# 头文件和类声明

参考：[2. 头文件与类声明 - 一杯明月 - 博客园 (cnblogs.com)](https://www.cnblogs.com/yibeimingyue/p/12286668.html)

### 文件扩展名不一定是.h或者.cpp，也可能是.hpp（头文件和主程序放在一个文件中实现）。

### 2.一个简单的C++程序示例（输出一个变量的值到屏幕上）：

[IMG_256](javascript:void(0);)

#include<iostream>

using namespace std;

int main()

{

int i = 7;

cout << "i=" << i << endl;

system("pause");

return 0;

}

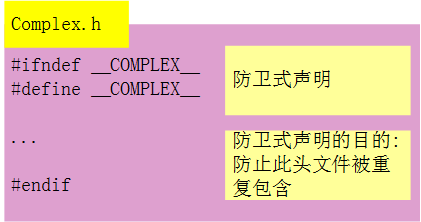
[IMG_257](javascript:void(0);)

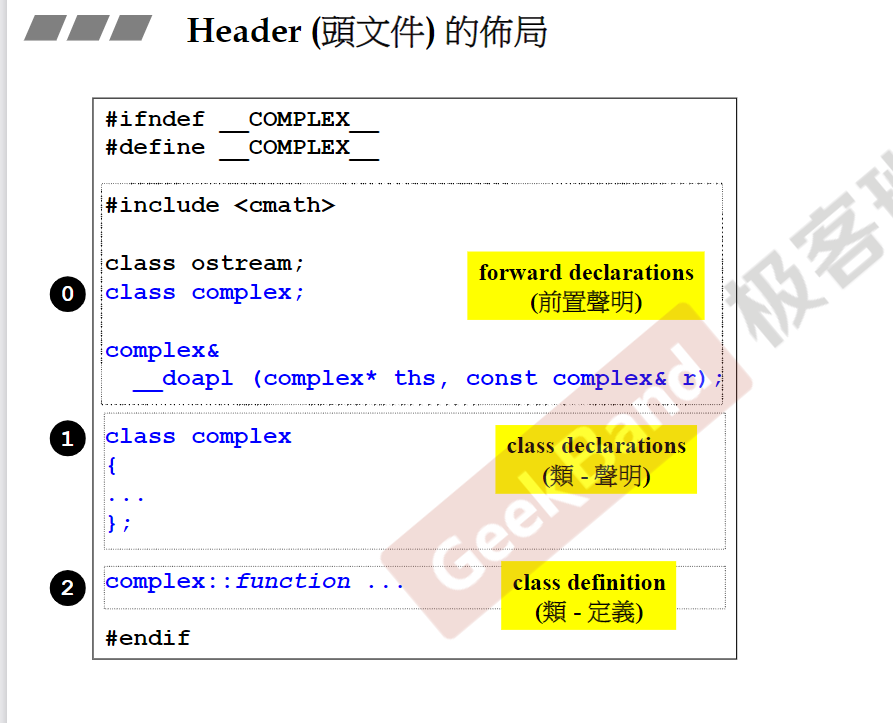
注意：

（1）#include<iostream>也可以写成：#include<iostream.h>；

（2）把cout想象成是一块屏幕，cout << "i=" 相当于是把 i= 丢到屏幕上进行显示。

### 3.头文件里面防卫式声明的写法（注：这是大气，出手不凡，规范的写法）：





还有一种防止头文件被重复包含的办法：

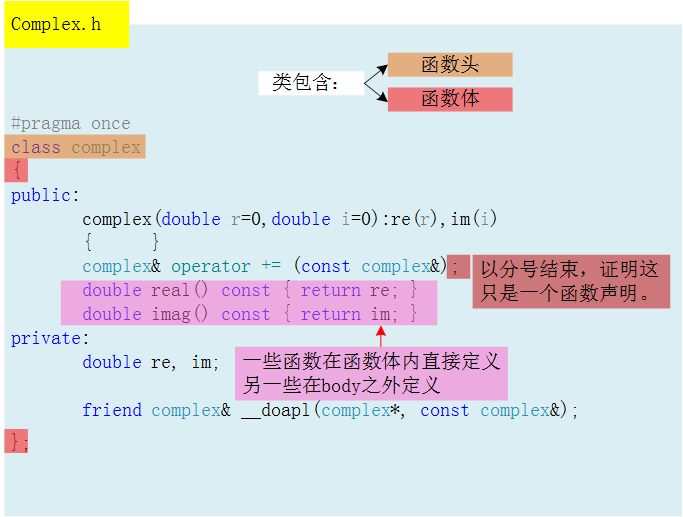
#pragma once

// contents of the header

...

区别：[#ifndef可以针对一个文件中的部分代码](https://blog.csdn.net/fengbingchun/article/details/78696814" \t "https://www.cnblogs.com/yibeimingyue/p/_blank)，而#pragma once只能针对整个文件。

### 4.类的组成样子。



### 5.定义了模板的类

第6条是把私有数据的类型写死了，定义示例的时候，数据类型受到限制。所以需要写一个模板类（含模板的类）。T写成什么都可以，比如K，U。

#pragma once  
template<typename T>  
class complex  
{  
public:  
complex(T r=0,T i=0):re(r),im(i)  
{ }  
complex& operator += (const complex&);  
T real() const { return re; }  
T imag() const { return im; }  
private:  
T re, im;

friend complex& \_\_doapl(complex\*, const complex&);  
};

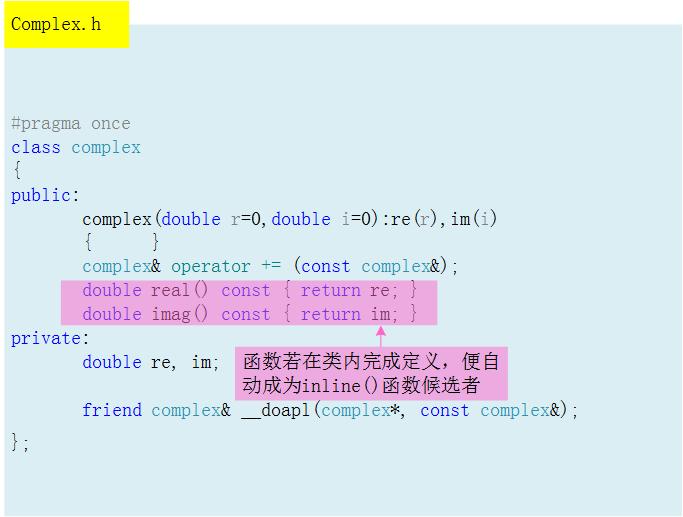
{  
complex<double> c1(2.5, 1.5);  
complex<int> c2(2,6)  
}

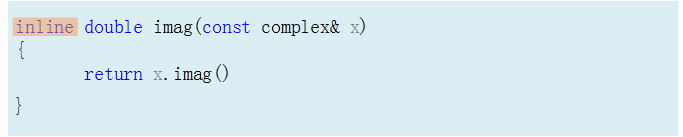
# 构造函数

### 1. inline(内联)函数

inline函数很好，其优点是：传的速度快。那所有函数都做成inline()函数岂不是很好，有的函数即便放在class body内完成定义，编译器也没办法变成inline()函数。简单的函数可以，复杂的不可以。如下面的两个函数，都是只是返回一个数据，如此简单的函数想必编译器有能力把它做成inline()函数。

如果函数没在class body内定义，而是在class body内声明，class body外定义，此时还是想把它写成inline()函数，那么只需要在函数的开头加上“inline” 关键字即可。





### 2. 访问级别

访问级别是指public:和private:所形成的区域。前者形成的区域是可以被外界访问的区域，后者形成的区域是不可以被外界访问的区域。

没有特殊情况的话：

* 数据一般放在private区域。
* 函数一般放在public区域。



### 3. 构造函数（大主题）

C++说，在利用类名创建一个对象的时候，有一个函数会被自动调用，这个函数就是构造函数。

构造函数的特点：

* 没有返回值类型，也不需要有，不必有，因为返回值肯定是当前构造函数对应的类，相当于省略；
* 用创建实例的时候传递进来的参数值去初始化私有变量；



* 可以在函数（大括号）内部去初始化私有变量，但是函数名后面冒号后初始化更好。大括号内部初始化未尝不可，但是动作稍微晚了一点，效率变低了一点（侯捷老师的话是：效率上差了一点）。



### 4. 析构函数

不带指针的class多半不用写析构函数，像上面这个例子就不用写析构函数。

### 5. 构造函数可以有很多个，这就是overloading(构造函数的重载)，在使用类创建对象的时候，有多少个想法就可以创建多少个形状不同的构造函数，形状不同指的是函数参数个数不同，类型不一样。

下面这个构造函数的重载形式不可以，因为当创建下列两个对象时，编译器会犯傻，不知道该调用哪个构造函数。

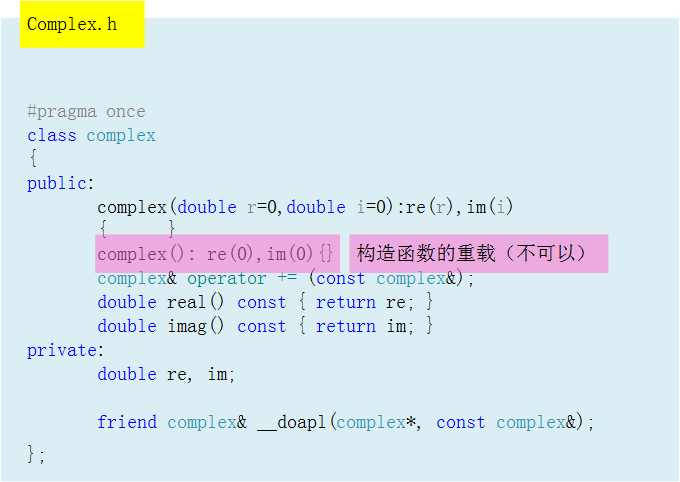
{

complex c1();

complex c2;

}

因为创建这两个对象的时候，都没有提供参数值，所有似乎调用哪个都可以（只要提供默认值的都可以），此时构造函数会起冲突。重载可以，带歧义的重载不可以。



### 6. 非构造函数的重载

重名函数，人看相同，编译器会把它们编译成名字不同的函数，编译后的名字人看不懂。



# 第三章 参数传递与返回值

### 1. 定义在私有区域中的构造函数（内容略）

### 2. 常量成员函数（const member functions）

double real() const { return re; }

double imag() const { return im; }

上面这两个函数只是把函数的实部和虚部拿出来，并不是写，不写就不会改变私有数据re 和 im，所以可以定义成常量成员函数。所谓常量成员函数就是不改变私有数据的成员函数，表面上看的话就是函数名和作用域之间加上const修饰符的函数，加const就意味着不打算改变私有数据，不改变私有数据就一定要加上const.函数前面需要加const修饰符的时候一定要加，不加的话可能会产生不希望的后果。比如：

const complex c1(2, 1);

这一句就是说，我创建对象的时候，我的数据不可以被改变，但是假如调用函数real()的时候，real()函数前面没有加const，就会产生矛盾，因为没有用const修饰的函数意味着：进来的数据有可能被改变。

[IMG_256](javascript:void(0);)

#include<iostream>

#include"complex.h"using namespace std;int main()

{

const complex c1(2, 1);

cout << c1.real() << endl;

