条款01：视C++为一个语言联邦

C++主要的次语言总共只有四个(sublanguage)

C：C++仍然是以C为基础，区块，语句，预处理器，内置数据类型，数组，指针等

Object-Oriented C++：这部分就是C with Class，类，封装，继承，多态，虚函数等

Template C++：这是C++的泛型编程，template metaprogramming (TMP 模板元编程)

STL：template程序库，由容器，迭代器，算法以及函数对象所组成

- 每个次语言都有自己的规约，C++高效编程守则视状况而变化，取决于你使用C++的那一部分

条款02：尽量以const，enum，inline替换 #define

可以改成宁可以编译器替换预处理器

因为define不视为语言的一部分，define出来的记号名称不会进入记号表（symbol table），这就可能导致debug时候的问题

当使用const替换define时，有两种特殊情况需要讨论

第一种是常量指针，我们有必要将指针（而不只是指针所指之物）声明为const 比如在头文件内定义一个常量字符串

`const char* const str = "hello world"`

string对象通常比字符数组合适 所以上面的代码往往定义成这样更好些

`const std::string str("hello world")`

第二种是class专属常量，为了将常量的作用域限制于class内，我们需要把它变成class的一个成员，而为了确保此常量至多只有一份实体，我们需要让他成为一个static成员

`static const int Num = 5`

包含在头文件中的这段代码是一个声明式而非定义式，通常C++要求我们对所使用的东西提供一个定义式，但是如果他是class专属常量又是static且为整数类型（注意不是int型，而是int,char,bool等）则需要特殊处理，只要不取他们的地址，就可以直接使用它们而不需要提供定义式，如果我们坚持使用他们的地址，或者编译器（错误的）坚持要看到一个定义式，我们就必须提供定义式如下

`const int GamePlayer::Num;`

这个要放到实现文件而非头文件中，由于声明时已经获得了初值，所以在这里可以不再设置初值

旧式编译器或许不支持上述语法，他们不允许static成员在其声明式上获得初值，此外所谓的in-class初值设定，也只允许对整数常量进行，如果编译器不支持，那么我们可以将初值放到定义式：

```
static const double Factor;
const double Estimate::Factor = 1.35;
```

还有一个例外就是如果你在class编译期间需要一个class常量值，比如用在数组声明式中（编译器坚持必须在编译期间知道数组的大小）。如果你的编译器（错误的）不允许static整数型class常量完成in class的初值设定，你可以改用所谓的‘the enum hack’补偿做法。其理论基础是：一个属于枚举类型的数值可权充int被使用

```
enum {Num = 5};
int scores[Num];
```

enum hack行为比较类似define而不是const，enum类型不允许取地址，所以如果你不想让别人获得一个pointer或reference指向你的某个整数常量 enum可以帮助你实现（反外挂手段）

使用inline代替宏，考虑如下代码

```
#define CALL_WITH_MAX(a, b) f((a) > (b) ? (a) : (b))    //以a和b中较大值调用f
int a = 5, b = 0;
CALL_WITH_MAX(++a, b);                  //a被加一次
CALL_WITH_MAX(++a, b + 10);         //a被加两次
```

- 使用inline可以获得宏带来的效率以及一般函数的所有可预料行为和类型安全性
- 对于单纯常量，最好以const或enum替换#define
- 对于形似函数的宏，最好改用inline函数替换#define

条款03：尽可能使用const

如果关键字const出现在星号左边，表示被指物是常量，如果出现在星号右边，表示指针自身是常量

如果被指物是常量，有的人会把const写在类型之前，有个人会把他写在类型之后，意义相同

```
void f1 (const Widget *pw);
void f2 (Widget const *pw);
```

都代表这参数是一个指向常对象的指针

STL迭代器的一个例子

```
std::vector<int> vec;
const std::vector<int>::iterator iter = vec.begin();        //相当于T* const
*iter = 10;                                                                    //改变iter所指没问题
++iter;                                                                          //iter不能改变
std::vector<int>::const_iteraotr cIter = vec.begin();   //相当于const T*
*cIter = 10;                                                                 //被指物是const 不能改变
++cIter;                                                                        //可以改变
```

