Effective C++ 笔记

By Ken 2013-03-28

（一） 让自己习惯 C++

- ✧ **条款 1：**视 C++为一个语言联邦. C++高效编程守则视状况而变化，取决于你使用 C++的哪一部分
  1) C
  2) Object-Oriented C++
  3) Template C++ (TMP, template metaprogramming)
  4) STL
- ✧ **条款 2：**尽量以 const, enum, inline 替换#define
  1) #define 宏被编译器展开后，symbol table 里面没有记录，不易调试；而 const 常量会记入 symbol table
  2) 类成员 const static integral types，在声明时可以赋以初值，请在实现文件(而非头文件)中提供定义，如下

```
class GamePlayer {
private:
    static const int NumTurns = 5;   // 常量声明式
    int scores[NumTurns];             // 使用该常量
};
// 实现文件中
const int GamePlayer::NumTurns; // NumTurns 的定义，不用再赋值
```

  3) 取一个 const 的地址是合法的，但取一个 enum 的地址就不合法。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 可以帮助你实现这个约束
  4) 优秀的编译器有可能使用“常量迭代”从而消除为整数型 const 对象分配额外的内存(除非你创建一个 pointer 或 reference 指向该对象)。Enums 和#defines 绝不会导致非必要的内存分配
  5) 在头文件中定义一个常量或 static 变量，然后该头文件被多个实现文件#include 不会造成重定义的错误，因为每个包含这个头文件的实现文件都会有一份私有拷贝。而在头文件中定义一个变量，这个头文件被多个实现文件包含的话，则会造成重定义错误
- ✧ **条款 3：**尽可能使用 const
  1) 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；如果出现在星号两边，表示被指物和指针两者都是常量
  2) const vector<int>::iterator iter = vec.begin(); iter 的作用像个 T* const 表示 iter 本身是 const 的，但 iter 指向的对象可以被改变；vector<int>::const_iterator iter = vec.begin(); iter 的作用像 const T*表示 iter 所指向的对象不能被改变
  3) 令函数返回一个常量值，往往可以降低客户错误而造成的意外，而又不至于放弃安全性和高效性

class Rational {…}
const Rational operator* (const Rational& lhs, const Rational& rhs);
Rational a, b, c;
(a * b) = c ; // prevent this kind of error
if (a*b = c) …

4) const 成员函数不能修改对象的 data member；const 接口表达的意思是 readonly access 对象

5) const 参与函数重载

```
class TextBook {
public:
  const char& operator[](size_t pos) const;  // for const 对象
  char& operator[](size_t pos);  // for non-const 对象
};
```

What about these ?

```
char& operator[](size_t pos) const;
```

// gcc complains: error: invalid initialization of reference of type ‘char&’ from expression of type ‘const char’; want logical constness ?

```
(const) char operator[](size_t pos) const;
```

// valid, for built-in types, it is the almost the same as returning const char&, or const char; but for complex object, it will be inefficient, coz a copy happens

```
const char& operator[](size_t pos);
```

// compile, but this interface is malformed, it has two total opposite meanings and tb[0] = ‘x’ is invalid now

```
char operator[](size_t pos);
```

// compile, but tb[0] = ‘x’ is invalid now

6) bitwise constness 是 C++对常量性的定义，意思是说不改变一个对象内的任何一个 bit (static 成员变量除外)

7) mutable 关键字释放掉 non-static 成员变量的 bitwise constness 的约束

8) 必要时“通过转型，利用 const 实现 non-const 成员函数”；这项技术是安全的，因为能够调用 non-const 成员函数的对象肯定是个 non-const 对象，改变这个对象的 data 是不会有问题的。反之通过转型利用 non-const 成员函数实现 const 成员函数是有问题的，因为 non-const 成员函数本身有可能改变对象数据

```
class TextBook {
  const char& operator[](size_t pos) const
  {
    …  // 边界检查
    …  // 志记数据访问
    …  // 检验数据完整性
    return text[pos];
  }
  char& operator[](size_t pos)
```

```
    {
        …   // 边界检查
        …   // 志记数据访问
        …   // 检验数据完整性
        return text[pos];
    }
…
};
```

➔

```
class TextBook {
    const char& operator[](size_t pos) const
    {
        …   // 边界检查
        …   // 志记数据访问
        …   // 检验数据完整性
        return text[pos];
    }
    char& operator[](size_t pos)
    {
        return const_cast<char&>(static_cast<const TextBook&>(*this)[pos]);
    }
…
};
```