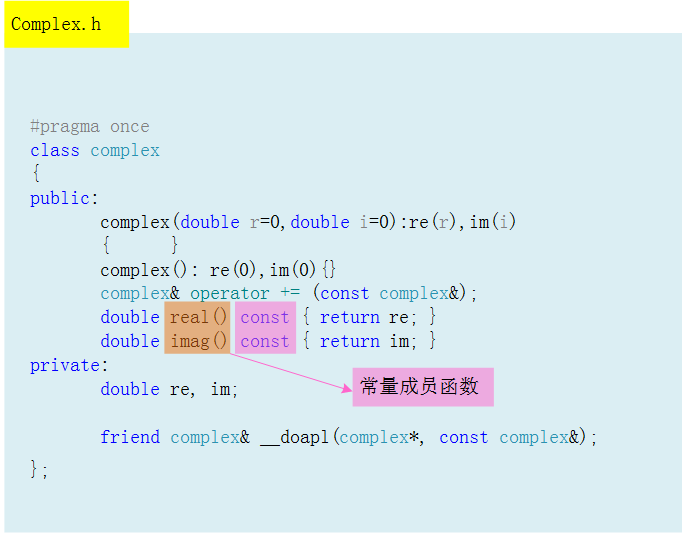
cout << c1.imag() << endl;

system("pause");

return 0;

}

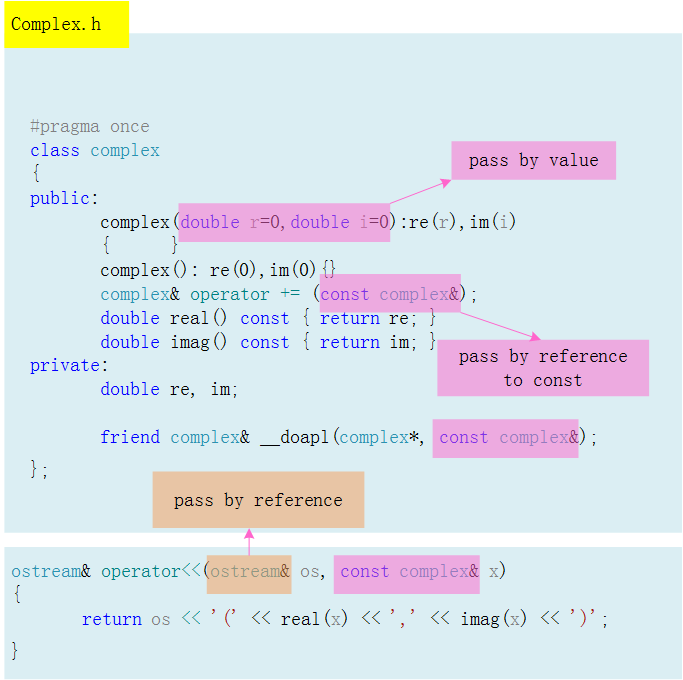
[IMG_257](javascript:void(0);)



### 3. 参数传递（pass by value or pass by reference(to const)）

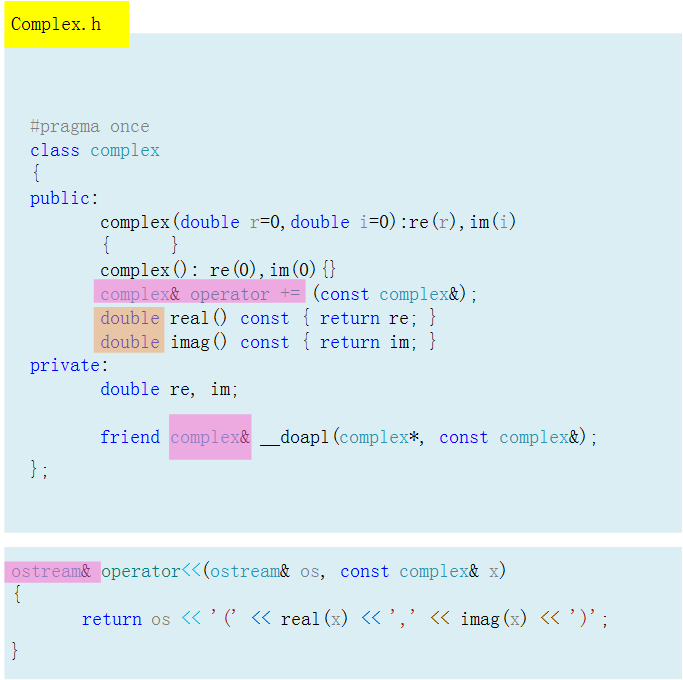
尽量不要pass by value，因为字节数（可能比较大，一大包东西）不定。C语言传的是指针，永远是4个bytes或者是8给bytes（看系统）。C++有一个东西像指针，但是更漂亮，这就是引用。引用很抽象，引用在底层就相当于一个指针，所以传引用就会像传指针那样的快。最好所有的参数传递都传引用。

* 传引用到const变量的话，值不会被改变；
* 传引用到非const变量的话，值可能会被改变；
* 基础数据类型传值，自定义类型传引用（来自于网友）。



### 4. 返回值传递（pass by value or pass by reference(to const)）

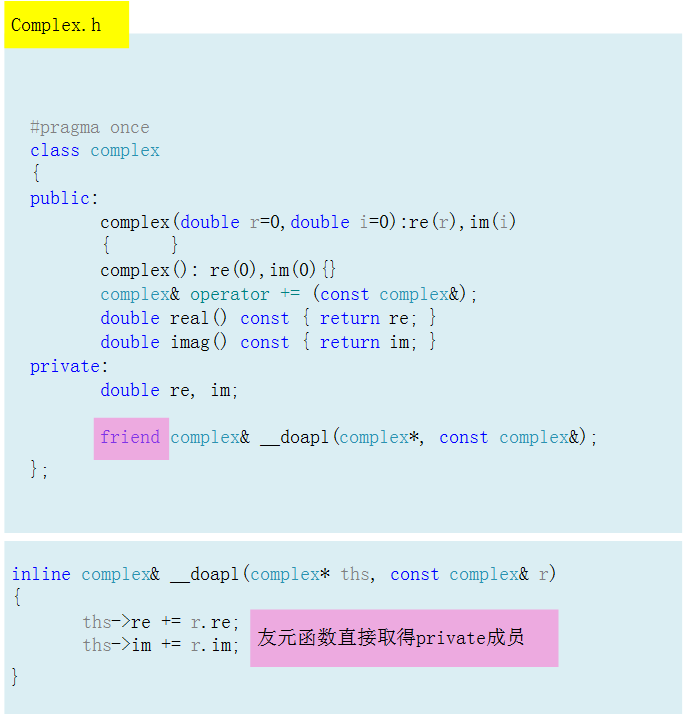
选择pass by reference多半是考虑到效率。



### 5. 友元函数（friend）

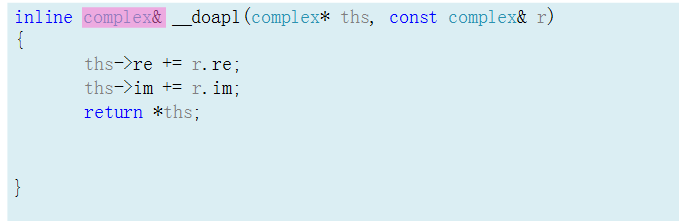
特点：

* 可以自由取得私有(区域)变量。



### 6. 不能返回局部变量的引用

函数之间在进行参数传递和值传递的时候，都是要尽量传递引用，但是在进行值传递的时候，有一种情况不能传递引用，这就是值被赋给一个函数内的临时变量的时候，因为临时变量的生命周期只在函数内，不在函数外。下面这个例子，ths本来就在，是非临时变量，所以可以传引用。



# 第四章 操作符重载和临时对象

### 1. 操作符重载之一：把操作符重载成成员函数

在C++里面，操作符是一种函数，这是C++的一大特点。

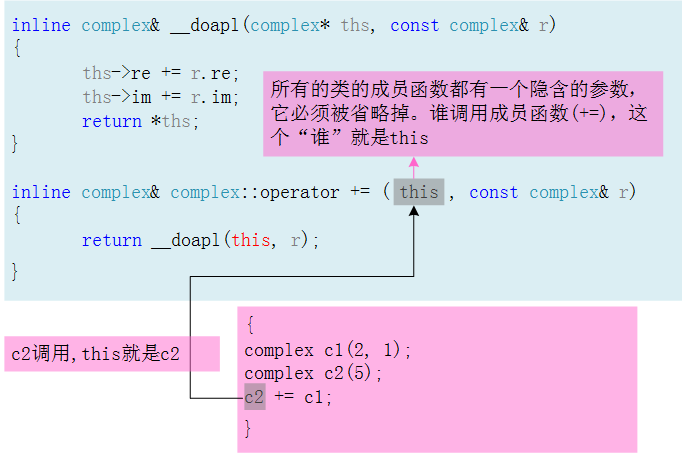
操作符是作用于左边的，比如a=b+c，最终会作用到a

操作符重载的原因：

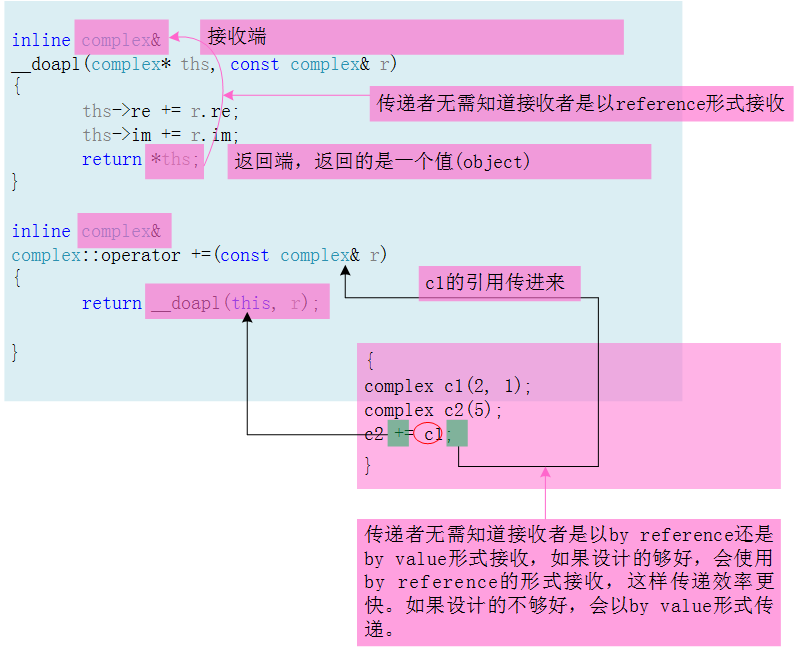
* 因为函数库中没有两个复数直接相加的函数，只有实数和实数相加的函数，如1.2+2.3=3.5。

对于成员函数：+=

* c2调用了+=，c2就是this，或者说this此时就指向了c2(this是一个指针).

