文档目的：记录关于c++不明白的东西,不懂就在这里提问题，然后在这里进行解答

c++编程规范：

<https://zh-google-styleguide.readthedocs.io/en/latest/google-cpp-styleguide/>

# 一.面向对象

# 二.基础知识

# 三.STL

## 1. 静态变量static

### 1.1静态变量的目的

**为了数据的共享**

### 1.2 静态数据成员

1. 静态数据成员可以实现多个对象之间的数据共享，是类的所有对象共享的成员，而不是对象的成员。
2. 静态数据成员可以节省内存，对于多个对象来说，静态数据成员只存在一处，只要静态数据成员更新一次，所有对象存取更新后的相同的值。
3. 静态数据成员在对象生成之前已经分配了空间（在编译期间），在程序结束释放空间。
4. 既可以通过类名进行引用，也可以使用对象名对静态成员进行引用。
5. 注意事项：初始化使用下面模板，不用加变量的权限

<数据类型> <类名>::<静态数据成员名> = <值>

引用静态数据成员使用下面模板：

<类名>::<静态成员名>

### 1.3 静态局部变量

1. 静态局部变量在函数中进行定义，不像自动变量在调用函数时存在，退出函数消失静态局部变量始终是存在的，生命周期是整个源程序。而**该变量只在第一次进入函数的时候进行初始化，后面再进行调用会跳过该初始化**。
2. 静态局部变量的生命周期是整个源程序，但是作用域还是与自动变量相同，在函数中可以使用，即使退出函数，变量任然存在，但是不能进行调用。
3. 对基本类型的静态局部变量若在说明时未赋以初值，则系统自动赋予0值。而对自动变量不赋初值，则其值是不定的。
4. 虽然调出函数不能使用局部静态变量，但是局部静态变量任然保存的是前次调用后留下来的值。

### 1.4 静态全局变量

1. 全局变量本身是静态存储方式，静态全局变量也是静态存储方式，两者在存储方式上并无不同。
2. 这两者的区别是，非静态全局变量的作用域是整个源程序，当一个源程序由多个源文件组成，非静态的全局变量在其他源文件也是有效的。但是静态的全局变量只在定义该变量的源文件中是有效的。

### 1.5 静态方法

静态方法不能调用一般成员。可以使用对象引用

## 2. 为什么使用初始化列表，而不使用赋值

### 2.1 构造函数执行的顺序

构造函数初始化分为两个步骤分别是，初始化阶段和计算阶段

初始化阶段：在类中，所有的成员必须进行初始化，初始化阶段可以显示的对成员变量进行初始化，初始化列表正是在这个阶段。

计算阶段：计算阶段主要进行处理，赋值操作以及其他操作，计算阶段一定在初始化阶段后完成。  
**using namespace** std;  
#include**<iostream>  
class** Base  
{  
**public**:  
 Base()cout<<**"Constructor for Base"**<<endl;}  
 Base(**const** Base& t){cout<<**"Copy Constructor for Base"**<<endl;a=t.a;}  
 Base& **operator**=(**const** Base& t){cout<<**"Assignment for Base"**<<endl;a=t.a;**return** \***this**;}  
**private**:  
 **int** a;  
};  
**class** Test  
{  
**public**:  
 Test(Base& t)b=t;}  
**private**:  
 Base b;  
};  
**int** main(){  
 Base b;  
 std::cout<<**"Test is running!"**<<std::endl;  
 Test t(b);  
}

上面例子输出：多了一次默认构造函数的调用

**"Constructor for Base"**

**“Test is running”**

**"Constructor for Base"//初始化阶段**

**"Assignment for Base"//计算阶段**

如果将Test改为

**class** Test{  
**public**:  
 Test(Base& t):b(t){}  
**private**:  
 Base b;  
};

则输出为：

**"Constructor for Base"**

**“Test is running”**

**"Copy Constructor for Base"//初始化阶段**

## 三.c字符串和以及字符串的种类

**c++中提供两种字符串的表示形式，一个是c风格字符串，一个是c++引入的string类型字符串。**

### 3.1 c风格字符串

字符串是一null字符'\0'结尾的一字符数组。初始化c字符串有两种方式:

一种是:char greeting[6] = {'h','e','l','l','o','\0'}

一种是:char greeting[6] = "hello";

