Const 用法：

修饰变量与参数：限定指针与指向内容是否可修改。

int \* const a = &b; //指针不能被修改

//a = &x;//错误：指针不能被修改

\*a = 100;//正确：指针指向得内容可以修改

const int \* d= &b ; //指针指向得内容不能被修改

//\*d = 100;//错误：指针指向得内容不能被修改

d = &x;//正确：指针可以修改

const int \* const c = &b;//指针与指针指向内容都不能修改

//\*c = 100;//错误：指针指向的内容不能被修改

//c = &x;//错误：指针不能修改

栈内存的申请方式是从高地址到低地址分配。（可想而知，栈的空间很有限的）

对的内存申请方式是从低地址到高地址分配。

Const和#define的区别：

Const是由编译器处理的，提供类型检查和作用域检查

#define是预处理器处理，单纯的文本替换

例如：

Const int a=10;const int b=20;

#define aa 10 #define bb 20

Int z[a+b]//可以通过编译

Int zz[aa+bb] //不能通过编译

//这三个函数看着长得差不多，其实天差地别：

class Test

{

public:

Test()

{

cout << "construct" << endl;

}

Test(Test& t)

{

this->age = t.age;

cout << "copy construct" << endl;

}

int age;

};

void Getage1(Test\* t)

{

cout << t->age << endl;

}

void Getage2(Test &t) //这里用的是引用，相当与别名

{

cout << t.age << endl;

}

void Getage3(Test t) //这是用的形参，相当于把t的数据拷贝了一份传过去，会调用拷贝构造函数

{

cout << t.age << endl;

t.age = 200;//在这里的修改不会影响原值，因为修改的拷贝进来的,上面两处都会修改

}

引用的基础知识：

引用也占内存，和指针差不多，相当于一个指针的别名吧。32位占4字节，64位占8字节。

\*引用实际是常量指针吗？

1. 引用基础

\*声明的引用类型变量一定要有初始值

\*将引用类型用作参数，可以不需要初始化。

Int &b=a;背后做了什么？：

1. 引用当左值

函数的返回值当引用（基础类型）：

int &foo1(){

int a=100;

return a;

}

若返回栈变量，如上情况：

\*返回值不能成为另一个引用的初始值

\*不能作为左值使用

若返回的位静态变量或全局变量：

\*可以作为另一个引用的初始值

\*可以作为左值使用，也可以作为右值。

Ex:

int & foo2()

{

static int a = 100;

return a;

}

三．指针的引用

什么情况要用二级指针呢？：

struct Teacher

{

char name[64];

int age;

};

例如：一个指针变量，需要通过一个函数去修改这个指针变量里面的值，就需要二级指针。

setTeachaer(Teacher\*\* p)

{

Teacher \*tmp =(Teacher\*)malloc(sizeof(Teacher));

Tmp->age=3000;

\*p=tmp;

}

//---main

Teacher \* temp;

setTeacher(&temp) //这时候就需要使用二级指针

----------------改进：使用指针引用：

setTeacher(Teacher \* & p)

{

Teacher\* tmp=(Teacher\*)malloc(sizeof(Teacher));

Tmp->age=30000;

P=temp;

}

//--main

Teacher\* temp;

setTeacher(temp);

四．Const引用

Const int& b=a

B=100;//错误，常量引用不能修改. 就是为了让引用拥有只读属性

常引用创建的两种方式：

\*通过另一个变量赋值

Int a=200;