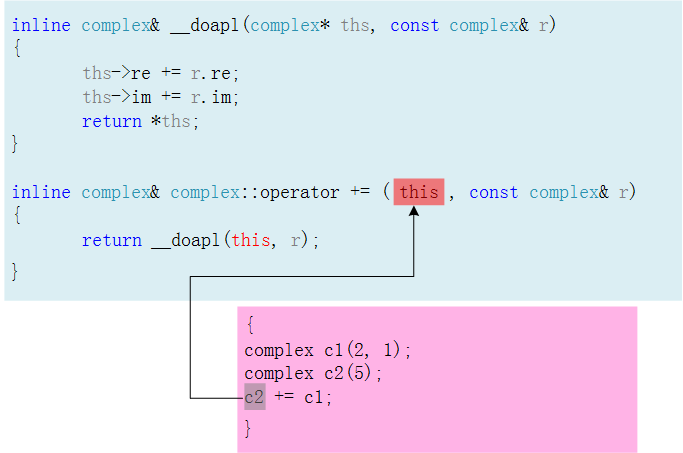


### 2. return by reference的故事（传递者无需知道接收端是以by reference还是by value的形式接收object,如果是前者，接收的就是object的地址）

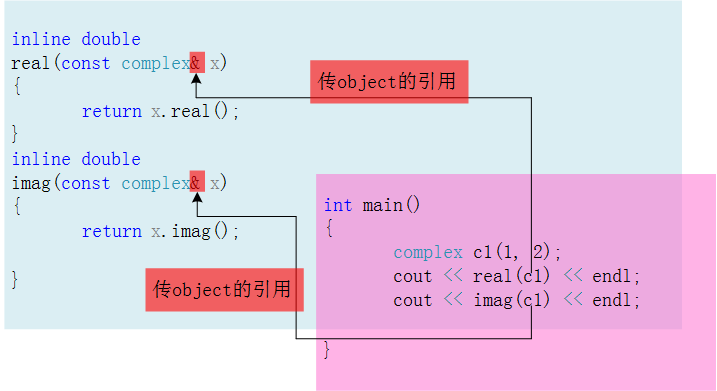


### 3. 返回值是写void还是写一个具体类型的故事

如果使用者只是这样调用+=这个重载函数：c2+=c1，那么返回类型complex&写成void是没有问题的。但如果这样调用：c3+=c2+=c1，返回void类型的话，就有问题了。



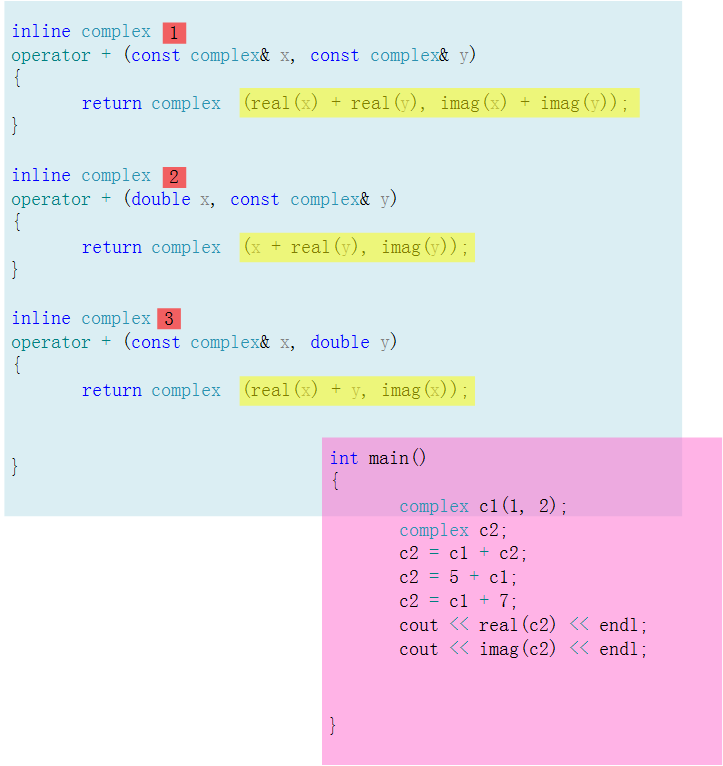
### 4. 在class body（类区域）外面定义函数



### 5. 操作符重载（操作符是非成员函数的情况，无隐含参数this(指针)）

为了应付客户的3种可能用法，需要在类体外写3个对应的处理加法的函数

* （1）复数+复数 c1+c2
* （2）实数+复数 5+c1
* （3）复数+实数 c1+7

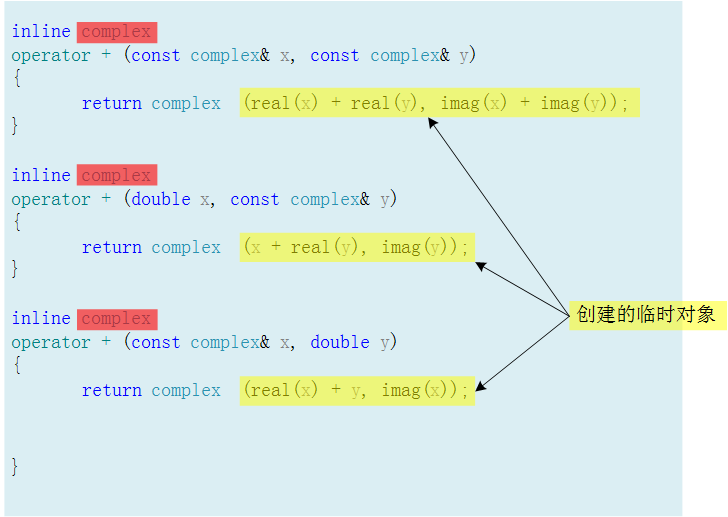


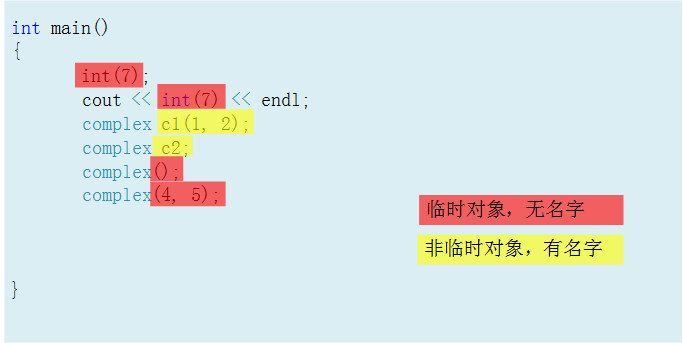
### 5. 临时对象（temp object）的故事

下面这些函数绝对不可以return by reference, 因为，它们返回的必定是个local object.

一定要return by value.

typename();(类名后面+小括号)，这是一个特殊语法，是创建临时对象的特殊语法。如下面函数体中的complex()就是在创建临时对象，临时对象的生命周期就是这创建的那一行，到下一行生命就结束了。

 再比如：

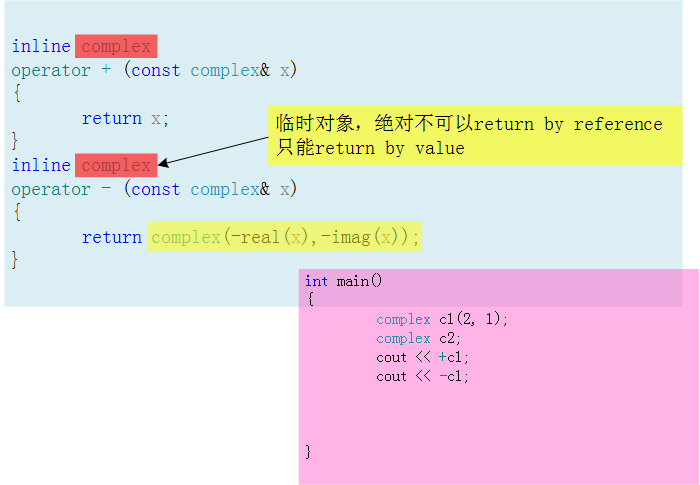


### 6. 单目运算符+，-的重载（正号操作，取反操作）

第一个正号操作，返回的是原来的东西，没有产生新的东西（新的local object，而下面代码中取反的操作产生了新的东西），那完全可以retrun by reference啊，这个是标准库里面的东西，那么厉害的人会注意不到这一点吗？有可能。这里其实可以retrun by reference。complex后面可以加个&

网友1：因为参数是const complex& x（const 型变量引用），如果返回值是引用的话，相当于去掉了const. 所以还是应该return by value. 实验了一下，果然是这样，retrun by reference（即写成inline complex&）的话，生成解决方案的时候会出错：错误C2440“return”: 无法从“const complex”转换为“complex &”

