目录

**[c++ part](#_Toc19269_WPSOffice_Level1)** **[3](#_Toc19269_WPSOffice_Level1)**

[1-1多态 & virtual](#_Toc20096_WPSOffice_Level1) [3](#_Toc20096_WPSOffice_Level1)

[1-2基类用virtual虚析构函数:](#_Toc31243_WPSOffice_Level1) [4](#_Toc31243_WPSOffice_Level1)

[1-3 虚继承](#_Toc14903_WPSOffice_Level1) [4](#_Toc14903_WPSOffice_Level1)

[1-4 virtual 应用场景](#_Toc29625_WPSOffice_Level1) [5](#_Toc29625_WPSOffice_Level1)

[1-5抽象类](#_Toc20944_WPSOffice_Level1) [6](#_Toc20944_WPSOffice_Level1)

[1-6 虚函数底层实现](#_Toc28071_WPSOffice_Level1) [6](#_Toc28071_WPSOffice_Level1)

[1-7 c++ 中的 overload overwriter](#_Toc13828_WPSOffice_Level1) [6](#_Toc13828_WPSOffice_Level1)

[1-8 重载与重写（覆盖）区别](#_Toc3763_WPSOffice_Level1) [6](#_Toc3763_WPSOffice_Level1)

[1-9 const](#_Toc7933_WPSOffice_Level1) [7](#_Toc7933_WPSOffice_Level1)

[1-10引用 &与指针区别：](#_Toc20490_WPSOffice_Level1) [7](#_Toc20490_WPSOffice_Level1)

[1-11 Vector 越界 map 清内存](#_Toc17494_WPSOffice_Level1) [7](#_Toc17494_WPSOffice_Level1)

[1-12 c++内存的注意](#_Toc2440_WPSOffice_Level1) [8](#_Toc2440_WPSOffice_Level1)

[1-13 new() malloc区别 delete 和delete[]](#_Toc27231_WPSOffice_Level1) [8](#_Toc27231_WPSOffice_Level1)

[1-14 c++ 11 智能指针, lamed ,function bin](#_Toc12928_WPSOffice_Level1) [8](#_Toc12928_WPSOffice_Level1)

[1-15 C++ Static](#_Toc26206_WPSOffice_Level1) [8](#_Toc26206_WPSOffice_Level1)

[1-16 全局变量和局部变量有什么区别？是怎么实现的？](#_Toc370_WPSOffice_Level1) [9](#_Toc370_WPSOffice_Level1)

**[QT part](#_Toc14808_WPSOffice_Level1)** **[10](#_Toc14808_WPSOffice_Level1)**

[2-1 信号和槽](#_Toc10931_WPSOffice_Level1) [10](#_Toc10931_WPSOffice_Level1)

[2-2 Qt线程的两种使用方式](#_Toc9516_WPSOffice_Level1) [11](#_Toc9516_WPSOffice_Level1)

[2-3 Qt的刷新机制](#_Toc15638_WPSOffice_Level1) [11](#_Toc15638_WPSOffice_Level1)

[2-5 Qt的事件管理机制，（从底层窗口传到顶层窗口）](#_Toc21512_WPSOffice_Level1) [12](#_Toc21512_WPSOffice_Level1)

[Gstreamer part](#_Toc21780_WPSOffice_Level1) [12](#_Toc21780_WPSOffice_Level1)

[3-1 Gstreamer播放视频](#_Toc6781_WPSOffice_Level1) [12](#_Toc6781_WPSOffice_Level1)

**[network part](#_Toc25097_WPSOffice_Level1)** **[14](#_Toc25097_WPSOffice_Level1)**

[4-1 UDP通信](#_Toc12223_WPSOffice_Level1) [14](#_Toc12223_WPSOffice_Level1)

[4-2 Tcp/ip 握手过程](#_Toc22140_WPSOffice_Level1) [14](#_Toc22140_WPSOffice_Level1)

[4-3 HTTP](#_Toc31700_WPSOffice_Level1) [14](#_Toc31700_WPSOffice_Level1)

[4-4 加密传输](#_Toc9106_WPSOffice_Level1) [14](#_Toc9106_WPSOffice_Level1)

[4-5组播广播](#_Toc30648_WPSOffice_Level1) [14](#_Toc30648_WPSOffice_Level1)

**[Linux 操作系统进程线程](#_Toc8467_WPSOffice_Level1)** **[14](#_Toc8467_WPSOffice_Level1)**

[5-1 进程同步方式](#_Toc23500_WPSOffice_Level1) [14](#_Toc23500_WPSOffice_Level1)

[5-2 线程同步方式](#_Toc20146_WPSOffice_Level1) [14](#_Toc20146_WPSOffice_Level1)

# c++ part

## 1-1多态 & virtual

**什么是多态，怎么实现多态的？**

1. 多态即：一个接口多个实现
2. 编译时多态（静态多态）： 通过重载实现，编译器决定使用那个可执行的代码块。
3. 运行时多态（动态多态）： 用过**继承机制和虚函数**实现，通过指向派生类的基类指针或引用，访问派生类中同名覆盖成员函数。