令函数返回一个常量值，往往可以降低因客户错误而造成的意外

考虑如下声明式`const Rational operator (const Rational &lhs, const Rational &rhs)`

为什么要返回常对象？否则用户可能实现这样的暴行 (a * b) = c;

将const实施与成员函数的目的，是为了确认该成员函数可作用于const对象身上，这一类成员函数之所以重要，第一，它们使class接口比较容易被理解——得知那个函数可以改动对象内容而那个函数不行，第二，它们使操作const对象成为可能，这对高效编写代码是个关键（改善c++程序效率的根本办法是以pass by reference to const的方式传递对象，而这项的前提是我们由const成员函数来处理取得的const对象）

C++的一个重要特性：两个成员函数如果只是常量性不同，是可以被重载的

成员函数如果是const意味什么？

两个流行概念，bitwise constness和logical constness

对于bitwise constness，只要不改变对象内的任意一个bit，就可以说是const，但是一个更改了指针所指物的成员函数虽然不能算是const，但是如果只有指针属于该对象，那么称此函数为bitwise const不会引发编译错误

对于logical constness，一个函数可以修改它所处理的对象内的某些bits，但只有在客户端侦测不出的情况下才得以如此（即逻辑常函数，用户不会涉及到即可，为了高效性我们还是可以部分改变的）

在logical constness的实现中，如果要在const成员函数来修改值怎么办？ 使用C++的一个与const相关的摆动场：mutable（可变的）mutable可以释放掉non-static成员变量的bitwise constness约束

假设我们在实现过程中需要执行边界检查，日记访问等操作

```
class TextBlock {
public:
    const char& operator[] (std::size_t position) const {
        ... //边界检查
        ... //数据访问等
        return text[position];
    }
    char& operator[] (std::size_t position)  {
        ... //边界检查
        ... //数据访问等
        return text[position];
    }
}
```

如果每个函数我们都要实现类似这样的代码，肯定是很麻烦的

所以我们应该实现operator[]的机能一次并使用它两次，也就是说，我们应该让一个调用另一个，这将促使我们将常量性转除

上面的例子中，const函数做了non-const函数所做的一切，唯一的不同是返回类型加了一个const修饰

我们令non-const operator调用其const兄弟是一个避免代码重复的安全做法

```
const char &operator(std::size_t position) const {
    return text[position];
}
char &operator(std::size_t position) {
    return const_cast<char&>(static_cast<const TextBlock&>(*this)[position]);
}
```

解释一下上面的代码，首先将*this将其原始类型TextBlock& 转换成了const TextBlock&，为其添加上const，然后在接下来调用的时候使用的就是const版本的函数，再从其返回值中移除const

有关static_cast 和 const_cast 之后会提到

还有一点，令const版本调用non-const版本是我们不应该做的事，const成员函数承诺绝不改变其对象的逻辑状态（logical state），non-const成员函数没有这样的承诺，如果你在const内部调用了non-const成员函数，就违背了承诺，这样就承担了在其过程中改变对象的风险。

- 将某些东西声明为const可帮助编译器侦测出错误用法。const可被施加于任何作用域内的任何对象，函数参数，函数返回类型，成员函数本体。
- 编译器强制实施bitwise constness，但你编写程序的时应该使用“概念上的常量性”（conceptual constness）

- 当const和non-const成员函数有着实质等价的实现时，令non-const版本调用const版本可避免代码重复

条款04：确定对象被使用前已先被初始化

这里指的对象包括用户自定义的数据类型，以及内置数据类型（比如int，char等）

注意，在某些平台上，仅仅只是读取未初始化的值，就可能让你的程序终止运行

```
int x = 0;          //对int进行手工初始化
const char* text = "hello world"    //对指针进行手工初始化
double d;
std::cin >> d;      //以读取input stream的方式完成初始化
```

对于内置类型以外的数据，确保每一个构造函数都将对象的每一个成员初始化

```
ABEntry::ABEntry(const std::string &name, const std::string& address, const
std::list<PhoneNumber>& phones) {
    theName = name;             //这些都是赋值(assignments)
    theAddress = address;   //而非初始化(initializations)
    thePhones = phones;
    numTimesConsulted = 0;
}
```