网友2：因为传入的参数是const类型的引用，如果将返回值改成引用，相当于将const类型转换为非const类型，这在编译上是通不过的。



### 7. 双目操作符“==”的重载

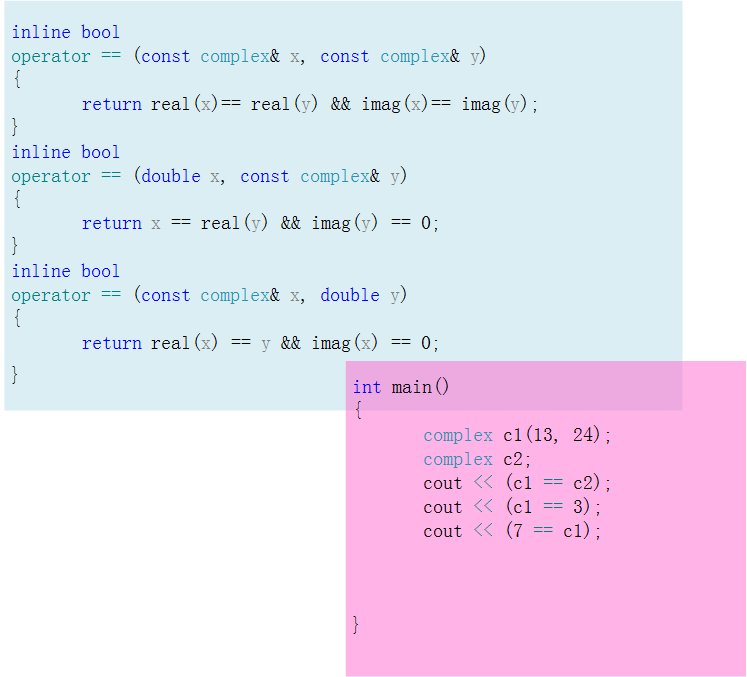
3种情况：

* 判断一个复数是否等于另一个复数
* 判断一个实数是否等于另一个复数

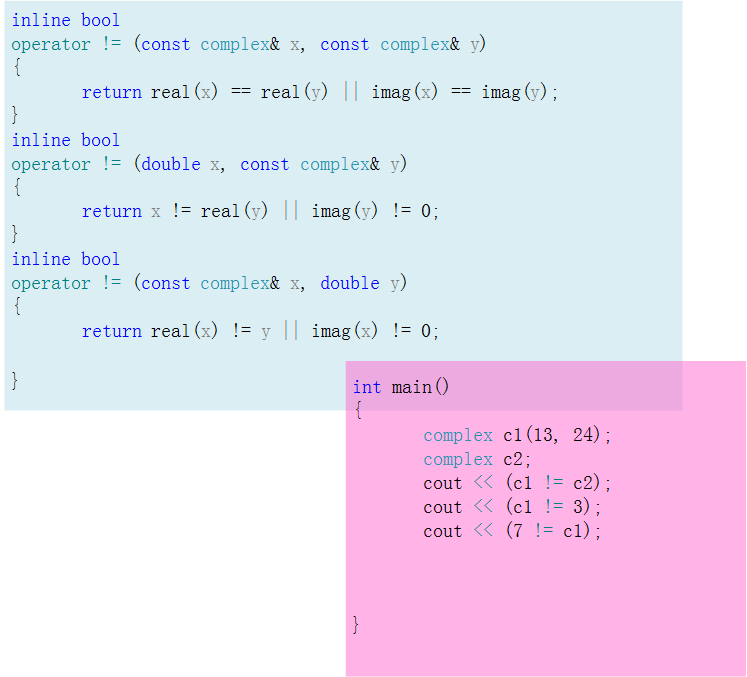
判断一个复数是否等于另一个实数

写一个函数一般考量两个点，这两点影响着程序执行的效率：

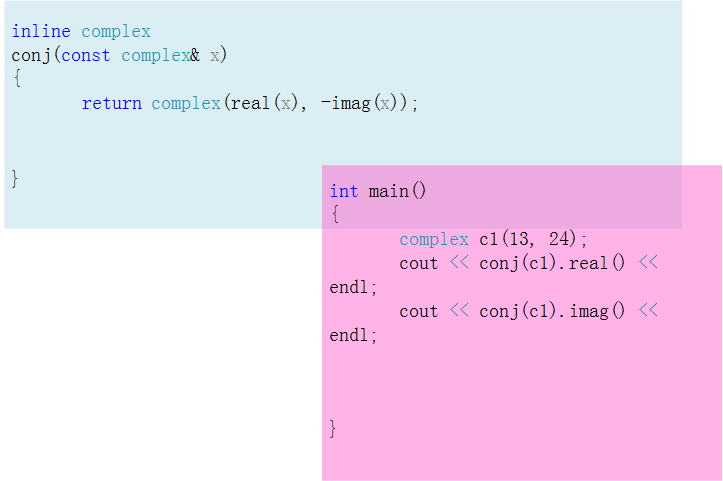
* 参数传递要不要by reference?
* 返回值传递要不要by reference?



### 8. 双目操作符“！=”的重载



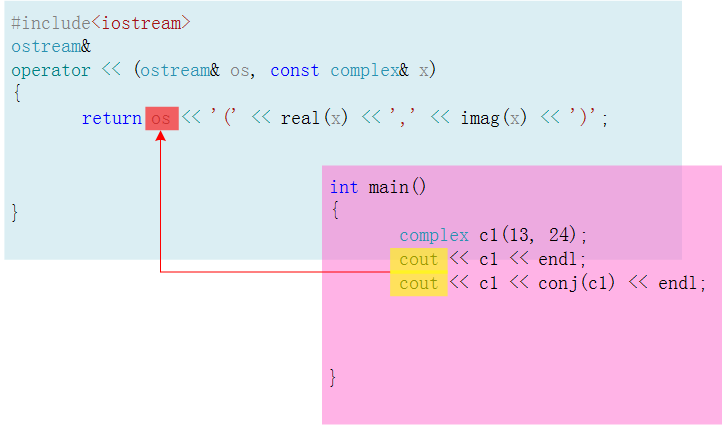
### 9. 共轭复数



### 9. 重载输出操作符“<<”进行复数或者共轭复数的输出

可以说的有以下几点：

* 参数传递，均是pass by reference；
* 返回值传递，return by reference；
* 如果不是连续输出，函数的返回值类型可以设计成void，如果使用连续输出，函数的返回值类型不能是void类型；
* “<<”是双目运算符，需要提供2个参数；
* 运算符永远是作用在左边的对象上，没有作用在右边上的；
* “<<”左边的“cout”是一个对象，这个对象的类别是：ostream，标准库中有这样一句：IMG_256  
  就像对象c1的类别是complex，对象7的类别是int，对象7.9的类别是double一样；
* cout<<c1; 这一句会返回一个ostream类型的对象（的引用）；
* 返回类型前不可以加"const",因为连续输出的时候被丢到屏幕上的东西一直在变，即每有一个新的输出变量“os”的状态就会发生变化（由于这个原因，ostream& os的前面不可以加const修饰符），相当于一个变量的值一直在发生变化；
* 这段代码是标准库中提供的例子。
* <<不可以写成成员函数，因为返回值是ostream类型的对象，这个类(ostream类)是标准库中定义好的，所以只能写成全局函数，不能写成局部函数。



 注解：

1. 函数体重多个<<相当于连续调用<<函数。
2. 如果<<的返回值类型是void,则无法连续调用，必须返回ostream类型的对象，才可以被继续调用。
3. return os <<'('<< real(x) << ',' << imag(x) << ')';  这一句其实很奇妙，因为：os <<'('  相当于是调用了<<函数，调用后的返回值是ostream这种类型的对象，这种对象作为参数继续被<<调用，继续输出x的实部(real())到屏幕上，然后再次返回一个ostream这种类型的对象，继续作为参数被<<调用，输出','到屏幕上，循环往复下去，直到输出所有你想输出到屏幕上的东西。
4. 变量尽量写在private区，因为变量不想被外界看到。
5. 函数尽量写在public区，因为函数主要用来被外界调用的。

# 复习Complex类是实现过程

### 1. 编程实例（来自于standard library）

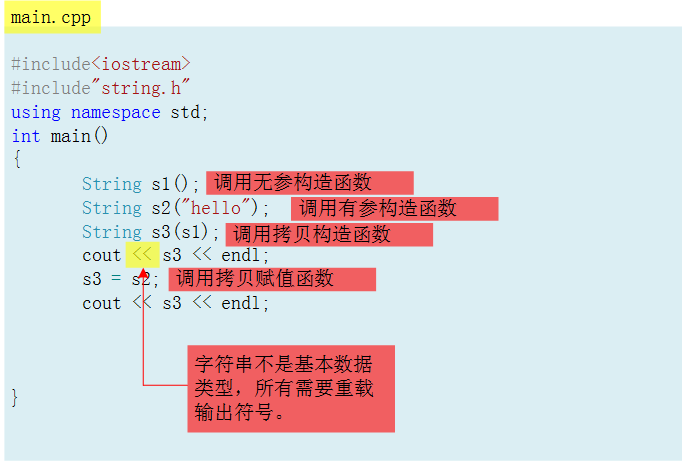
* 传递引用，占用的空间大小是4个字节（和系统有关）；
* double类型的数据，占用的空间是4个字节（和系统有关）；
* 传递引用是为了节约空间。

# 拷贝构造函数、拷贝赋值函数、析构函数

字符串类的例子，是侯老师自己写的，因为标准库的例子功能太复杂了。

编译器提供的拷贝赋值只是拷贝了对象的指针，没有真正拷贝对象。

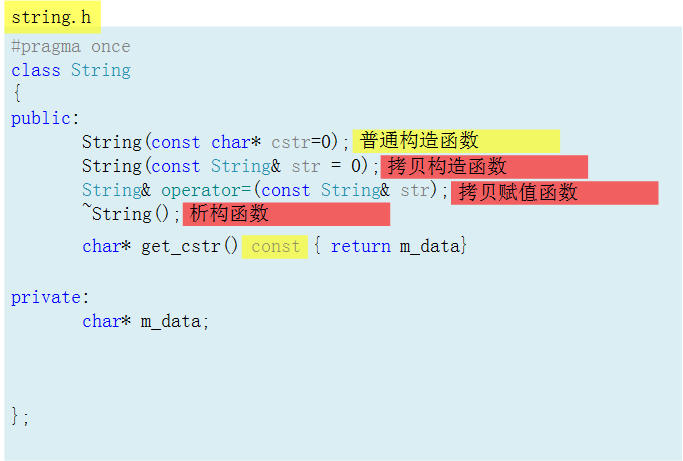
### 1. 拷贝构造函数和拷贝赋值函数



### 2. 3个特殊函数

只要类里面带有指向某个数据类型的指针，一定需要写拷贝构造函数和拷贝赋值函数。3个特殊函数（Big three）：

* 拷贝构造函数
* 拷贝赋值函数
* 析构函数



### 3. 构造函数和析构函数

* new就是要分配一块内存。
* 如果类里面含有指针，多半需要用new动态分配内存。
* 动态内存别忘记用delete 指针变量的形式释放掉，在new所在那一行离开作用域之前。
* 构造函数所占用的内存不是动态分配的，在离开它们所在的作用域的时候，析构函数自动被调用。
* 在为字符串分配内存的时候，都要多分配一个字节，用来存放结束符'\0'

 string.h

[IMG_256](javascript:void(0);)

#pragma onceclass String

{public:

String(const char\* cstr=0);

String(const String& str = 0);

String& operator=(const String& str);

~String();

char\* get\_cstr() const { return m\_data}

private:

char\* m\_data;

};

inline

String::String(const char\* cstr=0)

{

if (cstr)

{

m\_data = new char[strlen(cstr) + 1];

strcpy(m\_data, cstr);

}

else

{

m\_data = new char[1];

\*m\_data = '\0'; //这里也许可以这样写：\*m\_data='';(网友观点)

}

}

inline

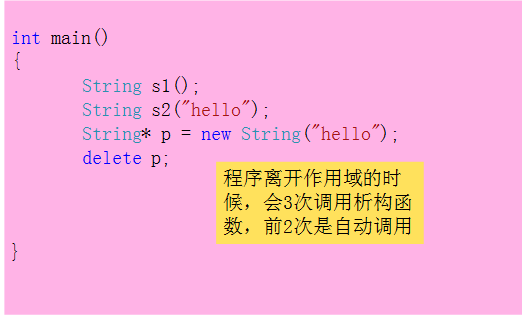
String::~String()

{

delete[] m\_data;

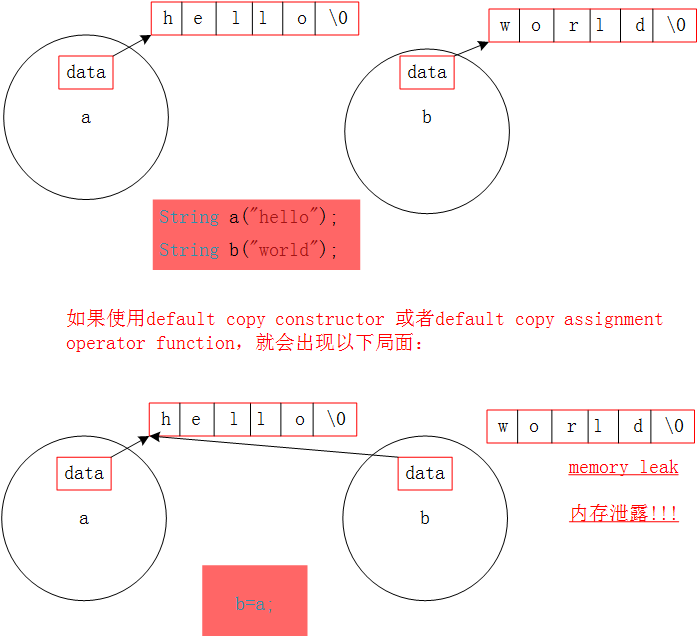
}

[IMG_257](javascript:void(0);)