## 1-2基类用virtual虚析构函数:

**- C++中基类采用virtual虚析构函数是为了防止内存泄漏**。

详解：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。

当使用基类指针指向子类的对象时，delete这个指针时不会发生动态绑定，只会调用基类的析构函数， 而不调用子类的析构函数，因此没有释放子类的内存，发生内存泄漏。

- 虚函数即动态绑定的是怎么实现的？

1. 为每个含义虚函数的类创建一个虚函数表VTable，存到常量区，依次存放虚函数的地址。对于每个派生类来说，如果没有重写基类的虚函数，那么派生类的虚函数表中的函数地址还是基类的那个虚函数地址。

（2） 为每个含有虚函数的对象创建一个指向VTable的指针VPtr，所以说同类对象的VPtr是一样的。

（3） 当基类指针指向派生类时，放生了强制转换，基类的指针指向了派生类的VPtr，这样当pBase->func()时，就可以调用派生类的func()了。

（4）没有虚函数的类也就没有VTable表了，或者这个表为空。这样基类指针自然调用不到派生类的函数了。

（5）父类指针指向派生类对象作用

为了实现C++的多态性。比如简单工厂模式，一开始我们并不知道需要生产哪种产品，也就不知道返回的是什么类型的产品，此时只能用基类指针去指向返回值。

## 1-3 虚继承

**再多继承的情况下，防止类继承关系中成员访问的二义性。**

在多继承下，虚继承就是为了解决菱形继承中，例如class A，为公共基类，B,C都继承了A，D继承了B,C，那么D关于 A的引用只有一次，而不是 普通继承的 对于A引用了两次。

代码详解：

class A{

public:

int t;

A(int a)

{

t = a;

}

void fun();

};

class B:virtual public A

{

public:

B(int a, int b) :A(a+10)

{

t1 = b;

}

~B();

int t1;

};

class C :virtual public A

{

public:

C(int a,int c):A(a+20)

{

t2 = c;

}

~C();

int t2;

};

class D :public B,public C

{

public:

D(int a,int b,int c,int d) :B(a,b),C(a,c),A(a){}//在此必须要给虚基类传参

~D();

};

//C++编译系统在实例化D类时，只会将虚基类的构造函数调用一次，忽略虚基类的其他派生类（class B，class C）对虚继承的构造函数的调用，从而保证了虚基类的数据成员不会被多次初始化。

## **1-4 virtual 应用场景**

当我们一个**基类指针指向了子类**，或者引用子类那么我们希望调用函数的时候是调用子类的函数，

但是实际情况却不是这样。这时候就需要virtual关键字了

## **1-5抽象类**

# 纯虚函数： 含有 它的类叫抽象类 ，不能被实例化 即new()

# 纯虚函数子类必须实现它

## **1-6 虚函数底层实现**

# **实现原理：**虚函数表+虚表指针

# **关键字：虚函数底层实现机制；虚函数表；虚表指针**

# 编译器处理虚函数的方法是：为每个类对象添加一个隐藏成员，隐藏成员中保存了一个指向函数地址数组的指针，称为虚表指针（vptr），这种数组成为虚函数表（virtual function table, vtbl），即，**每个类使用一个虚函数表，每个类对象用一个虚表指针。**

# **举个例子：**基类对象包含一个虚表指针，指向基类中所有虚函数的地址表。派生类对象也将包含一个虚表指针，指向派生类虚函数表。看下面两种情况：

# 如果派生类重写了基类的虚方法，该派生类虚函数表将保存重写的虚函数的地址，而不是基类的虚函数地址。

# 如果基类中的虚方法没有在派生类中重写，那么派生类将继承基类中的虚方法，而且派生类中虚函数表将保存基类中未被重写的虚函数的地址。注意，如果派生类中定义了新的虚方法，则该虚函数的地址也将被添加到派生类虚函数表中。

## **1-7 c++ 中的** **overload overwriter**

C++增加了重载（overloaded）、内联（inline）、const 和 virtual

四种新机制。其中重载和内联机制既可用于全局函数也可用于类的成员函数，**const 与**

**virtual 机制仅用于类的成员函数**

## **1-8 重载与重写（覆盖）区别**

**重载：**

成员函数被重载的特征：

（1）相同的范围（在同一个类中）；

（2）函数名字相同；

（3）参数不同；

（4）virtual 关键字可有可无。

覆盖是指派生类函数覆盖基类函数，特征是：

（1）不同的范围（分别位于派生类与基类）；

（2）函数名字相同；

（3）参数相同；

（4）基类函数必须有 virtual 关键字。

# 函数隐藏：指派生类的函数屏蔽了与其同名的基类函数 1.两个函数在不同的类域 2.函数名称相同 3.函数参数不同 4.如果派生类函数与基类函数参数相同，但是在基类函数中没有virtual关键字，发生函数隐藏

## **1-9 const**

**- const 与 #define 区别**

1)const **有类型检测** 后者没有

2)有些集成化的调试工具可以对 const 常量进行调试,但是不能对宏常量进行调试。

**- 类中的const**

1)企图在类声明中初始化 const 数据成员 错误

2)数据成员的初始化只能在类构造函数的**初始化表中进行** A::A(int size) : SIZE(size) // 构造函数的初始化表

3)整个类中都恒定的常量呢?别指望 const 数据成员了,应该用类中 的枚举常量来实现：enum { SIZE1 = 100, SIZE2 = 200};