Const int &b=a

\*直接赋值右值

Const int &c=200;//编译器会偷偷的给c分配内存

内联函数：

\*再最终生成的代码中是没有定义的。

\*inline函数由编译器处理，会进行编译。#define由预编译器处理，没有任何编译，直接替换片段。

\*在编译时直接将函数体插入到函数调用的地方，相当于宏片段替换

\*编译器可能会进行优化，对没有加inline的函数优化为inline函数。

\*是一种请求，由编译器自己决定

\*省去了普通函数调用时压栈、调转、返回的开销。

限制：

不能有循环，不能有太多的判断语句，函数体不能过大，不能对函数进行取地址操作

函数的占位参数：

Int foo(int a,int b,int)//最后一个就是占位参数

\*调用的时候必须写够参数

\*函数体内部不能用占位参数，没什么卵用。

函数指针：

\*声明一个函数类型

Typedef void Foo3(int a,int b);//用来定义函数指针

Foo4 \* ff1 = nullptr;//定义一个函数指针，指针指向函数的入口地址

\*声明一个函数指针类型

typedef void(\*pFoo5)(int a, int b); //声明了一个指针的数据类型

pFoo5 ff2 = nullptr;//这里就不用加\*号了

\*定义一个函数指针变量

void(\*Foo6)(int a, int b);//直接定义一个函数指针变量

Foo6=PP;

函数对象：

\*相比于函数指针：

函数指针笨拙，危险，使用函数对象来代替其。

函数对象可以有自己的状态，可在类中定义状态变量，一个函数对象可在多次调用共享这个状态。

函数对象有自己特友的类型，例如在stl中sort的cmp可以传递不同的参数来实例化模板。

//通过虚函数来增加OOP的灵活性

class Func{

public:

virtual ~Func();

virtual double operator()(double) = 0;

};

class NMFunc : public Func{

public:

NMFunc(double(\*f)(double)) :f\_(f) {}

double operator()(double d) { return f\_(d); }

private:

double(\*f\_)(double);

};

//函数使用实体

double integrate(Func& f, double low, double high);

double aFunc(double x) {}

int main(){

//如何利用多态性来灵活使用函数对象

NMFunc g(aFunc);

double area = integrate(g, 0.0, 2.7);

}

函数指针遇到函数重载：

函数指针所接受的函数，会严格按照函数指针类型声明的参数进行匹配。

Ex：

Typedef void(\*Foo5)(int a,int b);

Void PP(int a,int b){}

Void PP(int a){}

//main--------

Foo5 ff5=nullptr;

Ff5=PP; //用起来就和C#里的委托一样。

Ff5(1,2)//通过

Ff5(1)//不通过

Ctrl+shift+u把小写变大写。

Struct和class的区别：

Struct：

不写访问修饰符默认时pulic。

Class：

不写访问修饰符默认时privat。

拷贝构造函数与赋值构造函数调用时机：

\*对象间的赋值与复制构造

Test a1=a2或Test a1(a2);//调用拷贝构造函数

a1=a2; //注意：这里调用赋值构造函数

\*函数参数使用值类型来传递对象的时候：

Void F(Test t1)//调用复制构造

Void F1(Test & t1)//不调用

Void F2(Test \* t1)//不调用

\*函数返回值以值类型返回：

Test rFF1(){ //调用，以值类型返回了

Test t;

return t;

}

Test& rFF2(){ //不调用，以引用类型返回

Test t;

return t;

}

Test\* rFF3(){ //不调用，返回指针

Test t;

return &t;

}

接收过程：

以值类型作为返回值，无论如何都会调用拷贝构造函数：

Test xx0 = rFF1();//调用

Test & xx1 = rFF1();//调用

Test\* xx2 = &rFF1();//调用

以引用类型作为返回值，转变为值类型会调用

Test xx0 = rFF2();//调用

Test & xx1 = rFF2();//不调用

Test\* xx2 = &rFF2();//不调用

以指针类型作为返回值，转变为为值类型会调用：

Test xx0 = \*rFF3();//调用

Test & xx1 = \*rFF3();//不调用

Test\* xx2 = rFF3();//不调用

构造函数知识小结：

\*类中没有任何实现任何构造函数，默认提供构造与拷贝构造

\*类中只实现了拷贝构造函数，不会提供默认无参构造函数。

\*类中实现了非无参构造函数，不会提供默认无参构造函数

对象有指针一定要使用深拷贝：

Ex：

Class Test{

Char\* str; //有指针变量，要是重写拷贝构造函数，实现深拷贝

Int a=0;