虽然这会导致ABEntry对象带有你期望的值，但这并不是最佳做法。

C++规定，对象的成员变量的初始化动作发生在进入构造函数本体之前

（但是对于内置类型，不保证一定在你所看到的那个赋值动作的时间点之前获得初值）

还记得初始化列表吗？member initialization list

```
ABEntry::ABEntry(const std::string &name, const std::string& address, const
std::list<PhoneNumber>& phones)
    :theName(name),
     theAddress(address),
     thePhones(phones),
     numTimesConsulted(0)
{ }                                 //现在，这些都是初始化。构造函数的本体不必有任何动作
```

基于赋值的版本首先调用了default构造函数设定初值，然后在对他们进行赋值

而初始化列表直接用实参对成员进行初始化，本例中即调用了copy构造

由于编译器会为用户自定义类型的成员变量自动调用default构造函数——如果那些成员变量没有在初始化列表中指定初值的话，这可能引起一些夸张的写法

请立下一个规则，规定总是在初始化列表中列出所有的成员变量

有些情况下，即使面对的成员变量属于内置类型，也一定要用初始化列表，比如成员变量是const或者references，他们就一定需要初始化

C++有着十分固定的成员初始化次序，base classes更早与其derived classes被初始化，而class的成员变量总是以其声明的次序被初始化

为了避免一些晦涩的错误，请你在初始化列表罗列各个成员时，最好总是按照其声明次序来罗列

最后就只剩一个内容了，但是这个也是最重要的一个

不同编译单元内定义之non-local static对象的初始化次序

static对象，其寿命是从被构造出来到程序结束为止，函数内的static对象被称为local static对象（因为他们对函数而言是local），其他的static对象，包括global对象，定义于namespace作用域内的对象，在classes内，在函数内，以及在file作用域内的static对象就是non-local static对象

所谓编译单元，就是指产出单一目标文件的那些源码，基本上他是单一源码文件加上其所含入的头文件（single object file and #include files）

现在，我们所关心的问题涉及到至少两个源码文件，每一个内含至少一个non-local static对象，如果编译单元内的某个non-local static对象的初始化动作使用了另一个编译单元的某个non-local static对象，它所用到的这个对象可能尚未被初始化，因为C++对定义于不同编译单元内的non-local static对象的初始化次序并无明确定义

这是有原因的：决定它们的初始化次序相当困难，非常困难，根本无解。在其最常见形式，也就是多个编译单元内的non-local static对象经由模板隐式具现化implicit template instantiations形成（而后者自己可能也是经由模板隐式具现化形成），不但不可能决定正确的初始化次序，甚至往往不值得寻找可决定正确次序的特殊情况

幸运的是一个小小的设计就可以完全消除这种问题，唯一需要做的就是，将内个non-local static对象搬到自己的专属函数内，这些函数返回一个reference指向它所含的对象，然后用户调用这些函数，而不直接指涉这些对象，换句话说，non-local static对象被local static对象替换了。这是Design Patterns中，Singleton模式的一个常见的实现手法

因为C++保证，函数内的local static对象会在函数被调用期间，首次遇上该对象的定义式时被初始化。

更棒的是，如果你从未调用non-local static对象的“仿真函数”，就绝不会引发构造和析构成本

还有一点，任何一种non-const static对象，不论他是local还是non-local，在多线程环境下等待某事发生都会有麻烦，处理这个麻烦的一种做法是，在程序的单线程启动阶段(single threaded startup portion)手工调用所有reference-returning函数，这可以消除与初始化有关的“竞速形势”(race conditions)

当然了，上述方法能够给成功前提是你要有一个合理的初始化次序。

- 为内置型对象进行手工初始化，因为c++不保证初始化他们
- 构造函数最好使用初始化列表，而不要在构造函数本体内使用赋值操作。初始化列表列出的成员变量，其排列次序应该和他们在class中的声明次序相同。
- 为免除“跨编译单元的初始化次序”问题，请以local static对象替换non-local static对象

条框05：了解C++默默编写并调用那些函数

一个empty class，在C++处理之后，编译器就会为他声明一个copy构造函数，一个copy assignment操作符和一个析构函数，此外如果你没有声明构造函数，编译器也会帮你声明一个default构造函数，所有的函数都是public且inline的

```
//如果你写下这样的代码
class Empty { };
//就好像你写下了
class Empty {
public:
    Empty() {}
    Empty(const Empty& rhs) {}
    ~Empty() {}
    Empty& operator=(const Empty& rhs) {}
};
```