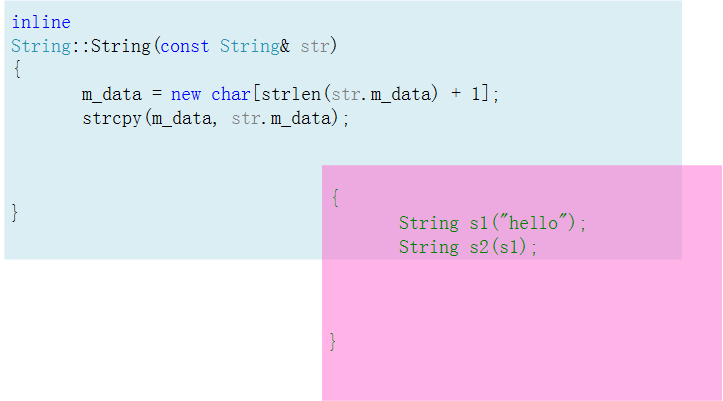


### 4. 只要类里面有指针，一定需要写拷贝构造函数和拷贝赋值函数，为什么？如果用标准库提供的拷贝构造函数或者拷贝赋值函数会怎样？

答：会出错，因为这是浅拷贝，执行b=a；后，本来是想把a的字符串值赋给b，但是这样只会让b指向同一块内存（所包含的内容），还会造成这样一个后果，假如改变a，b也会跟着改变，因为它们指向相同的内容。此时b叫做a的别名（alias），在编程里面，别名的存在是一件危险的事情。



### 5. 拷贝构造函数（深拷贝）



### 6. 拷贝赋值函数

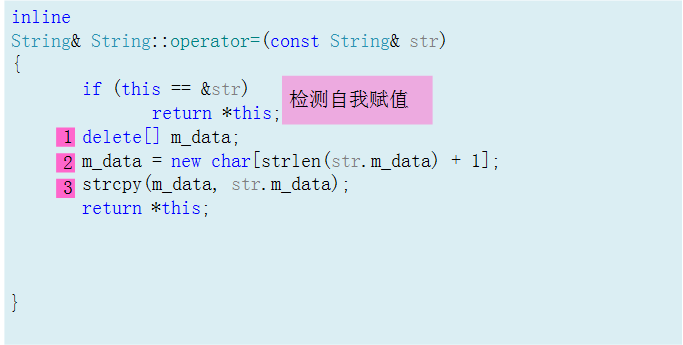
首先检测自我赋值，如果没有检测，会出现不好的结果。

拷贝赋值的具体步骤是：

* 杀掉自己；
* 开辟和赋值过来的对象一样大小的空间；
* 字符串复制。

拷贝赋值操作语句：a=b;如果指针a和b本来就指向同一块内存空间，杀掉a之后，相当于把b指向的内容也杀掉了，所以没办法做第2步了。

第2行里面的&代表对象的引用，第4行里面的&代表取地址。



### 7. 字符对象的输出----输出操作符重载

不可以写成成员函数，因为输出方向会相反。用户用的时候不方便：s1<<cout;

实际上这样用比较方便：cout<<s1;

网友1：当“<<”函数输出指向字符串的指针时，它会自动输出整个字符串，然而如果这个指针指向的是其它数据类型，则输出的指针变量的右值。

# 第七章 堆、栈与内存管理

### 1. 栈（stack）和堆（heap）

stack，是存在与某作用域（scope）的一块内存空间（memory space），例如当你调用函数，函数本身即会形成一个stack用来放置它所接收的参数，以及返回地址。

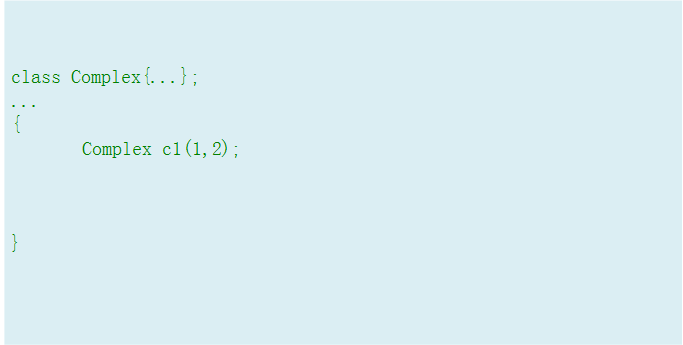
在函数本体（function body）内声明的任何变量，其所使用的内存块都取自上述stack.

Heap, 所谓system heap， 是指由操作系统提供的一块 global 内存空间，程序可动态分配（dynamic allocated）从其中获得若干区块（blocks）.

当离开作用域{}的时候,c1的生命自然就消失了， 即从栈中获得的内存，离开作用域的时候会自动释放。

当离开作用域{}的时候，动态分配的内存不会消失，即从堆中动态取得的内存不会自动消失，需要手动释放(delete 掉)。

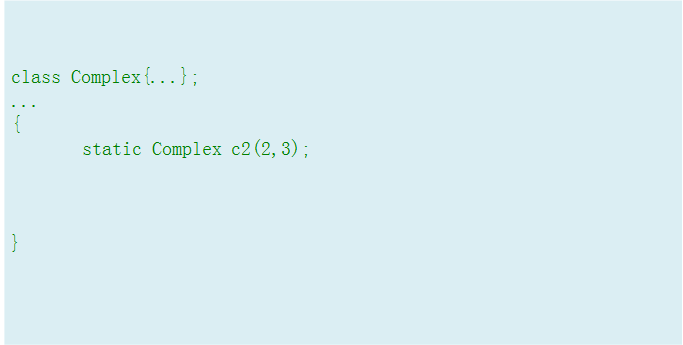
c1所占用的空间来自stack，它就是所谓的栈对象（stack object），其声明在作用域（scope）结束之际结束。这种作用域内的对象（object），又叫做local object, auto object, 因为它会被自动清理（意思是析构函数被自动调用）。



### 2. 静态对象

一个对象前面加上 static 修饰符后，c2既变成所谓的静态对象（static object）,其生命在作用域（scope）结束之后仍然存在，直到整个程序的结束。

如果你需要静态对象的这种特性，就要这样定义静态对象。

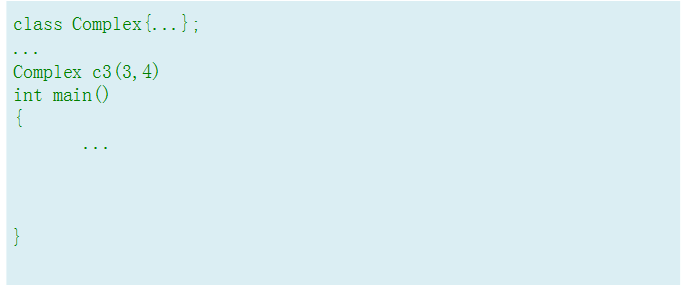


### 3. 全局对象

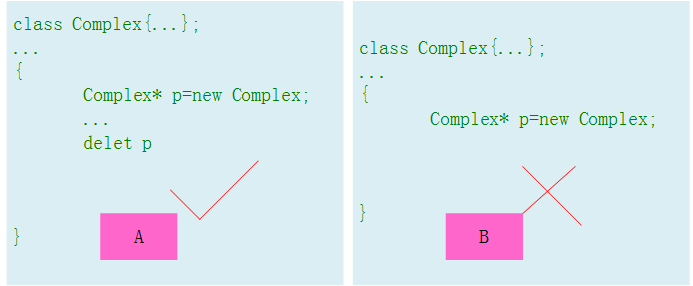
定义在任何作用域或者说大括号之外的对象。

一个对象的存在和消失取决于构造函数和析构函数何时被调用。

c3就是所谓的global object,其生命在整个程序结束之后才结束，你也可以把它视为一种static object,其作用域是整个程序。



### 4. heap object的生命周期



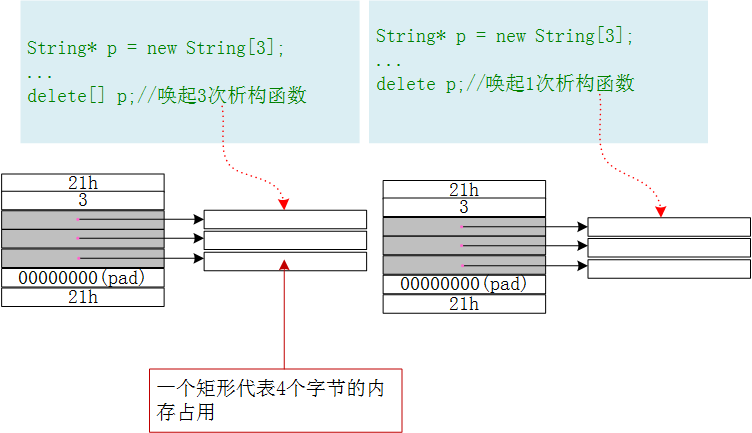
 A：p所指向的便是heap object(堆对象)，因为是动态从操作系统某个地方分配的内存，其生命周期在它被delete之际（此时默认析构函数或者手写析构函数被调用）结束。

B：会出现内存泄露（memory leak），因为当作用域结束后，p所指向的heap object仍然存在，但是指针p的生命却结束了，作用域之外再也看不到p,也就没机会delete p.

### 5. array new一定要搭配array delete,在使用指针新建字符数组的情况下，在手动释放内存的时候，如果delete[] p;写成了delete p;会造成内存泄露。

21h代表一个16进制的数（32）；

3代表新创建了3个字符串；



# 第八章 复习String类的实现过程

### 设计一个class，总是先思考需要什么样的(私有)数据,字符串里面一定会放很多字符。想法1是利用数组存放字符，但是这种想法不好，因为不知道要存放的字符有多少，占多少空间不知道，所以预先设定的大小不好确定。所以私有数据应该是一个指针(第12行)，将来放多大的内容，用new的方式动态分配内存大小。32位电脑上，一个指针占4个字节的大小（网友1:64位系统，vc中的指针变量大小也是4bytes）。

#pragma once

class String

{

public:

String(const char\* cstr = 0);

String(const String& str);

String& operator=(const String& str);

~String();

char\* get\_cstr() const { return m\_data; }

private:

char\* m\_data;

};

inline

String::String(const char\* cstr = 0)

{

if (cstr)

{

m\_data = new char[strlen(cstr) + 1];

strcpy(m\_data, cstr);

}

else

{//未指定初值

m\_data = new char[1];

\*m\_data = '\0';//空字符串也有最后的结束符

}

}

//拷贝构造函数

inline

String::String(const String& str)

{

m\_data = new char[strlen(str.m\_data) + 1];

strcpy(m\_data, str.m\_data);

}

//拷贝赋值函数

inline

String& String::operator=(const String& str)

{

if (this == &str)

return \*this;

delete[] m\_data;//先把自己（目的端）杀掉，

m\_data = new char[strlen(str.m\_data) + 1];//重新分配一块和来源端大小相同的空间

strcpy(m\_data, str.m\_data);//拷贝字符串

return \*this;

}

//操作符重载，输出字符对象

#include<iostream>

ostream& operator << (ostream& os, const String& str)

{

os << str.get\_cstr();//函数调用，取得字符对象的指针

return os;

}

inline

String::~String()

{

delete[] m\_data;//m\_data = new char[strlen(str.m\_data) + 1];属于array new,所以应该delete[] 指针

}

### 2. 接下来思考public:区域中需要设计哪些函数开放给外界调用。首先需要准备构造函数，main()函数中创建字符串，一定会调用构造函数（第5行），构造函数永远没有return type. 所以返回类型什么都不写。参数设计成指向字符串的指针，默认是0，对象一经构造出来，就不会改变，所以参数前面加上const修饰符。