枚举常量不会占用对象的存储空间,它们在编译时被全部求值。枚举常量的缺点是: 它的隐含数据类型是整数,其最大值有限,且不能表示浮点数(如 PI=3.14159)。

return int(x + y); // 创建一个临时变量并返回它

应该：

int temp = x + y;

return temp;

## **1-10引用 &与指针区别：**

使用：

(1)引用被创建的同时必须被初始化(指针则可以在任何时候被初始化)。

(2)不能有 NULL 引用,引用必须与合法的存储单元关联(指针则可以是 NULL)。

(3)一旦引用被初始化,就不能改变引用的关系(指针则可以随时改变所指的对象)

注意 当 数组 作为 函 数的 参数 进 行传 递时 , 该数 组自 动 退化 为同 类 型的 指针

区别：

1) 引用必须被初始化，指针不必。

2) 引用初始化以后不能被改变，指针可以改变所指的对象。

3) 不存在指向空值的引用，但是存在指向空值的指针。

**优点**

**Const 引用：**

**在C++中所有的临时对象都是const类型的，**而上面的程序试图将const对象赋值给非const对象，这必然会使程序报错。如果在函数func2的参数前添加const，则程序便可正常运行了。

总结

1.在引用的使用中，单纯给某个变量去别名是毫无意义的，引用的目的主要用于在函数参数的传递中，**解决大块数据或对象的传递效率和空间不如意的问题**

2.用引用传递函数的参数，能保证参数在传递的过程中不产生副本，从而提高传递效率，同时通过const的使用，**还可以保证参数在传递过程中的安全性**

3.引用本身是目标变量或对象的别名，**对引用的操作本质上就是对目标变量或对象的操作**。因此能使用引用时尽量使用引用而非指针

**1-11 Vector 越界 map 清内存**

**- 关于 std::vector 的下标越界检查**

超过了 vector 的下标范围，在几种方式的结果就有区别了。在 linux 平台只有 at(n) 会抛出 std::out\_of\_range 异常，而其他两个都不会，所以要使用at 做越界检查。

**- 清除vector或者map的所有数据并且释放memory**

方法是定义一个 空的vector或map， 让要被清除的vector或map的对象和 这个空的交换。

注意空的对象的类型，key和value的类型要保持一致。

**1-12 c++内存的注意**

（1）操作堆内存时要注意安全释放原则：可以做一个宏例如：

#define Safe\_free(p) if(p != NULL) {free(p); p = NULL;}

free();后要制 p = NULL,否则会产生野指针；

检查 p != NULL // 是为了不要释放 空指针，释放后用两次就over；

（2）在malloc后要检查是否分配成功(比如内存耗尽了)：

#define Safe\_check if(p == NULL) { exit(1);// 结束本程序} //或者用return; 结束本函数

**1-13 new() malloc区别 delete 和delete[]**

new() &delete:除了自动分配内存外，还可以调用函数的构造或者析构。

delete 和delete[]：delete[]释放对象数组

eg:

A \*a = new A[100];// 创建 100 个动态对象

就得用delete[]a;

**1-14** c++ 11 智能指针, lamed ,function bin

/\*

智能指针的使用 c++ 11

- 没有free 或者 delete 指针 就是：内存泄露

为什么要用智能指针：

- 更安全的使用 动态内存

- 自动释放所指向的对象

- 智能指针c++ 11 新标准库 提供(类似vector)三个模板： shared\_ptr,unique\_ptr,weak\_ptr

\*/

#include <iostream>

#include <stdlib.h>

#include <string.h>

#include <stdio.h>

#include <unistd.h>

#include <fstream>

#include <memory>

#include <string.h>

using namespace std;

#define TST\_SHARED\_P 1

#define TST\_UNIQUE\_P 1

#define TST\_WEAK\_P 1 //解决类引用 环回问题

#define SAFE\_FREE(p) if(NULL != p) {free(p); p = NULL; }

#define SAFE\_CHECK(p) if(NULL == p) { exit(1); } // 分配完内存后注意要检查 是否分配成功

class Derived;

class Base

{

public:

const int B;

Base();

Base(int x);

Base(const Base &b);

void foo(void); // inline 仅与函数声明放在一起 是没有效果的

void foo\_au(){} // --->自动地成为内联函数

void test\_sm();

void stcp(const char \*in,char \*\*out);

//void construct(const Base &b);

virtual ~*Base*(){cout<< " ~Base() " <<endl;}

weak\_ptr<Derived> m\_dm;

private:

int m\_b\_ma = 666;

};

inline void foo() // 好的写法 ：类内定义 类外实现

{

cout<<"Base : inline foo() "<<endl;

}

Base::Base(int x)

:B(x)

{

cout<< "Base(int x) x = " <<B<<endl;

}

void Base:: stcp(const char \*in,char \*\*out) //使用二级 指针是 能够传参成功 const的作用是 防止修改 \*in ，保证入参的安全

{

char \*t;

\*out = (char \*)malloc(sizeof(in)); // 开辟内存后才能 cpy 最后要安全删除

SAFE\_CHECK(out); //检查 是否malloc 成功

memcpy(\*out,in,sizeof(in));

cout<< "stcp() out = "<< \*out<<endl;

//SAFE\_FREE(out);

//strcpy(in,"yq"); // 如果修改 const修饰的 入参 则会保错

}

Base::Base(const Base &b)

:B(22) //const 只能在函数参数表里赋值

{

cout<<"Base(const Base &b) m\_b\_ma = "<<b.m\_b\_ma<<endl;