✧ **条款 4：**确定对象使用前已先被初始化

1) 总是通过成员初始化列表来初始化所有成员变量，保持初始化顺序和成员变量声明顺序相同；在合理的情况下，创建一个 private 的初始化函数来初始化某些变量(赋值和初始化表现一样好)，这个初始化函数被多个 constructor 共享
2) 不同实现文件中的 non-local static 对象初始化顺序不确定；一个 workaround 的方法是把 non-local static 对象移到一个函数中，通过这个函数返回对象的 reference。这在多线程下会有不确定性，一个 workaround 是在程序启动的时候调用这些带有 local static 的函数。(永远尽可能不用 static/global 对象)

（二） 构造/析构/赋值计算

✧ **条款 5：**了解 C++默认编写定调用哪些函数

1) class Empty{}; 编译器会自动为它声明编译器版本的一个 default 构造函数、一个 copy 构造函数、一个 copy assignment 操作符和一个析构函数，所有的这些函数都是 inline 的，就像下面声明的一样。惟有当这些函数被需要(调用)的时候，它们才被编译器创建出来

```
class Empty {
public:
    Empty () {…}
    Empty(const Empty& rhs) {…}
    ~Empty(){…}
    Empty& operator=(const Empty& rhs) {…}
};
```

2) 记住，编译器生成的析构函数是 non-virtual 的，除非这个 class 的 base class 自身声明有 virtual 的析构函数(这种情况下这个函数的属性 virtualness 主要来自 base class)
3) 编译器生成的 copy 构造函数和 copy assignment 函数，只是单纯地将源对象的每一个 non-static 成员变量逐 bit 拷贝到目标对象
4) 如果一个类里面包含了 reference 数据成员；const 对象数据成员；或 base class 将 copy assignment 操作符声明为 private，编译器将拒绝为其 derived classes 生成一个 copy assignment 操作符，这些情况下都必须自定义 copy assignment 操作符

✧ **条款 6：**若不想使用编译器自动生成的函数，就应该明确拒绝。

1) 把它们声明为 private 而且故意不去实现它们。Uncopyable 工作原理：如果任何人想 copy HomeForSale 对象，编译器便生成一个 copy 构造函数和一个 copy assignment 操作符，这些函数的“编译器生成版“会尝试调用其 base class 的对应兄弟，那些调用会被编译器拒绝，因为其 base class 的拷贝函数是 private

```
class Uncopyable {
protected:
   Uncopyable() {}
   ~Uncopyable() {}
private:
   Uncopyable(const Uncopyable&);
   Uncopyable& operator=(const Uncopyable&);
};
class HomeForSale: private Uncopyable {
   …
}
```

✧ **条款 7：**为**多态基类**声明 virtual 析构函数

1) C++标准明确指出，当 derived class 对象经由一个 base class 指针被删除，而该 base class 带有一个 non-virtual 析构函数，其结果是未定义的——实际执行时通常发生的是对象的 derived 成分没有被销毁
2) 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(因为 virtual interface 是希望被重写，所以这个 class 是被设计成一个基类)
3) 定义 pure virtual 析构函数，并为之提供一份定义。(析构是编译器必须调用它)

```
class AWOV {
public:
   virtual ~AWOV() = 0;
}
AWOV::~AWOV() {}
```

✧ **条款 8：**别让异常逃离析构函数

1) 为什么异常从析构函数中抛出有害？
   假设 vector<Widget> v 包含 10 个 Widget 对象，Widget 对象的析构函数可能抛出异常，当析构第一个 Widget 的时候，抛出一个异常，其它 9 个 Widget 还是应该被销毁(否则它们持有的资源都会发生泄漏)，因此 v 应该调用它们各个析构函数。但假设在那些调用期间，第二个 Widget 析构函数又抛出异常。现在有两个同时作用的异常，这对 C++而言太多了。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

2) int main(void)

```
{
  try {
    vector<Widget> v(1);
  } catch (…) {
    // catch 不到从 Widget 析构函数中抛出的异常
  }
}
```

3) 案例研究

```
class DBConn {
public:
  …
  ~DBConn()
  {
    db.close();
  }
private:
  DBConnection db;
};
```

如果 db.close()抛出异常，两个方法 a), b)不让异常逃离析构函数。这两种方法都没有什么吸引力。问题在于两者都无法对“导致 close 抛出异常”的情况做出反应。一个较佳的策略是重新设计 DBConn 接口，使客户有机会对可能出现的问题做出反应

a) 结束程序

```
DBConn::~DBConn(){
  try { db.close();
  catch(…) {
    // logging
    std::abort();
  }
}
```

b) 吞下异常

```
DBConn::~DBConn(){
  try { db.close();
  catch(…) {
    // logging
  }
}
```

c) 重新设计接口是得 client 有机会处理这种异常，如果 client 不 care 这种异常，那么就吞下它

```
class DBConn {
public:
  …
```

```
    void close()
    {
      db.close();
      closed = true;
    }
    ~DBConn()
    {
      if (!closed) {
        try {
          db.close();
        } catch (…) {
          // logging
        }
      }
    }
  private:
    DBConnection db;
    bool closed;
  };
```