### 3. 类里面含有指针，所以一定要考虑三大函数（拷贝构造，拷贝赋值，析构函数），拷贝构造函数也是构造函数，所以没有返回类型，既然是拷贝构造，参数就是它自己这种类型的东西（第6行），首先考虑用reference 传参，要不要加const? 传进来的知识一个等待赋给别人的蓝本，函数里面不会去改传进来的蓝本，所以加上const修饰符。

### 4. 拷贝赋值函数，参数是字符串类，用引用传参，赋值是把来源端拷贝到目的端，所以来源端不会被改变，所以加上const。来源端拷贝到目的端，得到的结果就是目的端这种东西，所以返回类型是String类的对象（第7行），返回值能不能传引用？因为函数执行的结果是放在了目的端对象上，而目的端对象本来就存在，并不需要在这个函数中去创建，所以不是临时对象，所以可以传引用。

### 5. 析构函数（第8行）。至此，三大函数的接口设计完毕。

### 6. 考虑下还要不要设计什么辅助函数了，由于后续希望把字符串丢到cout进行打印输出，所以应该再设计一个取得私有数据（字符串）的辅助函数（第9行）。m\_data是指针指向字符的字符串，是C风格的字符串，所以函数的名字起作get\_cstr(). 此函数如此简单，所以可以直线在class body内实现。要不要写成常量成员函数？答：要。因为只是取得私有数据，没有改变私有数据。构造函数，拷贝构造函数，拷贝赋值函数要不要设计成常量成员函数呢？答：不可以，因为其实那三个函数都有更改指针，构造函数就是要写私有数据的，拷贝构造也是写私有数据，拷贝赋值也是要写私有数据。实际上，所有类的big three都不能设计成常量成员函数。

### 7. 构造函数的实现

考虑分配足够的内存空间放初始的字符串，如“hello!”,判断传进来的指针是否指向空的内容，如果指向不为空的话，就动态分配传进来的字符串的长度+1这样长度的空字符数组，用来存放传进来的字符串，然后使用C的字符串拷贝函数把传进来的字符串拷贝到私有指针变量指向的新分配的空间中来。如果传进来的是空，也要准备一个字节的内存空间，放字符串结束符'\0';即空字符串有个含有一个字符，它是结束符'\0'。最后再写上inline修饰符，建议编译器帮忙做成内联函数。

### 8. 析构函数的实现

析构就是要把自己清理干净（第60行），也就是释放之前动态分配的内存。new的时候用的是数组，delete的时候也要这样写：delete[] m\_data;而不要这样写：delete m\_data; 最后要求编译器把它做成inline函数。

### 9. 拷贝构造函数的实现

参数：自己这种类型。

返回类型：无，因为拷贝构造函数也属于构造函数。

来源端当成蓝本，拷贝到目的端，实现过程跟上一个构造函数的思考差不多。

### 10. 拷贝赋值函数的实现

参数：字符串类型。

返回类型：来源端赋值到目的端，目的端就是自己这种东西，就是字符串类，传引用。

目的端是本来已经存在的东西，所以先把自己杀掉，杀掉后重新分配一块足够盛放来源端大小的内存，最后把来源端拷贝到目的端。如果是连续赋值，那么就要有返回值，返回值就是自己这种东西。如果不是连续赋值，可以把返回值设计成void。

拷贝赋值一定要关注自我赋值，即来源端和目的端本来就指向同一块内存。此时不需要做下面的拷贝动作，直接返回自己就行。

[IMG_256](javascript:void(0);)

//拷贝赋值函数

inline

String& String::operator=(const String& str)

{

if (this == &str)

return \*this;

delete[] m\_data;//先把自己（目的端）杀掉，

m\_data = new char[strlen(str.m\_data) + 1];//重新分配一块和来源端大小相同的空间

strcpy(m\_data, str.m\_data);//拷贝字符串

return \*this;

}[IMG_257](javascript:void(0);)

注：C++中，&符号出现的位置不同，其含义不同，第3行，代表一个类的引用，typename&，第5行代表取地址，得到是一个指针。

# 扩展补充：类模板，函数模板及其他

### 1.静态数据与非静态数据的区别

静态数据只有一份，位于内存的某个位置，比如，100万个人去开银行账户，那么账户的数量就是100万个，但是年利率都一样，所以年利率数据应该设计成静态数据，它就一份。

### 2. 静态成员函数与非静态成员函数的区别

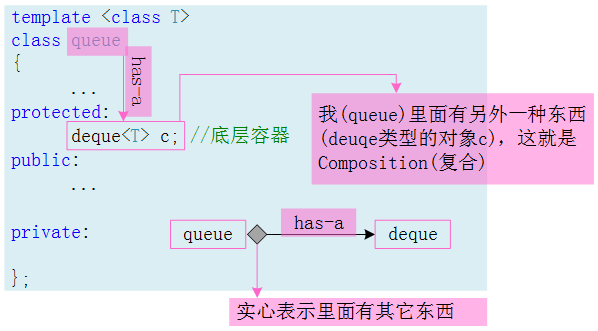
非静态成员函数有this 指针，静态成员函数没有 this 指针。

# 第十章 组合与继承

* Inheritance(继承)
* Composition(复合)
* Delegation(委托)

像字符串类，复数类一把不会和其他类发生关联，但有的类可能需要和其它类发生关联，这就是面向对象的思想。

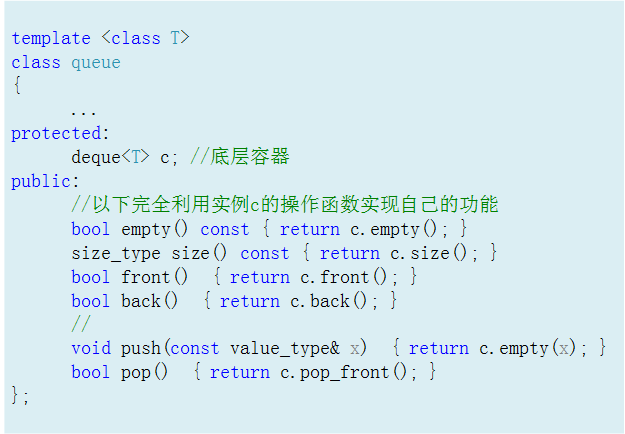
### 1.Composition(复合) ，表示has-a



注解：

1. queue跟deque的关系就是复合关系，这就是has-a关系，表示‘我’里面有另外的东西。
2. 联想C的结构(体)，C的结构体里面可以有别的结构体，也可以由别的整型，浮点型，字符型对象，这也是has-a的关系。
3. 所以，Composition表示一个类里面有其他类的实例(来自网友弹幕观点)。
4. 其实queue是个容器，它容纳了另外一个东西：deque类的实例。

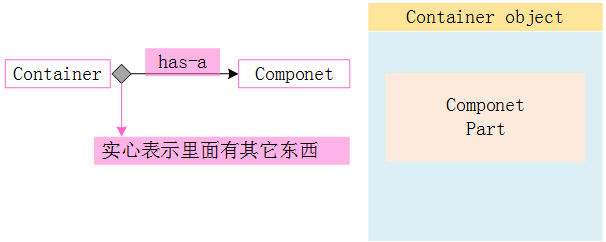
补全一点：



 注解：

1. queue的public区域中6个功能完全利用deque的实例c来实现。
2. 类复合的意义在于：A拥有B的实例，A的所有功能都利用B的实例来实现。
3. 其实这是标准库中的例子。queue队列：先进先出，deque(double ended queue)队列：两端进出，显然后者功能更加强大。
4. 最后一句，deque的pop\_front()什么都没做，直接改装成了queue的pop()。
5. 说不定queue里面的deque有100个功能，但是只开放了6个功能给queue.或者说queue只使用了deque里面的6个功能。
6. 这种改造叫做Adapter(改造，适配)。把已有的类的功能改造成客户所需要的其它类的功能。
7. 并不是所有的Composition都长这个样子哦，这只是一个特别的例子。

* Composition(复合)关系下的构造和析构



构造由内而外

Container的构造函数首先调用Component的default构造函数，然后才执行自己。

IMG_259

 注解：

1. 就像建造大楼一样，先建造里面，再建造外面。
2. 红色的Component()表示先调用内部的构造函数(如果不符合自己的要求，则要自己写哦)，{...}表示后做自己的事情。

析构由外而内

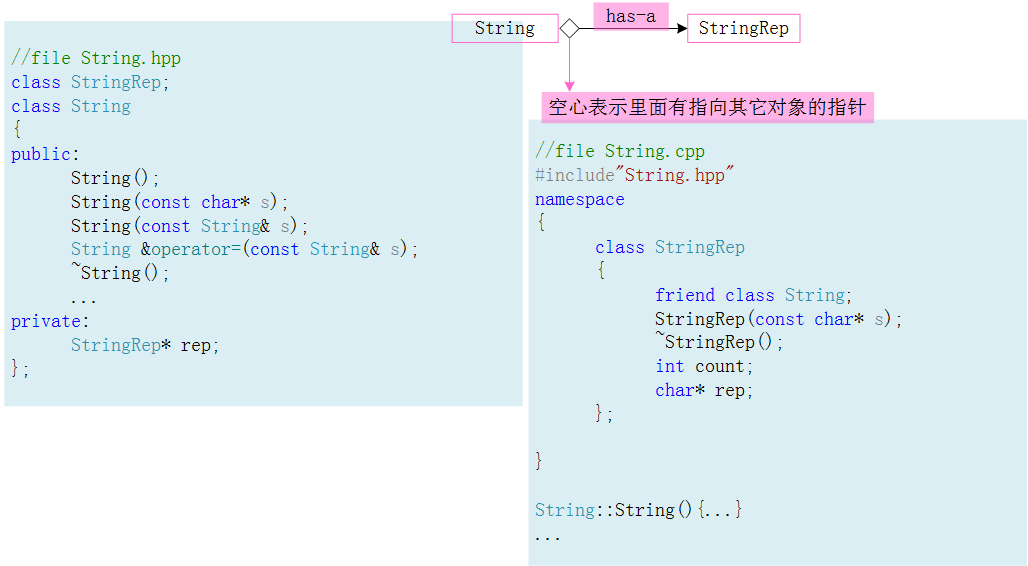
Container的析构函数首先执行自己，然后才调用Component的析构函数。