}

void Base::test\_sm()

{

cout<< ""<<endl;

}

class Derived : public Base

{

public:

Derived(int x,int y);

~*Derived*(){cout<< " ~Derived() " <<endl;}

weak\_ptr<Base> m\_bm;

private:

int m\_d;

int m\_b;

};

Derived::Derived(int x,int y)

:Base(x),m\_d(x),m\_b(y) //在初始化表里调用基类的构造函数 1) && 也可以初始化 内部成员变量

{

cout<<"init Derived m\_d = "<< m\_d <<endl;

}

int main()

{

#if TST\_SHARED\_P

cout<< "##### smart\_pointer : tst\_shared\_ptr #####"<<endl;

char \*\_in = "yyll52"; // char in[] = "yyll52"; 效果一样

char \*\_out =NULL;

shared\_ptr<Base> sp(new Base(5656)); // 使用 智能指针

sp->stcp(\_in,&\_out);

SAFE\_CHECK(\_out);

cout<< " \_out = "<<\_out<<endl;

//Base nb(5454); // 相比智能指针，这一句不会 自动析构

shared\_ptr<Base> ssp = make\_shared<Derived>(8888888,9999); /// 传递给make\_shared函数的参数必须和shared\_ptr所指向类型的某个构造函数相匹配 相当于 new()

if(ssp.get()) //get()是判断智能指针是否为空

{

cout<< "use\_count() = "<<ssp.use\_count()<<endl;

ssp = sp;

cout<<"ssp.m\_b\_ma"<<ssp->B<<endl;

}

shared\_ptr<Derived> sm\_deptr(new Derived(66666,777777));

cout << "reset() use "<<endl;

sm\_deptr.reset();

shared\_ptr<Base> sb = make\_shared<Base>(9999);

sb->test\_sm();

cout<<"ronding test sb= "<< sb.use\_count()<<endl;

shared\_ptr<Derived> sd = make\_shared<Derived>(11111,222222);

cout<<"ronding test sd = "<< sd.use\_count()<<endl;

#endif

#if TST\_UNIQUE\_P

cout<< "##### smart\_pointer : tst\_unique\_p #####"<<endl;

unique\_ptr<Base> ub (new Base(999990));

unique\_ptr<Base> ub\_t (new Base(88880));

// ub = ub\_t; unique\_ptr 没有重载 = 赋值 比auto\_ptr 安全

#endif

}

**1-15 C++ Static**

**- 静态全局变量好处：**  
      （1）静态全局变量**不能被其它文件所用**；  
      （2）其它文件中可以**定义相同名字的变量，不会发生冲突**；

**- 静态局部变量有以下特点：**

（1）该变量在全局数据区分配内存；

（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；

（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；

（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，当定义它的函数或语句块结束时，其作用域随之结束；

通常，在函数体内定义了一个变量，每当程序运行到该语句时都会给该局部变量分配栈内存。但随着程序退出函数体，**系统就会收回栈内存**，局部变量也相应失效。

　　但有时候我们需要在两次调用之间对变量的值进行保存。通常的想法是定义一个全局变量来实现。但这样一来，变量已经不再属于函数本身了，不再仅受函数的控制，给程序的维护带来不便。

静态局部变量正好可以解决这个问题。静态局部变量保存在全局数据区，而不是**保存在栈中，每次的值保持到下一次调用，直到下次赋新值**

Eg:

Void fun()

{

Int a,b,c;//栈区

C= b+a;

Static tp;//分配再全局数据区 可以保存函数临时值，而不会随着函数结束而消亡，替代全局变量

}

- 静态函数

1. 和静态全局变量一样，可以解决名字冲突，只作用于本文件。
2. 只能调用static的类中的变量或者函数，无法访问非静态的成员
3. 同全局变量相比，使用静态数据成员有两个优势：1静态数据成员没有进入程序的全局名字空间，因此不存在与程序中其它全局名字冲突的可能性；2可以实现信息隐藏。静态数据成员可以是private成员，而全局变量不能；

　与静态数据成员一样，我们也可以创建一个静态成员函数，它为类的全部服务而不是为某一个类的具体对象服务。静态成员函数与静态数据成员一样，都是类的内部实现，属于类定义的一部分。普通的成员函数一般都隐含了一个this指针，this指针指向类的对象本身，因为普通成员函数总是具体的属于某个类的具体对象的。通常情况下，this是缺省的。如函数fn()实际上是this->fn()。**但是与普通函数相比，静态成员函数由于不是与任何的对象相联系，因此它不具有this指针。从这个意义上讲，它无法访问属于类对象的非静态数据成员，也无法访问非静态成员函数，它只能调用其余的静态成员函数**

**<https://blog.csdn.net/Hackbuteer1/article/details/7487694>**

总结：

* 出现在类体外的函数定义不能指定关键字static；
* 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；
* 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；
* 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；
* 由于没有this指针的额外开销，因此静态成员函数与类的全局函数相比速度上会有少许的增长；
* 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，也可以直接使用如下格式：