4) 如果某个操作可能在失败时抛出异常，而又存在某种需要必须处理改异常，那么这个异常必须来自析构函数以外的某个函数

✧ **条款 9：**决不在构造和析构过程中调用 virtual 函数

1) 在 base class 构造期间，virtual 函数不是 virtual 函数。由于 base class 构造函数的执行更早于 derived class 构造函数，当 base class 构造函数执行时 derived class 的成员变量尚未初始化。如果此期间调用的 virtual 函数下降至 derived class 阶层，要知道 derived class 的函数几乎必然取用 local 成员变量，而那些成员变量都还未初始化。还有一个最根本的理由是：在 derived class 对象的 base class 构造期间，对象的类型是 base class 而不是 derived class。不只 virtual 函数会被编译器解析至 base class，若使用 RTTI(dynamic_cast 或 typeid)，也会把对象视为 base class 类型。相同的道理也适用于析构函数，一旦 derived class 析构函数开始执行，对象内的 derived class 成员变量便呈现未定义值，所以 C++视它们仿佛不存在。进入 base class 析构函数后对象就成为一个 base class 对象

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

1) 为了实现“连锁赋值”

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

2) 这个协议也适用于+=，-=，*=等等。string/vector/complex/shared_ptr 等都遵循

这个协议

✧ **条款 11：**在 operator=中处理“自我赋值”

1) 案例研究

```
// 如果 rhs 和 this 是同一个对象，死定了
Widget& Widget::operator=(const Widget& rhs)
{
   delete pb;
   pb = new Bitmap(*rhs.pb);
   return *this;
}
// 如果 new Bitmap 失败了，又死定了，pb 已经 delete 掉了
Widget& Widget::operator=(const Widget& rhs)
{
   if (this == &rhs) return *this;

   delete pb;
   pb = new Bitmap(*rhs.pb);
   return *this;
}
// 正确版本 1
Widget& Widget::operator=(const Widget& rhs)
{
   // if (this == &rhs) return *this; //必要时可以加上，但不加也无妨

   Bitmap* pOrig = pb;
   pb = new Bitmap(*rhs.pb);
   delete pOrig;
   return *this;
}
// 正确版本 2, copy-and-swap
class Widget {
…
void swap(Widget& rhs); // no exception
…
};
Widget& Widget::operator=(const Widget& rhs)
{
   Widget temp(rhs);
   swap(temp);
   return *this;
}
```

✧ **条款 12：**复制对象时勿忘其每一个成分

1) 如果你为 class 添加了成员变量，你必须同时修改自定义的 copying 函数
2) 任何时候只要你承担起“为 derived class 撰写 copying 函数”的重责大任，必

须很小心地也复制其 base class 部分。那些成分往往是 private 的，所以你复发直接访问它们，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;
}
```

（三） 资源管理

✧ **条款 13：**以对象管理资源

1) 永远把资源放进对象 RAII，利用对象的析构函数确保资源被释放
2) shared_ptr/scoped_ptr/scoped_array/shared_array

✧ **条款 14：**在资源管理类中小心 copying 行为

1) 多个资源管理类同时指向相同的资源有可能造成多次释放的错误，解决方案有
   - 禁止复制资源管理类(scoped_ptr)
   - 对底层资源祭出“引用计数法”(shared_ptr)
   - 复制底部资源(Deep copy)
   - 转移底部资源的拥有权(auto_ptr)

✧ **条款 15：**在资源管理类中提供对原始资源的访问

1) 提供对原始资源的访问是因为 API 往往要访问原始资源
   - 采用显式转换函数(shared_ptr.get()) // Prefer this way, Python: Explicit is better than implicit
   - 采用隐式转换(operator Type() const) (语法自然，容易使用，但容易滋生错误)
2) RAII class 内的返回原始资源的函数，与“封装”发生一定的矛盾。但它谈不上什么设计灾难。RAII class 并不是为了封装某物而存在；它们的存在是为了确保一个特殊行为——资源释放——会发生

✧ **条款 16：**成对使用 new 和 delete

1) new/delete 成对使用
2) new []/delete []成对使用

✧ **条款 17：**以独立语句将 newed 对象置入智能指针

1) int priority();
   void processWidget(shared_ptr<Widget> pw, int priority);
   processWidget(shared_ptr<Widget>(new Widget), priority())
   上述调用可能有资源泄漏，C++有自由使用下面的调用顺序，如果 priority()调用有异常，Widget 就泄漏了
   1. 执行”new Widget”
   2. 调用 priority()
   3. 调用 shared_ptr 构造函数

解决方法：

```
shared_ptr<Widget> pw(new Widget);
processWidget(pw, priority());
```

（四） 设计与声明

✧ **条款 18**：让接口容易被使用，不易被误用

1) 许多客户端错误可以因为导入新类型而获得预防。利用 C++的类型系统探测错误

案例：