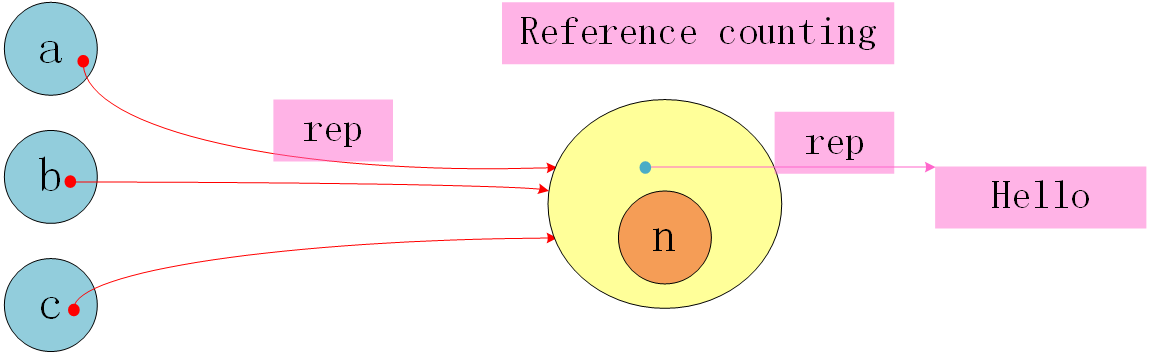
IMG_260

注解：

1. 就像拆大楼一样，先拆外面，后拆里面。就像剥葱一样，先剥外面的，再剥里面的。
2. {...}表示先做自己的事情，先执行自己的析构语句。

### 2.Delegation(委托). Composition by reference(两个类通过引用(指针)复合)

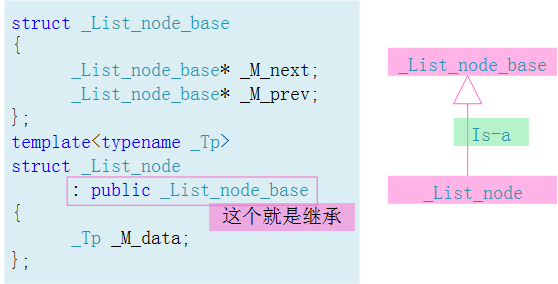




注解：

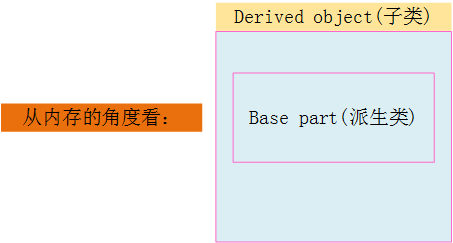
1. 左边仍然有一个右边的类的实例，但是这个“有”有点虚，不是那么扎实。只是有一个指向右边的类实例的指针，也可以说只是存储了另一个类的对象的首地址。这样的关系叫做Delegaion(委托)。
2. 在任何一个时间点，我想做事情的时候我就可以调用。
3. reference counting表示引用计数。a、b、c的内容都是Hello.
4. 内容共享，共享的前提是内容要一样。
5. 涉及到共享指针、写时复制、写时拷贝(copy on write)。

### 3.Inheritance(继承)，表示is-a



注解：

1. C++的继承方式有3种：public,private,protected,但是最重要的情况就是public.
2. 继承的特点：人是一种动物，动物是一种生物。香蕉是一种水果。这些都是is-a的关系。
3. 数据是可以完整的继承下来的。
4. \_List\_node所创建的实例将拥有三个数据成员，自己的1个加上继承自\_List\_node\_base的两个。这一点并不是继承这个语法最有价值的部分。最有价值的部分是虚函数概念。



注解：

1. 子类的对象有父类的成分在里面。
2. base class的dtor必须是virtual,否则会出现undefined behavior.

 构造有内而外(这样才坚固)

Derived的构造函数首先调用Base的default构造函数，然后才执行自己(...)。

IMG_265

 析构有外而内(这样也才坚固)

Derived的构造函数首先执行自己(...)，然后才调用Base的析构函数。

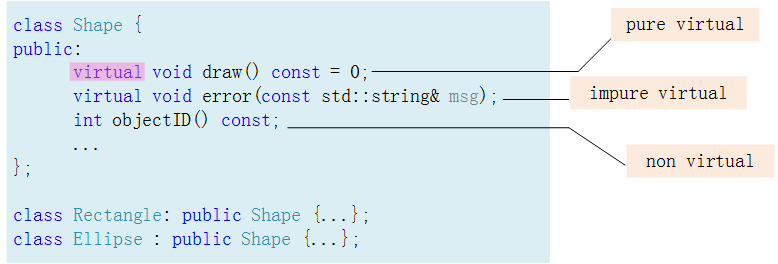
IMG_266

# 第十一章 虚函数和多态

### 详细介绍：[C++中的virtual关键字 | 从零开始的BLOG (hellozhaozheng.github.io)](https://hellozhaozheng.github.io/z_post/Cpp-virutal%E5%85%B3%E9%94%AE%E5%AD%97/)

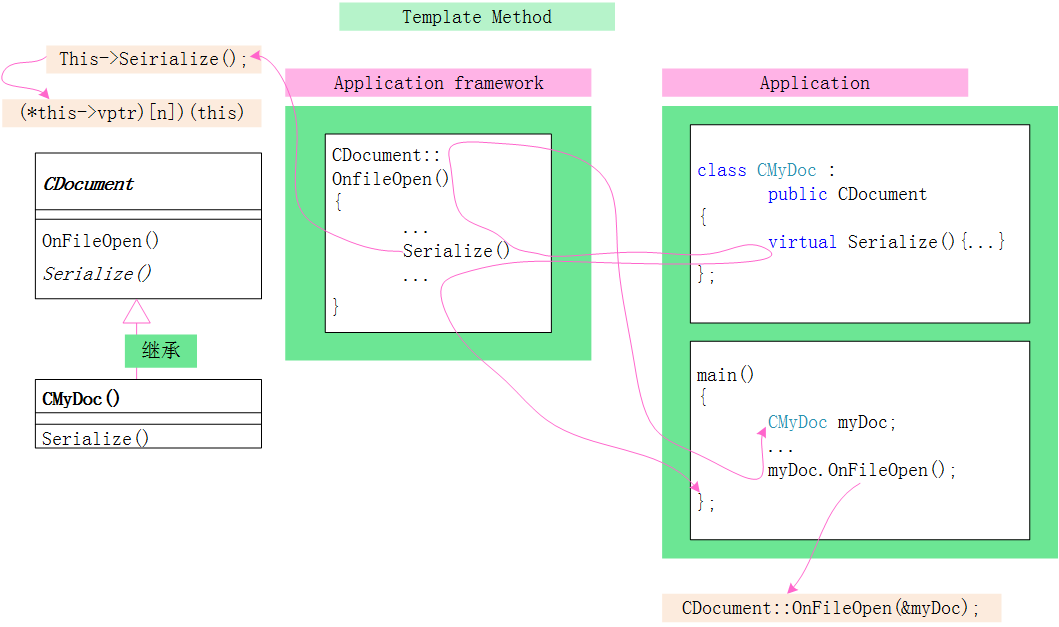
### Inheritance(继承) with virtual functions(虚函数)

1. non-virtual函数(不是虚函数)：你希望drived class(子类)不要重新定义(overide，覆盖)。
2. virtual(虚)函数：你希望devired class(子类)最好去重新定义(overide)，且你对它(指父类的虚函数)已有默认定义。
3. pure virtual(纯虚)函数：你希望devired class(子类)一定重新定义(overide)，你对它(指父类的虚函数)完全没有默认的定义(其实可以由定义，但你不去定义它)。
4. 写笔记的时候可以考虑多用点副词，以加深印象。



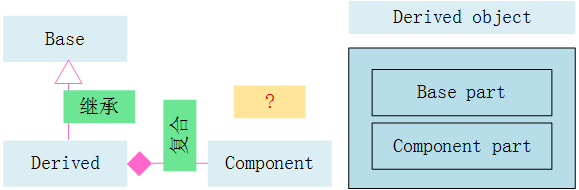
注解：

1. 任何一个成员函数前面加上一个virtual进行修饰，它就会成为虚函数。
2. 在继承的关系里面，所有的东西都可以被继承下来，这包括私有数据和函数的调用权限。函数的继承继承的是调用权。
3. 子类可以调用父类的所有成员函数，但子类在调用的时候要不要重新定义呢？见行文一开始的1、2 、3.

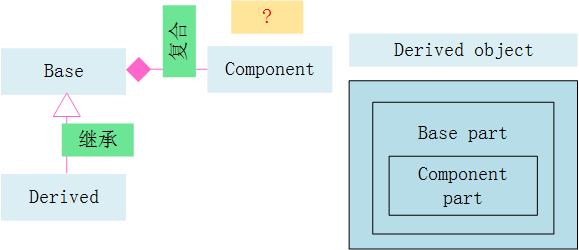
注解：

1. 父类有时候只设计了一个框架，有些动作父类不知情，所以完成不了，这时候就需要设计成虚函数，在子类中实现。
2. 在main()函数中，首先创建一个子类的对象，通过子类对象调用父类函数(在面向对象的编程中，这是一个非常经典的想法)。
3. 最关键的那一刻是：Serialize()的实现在子类中，调用路线是遇到父类中的Serialize()就跑去子类中找具体的实现。
4. 父类中的Serialize()函数也许3年前就设计好了，但是一直没实现，等着客户去实现，这就是设计模式、模板模式(Templat Method)。
5. myDoc调用父类的OnFileOpen(),myDoc(谁调用我的那个谁)就会成为隐藏的this pointer.从编译器的角度考虑，它会这样写CDocument::OnFileOpen(&myDoc);&myDoc，即myDoc的地址是隐藏的参数，它将被传到父类中的OnFileOpen()函数的参数中。
6. this->Seirialize(); 值通过this 调用Seirialize()，this是谁？this是&myDoc。

* Inheritance+Composition关系下的构造和析构函数的调用顺序(?)



注解：



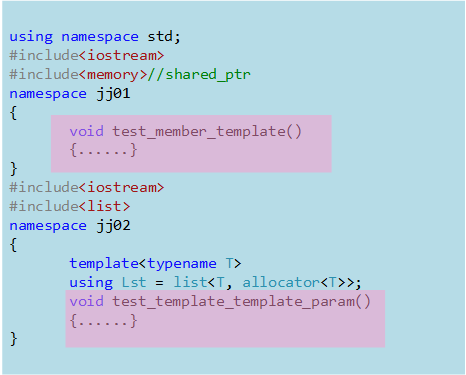
 注解：

1. Base,Component, Derived可以认为是类名。
2. 构造函数的调用顺序是：由内而外。
3. 析构函数的调用顺序是：跟构造函数调用顺序相反。

# 第十二章 namespace经验之谈

这是个小话题。

其实就是把一些东西区分开来，办公室A和办公室B缩写的function可能不一样，我们各自取个名字把自己的东西包起来。



int main()

{

jj01::test\_member\_template();

jj02::test\_template\_template\_param();

}

注解：

1. 这就是在不同的命名空间调用各自空间内的函数，即便函数重名也没关系。

# template模板类

不用模板的情况：

[IMG_256](javascript:void(0);)

class complex

{

public:

complex(double r=0,double i=0):re(r),im(i){}

complex& operator +=(const complex&);

double real() const { return re; }

double imag() const { return im; }

private:

double re, im;

friend complex& \_\_doapl(complex\* , const complex& r);

};[IMG_257](javascript:void(0);)

{

complex c1(2.5, 1.5);

complex c2(2, 6);

}

使用模板的情况：

[IMG_258](javascript:void(0);)

template<typename>

class complex

{

public:

complex(T r=0,T i=0):re(r),im(i){}

complex& operator +=(const complex&);

T real() const { return re; }

T imag() const { return im; }

private:

T re, im;

friend complex& \_\_doapl(complex\* , const complex& r);

};[IMG_259](javascript:void(0);)

{

complex<double> c1(2.5, 1.5);

complex<int> c2(2, 6);