Test(const Test& t){

This.a=t.a;

Str = new char[strlen(t.str)+1]//额外分配一个/0占位符

//c++中strcpy没有做安全判断，不建议使用

Strcpy\_s(str,strlen(t.str)+1,t.str)

}

~Test(){

If(str)

Delete str;//释放指针变量

Str=nullptr;//防止悬空指针。

A=0;}

构造函数初始化列表：

**\*类中有多个参数的对象,需要在参数列表中初始化：**

Ex：

class Father

{

public:

int a;

int b;

Father() {}

Father(int a, int b){

this->a = a;

this->b = b;

}

Father(const Father& t){

cout << "copy cons" << endl;

}

Father & operator = (const Father& t) {

cout << "copy operator" << endl;

return \*this;

}

};

class Test

{

public:

char\* str;

int a;

Father f1;

Test(int \_a, const char\* \_str) : f1(100, 100)//再初始化列表中做初始化才正确.

{

a = \_a;

//str = const\_cast<char\*> (\_str);

int len=strlen(\_str);

//str=(char\*)malloc(sizeof(char)\*(len + 1));

str = new char[len+1];

strcpy\_s(str,len+1, \_str);

**//这种也可以，但非常推荐这么做！！因为会多调用一个拷贝赋值函数**

//f1 = Father(1, 1);

}

};

**\*类中有const成员，需要在初始化列表中初始化:**

Class Test{

Const int a;

Test():a(100){

}};

**\*类中对象的调用顺序与对象的声明顺序有关，与初始化列表的顺序无关**

Father1 f2;

Father f1;

Test(int \_a, const char\* \_str) :cc(100), f1(100, 100), f2(200, 200)

{

a = \_a;

int len=strlen(\_str);

str = new char[len+1];

strcpy\_s(str,len+1, \_str);

cout << "wuhu" << endl;

}

Res：顺序为先执行f2的构造函数，在执行f1的构造函数。

Father f1;

Father1 f2;

Res：这种就限制性f1的再执行f2的。

匿名对象/临时对象：

**\*匿名对象/临时对象，在使用完就立马析构：**

Bt haha(){

Reutn bt(100);}

Res:最后再这个语句结束后，立刻执行一次构造，并立刻执行一次析构。

Bt haha(){

Bt x=bt(100); //把临时对象转正，相当于一次构造，不调用拷贝构造

Reutn x}

Res:这个会执行一次bt的构造，然后等函数体结束才执行析构

**\*如果在类的构造函数中再调用构造会是什么样的结果呢：**

Ex：

class bt {

public:

int a;

int b;

bt(int a)

{

this->a = a;

bt(a, 30000);

cout << "bt\_1" << endl;

}

bt(int a, int b)

{

this->a = a;

this->b = b;

cout << "bt\_2" << endl;

}

bt(const bt& b)

{

cout << "copy...." << endl;

}

~bt()

{

cout << "destory bt" << endl;

}

};

//main-------

Bt zzz(1)

Cout<<zzz.b<<endl;//结果会是什么，会是3000吗

Res：结果不是，因为在构造函数中调用构造函数，相当于使用临时对象，会把临时对象中的元素初始化，而且本身对象的元素并没有初始化。

静态成员变量与静态成员函数：

**\*静态成员变量必须在类外设置初始值，而且每个类型的模板类都有单一的静态实例：**

Class Test{

static int a;};

int Test::a = 100;

**\*普通成员函数可以访问静态成员与不同成员，但静态成员函数只能访问静态成员**

malloc，free和new，delete的区别：

\*malloc是c语言中的函数，执行时候只会分配大小不会调用构造函数。

New是c++中的关键字，不光会分配内存，还会调用类的构造函数。

\*free是c语言中的函数，执行的时候只会释放内存，不会调用析构函数。

Delete是c++中的关键字，不光释放内存，害调用类的析构函数。

转换函数：

可将一个对象转换为另一个种类型，用于你认为和合理的地方转换。

一个类中可以定义多个转换函数，只要是你认为和你就可以。

注意转换函数和对应运算操作符重载之间的二义性冲突。

class HH {

public:

void operator()(double d) { tt(d); }

operator float()const

{

return 100.f;