```
class Month {
public:
   static Month Jan() { return Month(1); }
   …
private:
   explicit Month(int m);
};
Date d(Month::Mar(), Day(30), Year(1995));
```

2) 预防客户错误的另外一个办法是，限制类型内什么事可做，什么事不能做。常见的限制是加上 const

3) 除非有好理由，否则应该尽量令你的 types 的行为与内置 types 一致。像 ints 一样行事

4) 任何接口如果要求客户必须记得做某些事情，就是有着“不正确使用”的倾向，因为客户可能忘记做那件事

```
// Not good
Investment* createInvestment();
void getRidOfInvestment(Investment*);
// Good
shared_ptr<Investment> createInvestment()
{
   shared_ptr<Investment>   retVal(static_cast<Invenstment*>(0),   getRidOfInvestment);
   retVal = … //  令 retVal 指向正确对象
   return retVal;
}
```

这种方法也消除了 cross-DLL problem (在一个 DLL 中 new，在另外一个 DLL 钟 delete)

✧ **条款 19**：设计 class 犹如设计 type

1) 如何设计高效的 classes，可以对下提出问题:

- 新 type 的对象该如何被创建和销毁？构造函数、析构函数、内存分配函数 operator new 和 operator new[]，内存释放函数 operator delete 和 operator delete []
- 对象的初始化和对象的赋值该有什么样的差别？构造函数和赋值操作符
- 新 type 的对象如果被 passed by value，意味着什么？copy 构造函数
- 什么是新 type 的“合法值”？class invariants
- 你的新 type 需要配合某个继承图吗？type conversion operator 或 non-explicit-one-argument 构造函数或 explicit type conversion function

- 你的新 type 需要什么样的转换？
- 什么样的操作符和函数对此新 type 而言是合理的？
- 什么样的标准函数应该驳回？
- 谁该取用新 type 的成员？
- 什么是新 type 的“未声明接口”？
- 你的新 type 有多么一般化？是否要定义成 class template？
- 你真的需要一个新 type 吗？单纯定义一个或多个 non-member 函数或 templates，是否能够到达目标？

✧ **条款 20：宁以 pass-by-reference-to-const 替换 pass-by-value**

1) 对于内置类型选择 pass-by-value 或许更高效，STL 迭代器或函数对象习惯上被设计为 passed by value。迭代器和函数对象的实现者有责任看看它们是否高效且不受切割问题影响
2) 某些编译器对待“内置类型”和“用户自定义类型”的态度截然不同。比如，某些编译器拒绝把只由一个 double 组成的对象放进缓冲器内，却乐意在一个正规基础上对光秃秃的 doubles 那么做。在这种情况下，pass-by-reference 对待这些对象更高效

✧ **条款 21：必须返回对象时，别妄想返回其 reference**

1)
```
inline const Rational operator * (const Rational& lhs, const Rational& rhs)
{
  return Rational(lhs.n * rhs.n, lhs.d * rhs.d);
}
```

✧ **条款 22：将成员变量声明为 private**

1) 为每个数据成员变量写一个 getter/setter 函数在绝大多数情况下是值得的，因为这为日后的所有可能的更改提供弹性，比如插入 logging，更新状态等等
2) protected 并不比 public 更具封装性

✧ **条款 23：宁以 non-member，non-friend 替换 member 函数**

1) 案例研究

```
class WebBrowser {
public:
  …
  void clearCache();
  void clearHistory();
  void removeCookies();
  void clearEverythin(); // call clearCache, clearHistory and removeCookies
};
void clearBrowser(WebBrowser& wb)
{
  clearCache();
  clearHistory();
  removeCookies();
}
// the above interface is better than WebBrowser::clearEverything(), why ?
```

越少代码可以看到数据(也就是访问它)，越多的数据可被封装，而我们也就越能自由地改变数据。越多函数可以访问数据，数据的封装性就越低。这说明

non-member non-friend 的 clearBrowser 函数更好的封装性

一个 alternative 方案是把 clearBrowser 实现成 WebBrowser 的 static member 函数。

如果和 WebBrowser 这个类有大量相关的 utility 函数，比如某些与书签有关，某些与打印有关，还有一些与 cookie 的管理有关…，一个合理的作法是把这些 utility 函数分别放入不同的头文件和实现文件，但同归属于一个 namespace，这样编译依赖性最低，因为某个 client 只对书签相关的函数感兴趣也要包含一堆和 cookie 相关的便利函数。C++标准库就是这么组织的。