注释：也可以将6进行省略。对字符数组进行初始化可以使用整体形式像第二种，但是不能对字符数组进行整体赋值

error: char greeting[6]; greeting = "hello"

### 3.2 字符指针

字符串指针变量是存放字符串的首地址为首的一串连续内存空间，并且以'\0'结束，

char \* ps = "hello";

顺序是1.分配内存给指针变量，2.分配内存给"hello",3.将字符串的首地址赋给指针变量。

### 3.3 string类型

***今天犯的错误***:

char \*p = new char;

p = "hello"

delete p;

这是错误的，第一：因为p开始指向的动态存储空间，但是又将p指向了静态存储空间，而静态存储空间是不能delete的。第二：将p指向了另外一个地址导致的是，new出来的存储空间没有进行delete导致内存泄露。

## 四.静态成员变量的初始化方式

1. **静态成员是整数或者是枚举型const可以在类声明中初始化。**
2. **其他的在类的定义中进行初始化**

## 五.构造函数和构造函数的调用？

5.1 copy constructor:在三种情况会被调用**。**

1. 通过使用另一个同类型的对象来进行初始化新创建的对象。
2. 复制对象把它作为参数传递给函数
3. 复制对象，并从函数返回这个对象。

**下面情况都将进行调用copy constructor：**

stringbad ditto(motto)

stringbad metoo = motto

stringbad also = stringbad(motto)

stringbad \* pstringbad = new stringbad(motto)

### 5.2 默认的复制构造函数的功能

默认构造函数一个一个**复制非静态成员**（成员复制也称为浅复制），复制的是成员的值。

stringbad new = old

**等价于：**

stringbad new;

new.str = old.str;

new.len = old.len

**Exception:**1.如果成员变量是类变量，则进行调用类的复制构造函数来复制构造函数。2.静态函数和静态变量不进行复制。

1. **当类成员是new初始化的，指向数据的指针，必须定义复制构造函数进行深度copy**，复制指向的数据而不是指针，这称为深度copy，浅复制只是复制了指针的信息。只有这样不同的对象在进行析构的时候，才能进行析构自己的成员变量，不然会导致double free or curruption.

### 5.3左值引用，右值引用？ move constructor作用？与copy constructor的区别？

**问题一**：

**左值和右值**：左值是**有名字的变量**或者是对象，可以**进行赋值**，可以在多条语句中进行使用。右值是**没有名字**的变量，只能在一条语句中出现，**不能进行赋值**。

**左值引用和右值引用**：左值引用的声明符号位&，右值引用的声明符号位&&。

void process\_value(int & value){ cout<<"L value"<<value<<endl;}

void process\_value(int && value){ cout<<"R value"<<value<<endl;}

int main()

{

int i =0 ;

process\_value(i);

process\_value(1);

return 0;

}

**注：有个问题是x是右值引用，指向的是右值，那x本身是右值还是左值呢？答案是x是左值。下面的结果会输出**

**L value 0, L value 0**

void process\_value(int & value){ cout<<"L value"<<value<<endl;}

void process\_value(int && value){ cout<<"R value"<<value<<endl;}

int main()

{

int i =0 ;

process\_value(i);

int && j = 0;

process\_value(j);

return 0;

}

**右值引用的目的：**

1. 简单的说，右值引用是为临时变量续命，因为右值在表达式结束后就消亡了，如果继续想使用右值，就必须调用昂贵的拷贝构造函数。
2. 支持转移语义，转移语义可以将资源从一个对象转移到另一个对象中，可以减少临时对象的创建，拷贝以及销毁，能够大幅度提高c++应用程序的性能，临时对象的维护对性能右严重影响。
3. 转移语义和拷贝语义是相对的，可以类比拷贝和剪切，剪切的速率肯定比拷贝的速率要快的多。
4. 对于右值的拷贝和赋值会调用转移构造和转移赋值，如果没有进行定义就会调用拷贝构造和赋值语句。

下面是移动构造和移动赋值的例子：

class MyString {

private:

char\* \_data;

size\_t \_len;

void \_init\_data(const char \*s) {

\_data = new char[\_len+1];

memcpy(\_data, s, \_len);

\_data[\_len] = '\0';

}

public:

MyString() {

\_data = NULL;

\_len = 0;

}

MyString(const char\* p) {

\_len = strlen (p);