}

private:

void(\*tt)(int a);

};

HH hh;

cout<<100.1+hh<<endl;//就可以正常运行

小tip：

Std::nothrow ：

Char\* p = new(std::nothrow) char[100];

当内存申请失败会返回一个空指针，而不是出发bad\_alloc，可以方便进行判空检查。

Std::copy和memcpy的区别：

最好多使用copy，memcpy是针对c语言风格Pod类型的。C++中如果是pod类型，编译器也会把std::copy替换为memcpy。Memcpy中的void\*可能会丢失信息，然后std::copy的性能也会稍微好一点。Std::copy还可以放入迭代器等，而不是平常的指针，更灵活没有东西的丢失。

EffectiveC++知识：

\*如果一个类不作为基类，那么虚函数不要声明为virtual

\*如果一个类中至少有一个虚函数成员，才将虚函数声明为virtual。

\*不要在构造或析构函数中调用虚函数。

例如创建了子类对象，但父类constructor中调用了一个虚函数，这个虚函数在子类中重写了，这时候实例化子类时候一定会先调用父类构造函数，所以这时候父类构造调用的这个虚函数就是父类的，因为在baseclass构造期间，虚函数不是虚函数。

\*重载等号运算符时，记得要处理自我复制的情况

就是判断赋值对象是否是当前的自己，如果是返回自身this。

If(&rhs==this)return \*this;

\*编写拷贝、移动系列函数别忘了漏下任何一个成员

如果有继承的话，别忘记再写子类的同时顺便调用父类的拷贝、移动系列函数。

继承的拷贝构造：

B(const B& b):A(b){

Xxx}

继承的拷贝赋值：

B& operator = (const B& b){

A::operator=(b)

Xxx}

\*new和delete要匹配的出现：

例如string\* sArray = new string[100];就要delete[] sArray;

例如string str = new string; 就要delete str;

\*子类虚函数调用父类版本时

不要想着通过类型转换调用，直接显示的baseclass::baseVirtual()调用。

Inline：

函数指针中的函数是一个inline函数，调用时不会进行inline优化，因为它通过一个函数指针达成

类中成员构造函数调用顺序：

从子类先一直跑到父类，然后开始执行父类中的东西，执行父类的构造函数。然后先调用成员中对象的构造函数，调用顺序与成员的声明顺序有关。然后调用自身的构造函数。然后在跑到下一级子类中，重复上面，然后知道到最后一个子类。

Stl中常见容器：

顺序容器：vector,list,deque.

关联容器：map,multimap,set,multiset

容器适配器：stack,queue

Stl中使用empty()代替size()==0

因为在一些list容器中，size()是线性的，而empty使用是常熟。

给容器全新的赋值（清楚原有数据）要记得使用assign函数：

要比循环更快，给容易全新的元素（operator=无法完成），可读性高。

声明函数指针时

Void Test(Int(\*pf)());和void Test(int pf())是一样的。

使用指针容器要记得释放指针对象

如果存入的是对象副本，那么容器会自动释放。如果是指针，就需要自己释放了，具体参考：<http://blog.sina.com.cn/s/blog_4b3c1f950100kgps.html>

Stl容器中，每种容器的删除策略都不一样，效率也不一样，记得抽空看看。

--去除一个容器中特有值得所有对象：

如果是vector,string,deque,使用erase-remove的方法

如果是list，使用list::move()

如果是标准关联容器，使用erase()成员函数

--去除一个容器中满足一个特定判定式的所有对象：

如果vector,string,deque,使用erase-remove\_if

如果是list，使用list::remove\_if()