如果其中声明了一个构造函数，那么编译器就不会再为他创建default构造函数

对于编译器生成的copy构造函数，他只是单纯的将对象的每一个non static成员变量拷贝到目标对象

对于copy assignment函数来说，其行为基本上与copy构造函数如出一辙，但是会有特殊情况出现

考虑如下代码

```
template<class T>
class NamedObject {
public:
    NamedObject(std::string &name, const T &value);
private:
    std::string &nameValue;
    const T objectValue;
};
std::string newDog("A");
std::string oldDog("B");
NamedObject<int> p(newDog, 2);
NamedObject<int> s(oldDog, 3);
p = s;
```

赋值之前，p和s的nameValue都指向着string对象，但是由于reference是不可被改动的，所以编译器会拒绝这一行的赋值动作。

注意，copy assignment会被拒绝，但是copy构造不会，因为他是在初始化的时候将引用作为初值，并不是在改变引用自身，即const和reference可以被初始化但不能被赋值

如果你打算在一个内含reference成员，或者内含const成员的class内支持赋值操作，那么你必须自己定义copy assignment操作符。

同时，如果一个base class的copy assignment操作符声明为了private，那么编译器同样会拒绝对其derived class生成copy assignment操作符，因为他们无法调用derived class无法调用的函数

- 编译器可以暗自为class创建default构造函数，copy构造函数，copy assignment操作符以及析构函数

条框06：若不想使用编译器自动生成的函数，就该明确拒绝

条款05有提到，对于copy构造函数和copy assignment操作符来说，即使你不声明他们，编译器也会在他们被使用的时候自动帮你声明，那么就出现了一个问题，倘若我们不希望我们的class拥有拷贝或者赋值的功能怎么办。

答案的关键是，编译器产出的函数都是public的，你可以将copy构造函数和copy assignment操作符声明为private，这样你阻止了编译器自动创建其专属版本，也阻止了人们调用它

但是这并不是绝对安全的，因为member函数和friend函数还是可以调用你的private函数，除非你足够聪明，不去定义它们，这样如果某些人不慎调用任何一个的时候，会获得一个连接错误。

这一个技巧被广泛应用，看看手中的ios_base,basic_ios等库，你会发现他们的copy构造函数和copy assignment操作符都是被声明为private且没有定义

当你需要使用相关的类的时候，创建一个专门为了组织copying动作而设计的base class中

```
class Uncopyable {
protected:
    Uncopyable() {}
    ~Uncopyable() {}
private:
    Uncopyable() {const Uncopyable&};
    Uncopyable& operaotr=(const Uncopyable);
};
```

然后让你的class去集成这个class即可

虽然这会涉及到很多别的小问题，但是通常情况下你都可以忽略这些点

- 为驳回编译器自动（暗自）提供的机能，可将相应的成员函数声明为private并且不予实现。使用像Uncopyable这样的base class也是一种做法

条款07：为多态基类声明virtual析构函数

考虑如下代码

```
class Timekeeper {
public:
    TimeKeeper();
    ~TimeKeeper();
}
class AtomicClock: public TimeKeeper {};
class WaterClock: public TimeKeeper {};
classWristWatch: public TimeKeeper {};
```

我们可以通过设计factory（工厂）函数，返回指针指向一个计时对象，factory函数会返回一个base class指针，指向新生成的derived class对象

`TimeKeeper* getTimeKeeper()`

注意，被TimeKeeper返回的对象必须位于heap。因此为了避免泄露内存和其他资源，将factory函数返回的每一个对象适当的delete掉很重要

getTimeKeeper返回的指针指向一个derived class对象，但是却由一个base class指针被删除，同时，目前的base class有一个non-virtual 析构函数

C++明确指出，当derived clas对象由一个base class指针被删除，而该base class带有一个non-virtual 析构函数，其结果未有定义

