# Effective C++

## 条款1：尽量用const和inline而不用#define

const(内容) char \* const(指针) authorName = "Scott Meyers";

class GamePlayer {--

 private: static const int NUM\_TURNS = 5; // 声明

 int scores[NUM\_TURNS]; // use of constant ...

};

const int GamePlayer::NUM\_TURNS;定义

常量必须如此定义和声明,static const.

或者

class GamePlayer {

private: enum { NUM\_TURNS = 5 }

int scores[NUM\_TURNS];

};

replace #define max(a,b) ((a) > (b) ? (a) : (b))

with

inline int max(int a, int b) { return a > b ? a : b; }

or

template<class T> inline const T& max(const T& a, const T& b) { return a > b ? a : b; }

不同于C#, class T 只是类型标识，也可以使用int作为T.

C#也可以使用int或class，但不能通用，可以通过where来限制类型

## 条款2：尽量用<iostream>而不用<stdio.h>

scanf和printf不是类型安全的，而且没有扩展性。另外，scanf/printf系列函数把要读写的变量和控制读写格式的信息分开来.

scanf/printf的这些弱点正是操作符>>和<<的强项:

int i; Rational r;// r 是个有理数 ...

cin >> i >> r;

cout << i << r;

上面的代码要通过编译，>>和<<必须是可以处理Rational类型对象的重载函数(可能要通过隐式类型转换)。

class Rational {

 public: Rational(int numerator = 0, int denominator = 1); ...

private: int n, d;// 分子，分母

**friend ostream& operator<<(ostream& s, const Rational& );** };

ostream& operator<<(ostream& s, const Rational& r)

{ s<< r.n << '/' << r.d; return s; } 1

operator<<不是成员函数, 而且传递给operator<<的不是Rational对象，而是定义为const的对象的引用

如果使用了#include <iostream>, 得到的是置于名字空间std下的iostream库的元素；如果使用#include <iostream.h>，得到的是置于全局空间的同样的元素。

## 条款3：尽量用new和delete而不用malloc和free

malloc和free(及其变体)会产生问题的原因在于它们太简单：他们不知道构造函数和析构函数。

new/delete和malloc/free的不兼容性常常会导致一些严重的复杂性问题。特别是从C函数中返回的指针忘了free.

## 条款4：尽量使用c++风格的注释

## 条款5：对应的new和delete要采用相同的形式

注意delete p 和 delete []p 的区别.

# 条款6：析构函数里对指针成员调用delete

删除空指针是安全的(因为它什么也没做)。

考虑**智能指针**

# 条款7：预先准备好内存不够的情况

有人还会自定义(重载)operator new，所以程序里会包含任意个使用new的语法形式

指定出错处理函数时要用到set\_new\_handler函数，它在头文件<new>里大致是象下面这样定义的：

**typedef void (\*new\_handler)();**

new\_handler set\_new\_handler(new\_handler p) throw();

operator new不能满足内存分配请求时，new-handler函数不只调用一次，而是不断重复，直至找到足够的内存

void \* x::operator new(size\_t size) {  **//注意operator new 的size参数是自动赋予的**

new\_handler globalhandler = // 安装x的new\_handler

    std::set\_new\_handler(currenthandler);

void \*memory;

try {

 // 尝试分配内存

memory = ::operator new(size);

}

catch (std::bad\_alloc&) {

// 恢复旧的new\_handler

std::set\_new\_handler(globalhandler);

throw; // 抛出异常

}

std::set\_new\_handler(globalhandler); // 恢复旧的new\_handler return memory;

}

可以使用模板类来支持new handler.

template<class t> // 提供类set\_new\_handler支持的

class newhandlersupport { // 混合风格”的基类

public:

static new\_handler set\_new\_handler(new\_handler p);

static void \* operator new(size\_t size);

private:

static new\_handler currenthandler;

};

template<class t>

new\_handler newhandlersupport<t>::set\_new\_handler(new\_handler p)

{

......