```
//头文件"webbrowser.h"包含对 class WebBrowser 自身以及其核心功能
namespace WebBrowserStuff {
class WebBrowser { … };
… // 核心机能，几乎每个 client 都需要访问的 non-member 函数
}
// 头文集"webbrowserbookmarks.h"
namespace WebBrowserStuff {
… // 与书签有关的便利函数
}
// 头文集"webbrowsercookies.h"
namespace WebBrowserStuff {
… // 与 cookie 有关的便利函数
}
```

✧ **条款 24：**若所有参数皆需类型转换，请为此采用 non-member 函数

1) 案例研究

```
class Rational {
public:
   Rational (int numerator = 0, int denominator = 1);
   int numerator() const;
   int demoninator() const;
   const Rational operator* (const Rational& rhs) const;
…
};
Rational oneEighth(1, 8);
Rational oneHalf(1, 2);
Rational result = oneEighth * oneHalf;    // good !
result = oneHalf * 2;      // good !
result = 2 * oneHalf;    // hmm…
```

方案：

```
// non-member function
const Rational operator* (const Rational& lhs, const Rational& rhs)
{
   return Rational(lhs.numerator() * rhs.numerator(),
                   lhs.demoninator () * rhs.demoninator ());
}
```

✧ **条款 25：**考虑写出一个不抛出异常的 swap 函数

1) 案例研究

```
class WidgetImpl {
public:
…
private:
   int a, b, c;
   std::vector<double> v;
};

class Widget {
public:
   Widget (const Widget& rhs);
   Widget& operator= (const Widget& rhs)
   {
      …
      *pImpl = *(rhs.pImpl);
      …
   }
   // 1) 提供一个高效的 member swap 函数
   void swap (Widget& other)
   {
      using std:swap;    // don't forget this
      swap (pImpl, other.pImpl);
   }
private:
   WidgetImpl* pImpl;
};

// 2) None-member swap 函数，并不属于 std 命名空间
template<typename T>
void swap (Widget<T>& a, Widget<T>& b)
{
   a.swap(b);
}

// 3) 提供一个全特化 std::swap，如果我们在编写一个 class (而非一个 class template)
namespace std {
template<>
void swap<Widget> (Widget& a, Widget& b)
{
   a.swap(b);
}
```

}

注意，如果我们在编写一个 class template，提供如下的偏特化的 swap 是不能编译的，function 不能被偏特化

```
namespace std {
template<typename T>
void swap<Widget<T> > (Widget<T>& a, Widget<T>& b)
{
  a.swap(b);
}
}
```

2) 往 std namespace 中添加东西结果是未定义的，但我们可以为标准 templates 制造特化版本

（五） 实现

✧ 条款 26：尽可能延后变量定义式的出现时间

1) 案例研究

```
// 方法 A：定义于循环外
Widget w;
for (int i = 0; i < n; ++i)
{
  w = 取决于 i 的某个值;
  …
}
```

```
// 方法 B：定义于循环内
for (int i = 0; i < n; ++i)
{
  Widget w(取决于 i 的某个值);
  …
}
```

- 做法 A：1 个构造函数 + 1 个析构函数 + n 个赋值操作
- 做法 B：n 个构造函数 + n 个析构函数

除非(1)赋值成本被“构造+析构”成本低，(2)你正在处理 performance-senstive 的部分，否则应该使用做法 B

✧ 条款 27：尽量少做转型动作

1) 任何一个类型转换(不论是通过转型操作而进行的显示转换或是通过编译器完成的隐式转换)往往真的令编译器编译出运行期执行的代码

比如下面的转型几乎都会产生额外的代码

```
int x, y;
…
double d = static_cast<double>(x)/y;
Derived d;
Base* pb = &d;
```

2) static_cast<Widget>(*pDerivedWidget)会产生一个临时对象
3) dynamic_cast 性能较差，因为一般的编译器实现是基于继承体系类的类名的字符串比较，如果一个类有比较深的继承体系，那么每次转型时 strcmp 调用的次数就越多
4) 绝对避免 cascading dynamic_cast(把 dynamic_cast 作为 if 判断然后做 dispatch)

✧ 条款 28：避免返回 handles 指向对象内部成分 (注意和条款 15 对比)

1) 案例研究

```
class Point {
public:
```

```
    Point (int x, int y);
    void setX (int x);
    void setY (int y);
};
class RectData {
    Point ulhc;
    Point lrhc;
};
class Rectangle {
…
private:
    Point& upperLeft() const { return pData->ulhc; }
    Point& lowerRight() const { return pData->lrhc; }
…
};
```

这样的设计可以编译通过，但却是错误的。实际上它是自我矛盾的。一方面 upperLeft 和 lowerRight 被声明为 const 成员函数，因为它们的目的只是为了提供一个得之 Rectangle 相关坐标点的方法，而不是让客户修改 Rectangle。另外一方面两个函数却都返回 references 指向 private 内部数据，调用者于是可通过这些 reference 更改内部数据