\_init\_data(p);

}

MyString(const MyString& str) {

\_len = str.\_len;

\_init\_data(str.\_data);

std::cout << "Copy Constructor is called! source: " << str.\_data << std::endl;

}

MyString& operator=(const MyString& str) {

if (this != &str) {

\_len = str.\_len;

\_init\_data(str.\_data);

}

std::cout << "Copy Assignment is called! source: " << str.\_data << std::endl;

return \*this;

}

virtual ~MyString() {

if (\_data) free(\_data);

}

};

int main() {

MyString a;

a = MyString("Hello");

std::vector<MyString> vec;

vec.push\_back(MyString("World"));

}

在上面的程序中，MyString("Hello")和MyString("World")是右值，但是没有定义转移构造和转移赋值，所以上面会先调用赋值和拷贝构造。

因为右值变量在语句结束后不再使用，所以完全可以不用拷贝构造和赋值，而使用移动构造函数和移动赋值,下面添加move constructor和move assignment在上面代码将调用这两个函数，将避免了内存的浪费。

**//移动构造函数**

MyString(MyString&& str) {

std::cout << "Move Constructor is called! source: " << str.\_data << std::endl;

\_len = str.\_len;

\_data = str.\_data;

str.\_len = 0;

str.\_data = NULL;

}

**//移动赋值**

MyString& operator=(MyString&& str) {

std::cout << "Move Assignment is called! source: " << str.\_data << std::endl;

if (this != &str) {

\_len = str.\_len;

\_data = str.\_data;

str.\_len = 0;

str.\_data = NULL;

}

return \*this;

}

## 六.返回对象还是引用

### 6.1 返回指向const对象的引用

如果函数返回传递给他的对象，可以通过返回引用来进行提高效率，因为**返回对象需要调用copy constructor,**而返回引用不会。下面有两个版本：

1.返回对象会调用copy constructor,返回引用不会，version2做的工作最少，效率更高。

2. version2中，传入的参数为const引用返回也必须是const引用。

version1:

**Point Max(const Point &p1,const Point &p2){**

**if(p1.x > p2.x){**

**return p1;**

**}else{**

**return p2;**

**}**

**}**

**version2:**

**const Point & Max(const Point &p1,const Point &p2){**

**if(p1.x > p2.x){**

**return p1;**

**}else{**

**return p2;**

**}**

**}**

### 6.2 返回对象

如果被返回的对象是函数中的局部变量，则不能按照去引用他，因为被调用的函数执行完毕，局部变量已经被析构。则**必须返回对象**。则存在调用copy constructor，这是不可避免的。

**注释**：gcc后优化方式是不一样

C func(){

C tmp;

return tmp;

}

如果调用C newC = func(),则一次copy constructor都不会调用，因为C函数中tmp的地址和newC地址是一样的。也就是编译器在func函数结束的时候，不会撤销这个对象，而将newC和tmp关联了起来。也就是说现在的编译器优化程度很高。。6666

## 7. 析构函数什么时候将被调用？

1. 如果对象是动态变量，则当执行完定义该对象的***程序块***的时候，将调用该对象的析构函数
2. 如果对象是静态变量，则在程序结束时候，将调用对象的析构函数
3. 如果对象是new创建的，则仅当显式使用delete删除对象时，其析构函数才被调用。

class Act{};

'''

Act act;

''''

int main(){

Act \* pt = new Act;

{

Act ptt;

} **//执行到代码块最后，ptt将执行析构调用**

delete pt;**//delete后调用析构函数，pt**

}//程序执行完，act才调用析构

## 8. 析构函数为什么常常是虚的？

当使用指针的时候，加入指针指向的对象是基类对象，

1. 如果析构函数为虚的，则delete会先释放子类然后在释放基类
2. 如果析构函数不为虚的，则delete只会释放基类对象不会释放子类对象，导致内存泄漏。代码如下：

testcons \* p = new test();

delete p;

class **testcons**{

public:

testcons(){cout<<"construct testcons()"<<endl;};

~testcons(){cout<<"desconstruct testcons()"<<endl;};

};

class **test** : public testcons{

public:

test(){cout<<"construct test()"<<endl;}

~test(){cout<<"desconstruct test()"<<endl;}

};