}

# 条款8: 写operator new和operator delete时要遵循常规

自己重写operator new时，函数提供的行为要和系统缺省的operator new一致。处理零字节请求的技巧在于把它作为请求一个字节来处理。operator new内部包含一个无限循环

注意继承：

void \* base::operator new(size\_t size)

{

  if (size != sizeof(base))             // 如果数量“错误”，让标准operator new

    return ::operator new(size);        // 去处理这个请求

                                        //

  ...                                   // 否则处理这个请求

}

c++保证删除空指针永远是安全的

void base::operator delete(void \*rawmemory, size\_t size)

{

  if (rawmemory == 0) return;      // 检查空指针

  if (size != sizeof(base)) {      // 如果size"错误"，

    ::operator delete(rawmemory);  // 让标准operator来处理请求

    return;

  }

# 条款9: 避免隐藏标准形式的new

在类里定义了一个称为“operator new”的函数后，会不经意地阻止了对标准new的访

问。

一个办法是在类里写一个支持标准new调用方式的operator new，它和标准new做同样

的事。这可以用一个高效的内联函数来封装实现。

class x {

public:

  void f();

  static void \* operator new(size\_t size, new\_handler p);

  static void \* operator new(size\_t size)

  { return ::operator new(size); }

};

x \*px1 =

  new (specialerrorhandler) x;      // 调用 x::operator

                                    // new(size\_t, new\_handler)

x\* px2 = new x;                     // 调用 x::operator

                                    // new(size\_t)

另一种方法是为每一个增加到operator new的参数提供缺省值(见条款24)：

class x {

public:

  void f();

  static

    void \* operator new(size\_t size,                // p缺省值为0

                        new\_handler p = 0);         //

};

# 条款10: 如果写了operator new就要同时写operator delete

缺省的operator new和operator delete具有非常好的通用性，它的这种灵活性也使得在某些特定的场合下，可以进一步改善它的性能，比如需要动态分配大量的但很小的对象。

在于operator new和operator delete之间需要互相传递信息，消耗额外内存。

你不会得到一块看起来象这样的内存块：

    pa——> airplane对象的内存

而是得到象这样的内存块：

    pa——> 内存块大小数据 + airplane对象的内存

内存池的最小接口：

class pool {

public:

  pool(size\_t n);                      // 为大小为n的对象创建

                                       // 一个分配器

  void \* alloc(size\_t n)  ;            // 为一个对象分配足够内存

                                       // 遵循条款8的operator new常规

  void free(  void \*p, size\_t n);      // 将p所指的内存返回到内存池；

                                       // 遵循条款8的operator delete常规

  ~pool();                             // 释放内存池中全部内存

};

条款11: 为需要动态分配内存的类声明一个拷贝构造函数和一个赋值操作符

因为没有自定义的operator=可以调用，c++会生成并调用一个缺省的operator=操作符。这个缺省的赋值操作符会执行从a的成员到b的成员的逐个成员的赋值操作，对指针(a.data和b.data) 来说就是逐位拷贝。

如果没有重载：

string a("hello");

string b("world");

a = b;

第一，b曾指向的内存永远不会被删除，因而会永远丢失。这是产生内存泄漏的典型例子。第二，现在a和b包含的指针指向同一个字符串，那么只要其中一个离开了它的生存空间，其析构函数就会删除掉另一个指针还指向的那块内存。

只要类里有指针时，就要写自己版本的拷贝构造函数和赋值操作符函数。在这些函数里，你可以拷贝那些被指向的数据结构，从而使每个对象都有自己的拷贝；或者你可以采用某种引用计数机制（见条款 m29）去跟踪当前有多少个对象指向某个数据结构。

# 条款12: 尽量使用初始化而不要在构造函数里赋值

在写namedptr构造函数时，必须将参数值传给相应的数据成员。有两种方法来实现。第一种方法是使用成员初始化列表：

template<class t>

namedptr<t>::namedptr(const string& initname, t \*initptr  )

: name(initname), ptr(initptr)

{}

第二种方法是在构造函数体内赋值：

template<class t>

namedptr<t>::namedptr(const string& initname, t \*initptr)

{

  name = initname;

  ptr = initptr;