```
const Rectangle rec(x, y, z, zz);
rec.upperLeft().setX(50);
```

正确的修正是加上返回 reference to const

两个教训：

- 成员变量的封装性最多只等于“返回其 reference”的函数的访问级别。本例中虽然 ulhc 和 lrhc 都被声明为 private，但实际上却是 public
- 如果 const 成员函数传递出一个 reference，后者所指数据与对象自身有关联，而它又被存储于对象之外，那么这个函数的调用者可以修改那笔数据。(注意如果把 shared_ptr 改为 RectData data，那么编译通不过)。这正是 bitwise constness 的一个附带结果

2) string/vector 允许你通过 operator[]“摘采”个别元素，而这些 operator[]就是返回 reference 指向“容器内部的数据”，那些数据会随着容器被销毁而销毁。尽管如此，这样的函数毕竟是例外，不是常态

✧ **条款 29：为“异常安全”而努力是值得的**

1) 异常安全函数即使发生也不会泄露资源或允许任何数据结构败坏。这样的函数区分为三种可能的保证：基本保证，强烈保证，不抛出异常
2) “强烈保证”往往能够以 copy-and-swap 实现出来，但“强烈保证”并非对所有函数都可实现或具备现实意义

✧ **条款 30：透彻了解 inlining 的里里外外**

1) implicit inline，直接定义在 class 内部的成员函数
2) explicit inline，加上 inline 关键字的函数。
3) template 的具现化与 inlining 无关，如果一个 template 的所有具现化都应该是 inlined，那么需将此 template 声明为 inline
4) 复杂函数、virtual 成员函数，被取地址的函数编译器都将生成 outlined 函数本

体

5) 构造函数和析构函数往往是 inline 的糟糕候选，因为它们会调用父类的和各个成员变量的构造函数和析构函数

6) inline 函数改动后需要重新编译代码，而且可能导致代码膨胀

✧ **条款 31：将文件件的编译依存关系降至最低**

1) 采用 pImpl 手法(handle classes)或 interface classes + static factory 函数来降低编译依存性

2) 如果能够使用 object reference 或 object pointer 完成任务，就不要使用 objects

3) 如果能够，尽量以 class(前向)声明替换 class 定义式。<iosfwd>

（六） 继承与面向对象设计

✧ **条款 32：确定你的 public 继承塑模出 is-a 关系**

1) public 继承意味着 is-a。适用于 base classes 身上的每一件事情一定也适用于 derived classes 身上，因为每一个 derived class 对象也都是一个 base class 对象

✧ **条款 33：避免遮掩继承而来的名称**

1) 案例研究

```
class Base {
public:
  virtual void mf1() = 0;
  virtual void mf1(int);
  virtual void mf2();
  void mf3();
  void mf3(double);
};
class Derived: public Base {
public:
  // using Base::mf1;
  // using Base::mf3;
  virtual void mf1();
  void mf3();
  void mf4();
};
```

Base class 内所有名为 mf1 和 mf3 的函数都被 derived class 内的 mf1 和 mf3 函数遮掩掉了。从名称查找观点来看，Base::mf1 和 Base::mf3 不再被 Derived 继承

```
Derived d;
int x;
d.mf1();
d.mf1(x); // error !可以使用 using 暴露 base 的成员函数定义
d.mf2();
d.mf3();
d.mf3(x); // error !可以使用 using 暴露 base 的成员函数定义
```

✧ **条款 34：区分接口继承和实现继承**

1) 在 public 继承之下，derived classes 总是继承 base class 的接口

2) 声明一个 pure virtual 成员函数的目的是为了让 derived classes 只继承函数接口

3) 声明一个 impure virtual 成员函数的目的是让 derived classes 继承该函数的接口和缺省实现
4) 声明 non-virtual 成员函数的目的是为了令 derived classes 继承函数的接口及一份强制实现
5) Shape * ps = new Rectangle; ps->Shape::draw(); //调用 Base 的 virtual member function

✧ **条款 35：考虑 virtual 函数意外的其他选择**

1) Template method (NVI – Non-Virtual Interface which calls private or protected virtual functions)
2) Strategy (function pointer or tr1::function)，可以动态改变要 call 的 function

✧ **条款 36：绝不重新定义继承而来的 non-virtual 函数**

1) Non-virtual 函数都是静态绑定的
2) B *pB = new D; pB 的静态类型是 B*

✧ **条款 37：绝不重新定义继承而来的缺省参数值**

1) virtual 函数是动态绑定，而缺省参数值是静态绑定的。C++为了注重运行期效率，对缺省参数进行静态绑定
2) 案例研究