＜类名＞::＜静态成员函数名＞（＜参数表＞）

## **1-16 全局变量和局部变量有什么区别？是怎么实现的？**

**区别：**生命周期不同：

全局变量随主程序创建和创建，随主程序销毁而销毁；局部变量在局部函数内部，甚至局部循环体等内部存在，退出就不存在；

使用方式不同：通过声明后全局变量程序的各个部分都可以用到；局部变量只能在局部使用**分配在栈区**。

操作系统和编译器通过内存分配的位置来知道的，全局变量分配在全局数据段并且在程序开始运行的时候被加载。局部变量则分配在堆栈里面 。

**1-17 c++ 11 -智能指针**

- 没有free 或者 delete 指针 就是：内存泄露

为什么要用智能指针：

- 更安全的使用 动态内存

- 自动释放所指向的对象

- 智能指针c++ 11 新标准库 提供(类似vector)三个模板： shared\_ptr,unique\_ptr,weak\_ptr

* **如果程序要使用多个指向同一个对象的指针**，应选择shared\_ptr。这样的情况包括：有一个指针数组，并使用一些辅助指针来标示特定的元素，如最大的元素和最小的元素；两个对象包含都指向第三个对象的指针；STL容器包含指针。很多STL算法都支持复制和赋值操作，这些操作可用于shared\_ptr，但不能用于unique\_ptr（编译器发出warning）和auto\_ptr（行为不确定）。如果你的编译器没有提供shared\_ptr，可使用Boost库提供的shared\_ptr。
* **如果程序不需要多个指向同一个对象的指针**，则可使用unique\_ptr。如果函数使用new分配内存，并返还指向该内存的指针，将其返回类型声明为unique\_ptr是不错的选择。这样，所有权转让给接受返回值的unique\_ptr，而该智能指针将负责调用delete。可将unique\_ptr存储到STL容器在那个，只要不调用将一个unique\_ptr复制或赋给另一个算法（如sort()）。
* **shared\_ptr**

-make\_shared函数

  最安全的分配和使用动态内存的方法是调用一个名为make\_shared的标准库函数。此函数在动态内存中分配一个对象并初始化它，返回指向此对象的shared\_ptr.

**-特点**

**1)每个shared\_ptr都有一个关联的计数器，通常称为引用计数。**

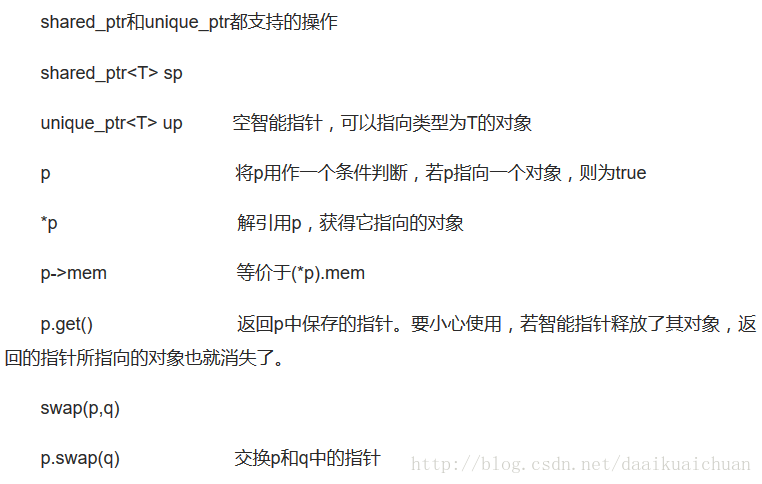
**2)无论何时我们拷贝一个shared\_ptr，计数器都会递增；当我们给shared\_ptr赋予一个新值或者shared\_ptr被销毁时，引用计数会递减。**

**3)一旦引用计数变为0，shared\_ptr就会自动释放自己所管理的对象**

**4)在多线程程序中，一个对象如果被多个线程访问，一般使用shared\_ptr，通过引用计数来保证对象不被错误的释放导致其他线程访问出现问题。**

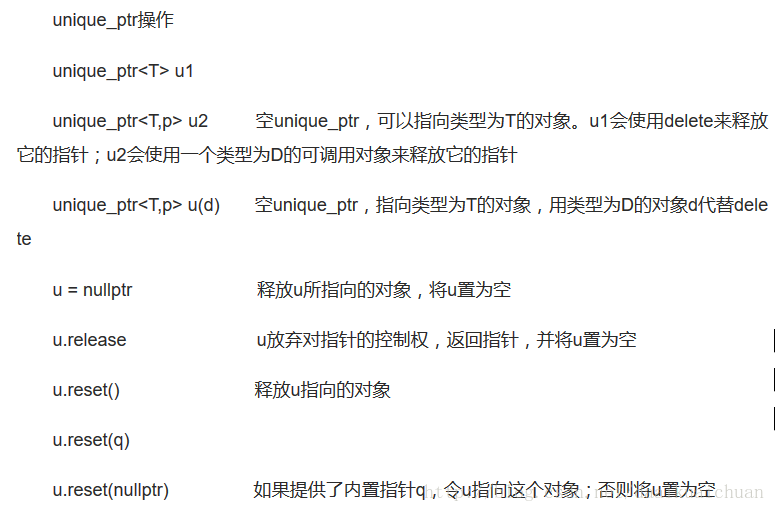
**循环引用的时候 会造成内存泄漏。可以用weak\_ptr来解决**

**把class A或者class B中的shared\_ptr改成weak\_ptr即可，由于weak\_ptr不会增加shared\_ptr的引用计数，（refer to https://blog.csdn.net/daaikuaichuan/article/details/88813992#font\_size5weak\_ptrfont\_221）**