}

const和引用数据成员只能用初始化，不能被赋值

对象的创建分两步：

1. 数据成员初始化

2. 执行被调用构造函数体内的动作。

有一种情况下，对类的数据成员用赋值比用初始化更合理。这就是当有大量的固定类型的数据成员要在每个构造函数里以相同的方式初始化的时候。

请注意static类成员永远也不会在类的构造函数初始化。静态成员在程序运行的过程中只被初始化一次，

# 条款13: 初始化列表中成员列出的顺序和它们在类中声明的顺序相同

类成员是按照它们在类里被声明的顺序进行初始化的，和它们在成员初始化列表中列出的顺序没一点关系，下面的初始化就可能有问题

array<t>::array(int lowbound, int highbound)

: size(highbound - lowbound + 1),

  lbound(lowbound), hbound(highbound),

  data(size)

{}

对一个对象的所有成员来说，它们的析构函数被调用的顺序总是和它们在构造函数里被创建的顺序相反

# 条款14: 确定基类有虚析构函数

当通过基类的指针去删除派生类的对象，而基类又没有虚析构函数时，结果将是不可确定的.实际运行时经常发生的是，派生类的析构函数永远不会被调用\

当一个类不准备作为基类使用时，使析构函数为虚一般是个坏主意。实现虚函数需要对象附带一些额外信息，以使对象在运行时可以确定该调用哪个虚函数。对大多数编译器来说，这个额外信息的具体形式是一个称为vptr（虚函数表指针）的指针。vptr指向的是一个称为vtbl（虚函数表）的函数指针数组

注意当类里没有虚函数的时候，也会带来非虚析构函数问题。比如子类新增了指针，这个子类被cast成基类并且被当作基类delete时，子类的新增内存会泄露.

有些时候，你想使一个类成为抽象类，但刚好又没有任何纯虚函数，可将析构设为virtual=0

如果声明virtual析构函数为inline，将会避免调用它们时产生的开销，但编译器还是必然会在什么地方产生一个此函数的拷贝

# 条款15: 让operator=返回\*this的引用

赋值操作可以象下面这样链起来：

int w, x, y, z;

w = x = y = z = 0;

c& c::operator=(const c&);

一般情况下几乎总要遵循operator=输入和返回的都是类对象的引用的原则，然而有时候需要重载operator=使它能够接受不同类型的参数

不能返回右边的值，因为operator返回值不是const，而右边是const，为什么右边是const，因为编译器产生的临时值可能是const.

x = "hello"; // 和x.op = ("hello"); 相同

因为赋值语句的右边参数不是正确的类型——它是一个字符数组，不是一个string——编译器就要产生一个临时的string对象（通过stirng构造函数——参见条款m19）使得函数继续运行。就是说，编译器必须产生大致象下面这样的代码：

const string temp("hello"); // 产生临时string x = temp; // 临时string传给operator=

编译器一般会产生这样的临时值（除非显式地定义了所需要的构造函数——见条款19），但注意临时值是一个const。这很重要，因为它可以防止传递到函数内的临时值被修改

# 条款16: 在operator=中对所有数据成员赋值

写赋值运算符时，必须对对象的每一个数据成员赋值.

  \*ptr = \*rhs.ptr;        // 对于ptr，赋的值是指针所指的值，

                               // 不是指针本身

派生类的赋值运算符也必须处理它的基类成员的赋值

// 正确的赋值运算符

derived& derived::operator=(const derived& rhs)

{

  if (this == &rhs) return \*this;

  base::operator=(rhs);    // 调用this->base::operator=

  y = rhs.y;

  return \*this;

}

derived的拷贝构造函数必须保证调用的是base的拷贝构造函数而不是base的缺省构造函数

class derived: public base {

public:

  derived(const derived& rhs): base(rhs), y(rhs.y) {}

  ...