实际执行时通常发生的是对象的derived成分没被销毁，因为derived class的析构函数没执行起来，就导致了一个局部销毁的对象

解决问题的方法很简单，给base class一个virtual 析构函数，他就会销毁整个对象。

这种base class通常还有其他的virtual函数，任何class 只要带有virtual函数都几乎确定应该也有一个virtual析构函数

当声明了virtual函数以后，对象将会带有vptr（virtual table pointer）指针，该指针指向一个由函数指针构成的数组，成为vtbl（virtual table），虚函数表记录了该对象在调用函数时应该调用的函数指针。

倘若在不需要虚析构函数的情况下声明虚函数，就会导致一定的浪费以及一些其他的问题。

所以无端的将所有的函数都声明为virtual是错误的。

部分的class没有virtual析构函数，也就不希望你取继承他们

有时候让class带一个pure virtual函数较为便利，倘若你想拥有一个抽象的class，但是没有任何pure virtual函数怎么办？ 你可以定义一个pure virtual析构函数

但是要记得为这个析构函数提供一份定义

- polymorphic（带多态性质的）base classes应该声明一个virtual析构函数，如果class带有任何virtual函数，他就应该拥有一个virtual析构函数
- classes的设计目的如果不是作为base classes使用，或不是为了具备多态性（polymorphically），就不应该声明virtual析构函数

条款08：别让异常逃离析构函数

C++并不禁止析构函数吐出异常，他只是不建议你这样做

假设在析构函数中被抛出了异常，考虑你当前有若干个这样的对象，那么就有可能出现若干个同时作用的异常，在两个异常同时存在的情况下，程序若不是结束执行就是导致不明确行为

假设你当前用一个class负责数据库连接

```
class DBConnection {
public:
    static DBConnection create();   //返回DBConnection对象
    void close();       //关闭联机，失败则抛出异常
};
```

为了确保客户不忘记在DBConnection对象身上调用close()，一个合理的想法是创建一个用来管理DBConnection资源的class，并在其析构函数中调用close

```
class DBConn {
public:
    ~DBConn() {     //确保数据库连接总是会被关闭
        db.close();
    }
private:
    DBConnection db;
};
```

那么用户就可以写出这样的代码 `DBConn dbc(DBConnection::create());`

只要close调用成功，一切都是美好的，但是如果该调用出现异常，DBConn析构函数就会传播该异常

有两个方法可以避免这个问题

```
//如果close抛出异常就结束程序，通常可以用abort完成
DBConn::~DBConn() {
    try {db.close()}
    catch (...) {
        //制作运转记录，记下对close调用失败
        std::abort();
    }
}
```

tip：abort函数和exit函数的区别，exit会释放所有的静态的全局对象，缓存，关闭所有的I/O通道，最后关闭程序，但是对象还是不会被正确析构的。而对于abort来说，就是直接terminate程序，没有任何的清理工作。

那么上面的做法就是调用abort，在异常传播出去之前结束程序

```
//吞下因调用close而发生的异常
DBConn::~DBConn() {
    try{db.close();}
    catch (...) {
        //制作运转记录，记下对close调用失败
    }
}
```

一般来说，吞掉异常是个坏主意，因为他压制了某些动作失败的重要信息，但是这有时也比草率结束程序或不明确行为带来的风险好

一个比较好的方法是重新设计DBConn接口，让用户有一个机会可以处理因该操作而产生的异常

```
class DBConn{
public:
    void close() {
        db.close();
        closed = true;
    }
    ~DBConn() {
        if (!closed) {
            try {
                db.close();
            } catch (...) {
                //制作运转记录，记下对close调用失败
            }
        }
    }
private:
    DBConnection db;
    bool closed;
}
```

这样就把调用close的责任从DBConn析构函数的手上转到了用户的手上

如果他们自己没有调用close或者没有去处理调用失败的异常，出现了错误的时候，用户没有立场抱怨，因为他们曾经有机会第一手处理问题，而他们选择了放弃

- 析构函数绝对不要吐出异常。如果一个被析构函数调用的函数可能抛出异常，析构函数应该捕捉任何异常，然后吞下他们（不传播）或结束程序
- 如果客户需要对某个操作函数运行期间抛出的异常做出反应，那么class应该提供一个普通函数（而非在析构函数中）执行该操作

条款09：绝不在构造和析构过程中调用virtual函数

首先考虑如下代码

```
class Transaction {
public:
    Transaction();
    virtual void logTransaction() const = 0;
};
Transaction::Transaction() {
    logTransaction();
}
class BuyTransaction: public Transaction {
public:
    virtual void logTransaction() const;
};
```