* **unique\_pt**

unique\_ptr 独占智能指针，某个时刻只能有一个unique\_ptr 指向一个给定对象。当unique\_ptr 被销毁时，它所指向的对象也被销毁。unique\_ptr 不支持拷贝赋值等操作，除非这个unique\_ptr将要被销毁,这种情况，编译器执行一种特殊的"拷贝"



* **weak\_ptr**



**1-18 类模板**

模板是C++支持参数化多态的工具，使用模板可以使用户为类或者函数声明一种一般模式，使得类中的某些数据成员或者成员函数的参数、返回值取得任意类型。

　　模板是一种对类型进行参数化的工具；

　　通常有两种形式：函数模板和类模板；

　　函数模板针对仅参数类型不同的函数；

　　类模板针对仅数据成员和成员函数类型不同的类。

　　使用模板的目的就是能够让程序员编写与类型无关的代码。比如编写了一个交换两个整型int 类型的swap函数，这个函数就只能实现int 型，对double，字符这些类型无法实现，要实现这些类型的交换就要重新编写另一个swap函数。使用模板的目的就是要让这程序的实现与类型无关，比如一个swap模板函数，即可以实现int 型，又可以实现double型的交换。模板可以应用于函数和类。下面分别介绍。

注意：模板的声明或定义只能在全局，命名空间或类范围内进行。即不能在局部范围，函数内进行，比如不能在main函数中声明或定义一个模板。

<https://blog.csdn.net/biu__biu_biu/article/details/78020620>

**1-19 设计模式**

* **单例**

\* 保证一个类仅有一个实例，并提供一个访问它的全局访问点

4 \* 线程安全的实现方式

5 \* 单例模式的要点有三个：

6

7 单例类有且仅有一个实例

8 单例类必须自行创建自己的唯一实例

9 单例类必须给所有其他对象提供这一实例

10

11 从具体实现角度来说，可分为以下三点：

12

13 提供一个 private 构造函数（防止外部调用而构造类的实例）

14 提供一个该类的 static private 对象

15 提供一个 static public 函数，用于创建或获取其本身的静态私有对象（例如：GetInstance()

**1-20 二叉树**

种一经建立就很少改动、而又经常需要查找的线性表。

**二叉树的使用 搞明白 哈希表 队列（数据结构）**

1-21 哈希表

1-22 shell

**shell 怎么接受前面的命令的结果 $()**

1-23 makefile

**makefie := =区别 := 不会引索后面的值**

例如：

接着理解：=和=

这个是很常用的，比=用得多

看下面的例子：

    x = sb；

    y = u$(x)

    x = all

运行结果x 是all    y是uall

将=改成:=

    x := sb；

    y := u$(x)

    x := all

运行结果x是all  y是usb

怎么样？有感觉了吧。这里=的话算$(x)的时候会一直搜索到文件最后，最后再赋值。

而：=则不会搜索后面的x值。也就是前面的是有效的而引用变量之后的就是无效的。

**1-24 析构和构造的顺序**

父类先构造，再子类，析构子类再父类。

* 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。

析构函数的调用顺序与构造函数的调用顺序正好相反，将上面3个点反过来用就可以了，首先调用派生类的析构函数；其次再调用成员类对象的析构函数；最后调用基类的析构函数。

    析构函数在下边3种情况时被调用：

* 1.对象生命周期结束，被销毁时(一般类成员的指针变量与引用都i不自动调用析构函数)；
* 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。