```
class Shape {
public:
  enum ShapeColor {Red, Green, Blue};
  virtual void draw (ShapeColor color = Red) const = 0;
};
class Rectangle: public Shape {
public:
  virtual void draw (ShapeColor color = Green) const;
};
class Circle: public Shape {
public:
  virtual void draw (ShapeColor color) const;
  // 注意，Circle::draw 这样写，当客户以对象调用此函数，
  // 一定要指定参数值。因为静态绑定下这个函数并不从
  // base 继承缺省参数值。但若以指针或 reference 调用此函数
  // 可以不指定参数值，因为动态绑定下这个函数会从 base
  // 继承缺省参数
};
Shape* ps = new Rectangle;
ps->draw();
```

ps 静态类型是 Shape*，动态类型是 Rectangle*，缺省参数是静态绑定的，所以最终 color 的缺省值是 Red 而非 Green

如果改为：

```
class Rectangle: public Shape {
public:
  virtual void draw (ShapeColor color = Red) const;
};
```

又是糟糕的代码——base 改了，derived 也得跟着改

正确的设计是使用 NVI 手法(永远尽量不要使用 virtual public interface)

```
class Shape {
public:
   enum ShapeColor {Red, Green, Blue};
   void draw (ShapeColor color = Red) const
   {
      doDraw(color);
   }
private:
   virtual void doDraw (ShapeColor color) const = 0;
};
```

✧ **条款 38：通过复合塑模出 has-a 或“根据某物实现出”**

✧ **条款 39：明智而审慎地使用 private 继承**

1) 如果 classes 之间的继承关系是 private，编译器不会自动将一个 derived 对象转换为一个 base class 对象
2) Private 继承在软件设计层面没有意义，其意义在于软件实现层面
3) 尽量使用复合，而不要使用 private 继承，只有当 protected/virtual 函数牵扯竟来的时候才考虑使用 private 继承；另外一个激进的情况，是 Empty base class，并且空间利害关系要紧时(Empty base optimization)。“独立(非附属)”对象的大小一定不为 0.

✧ **条款 40：明智而审慎地使用多重继承**

1) 非必要不使用 virtual bases。平常使用 non-virtual 继承。
2) 如果必须使用 virtual bases，尽可能避免在其中放置数据
3) 多重继承的确有正当的用途。其中一个情节设计“public 继承某个 interface class”和“private 继承某个协助实现的 class”的两相组合

（七） 模板与泛型编程

✧ **条款 41：了解隐式接口和编译器多态**

1) Template，接口是 implicit，多态则是通过 template 具现化和函数重载解析发生在编译期

✧ **条款 42：了解 typename 的双重意义**

1) 当 template 内出现的类型名称相依于某个 template 参数时，需要使用 typename 显式告诉编译器相依的名称是一个类型
2) 例如：
3) template <typename C>
4) void f (const C& container, typename C::iterator iter);
5) 请使用 typename 标志嵌套从属类型名称；但不得在 base class list 或 member initialization list 内以它作为 base class 的修饰符

✧ **条款 43：学习访问模板化基类的名称**

1) 案例研究

```
class MsgInfo {…};
template<typename Company>
class MsgSender {
public:
```

```
    void sendClear (const MsgInfo& info)
    {
        string msg;
        // 根据 info 产生 msg
        Company c;
        c.sendClearText (msg);
    }
    void sendSecret (const MsgInfo& info) { …c.sendEncrypted(msg); }
};
template<typename Copmany>
class LoggingMsgSender: public MsgSender<Company> {
public:
    void sendClearMsg (const MsgInfo& info)
    {
        // 将“传送前”的信息写至 log;
        sendClear(info);
        // 将“传送后”的信息写至 log;
    }
};
```

编译通不过：原因当编译器遇到 class template LoggingMsgSender 定义时，并不知道它继承什么样的 class。(它继承自 MsgSender<Company>，但其中的 Company 是 template 参数，不到后来具现化，无法知道它是什么。想想 MsgSender<Company>可以全特化或偏特化，而特化版本压根就没有 sendClear 函数)

解决方案 1，加 this->：

```
    void sendClearMsg (const MsgInfo& info)
    {
        // 将“传送前”的信息写至 log;
        this->sendClear(info);
        // 将“传送后”的信息写至 log;
    }
```

方案 2，使用 using 声明

```
    using MsgSender<Company>::sendClear;
    void sendClearMsg (const MsgInfo& info)
    {
        // 将“传送前”的信息写至 log;
        sendClear(info);
        // 将“传送后”的信息写至 log;
    }
```

方案 3，显式地调用位于 base class 内的函数。(但关闭了多态的可能性)

```
    void sendClearMsg (const MsgInfo& info)
    {
        // 将“传送前”的信息写至 log;
        MsgSender<Company>::sendClear(info);
```