};

# 条款17: 在operator=中检查给自己赋值的情况

一是效率，而是正确性（比如赋值前需要释放内存）。任何时候写一个函数，只要别名有可能出现，就必须在写代码

时进行处理。

# 条款18: 争取使类的接口完整并且最小

不要忘记友元函数在所有实际应用中都是类的接口的一部分。这意味着友元函数影响着类的接口的完整性和最小性。

# 条款19: 分清成员函数，非成员函数和友元函数

成员函数和非成员函数最大的区别在于成员函数可以是虚拟的而非成员函数不行。如果函数不必是虚拟的，情况就稍微复杂一点。

注意隐士转换， 比如 Rational \* 2

如果Rational 有 从2的构造，编译器会饮食生成一个const rational. 而 2\* rational 不行.

在这种情况下，你可能还是想支持混合型的算术操作，而实现的方法现在应该清楚了：使operator\*成为一个非成员函数，从而允许编译器对所有的参数执行隐式类型转换：

class rational {

  ...                               // contains no operator\*

};

// 在全局或某一名字空间声明，

// 参见条款m20了解为什么要这么做

const rational operator\*(const rational& lhs,

                         const rational& rhs)

{

  return rational(lhs.numerator() \* rhs.numerator(),

                  lhs.denominator() \* rhs.denominator());

}

有需要的话，使之成为Rational的有缘函数。

C++中的const == C#中的static????

只有所涉及的构造函数没有声明为explicit的情况下才会这样，因为explicit构造函数不能用于隐式转换。

如果想重载operator>>和operator<<来读写string对象，你会很快发现它们不能是成员函数，否则

s >> cin;                   // 合法, 但

                            // 有违常规

s << cout;                 // 同上

正确做法：注意第一个参数在左边。

istream& operator>>(istream& input, string& string)

{

  delete [] string.data;

  read from input into some memory, and make string.data

  point to it

  return input;

}

ostream& operator<<(ostream& output,

                    const string& string)

{

  return output << string.data;

}

·虚函数必须是成员函数。如果f必须是虚函数，就让它成为c的成员函数。

·operator>>和operator<<决不能是成员函数。如果f是operator>>或operator<<，让f成为非成员函数。如果f还需要访问c的非公有成员，让f成为c的友元函数。

·只有非成员函数对最左边的参数进行类型转换。

# 条款20: 避免public接口出现数据成员

 在public接口里放上数据成员无异于自找麻烦，所以要把数据成员安全地隐藏在与功能分离的高墙后。需任何代价地换来一致性和精确的访问控制。

# 条款21: 尽可能使用const

 在类的外面，它可以用于全局或名字空间常量（见条款1和47），以及静态对象（某一文件或程序块范围内的局部对象）。在类的内部，它可以用于静态和非静态成员（见条款12）。

const char \*p        = "hello";          // 非const指针,

                                         // const数据

char \* const p       = "hello";          // const指针,

                                         // 非const数据

const char \* const p = "hello";          // const指针,

                                         // const数据

如果const出现在\*的左边，指针指向的数据为常量；如果const出现在\*的右边，指针本身为常量

让函数返回一个常量值经常可以在不降低安全性和效率的情况下减少用户出错的几率

const rational operator\*(const rational& lhs,

                         const rational& rhs);

很多程序员第一眼看到它会纳闷：为什么operator\*的返回结果是一个const对象？因为如果不是这样，用户就可以做下面这样的坏事：

rational a, b, c;

...

(a \* b) = c;      // 对a\*b的结果赋值

仅在const方面有不同的成员函数可以重载

当且仅当成员函数不修改对象的任何数据成员（静态数据成员除外）时，即不修改对象中任何一个比特(bit)时，这个成员函数才是const的. 不幸的是，很多不遵守bitwise constness定义的成员函数也可以通过bitwise测试。特别是，一个“修改了指针所指向的数据”的成员函数，其行为显然违反了bitwise constness定义，但如果对象中仅包含这个指针，这个函数也是bitwise const的

当对非静态数据成员运用mutable时，这些成员的“bitwise constness”限制就被解除：它们可以在任何地方被修改.