当我们尝试构建derived class的对象时，首先会调用base class的构造函数，在构造函数中，他调用了virtual函数，但是他调用的函数是base class内的版本，即使当前正在构建的是derived class的对象

base class构造期间virtual函数绝不会下降到derived class阶层，因为如果是那样的话，derived class的virtual函数几乎必会使用local成员变量，但是那些是都没有被初始化的成员变量，这将会是一张通往不明确行为和彻夜调试大会的直达车票。

derived class在base class构造期间，对象的类型是derived class，不只是virtual函数会被解析为base class，dynamic_cast和typeid都会将对象视为base class类型，调用构造函数时，BuyTransaction的专属成分尚未被初始化，所以最安全的做法就是视他们不存在，derived class构造函数开始前他都不会是一个derived class对象。

相同的道理也适用于析构函数

有的时候编译器会为此发出一个警告信息，但有的时候不会

所以唯一避免此问题的做法是，确定你的构造函数和析构函数都没有（在对象被创建和被销毁期间）调用virtual函数，而他们调用的所有函数也都服从同一约束

还有其他的方案可以解决此问题，将class Transation内将logTranslation函数改为non-virtual，然后要求derived class构造函数内传递必要信息给Transaction构造函数

```
class Transaction {
public:
    explicit Transaction(const std::string &logInfo);
    void logTransaction(const std::string &logInfo) const;
};
Transaction::Transaction(const std::string &logInfo) {
    logTransaction(logInfo);
}
class BuyTransaction {
public:
    BuyTransaction(parameters) : Transaction(createLogString( parameters )) {...}
private:
    static std::string createLogString( parameters );
};
```

换句话说你无法使用virtual函数从base classes向下调用，在构造期间，你可以用“令derived classes将必要的构造信息向上传递至base class构造函数”替换之

explicit防止函数调用时的参数进行隐式的类型转换，因为在默认规定中，只有一个参数的构造函数也定义了一个隐式转换

private static函数，比起初始化列表，利用辅助函数创建一个值给base class构造函数往往比较方便（也比较可读）（？？？ 可读我理解，为什么方便呢）

令此函数为static，也就不可能意外指向对象中尚未初始化的成员变量，这很重要，我们就是为了防止使用未定义的成员变量才这样做的

- 在构造和析构期间不要调用virtual函数，因为这类调用从不下降至derived class（比起当前执行构造函数和析构函数的那层）

条款10：令 operator= 返回一个reference to *this

关于赋值，我们通常可以把他写成连锁形式

`x = y = z = 10`

赋值采用右结合律，即`x = (y = (z = 10))`

为了实现连锁赋值，赋值操作符必须返回一个reference指向操作符的左侧实参

```
class Widget {
public:
    Widget& operator=(const Widget &rhs) {
        return *this;
    }
};
```

这个协议适用于所有赋值相关的运算

有的时候或许你不用这个方法结果也是正确的，是因为利用了c++赋值采用右结合律的特性，当表达式加上括号的时候你就不总是那么幸运了。

- 令赋值（assignment）操作符返回一个 reference to *this

条款11：在 operator= 中处理“自我赋值”

对象的自我赋值就是用自己给自己赋值

一些隐含的情况`a[i] = a[j]`或`*px = *py`

当i等于j时，当px和py指向的是相同的东西时，就会出现这种状况

你可能会想，自我赋值怎么了？应该不会出现什么问题的才对，毕竟自我赋值没有改变什么

考虑如下代码

```
class Widget {
private:
    Bitmap *pb; //指向一个从heap分配得到的对象
}
Widget& Widget::operator=(const Widget& rhs) {
    delete pb;
    pb = new Bitmap(*rhs.pb);
    return *this;
}
```