**1-25 只能再函数初始化表里初始化的是？**

1.如果类存在继承关系，派生类必须在其初始化列表中调用基类的构造函数

2.数据成员是对象，并且这**个对象只有含参数的构造函数**，没有无参数的构造函数；

3.类成员为const类型

4.初始化引用成员数据

**1-26. coredump 文件调试**

gcc -g .c -o xxx

gdb 应用程序 coredump文件

**1-27. gdb命令**

gcc -g main.c //在目标文件加入源代码的信息

gdb a.out

(gdb) start //开始调试

(gdb) n //一条一条执行

(gdb) step/s //执行下一条，如果函数进入函数

(gdb) backtrace/bt //查看函数调用栈帧

(gdb) info/i locals //查看当前栈帧局部变量

(gdb) frame/f //选择栈帧，再查看局部变量

**whatis var 更详细的显示**

**(gdb) ptype var  //这里，会打印出var的结构定义**

**(gdb) print/p //打印变量的值**

print /x var s

可以支持的变量显示格式有：

x  按十六进制格式显示变量。

d  按十进制格式显示变量。

u  按十六进制格式显示无符号整型。

o  按八进制格式显示变量。

t  按二进制格式显示变量。

a  按十六进制格式显示变量。

c  按字符格式显示变量。

f  按浮点数格式显示变量。

(gdb) finish //运行到当前函数返回

**(gdb) set var sum=0 //修改变量值**

**(gdb) list/l 行号或函数名 //列出源码**

(gdb) display/undisplay sum //每次停下显示变量的值/取消跟踪

(gdb) break/b 行号或函数名 //设置断点

(gdb) continue/c //连续运行

(gdb) info/i breakpoints //查看已经设置的断点

(gdb) delete breakpoints 2 //删除某个断点

(gdb) disable/enable breakpoints 3 //禁用/启用某个断点

(gdb) break 9 if sum != 0 //满足条件才激活断点

(gdb) run/r //重新从程序开头连续执行

(gdb) watch input[4] //设置观察点

(gdb) info/i watchpoints //查看设置的观察点

(gdb) x/7b input //打印存储器内容，b--每个字节一组，7--7组

(gdb) disassemble //反汇编当前函数或指定函数

(gdb) si // 一条指令一条指令调试 而 s 是一行一行代码

(gdb) info registers // 显示所有寄存器的当前值

(gdb) x/20 $esp //查看内存中开始的20个数

**1-27 做库的时候 linux和win 暴露出的函数 是用什么来修饰**

**1-28 多个隐形的参数 FUN（\_fu , ...）参数个数 是在运行时还是编译时确定的**

1-29 **怎么实现一个只提供堆或者栈内存的class**

* **只在栈上分配，（即只能静态实例化class）要禁止通过 "new" 方式实例化对象，从而保证某种类型的类不会发生内存泄漏。**

- 一种最直接的方法是把**构造函数和析构函数声明为 private，但是可以只将析构函数设为 private就行（如下代码）**

* **只在堆中分配对象**，（即只能动态实例化class）只通过new的方式来实例化对象。可以将析构函数设置为 私有的。

**1-30设计个只能再栈上的class**

设计思想是 要operator new() 和delete 并且是私有的

此类的class，不能被继承

因为 子类会继承 父类的 operator 的 new，所以子类也不能用 堆（即用new)

# QT part

**2-1 信号和槽**

1. **Emit** 作用： 发出 signed（也可以 发出自定义的信号函数） 执行 connect 的 slot 函数

在你的程序中应该能找到类似这样的语句：  
  connect(obj,SIGNAL(changeCurrentShape(\*)),anotherobj,SLOG(FUN(\*)))  
  
当执行到 emit changeCurrentShape(Shape::Triangle) 时，QT的信号槽机制，会自动触发FUN()函数

Connect的直接连接、队列连接和自动连接。

在于函数的最后一个缺省值，

connect(const QObject \*sender, const QMetaMethod &signal,

const QObject \*receiver, const QMetaMethod &method,

Qt::ConnectionType type = Qt::AutoConnection); //默认自动的

直接连接的大概意思是：信号一旦发射，槽立即执行，并且槽是在信号发射的线程中执行的。

队列连接的大概意思是：信号发射后当事件循环返回到接收线程时槽函数就执行了，也就是说这种连接方式不是立即触发槽函数的，而是要排队等的，并且是在槽函数的线程中执行。

自动连接的大概意思是：信号发射对象如果和槽的执行对象在同一个线程，那么将是直连方式，否则就是队列方式。

阻塞队列方式：在槽函数返回之前槽函数所在的线程都是阻塞的。

**2-2 Qt线程的两种使用方式**

（1）子类化QThread（不使用事件循环），subclassing QThread，直接继承自QThread

QThread的实例存活在实例化它的主线程中，而不是调用run函数的线程中。也就是说QThread放入队列的槽都会在父线程中执行。构造函数在父线程中执行，run函数在新线程中执行。

（2）子类化QObject，moveToThread，the worker-object approach

MyObject object;

QThread thread;

object.moveToThread(&thread);

QObject::connect(&thread, SIGNAL(started()), &object, SLOT(start()));

thread.start();

moveToThread 之后Event Process会在目标线程中进行。这个操作会使object中的所有active Timer都先停止，然后再在目标线程中重启。被新放到object的Event都会在目标线程中处理。

其主要特点就是利用Qt的事件驱动特性，将需要在次线程中处理的业务放在独立的模块（类）中，由主线程创建完该对象后，将其移交给指定的线程，且可以将多个类似的对象移交给同一个线程。在这个例子中，信号由主线程的QTimer对象发出，之后Qt会将关联的事件放到worker所属线程的事件队列。由于队列连接的作用，在不同线程间连接信号和槽是很安全的。  
说说connect最后一个参数，连接类型：  
1)自动连接(AutoConnection)，默认的连接方式，如果信号与槽，也就是**发送者与接受者在同一线程**，等同于直接连接；如果发送者与接受者处在不同线程，等同于队列连接。  
2)直接连接(DirectConnection)，当信号发射时，槽函数立即直接调用。无论槽函数所属对象在哪个线程，**槽函数总在发送者所在线程执行。**

3)队列连接(QueuedConnection)，当控制权回到接受者所在线程的事件循环时，槽函数被调用。**槽函数在接受者所在线程执行**。

**2-3 Qt的刷新机制**

update()和repaint()函数实现上有什么不同。

Update不会立即重绘，当回到**主事件循环时才调用paintEvent**。这样能更少闪烁并更快。正常情况下在paintEvent之前会先擦除widget区域。

repaint()是立即调用paintEvent()，~~而update()是几次执行才调用一次paintEvent()~~。  
这样update()会造成这样的结果：paintEvent()中的任务没有执行完，就又被update().paintEvent()中被积压的任务越来越多。

**2-5 Qt的事件管理机制，（从底层窗口传到顶层窗口）**

通过事件过滤器, 可以让一个对象侦听拦截另外一个对象的事件

void QObject::installEventFilter(QObject \* filterObj) **安装过滤器**

bool QObject::eventFilter(QObject \* watched, QEvent \* event)**相当于创建了过滤器 ，**不想让它继续转发，就返回 true，否则返回 false