  mutable size\_t datalength;      // 这些数据成员现在

                                  // 为mutable；他们可以在

类c的一个成员函数中，this指针就好象经过如下的声明：

c \* const this;              // 非const成员函数中

const c \* const this;        // const成员函数中

  // 定义一个不指向const对象的

  // 局部版本的this指针

  string \* const localthis =

  const\_cast<string \* const>(this

# 条款22: 尽量用“传引用”而不用“传值”

通过值来传递一个对象”的具体含义是由这个对象的类的拷贝构造函数定义的。这使得传值成为一种非常昂贵的操作

为避免这种潜在的昂贵的开销，就不要通过值来传递对象

通过引用来传递参数还有另外一个优点：它避免了所谓的“切割问题（slicing problem）”。当一个派生类的对象作为基类对象被传递时，它（派生类对象）的作为派生类所具有的行为特性会被“切割”掉，从而变成了一个简单的基类对象。这往往不是你所想要的

# 条款23: 必须返回一个对象时不要试图返回一个引用

防止传递一个并不存在的对象的引用. 引用只是一个名字，一个其它某个已经存在的对象的名字。

写一个必须返回一个新对象的函数的正确方法就是让这个函数返回一个新对象。

# 条款24: 在函数重载和设定参数缺省值间慎重选择

第一，确实有那么一个值可以作为缺省吗？第二，要用到多少种算法？一般来说，如果可以选择一个合适的缺省值并且只是用到一种算法，就使用缺省参数（参见条款38）。否则，就使用函数重载。

对很多函数来说，会找不到合适的缺省值

# 条款25: 避免对指针和数字类型重载

void f(int x);

void f(string \*ps);

f(0); // 调用f(int)还是f(string\*)?

f(0); // 还是调用f(int)

f(static\_cast(null)); // 调用f(string\*)

f(static\_cast(0)); // 调用f(string\*)

Using anonymous null class instead

const // 这是一个const对象...

class {

public:

  template // 可以转换任何类型

    operator t\*() const // 的null非成员指针

    { return 0; } //

  template // 可以转换任何类型

    operator t c::\*() const // 的null成员指针

    { return 0; }

private:

  void operator&() const; // 不能取其地址

// (见条款27)

} null; // 名字为null

# 条款26: 当心潜在的二义性

class A {

public:

 A(const B&); // 可以从B构造而来的类A

};

class B {

public:

 operator A() const; // 可以从A转换而来的类B

};

what if f(B) yet f needs take a parameter with type of A?

多重继承中的重名函数

d.Base1::doIt(); // 正确, 调用Base1::doIt

d.Base2::doIt(); // 正确, 调用Base2::doIt

# 条款27: 如果不想使用隐式生成的函数就要显式地禁止它

显式地声明一个成员函数，就防止了编译器去自动生成它的版本；使函数为private，就防止了别人去调用它。通过声明而不定义，防止内部调用。

# 条款28: 划分全局名字空间

namespace本质上和使用前缀的方法一样，只不过避免了别人总是看到前缀而已

namespace sdm {

  const double book\_version = 2.0;

  class handle { ... };

  handle& gethandle();

}

void f1()

{

  using namespace sdm;           // 使得sdm中的所有符号不用加

                                 // 修饰符就可以使用

}

void f2()

{

  using sdm::book\_version;        // 使得仅book\_version不用加

                                 // 修饰符就可以使用

}

# 条款29: 避免返回内部数据的句柄

const string b("hello world");      // b是一个const对象，内部的char\* 可能依然会暴露

引用也很容易被滥用

柄的合法性失效的时间和它所对应的对象是完全相同的。这个时间可能比用户期望的要早很多，特别是当涉及的对象是由编译器产生的临时对象时。

# 条款30: 避免这样的成员函数：其返回值是指向成员的非const指针或引用，但成员的访问级比这个函数要低

返回值是某个访问级较低的成员的指针或引用。但同时，你又不想牺牲private和protected为你提供的访问限制。这种情况下，你可以通过返回指向const对象的指针或引用来达到两全其美的效果