但是为什么如果是虚的就会释放子类呢？？

## 9. const成员变量初始化方式？

1. 在类内部使用const关键字来声明const数据成员，const数据成员的值不能进行修改，初始化的方式必须使用**初始化列表**，不能在构造函数里面进行赋值。
2. 每一个构造函数都需要初始化这个const对象，并且复制构造函数也需要初始化const对象

## 10.有哪些智能指针？有什么区别？如何选择智能指针？注意的问题？

1. 智能指针解决的问题：

下面的函数中new出了一个指针，但是在函数结束的时候，p变量将会被回收，但是p指向的堆内存没有被回收，所以**需要记住**使用delete回收。智能指针需要解决的问题就是解决偶尔会忘记delete的情况。智能指针是在对象过期的时候，让其析构函数删除指向的内存。

void remodel(string & s){

string \* p = new string(s);

\*s = \*p;

delete p;

return;

}

1. 智能指针分类？区别

auto\_ptr，unique\_ptr, shared\_ptr,

**auto\_ptr,和unique\_ptr**有所有权的概念，对于特定的对象只有一个智能指针可以拥有他，因此只有拥有该对象的才会delete该对象。所以下面的程序是错误的

auto\_ptr<string> ptr1(new string("liangsun"))

auto\_ptr<string> ptr2;

ptr2 = ptr1;//在运行的时候会报错，因为ptr2获得拥有权

cout<<\*ptr1;

unique\_ptr<string> ptr3(new string(liangsun))

unique\_ptr<sgtring> ptr4;

ptr4 = ptr3;//编译期间将报错，因为ptr4获得拥有权

shared\_ptr<string> ptr3(new string(liangsun))

shared<string> ptr4;

ptr4 = ptr3;//正确

**shared\_ptr**跟踪引用的特定对象的数量，称为引用计数，当赋值的时候，计数会加1，变量过期将减1,仅当最后一个对象过期才会delete

**注：1.**有一种情况的赋值是允许的如下：因为当demo函数结束的时候，temp变量将被删除，返回的是tmep的副本，并且没有在使用temp的机会。

auto\_ptr<string> demo(const char \* p){

auto\_ptr<string> temp(new string(p));

return temp;

}

auto\_ptr<string> d;

d = demo("liangsun");

1. 如何选择智能指针？

shared\_ptr:如何程序中出现，多个指向同一个对象的指针，如指针数组，求出最大值和最小值

unique\_ptr:如果程序中**不会出现**多个同一个对象的指针。可以使用unique\_ptr

1. 注意的问题？
2. 不能将一个不是new出来的对象赋值给智能指针，如下面的代码是不允许的，因为vacation不是new出来的，**因此delete不能用于非堆内存**

string vacation("vacation");

shared\_ptr<string> ptr(&vacation)

1. move如何使用？

## 11. explicit 关键字？为什么需要使用该关键字？

1. **explicit:显示的**，与其相反的是non-explicit. c++中，如果构造函数中**只有一个参数**，那么编译器在编译的过程中会做一个隐式的转换，将该构造函数对应的数据类型转化为该对象。下面的代码可以将一个整数转化为一个对象。如果添加explicit则编译器不能让其通过。
2. google c++规范提到explicit的有点是可以防止不可适宜的转换，缺点无。所以google规定所有单参数的构造函数都必须是显式的。

class test{

private:

int \_i;

public:

**explicit** test(int i):\_i(i){}

};

int main()

{

test j = 1;

}

## 12. 为什么将析构函数声明为default?

**default可以解决两个问题，**1.减轻程序员的工作量，2.编译器将为显式声明的函数自动生成函数体。并且编译器生成的函数体比自己写的函数体效率要高。

13. extern用法以及作用？

14.define函数定义：

13. override 关键字的作用？

override:是保留字表示当前函数重写了基类的虚函数，该保留字有如下作用：

1. 在函数较多的情况下，可以提醒读者，该函数是重写了基类虚函数，不是派生类自己定义的。
2. 强制让编译器检查该函数是否重写了基类的函数。

14. vector assign用法？

可以对vector进行赋值，如下代码，b的结果为1

vector a,b;

a.push\_back(1);

a.push\_back(2);

b = a.assign(a.begin(),a.end()-1);

15. std::move和std::forward用法？

16. unique\_ptr和optional如何选择？

17. static\_cast如何？