考虑如果你在这个对象中自我赋值会出现什么情况

你会先销毁对象，这同时也销毁了rhs的bitmap，那么结果就是你就用指针指向了一个已经被销毁的对象

想要阻止这种错误，一个很传统的方法就是在前面加上 `if (this == &rhs) return *this`

虽然这个方法可以让你具有自我赋值安全性，但是如果new Bitmap导致异常，Widget最终仍然会持有一块被删除的Bitmap

但是好处在于，具备异常安全性的代码也具有自我赋值安全性，所以你只要注意“许多时候一群精心安排的语句就可以导出异常安全以及自我赋值安全的代码”

```
Widget& Widget::operator= (cont Widget& rhs) {
    Bitmap* pOrig = pb;
    pb = new Bitmap(*rhs.pb);
    delete pOrig;
    return *this;
}
```

或许这不是处理自我赋值的最高效办法，但是他行得通，因为就算他遇到了自我赋值的情况，他只是做了一个复件，删除了原件并指向了新的复件

有一个替代方案是使用所谓的copy and swap技术

```
Widget& Widget::operator= (const Widget &rhs) {
    Widget temp(rhs);
    swap(temp);
    return *this;
}
```

上述手法为rhs创造了一个副本，并与*this交换，函数结束时，会自动销毁掉这个创造的副本

还有一种手段，利用了以下事实(1)：某class的copy assignment操作符可能被声明为“以by value方式接受实参”；(2)：以by value的方式传递东西会造成一份副本：

```
Widget& Widget::operator=(Widget rhs) {
    swap(rhs);
    return *this;
}
```

上述手法通过pass by value的手段创造了一份副本，并将其和*this互换

巧妙的修补牺牲了清晰性，但是copying动作从函数本体移至函数参数构造阶段却可令编译器有时生成更高效的代码

- 确保当对象自我赋值时 operator= 有良好行为。其中技术包括比较“来源对象”和“目标对象”的地址，精心周到的语句顺序，以及copy-and-swap
- 确定任何函数如果操作一个以上的的对象，而其中多个对象是同一个对象时，其行为仍然正确

条款12：复制对象时勿忘其每一个成分

考虑这种情况，你在类的copying函数中写好了每个成员变量的赋值，但是你突然需要增加一个成员变量，这时如果你只是单纯的改变相关的逻辑，不改变copying函数的话，程序还是会正常编译的，但是由于你没有写新变量的copy，就会导致运行中的错误

注意copying函数表示的是copy构造函数和copy assignment操作符

上述的问题编译器是不会给你任何提示的，所以结论很明显，当你为class增加一个成员变量，你必须同时修改copying函数

再考虑，如果发生继承，就会出现一个潜藏的危机

```
class PriorityCustomer:public Customer {
public:
    PriorityCustomer(const PriorityCustomer& rhs);
    PriorityCustomer& operator=(const PriorityCustomer& rhs);
private:
    int priority;
};
PriorityCustomer::PriorityCustomer(const PriorityCustomer &rhs) : priority(rhs.priority) {}
PriorityCustomer& PriorityCustomer::operator(const PriorityCustomer &rhs) {
    priority = rhs.priority;
    return *this;
}
```

看起来copying函数好像赋值了PriorityCustomer里的每一个东西，但是他其实只是复制了derived class的变量，并没有复制他所继承的变量，因此PriorityCustomer对象的Customer部分会被不带实参的Customer构造函数进行构造

所以任何时候，只要你承担起为derived class撰写copying函数的重任，你必须很小心的也复制其base class成分，你应该让derived class的copying函数调用相应的base class函数

```
PriorityCustomer::PriorityCustomer(const PriorityCustomer &rhs) : Customer(rhs),
priority(rhs.priority) {}
PriorityCustomer& PriorityCustomer::operator(const PriorityCustomer &rhs) {
    Customer::operator=(rhs);
    priority = rhs.priority;
    return *this;
}
```

所以请确保你复制了所有local的成员变量，调用了所有base classes内的适当的copying函数

请注意，有时这两个函数往往有着相似的实现本体，诱惑着你想要用一个调用另一个来实现代码精简

但是这是不合理的，在赋值中进行构造，或者对一个未构造成功的对象进行赋值都是无意义甚至可能出现问题的

如果你发现你的copying函数有着相似的代码，消除重复的做法是建立一个新的成员函数给两者调用，这样的函数往往是private而且常被命名为init

- copying函数应该确保复制“对象内的所有成员变量”及“所有base class”成分
- 不要尝试以某个copying函数实现另一个copying函数。应该将共同机能放进第三个函数中，并由两个copying函数共同调用