如果是标准关联容器，使用remove\_copy\_if和swap，或写一个循环来遍历容器，当把迭代器传给erase时记得后置递增它。

--在循环内做某些事（除了删除对象之外）：

如果是标准序列容器，写个循环，当使用erase时，记得用它的返回值更新迭代器

如果是标准关联容器，写循环遍历，当吧迭代器传给erase时，记得后置递增。

Stl迭代器中，主要有三种迭代器

安插型迭代器：back\_inserter, front\_inserter, inserter fill\_n一般和back\_inserter配合。

流迭代器：

逆向迭代器：

自己总结的详细渲染管线（自己软渲染器的）：

先来协定一些基础单元：

以下这些是使用forwardRender的流程：

Framebuffer（帧缓存）：

用途：负责存储最终输出到屏幕上的像素显示

包含元素：包含colorbuffer，depthbuffer（两个容器） opengl绘制接口

VertexShader（顶点着色器）：

用途：对顶点进行顶点变换流水线：mvp变换到裁剪空间+透视除法到NDC坐标下+视口变换到屏幕空间下。面提出针对三角面，在vs之后，光栅化之前。裁剪与剔除也大多发生在这个阶段中。

包含元素：Vertex，V2f

Rasterization（光栅化）：

用途：使用坐标系或扫描线算法等对三角形进行光栅化，内部对v2f各个属性进行差。 在这里我们还要进行深度测试哦（正常来说测试系列都是在ps之后，我们这里提前为了性能），测试掉的深度不再显示。深度写入和调用ps（颜色写入）的实际基本一样。

包含元素：V2f，FragmentShader

FragmentShader（片段着色器）：

用途：在这之中进行光照计算等等

包含元素：v2f

输出：最后只是单纯的颜色，输出给framebuffer。

裁剪与剔除：

按照先后一共是三次：视锥剔除，视口剔除，背面剔除。遮挡剔除属于obj级别的剔除（应该是）

视锥剔除：在cpu阶段的粗略剔除，通过aabb或obb与视锥提做检测，直接剔除掉完全不可见的物体，运算量底，进度也低

视口剔除：在投影变换之后屏幕映射之前，所以我们就在NDC下进行。这里使用Sutherland–Hodgman裁剪算法

背面剔除：规定好三角形绘制顺序，根据三点求出平面法向量，和相机的观察方向做点成来判定。

大多数美术制作的图都是经过gamma矫正的，为了可以直接用在显示器上显示。由于pbr是在线性空间中进行的，所以在进行在读入图时候，先进行一个gamma补偿转变为线性空间的，然后计算后在写入framebuffer之前在进行一个gamma矫正。

当然也不是所有的都要进行gamma矫正，用来显示颜色的需要，例如diffuse||albedo等，normal||AO||bump等不太需要。在设计软渲染器texture类似要开启是否gamma补偿的选项。

宏定义和const的区别：

<https://blog.csdn.net/zhongad007/article/details/77005990>

**报错： 无法解析的外部符号：**

经过简单的搜索后，可以得到出现这个错误的错因大多数在于：

[0]出现无法解析可能是因为lib文件不正确,比如64位的编译配置,结果使用的是32位的lib包.

[1]只写了类声明，但还没有写实现类,造成调用时无法解析

[2]声明和定义没有统一，造成链接不一致，无法解析

[3]没有在项目属性页的链接器的命令行选项加入相应的类包。

[4]没有在c++包含目录和库目录加入相应的类包路径

[5]在测试工程中被测文件目录可能需要包含被测类的cpp定义文件

[6]ICE接口测试时，无法解析可能因为被测文件没有包含进相关的cpp文件，另外，在TestSuite\_ProjectRun.h文件中需要包含IProjectRun.h头文件，及相关的头文件(举例)。

[7]import相关的无法解析内容，解决办法是在链接器的依赖项中加入相应的动态库 [8]出现如下错误的原因一般是动态库没有包进来。

[9]error LNK2001: 无法解析的外部符号 \_\_imp\_\_\_CrtDbgReportW

工程属性，C/C++,代码生成，运行时库选择MDd,