# 条款31: 千万不要返回局部对象的引用，也不要返回函数内部用new初始化的指针的引用

所以要记住你的教训：写一个返回废弃指针的函数无异于坐等内存泄漏的来临。

假设inline const rational& operator\*(const rational& lhs,

                                 const rational& rhs)

{

  // create a new object on the heap

  rational \*result =

    new rational(lhs.n \* rhs.n, lhs.d \* rhs.d);

  // return it

  return \*result;

}

调用：

rational one(1), two(2), three(3), four(4);

rational product;

product = one \* two \* three \* four;

会产生3个临时对象，但是内存没有释放！

解决这一难题的唯一方案是叫用户这样写代码：

const rational& temp1 = one \* two;

const rational& temp2 = temp1 \* three;

const rational& temp3 = temp2 \* four;

delete &temp1;

delete &temp2;

delete &temp3;

# 条款32: 尽可能地推迟变量的定义

如果定义了一个有构造函数和析构函数的类型的变量，当程序运行到变量定义之处时，必然面临构造的开销；当变量离开它的生命空间时，又要承担析构的开销。这意味着定义无用的变量必然伴随着不必要的开销，所以只要可能，就要避免这种情况发生。

通过可靠的变量本身来消除对它不必要的注释。

# 条款33: 明智地使用内联

因为避免函数调用的开销仅仅是问题的一个方面。所以当内联一个函数时，编译器可以对函数体执行特定环境下的优化工作。这样的优化对"正常"的函数调用是

可能会增加整个目标代码的体积

大多数编译器拒绝内联"复杂"的函数（例如，包含循环和递归的函数）；还有，即使是最简单的虚函数调用，编译器的内联处理程序对它也爱莫能助。

那些由编译器的实现者写的、在编译其间插入到你的程序中的代码，必然也藏身于某个地方------有时，它们就藏身于你的构造函数和析构函数

# 条款34: 将文件间的编译依赖性降至最低

在将接口从实现分离这方面，C++做得不是很出色。尤其是，C++的类定义中不仅包含接口规范，还有不少实现细节

class Date;           // 提前声明

class Address;        // 提前声明

class Country;        // 提前声明

class Person {

public:

...

}

// 类PersonImpl将包含Person对象的实

// 现细节，此处只是类名的提前声明

class PersonImpl;

class Person {

public:

private:

  PersonImpl \*impl;                 // 指向具体的实现类

};

只要有可能，尽量让头文件不要依赖于别的文件；如果不可能，就借助于类的声明，不要依靠类的定义。其它一切方法都源于这一简单的设计思想。

· 如果可以使用对象的引用和指针，就要避免使用对象本身。

· 尽可能使用类的声明，而不使用类的定义。

· 不要在头文件中再（通过#include指令）包含其它头文件，除非缺少了它们就不能编译。

Person类仅仅用一个指针来指向某个不确定的实现，这样的类常常被称为句炳类(Handle class)或信封类(Envelope class)。（

# 条款35: 使公有继承体现 "是一个" 的含义

C++面向对象编程中一条重要的规则是：公有继承意味着 "是一个" 。

企鹅不会飞" 的指令是由编译器发出的，"让企鹅飞是一种错误" 只能在运行时检测到

# 条款36: 区分接口继承和实现继承

· 声明简单虚函数的目的在于，使派生类继承函数的接口和缺省实现。

· 定义纯虚函数的目的在于，使派生类仅仅只是继承函数的接口。

· 声明非虚函数的目的在于，使派生类继承函数的接口和强制性实现。

理解了纯虚函数、简单虚函数和非虚函数在声明上的区别，就可以精确地指定你想让派生类继承什么：仅仅是接口，还是接口和一个缺省实现？或者，接口和一个强制实现？

# 条款37: 决不要重新定义继承而来的非虚函数

class D: public B {

public:

  void mf();                  // 隐藏了B::mf; 参见条款50

  ...