多个过滤器安装到了一个对象，则最后安装的会首先激活。

事件过滤器和被安装过滤器的组件必须在同一线程，否则，过滤器将不起作用

TPTCom通信，

IPTCom接收和发送PD数据，以及接收和发送MD数据的过程。

：先INit的过程，把data-set 和tdc文件load到对应德API函数里，

然后在一个loop get/send data 我们可以通过一个大的结构体去send、rec data了

两个配置文件的作用（一个data-set文件，一个tdc.hosts文件）。

data-set：**需要被配置网络上要交互的数据的IPTCom**，~~例如关于如何的信息~~

~~处理数据，在哪里发送数据等。这些信息存储在配置数据库中。~~

~~数据库以XML格式或API函数从配置文件填充。~~

tdc.hosts ：h**ost文件中查找URI以检索终端设备或多播的IP地址**

# Gstreamer part

**3-1 Gstreamer播放视频**

1、rtsp建立连接的过程以及采用Rtp协议（UDP或者TCP）发送数据。

简单的rtsp交互过程:

C表示rtsp客户端,S表示rtsp服务端

1.C->S:OPTION request //询问S有哪些方法可用

1.S->C:OPTION response //S回应信息中包括提供的所有可用方法

2.C->S:DESCRIBE request //要求得到S提供的媒体初始化描述信息

2.S->C:DESCRIBE response //S回应媒体初始化描述信息，主要是sdp

3.C->S:SETUP request //设置会话的属性，以及传输模式，提醒S建立会

话

3.S->C:SETUP response //S建立会话，返回会话标识符，以及会话相关信息

4.C->S:PLAY request //C请求播放

4.S->C:PLAY response //S回应该请求的信息

S->C:发送流媒体数据

5.C->S:TEARDOWN request //C请求关闭会话

5.S->C:TEARDOWN response //S回应该请求

Gstreamer管道的建立过程：

gPlayerPipeline = gst\_pipeline\_new("pipeline");

h264parse = gst\_element\_factory\_make("h264parse","h264parse");

imxvpudec = gst\_element\_factory\_make("imxvpudec","imxvpudec");

videosink = gst\_element\_factory\_make("imxg2dvideosink","videosink");

src = gst\_element\_factory\_make("rtspsrc","src");

rtph264depay = gst\_element\_factory\_make("rtph264depay","rtph264depay");

g\_object\_set(G\_OBJECT(src), "protocols", 1, NULL);

g\_object\_set(G\_OBJECT(src), "retry", 10, NULL);

g\_object\_set(G\_OBJECT(src), "latency", 150, NULL);

g\_object\_set(G\_OBJECT(src), "async-handling", true, NULL);

g\_object\_set(G\_OBJECT(src), "udp-reconnect", false, NULL);

g\_object\_set(G\_OBJECT(src), "message-forward", true, NULL);

gst\_bin\_add\_many(GST\_BIN(gPlayerPipeline), src, rtph264depay, h264parse, imxvpudec, videosink, NULL);

gst\_element\_link\_many(rtph264depay, h264parse, imxvpudec, videosink, NULL); //链接元件

g\_signal\_connect(src, "pad-added", G\_CALLBACK (on\_pad\_added),rtph264depay);

bus=gst\_pipeline\_get\_bus(GST\_PIPELINE(gPlayerPipeline));

bus\_watch\_id = gst\_bus\_add\_watch(bus,bus\_callback,this);

gst\_object\_unref(bus);

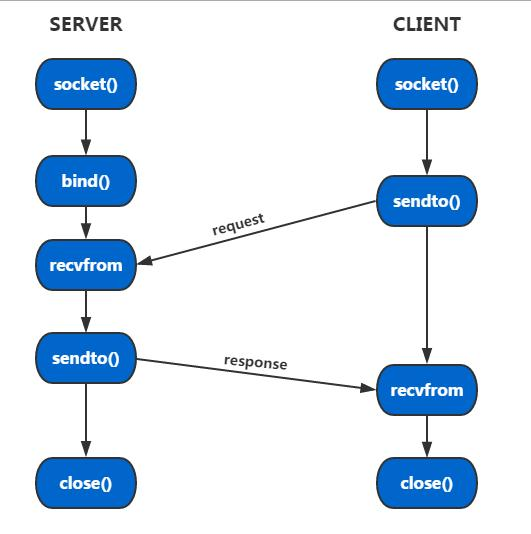
g\_object\_set(G\_OBJECT(src), "location", URI.c\_str(), NULL);

gst\_element\_set\_state(gPlayerPipeline, GST\_STATE\_PLAYING);

# network part

**4-1 UDP通信**

是一种**无连接的协议**，(与TCP比较少一个 bind(可选))每个数据报都是一个独立的信息**，包括完整的源地址或目的地址，它在网络上以任何可能的路径传往目的地**，因此能否到达目的地，到达目的地的时间以及内容的正确性都是不能被保证的。



**4-2 Tcp/ip 握手过程**

两种方式的过程详细:

**4-3 HTTP**

**4-4 加密传输**

**4-5组播广播**

# Linux 操作系统进程线程

**5-1 进程同步方式**

**5-2** **线程同步方式**