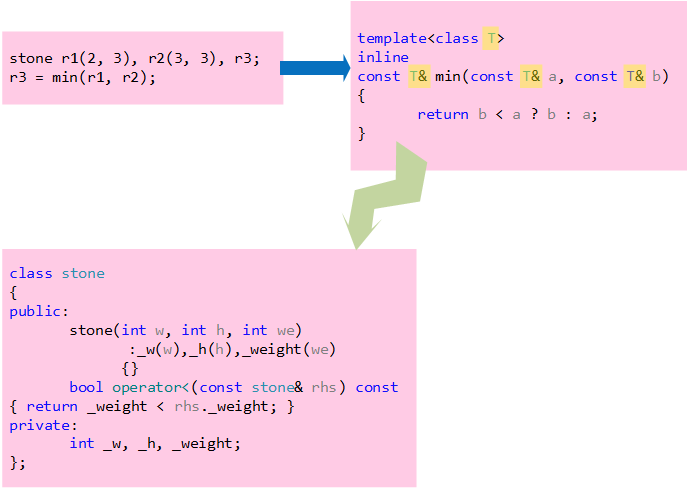
}

注解：

### 在设计一个class的时候，你认为你可以把哪些类型抽出来，允许用户在用的时候再指定，你就把这些类型抽出来。

# function template函数模板

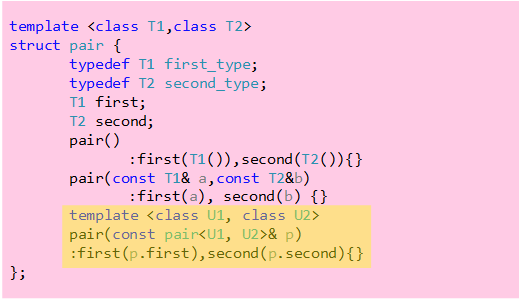
### 函数模板在使用的时候不必指明类型。



注解：

1. template<class T>里面的class 可以换成typename.
2. T是什么都可以。
3. 这样的函数模板的使用会比类模板更简单。类模板用的时候还要指定数据类型，函数模板在使用的时候不必指明它的type.即不必指明T是什么。为什么呢？因为函数模板在使用的时候一定是去调用，编译器会对function template 进行实参推导(argument deduction)，自动推导出所调用函数的参数类型。假如推出来类型是stone，那么它会想，它能不能比大小呢？此时它会找有没有"<"号的重载,如果找不到小于号"<"的重载定义，那么就无法编译，或者说会编译报错。

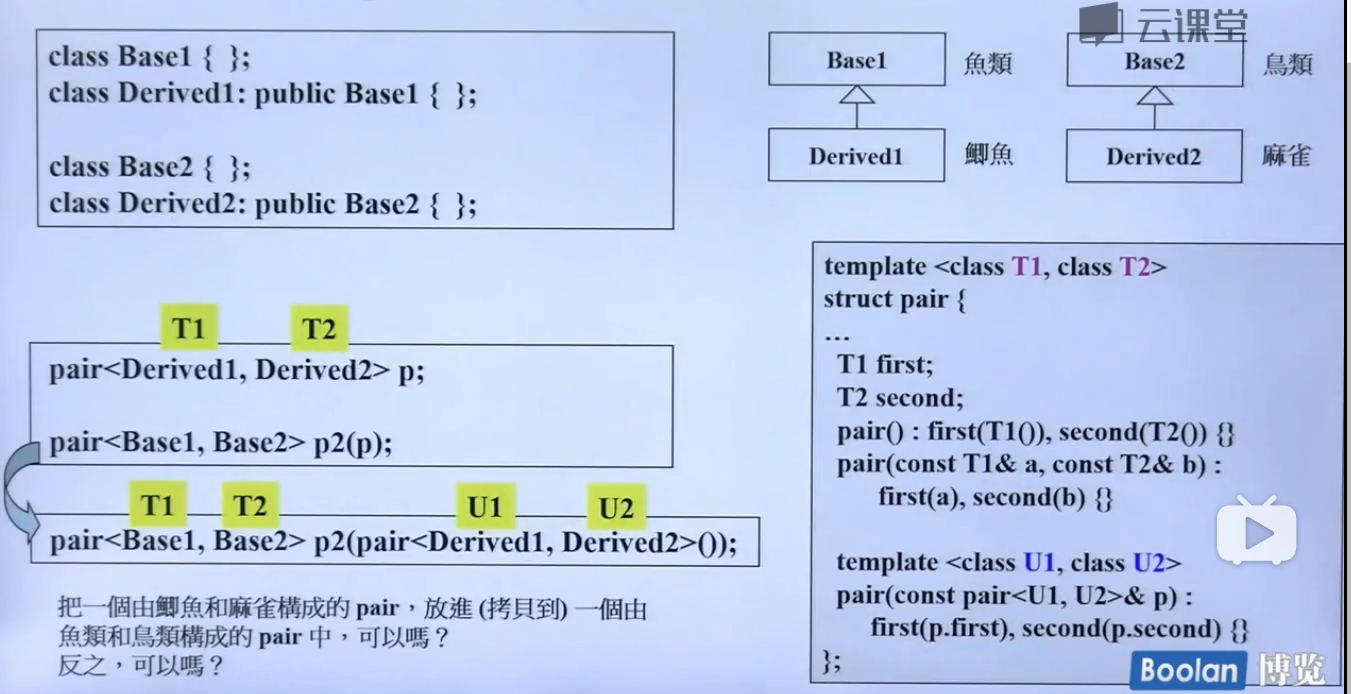
# member tamplate成员模板



注解：

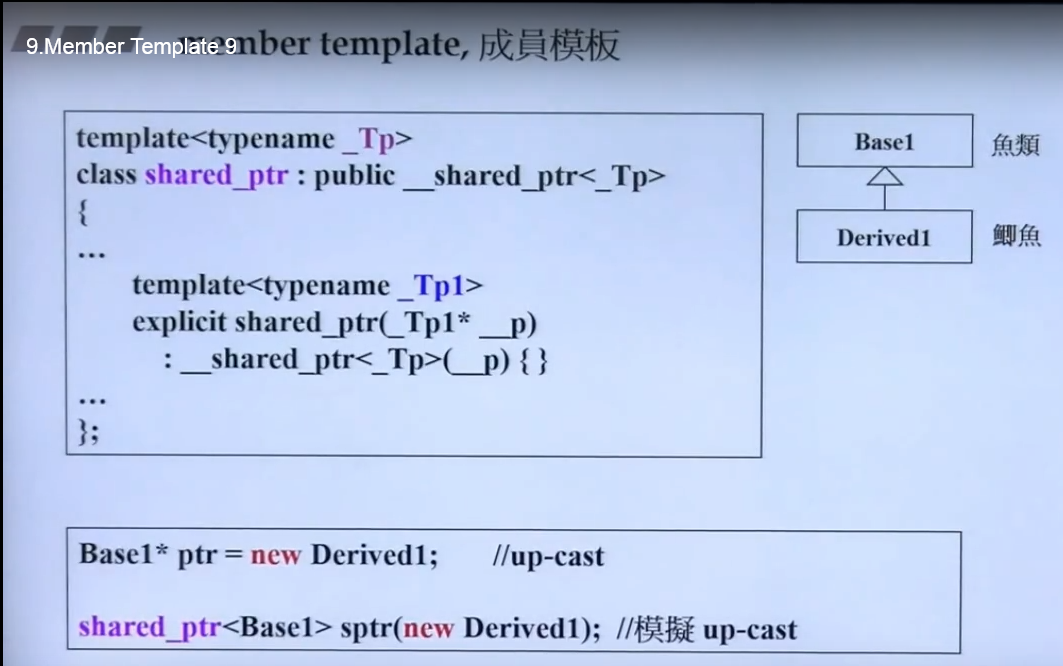
1. 黄色部分就是成员模板。
2. 黄色那一块是模板里面的一个成员，而它本身又是一个template, 黄色那一段就叫做成员模板。
3. 外头那个模板是个允许变化的东西，T1和T2可以变化。而加入T1和T2确定之后，里面的模板(黄色段落)又可以变化，U1和U2可以变化。

举一个例子：



 注解：

1. 首先制造一个p,p的类型是pair类。拿T1、T2来制造，实际上指的是鲫鱼和麻雀，把鲫鱼和麻雀做成一对。
2. pair<Base1,Base2> p2(p); 是把鸟类和鱼类做成一对，反之，则不行。
3. first(p.first),second(p.second)  :把初值的头尾放进来当成我本身的头尾。初值是鲫鱼和麻雀，放进来，鲫鱼是一种鱼类，麻雀是一种鸟类。
4. 成员模板是为了让构造函数更有弹性。



 注解：

1. Base1\* ptr = new Derived1;  如果new一个derived(鲫鱼)，但指针是指向鱼类，这是可以的。派生类一般画在下面，基类一般画在上面，这个动作有点相当于指针向上移动，这叫up-cast. 既然指针可以，智能指针也一定可以，为了这个可以，智能指针要写一个成员模板。
2. 派生类可以赋值给基类对象，反之不行(网友)。

# 智能指针

一个C++的class，做出来后，可能会像两种东西。

1. 所产生出来的对象像一个指针。所以叫做pointer-like classes.
2. 所产生出来的对象像一个函数，有点特别哦。

为何设计一个类产生的对象要像一个指针，因为你想要它做比指针更多的事情，所以通常这样做出来的东西，又叫做智能指针(更聪明的指针)。

[IMG_256](javascript:void(0);)

#pragma onceclass share\_ptr

{public: T& operator\*() const

{return \*px;} T\* operator->() const

{return px;}

shared\_ptr(T\* p): px(p){}private:

T\* px;

long\* pn;

......

};

[IMG_257](javascript:void(0);)

注解：

1. T\* px; px是指向T类型的指针。
2. C++里面的操作符重载是很常见的，且是很重要和强大的。智能指针的这两个写法几乎是固定的。
3. 聪明的指针都会有这么一个构造函数：

shared\_ptr(T\* p): px(p){}

1. 这个构造函数的参数接受天然的指针，C++的指针。

struct Foo

{

......

void method(void) {......}

};

shared\_ptr<Foo> sp(new Foo);

Foo f(\*sp);

sp->method();

px->method();

注解：

1. 假设现在写了一个Class,叫做Foo, 要把Foo这种天然的指针(new Foo，new就是得到一根指针)包装到这个聪明的指针里面去，即shared\_ptr指针里面。
2. sp->method();   智能指针调用method()方法。操作符'->'属于调用操作符重载哦。这句就相当于px->method(). 操作符->作用在指针对象sp上，得到指针对象px。

* Pointer-like classes的第二个大类是关于迭代器

迭代器就是要代表容器里面一个元素，因此它也像一个指针，也可以说它是一个智能指针。指针++就是向前移动，--就是向后移动。++、--都是智能指针的操作符重载。

[IMG_258](javascript:void(0);)

template <class T, class Ref， class Ptr>

struct \_\_list\_iterator { //这是一个链表

typedef \_\_list\_iterator<T, Ref， Ptr> self;

typedef Ptr pointer;

typedef Ref pointer;

typedef \_\_list\_node<T>\* link\_type;

link\_type node;

bool operator==(const self& x) const { return node == x.node; }

bool operator==(const self& x) const { return node != x.node; }

reference operator\*() const { return (\*node).data; }

pointer operator-> const { return &(operator\*())}

self& operator++() { node = (link\_type)((\*node).next); return \*this }

self& operator++(int) { self tmp = \*this; ++\*this; return tmp; }

self& operator--() { node = (link\_type)((\*node).prep); return \*this }

self& operator--(int) { self tmp = \*this; --\*this; return tmp; }

};[IMG_259](javascript:void(0);)