};

pB->mf();                     // 调用B::mf

pD->mf();                     // 调用D::mf

象B::mf和D::mf这样的非虚函数是静态绑定的

· 适用于B对象的一切也适用于D对象，因为每个D的对象 "是一个" B的对象。

· B的子类必须同时继承mf的接口和实现，因为mf在B中是非虚函数。

# 条款38: 决不要重新定义继承而来的缺省参数值

子类即使修改了父类的缺省值, 当子类被赋值给一个父类指针并被调用时,使用的是父类的缺省值因为缺省值是由指针的静态类型决定的(编译器为了提高效率).

# 条款39: 避免 "向下转换" 继承层次

"向下转换" 可以通过几种方法来消除。最好的方法是将这种转换用虚函数调用来代替，同时，它可能对有些类不适用，所以要使这些类的每个虚函数成为一个空操作。第二个方法是加强类型约束，使得指针的声明类型和你所知道的真的指针类型之间没有出入.

这种方法称为 "安全的向下转换"，它通过C++的dynamic\_cast运算符来实现

# 条款40: 通过分层来体现 "有一个" 或 "用...来实现"

避免 template<class T> class Set: public List<T> 因为 Set不是List. 使用 template <class T> class Set:

{

    private:

     List<T> list;

}

# 条款41: 区分继承和模板

· 当对象的类型不影响类中函数的行为时，就要使用模板来生成这样一组类。

· 当对象的类型影响类中函数的行为时，就要使用继承来得到这样一组类。

比如Stack类适合模板, Cat类适合继承.

# 条款42: 明智地使用私有继承

和公有继承相反，如果两个类之间的继承关系为私有，编译器一般不会将派生类对象（如Student）转换成基类对象（如Person）。

第二个规则是，从私有基类继承而来的成员都成为了派生类的私有成员，即使它们在基类中是保护或公有成员

私有继承意味着 "用...来实现"。私有继承纯粹是一种实现技术.

尽可能地使用分层，必须时才使用私有继承

class GenericStack { //类型不安全, 不可被实例化

protected:

  GenericStack();

  ~GenericStack();

  void push(void \*object);

  void \* pop();

  bool empty() const;

private:

  ...                             // 同上

};

template<class T>

class Stack: private GenericStack { //类型安全, 同时避免了多个GenericStack带来的代码膨胀,尽管本身Stack存在代码膨胀,但比GenericStack小得多.

public:

  void push(T \*objectPtr) { GenericStack::push(objectPtr); }

  T \* pop() { return static\_cast<T\*>(GenericStack::pop()); }

  bool empty() const { return GenericStack::empty(); }

};

# 条款43: 明智地使用多继承

Lottery和GraphicalObject中的draw函数都被声明为虚函数，所以子类可以重新定义它们（见条款36），但如果LotterySimulation想对二者都重新定义那该怎么办？ No Way.

class AuxLottery: public Lottery {

public:

  virtual int lotteryDraw() = 0;

  virtual int draw() { return lotteryDraw(); }

};

class AuxGraphicalObject: public GraphicalObject {

public:

  virtual int graphicalObjectDraw() = 0;

  virtual int draw() { return graphicalObjectDraw(); }

};

class LotterySimulation: public AuxLottery,

                         public AuxGraphicalObject {

public:

  virtual int lotteryDraw();

  virtual int graphicalObjectDraw();

}

class A { ... };

class B : virtual public A { ... };

class C : virtual public A { ... };

class D: public B, public C { ... };

                       A

                       /\

                      /  \

                     /    \

                    B    C

                     \    /

                      \  /

                       \/

                       D

是不是该让A成为虚基类呢？即，从A的继承是否应该是虚拟的呢？现实中，答案几乎总是 ---- 应该；只有极少数情况下会想让类型D的对象包含A的数据成员的多个拷贝。正是认识到这一事实，上面的B和C将A声明为虚基类。

如果A作为 "非虚" 基类，类型D的对象在内存中的分布通常占用连续的内存单元；如果A作为 "虚" 基类，有时，类型D的对象在内存中的分布占用连续的内存单元

A是非虚基类时D对象通常的内存分布：

           A部分+ B部分+ A部分 + C部分 + D部分

A是虚基类时D对象在某些编译器下的内存分布：

                           ------------------------------------------------

                           |                                                   |

                           |                                                  +

          B部分 + 指针 + C部分 + 指针 + D部分 + A部分

                                                      |                       +

                                                      |                        |

                                                      ------------------------

           A   virtual void mf();

