吗？return false会被执行吗？

将\_\_SOL\_TRACE、SOL\_TRACE\_\_换成\_\_SOL\_GUARD、SOL\_GUARD\_\_后呢？return true会被执行吗?return false会被执行吗？

\_\_SOL\_TRACE

SOL\_ASSERT(false, “”);’

Return true;

SOL\_TRACE\_\_

Return false;

\_\_SOL\_GUARD SOL\_GUARD\_与\_\_SOL\_TRACE SOL\_TRACE\_\_的区别是什么？

\_\_SOL\_GUARD\_EX \_\_SOL\_GUARD\_CATCHED SOL\_GUARD\_EX\_\_与\_\_SOL\_GUARD、SOL\_GUARD\_\_区别是什么？

## 类设计规范

类不需以“C”做开头。

类成员变量以m\_开头。

构造函数无比正确初始化各个成员变量，可抛出异常，但要保证资源不会泄露。

单参数构造函数声明为explicit。

析构函数必须正确释放资源，绝不能抛出异常，避免宕机。

子类重置虚函数不省略virtual关键字，必要时增加override关键字。

类中有引用类型，比如指针、引用、句柄等，则此类或者声明为noncopyable，或者自行实现拷贝构造函数和赋值操作符。

对delete基类指针这样的操作，必须保证析构函数为虚函数。

operator=需要规避右值是自己的情况。

不使用多重继承。

## 智能指针使用规范

智能指针是一个持有动态分配Object的指针的Object，其行为和C++的裸指针十分相似，且可在适当的时候自动delete所持有的Object，智能指针非常有助于在发生异常的时候确保销毁动态分配的对象。另外，在多个Owner之间共同持有一个Object时，也十分有用。从概念上来讲，智能指针就像一个指向Object的指针，并负责在不需要的时候delete掉指向的Object。

智能指针是一个对象，其主要作用是适时自动释放，异常安全。

禁止智能指针和裸指针混合使用。

禁止获取智能指针内含的裸指针，禁止使用智能指针的get、reset接口。

禁止使用裸指针变量初始化智能指针，建议使用boost::shared\_ptr<TypeName> sp(new TypeName)的形式初始化。

智能指针的类型转换使用（static/const/dynamic)\_pointer\_cast<T>的形式

智能指针可作为函数参数，传值即可不使用传引用。另外，一般不使用智能指针的引用。

智能指针可作为函数返回值，但必须是值返回而不是引用返回，时刻记住智能指针是一个对象。

智能指针可作为标准容器的元素。

智能指针相对裸指针在时间、空间效率上略有下降，但是在可接受范围内。

智能指针判空方式是if(ptr)或者if(!ptr)，或者跟nullptr判断。

智能指针变量的命名，建议是xxxPtr，不建议为pXXX。

## STL使用规范

容器外部不能以指针或引用形式引用容器内元素，如果必须这样，使用指针或智能指针作为容器元素。

在遍历容器的过程中，做出改变容器结构的操作（比如添加、删除）后，当前迭代器可能失效，必须重新获取有效的迭代器。

（此处应该有代码）

重载operator<操作符时，元素相等的的情况，必须返回false。

服务器端原则上，只有Routine底层才能使用STL。

总体原则是尽量不用，除非是能带来极大开发效率的提升。

一般STL用于长期存在对象的成员变量，例如服务器、场景，其他情况不要使用。

## CommonData、CommonFlag使用规范

CommonData和CommonFlag是公共资源，用于小量级的存储，避免为单独功能增加存储而使得逻辑复杂。

大家需要用到CommonData或者Commonflag存储的时候，需要向特定人员（目前是贺文鹏）申请ID，拿到指定ID之后再在程序中使用。

拿到ID之后需要在代码中增加这个枚举的定义（服务器是必须加的，客户端C#和lua中看使用情况增加必要枚举定义）。

同时注意这个存储是否需要即时同步给客户端，可以由不同的设定。

现在CommonData目前包含32/64位两种，数量有限请节约使用。

## 其他注意事项

* 1. switch语句中所有的case必须有break。
  2. 变量使用前必须进行初始化。
  3. 用宏定义表达式时，要使用完备的括号，将宏定义的多条表达式放在大括号里。
  4. 除法和求模运算时，对于除数变量需要做非0判断。
  5. 类型装换前必须进行有效性判断，不允许直接强转类型，优先使用C++的转型风格，禁止使用const\_cast和reinterpret\_cast。
  6. 尽可能使用const。
  7. 使用前置声明，避免在头文件中包含过多文件。
  8. 向下类型转化前，必须判断目标的类型，确定可以转换再转换。
  9. 使用模板还是面对对象的设计模式，关键看不同类型对行为是否产生影响。
  10. 消除所有的warning。
  11. 注意溢出错误、除0错误、数组越界。
  12. 不得使用std::vector<bool>，使用solar::bit。
  13. 可以使用solar::string代替char[]。
  14. 取当前时间使用solar::ansitime\_sec和solar::ansitime\_milsec，分别是取当前秒和毫秒。
  15. 小心使用dynamic\_cast，用于类层次间的上行转换和下行转换，下行转换时具有类型检查的功能。只能用于具有虚函数的类，否则会编译报错，这是因为运行时类型检查需要运行时类型信息，而这个信息存储在虚函数表中。
  16. 注意服务器在处理CG消息包的时候，会直接拿到客户端发来的数据。这些数据需要经过校验才能使用，恶意发送的数据不经过校验可能会造成巨大影响。
  17. 注意服务器在处理玩家利益相关操作的时候，需要先处理扣除，再处理给予。防止扣除不成功，但是直接给予利益的情况。
  18. 独立的功能应该在Config文件中增加开关，可以在线动态控制功能的运转，控制力度要跟策划商定。可能也会包含客户端的屏蔽。
  19. 注意功能是否涉及跨天、跨周、跨月、跨年、跨上下线等，多注意考虑极端情况。
  20. 可使用金钱兑换到的货币（比如元宝）禁止各种途径的放出（寄售行类除外）。
  21. 注意服务器避免短时间内大量复杂操作（比如某些全服操作，大循环，大量玩家操作并发），可能会影响性能。
  22. 尽量避免玩家利益回滚操作，可能造成刷钱的漏洞。
  23. pb里面定义的枚举，服务器使用没有的枚举值强制转换，导致PB底层抛异常，更严重导致宕机，例如：枚举里只有1,2 但是强制0转为这种枚举就会出现抛异常