             /\

            /  \

           /    \

          B    C   virtual void mf();

           \    /

            \  /

             \/

             D

如果A是非虚基类, 二义性; 如果A是虚基类, c:mf被调用

# 条款44: 说你想说的；理解你所说的

· 共同的基类意味着共同的特性。

· 公有继承意味着 "是一个"。

· 私有继承意味着 "用...来实现"。

· 分层意味着 "有一个" 或 "用...来实现"。优先于私有继承.

下面的对应关系只适用于公有继承的情况：

· 纯虚函数意味着仅仅继承函数的接口。

· 简单虚函数意味着继承函数的接口加上一个缺省实现。

· 非虚函数意味着继承函数的接口加上一个强制实现。

# 条款45: 弄清C++在幕后为你所写、所调用的函-

class Empty{};

和你这么写是一样的：

class Empty {

public:

  Empty();                        // 缺省构造函数

  Empty(const Empty& rhs);        // 拷贝构造函数-

  ~Empty();                       // 析构函数 ---- 是否

                                  // 为虚函数看下文说明

  Empty&

  operator=(const Empty& rhs);    // 赋值运算符--

  Empty\* operator&();             // 取址运算符

  const Empty\* operator&() const;

};

生成的析构函数一般是非虚拟的-，除非它所在的类是从一个声明了虚析构函数的基类继承而来

-

缺省拷贝构造函数（赋值运算符）对类的非静态数据成员进行 "以成员为单位的" 逐一拷贝构造（赋值）。

默认情况下，固定类型的对象拷贝构造（赋值）时是从源对象到目标对象的 "逐位" 拷贝，如果某个属性有拷贝构造函数，则调用拷贝构造函数。

如果想让一个包含引用成员的类支持赋值，你就得自己定义赋值运算符。因为C++没有办法让一个引用指向另一个不同的对象

# 条款46: 宁可编译和链接时出错，也不要运行时出错

Common Sense.

# 条款47: 确保非局部静态对象在使用前被初始化

非局部静态对象指的是这样的对象：

· 定义在全局或名字空间范围内（例如：theFileSystem和tempDir），

· 在一个类中被声明为static，或，

· 在一个文件范围被定义为static。

你绝对无法控制不同被编译单元中非局部静态对象的初始化顺序

虽然关于 "非局部" 静态对象什么时候被初始化，C++几乎没有做过说明；但对于函数中的静态对象（即，"局部" 静态对象）什么时候被初始化，C++却明确指出：它们在函数调用过程中初次碰到对象的定义时被初始化. 因此Singleton是个workaround.

# 条款48: 重视编译器警告

Common Sense.

# 条款49: 熟悉标准库

为了避免这种情况所造成的名字冲突，实际上标准库中的一切都被放在名字空间std中，并且头文件没有.h

库中的一切几乎都是模板, 在标准库中你无法避开模板

标准库都为它们定义了typedef，这样你就可以在编程时继续使用cin，cout，cerr等对象，以及istream，ostream，string等类型，不必担心cin的真实类型是basic\_istream<char>以及string的真实类型是basic\_string<char>。

不要再写你自己的基本容器类！标准库提供了下列高效的实现：vector（就象动态可扩充的数组），list（双链表），queue， stack，deque，map，set和bitset。string也是容器

标准库就提供了大量简易的方法， 算法将容器的内容当作序列（sequence），每个算法可以应用于一个容器中所有值所对应的序列，或者一个子序列（subsequence）。for\_each, find, count\_if, equal, search, copy, unique, rotate, sort....

· 对国际化的支持-

· 对数字处理的支持

· --诊断支持。标准库支持三种报错方式：C的断言（参见条款7），错误号，例外。

标准库中容器和算法这部分一般称为标准模板库（STL）。STL中实际上还有第三个构件 ---- 迭代子（Iterator）

# 条款50: 提高对C++的认识

Nothing.