```
    // 将“传送后”的信息写至 log;
  }
```

✧ **条款 44：将与参数无关的代码抽离 templates**

1) 分析函数的共性与变性，将共同的部分移至第三个函数，将变化的部分留在原函数
2) 因非类型模板参数(non-type template parameters)而造成的代码膨胀，往往可消除，做法是以函数参数或 class 成员变量替换 template 参数
3) 因类型参数而造成的代码膨胀，往往可以降低，做法是让带有完全相同的二进制表述的具现类型共享实现代码

✧ **条款 45：运用成员函数模板接受所有兼容类型**

1) 案例研究

```
template<class T>
class shard_ptr {
public:
  shared_ptr(shared_ptr const& r); // copy 构造函数
  template<class Y>                // 泛化的 copy 构造函数
  shared_ptr(shared_ptr<Y> const& r);
  shared_ptr& operator=(shared_ptr const& r); // copy assignment
  template<class Y>                // 泛化的 copy assignment
  shared_ptr& operator=(shared_ptr<Y> const& r);
};
```

2) 如果你声明 member templates 用于泛化 copy 构造或泛化 assignment 操作，你还是需要声明正常的 copy 构造函数和 copy assignment 操作符

✧ **条款 46：需要类型转换时请为模板类定义非成员函数**

1) 在 template 实参推导过程中从不将隐式类型转换函数纳入考虑
2) 案例研究

```
template<typename T>
class Rational {
public:
  Rational(const T& numerator = 0, const T& denominator = 1);
  const T numerator () const;
  const T denominator () const;
  …
};
template <typename T>
const Rational<T> operator* (const Rational<T>& lhs, const Rational<T>& rhs)
{
  …
}
Rational<int> oneHalf(1, 2);
Rational<int> result = oneHalf * 2;
// 上述代码编译通不过，原因就是 1). 第二个参数 2，不能隐式转换为 Rational<int>对象
解决方案：为了让类型转换可能发生在所有参数身上，我们需要一个
```

non-member 函数(条款 24)；为了令这个函数被自动具现化，我们需要将它声明在 template class 内部；而在 class 内部声明 non-member 函数的唯一办法是令它成为一个 friend

```
template<typename T>
class Rational {
public:
  …
  // 在 Rational template 内部提供 friend function 的定义是必须的
  friend Rational operator* (const Rational& lhs, const Rational& rhs)
  {
    return doMultiply(lhs, rhs);    // call helper function template
  }
  // friend Rational<T> operator* (const Rational<T> & lhs,
                                   const Rational<T> & rhs);
};
template<typename T>
const Rational<T> doMultiply(const Rational<T> & lhs,
                             const Rational<T> & rhs)
{
  return Rational<T>(lhs.numerator() * rhs.numerator(),
                     lhs.denominator() * rhs. denominator ())
}
```

3) 当我们编写一个 class template，而它提供的“与此 template 相关的”函数支持“所有参数的隐式类型转换”时，请将那些函数定义为“class template”内部的 friend 函数

✧ 条款 47：请使用 traits classes 表现类型信息

1) 对于 5 种迭代器，C++标准分别提供专属的卷标结构(tag struct)加以区分

```
struct input_iterator_tag {};
struct output_iterator_tag {};
struct forward_iterator_tag: public input_iterator_tag {};
struct bidirectional_iterator_tag: public forward_iterator_tag {};
struct random_access_iterator_tag: public bidirectional_iterator_tag {};

template<…>
class deque {
public:
  class iterator {
    typedef random_access_iterator_tag iterator_category;
    …
  };
…
};
template<…>
class list {
```

```
public:
   class iterator {
      typedef bidirectional_iterator_tag iterator_category;
      …
   };
…
};

template<typename IterT>
struct iterator_traits {
   typedef typename IterT::iterator_category iterator_category;
   …
};

// 一组偏特化，partial template specification
template<typename IterT>
struct iterator_traits <IterT*>{
   typedef random_access_iterator_tag iterator_category;
   …
};

// 一组函数重载，消除 if…else
template<typename IterT, typename DistT>
void doAdvance(IterT& iter, DistT d, random_access_iterator_tag)
{
   iter += d;
}

template<typename IterT, typename DistT>
void doAdvance(IterT& iter, DistT d, bidirectional_iterator_tag )
{
   if (d >= 0) { while (d--) ++iter; }
   else { while (d++) --iter; }
}

template<typename IterT, typename DistT>
void doAdvance(IterT& iter, DistT d, input_iterator_tag )
{
   if (d < 0) { throw out_of_range("Negative distance"); }
   while (d--) ++iter;
}

// advance
template <typename IterT, typename DistT>
```

```
void advance(Iter& iter, DistT d)
{
   doAdvance(iter, d, typename iterator_traits<IterT>::iterator_category());
}
```

✧ **条款 48：认识 template 元编程 (TMP)**

1) policy based design

（八） 定制 new 和 delete

✧ **条款 49：了解 new-handler 的行为**

1) 当 operator new 无法满足某一内存分配需求时，它会抛出 bad_alloc 异常(以前它会返回 NULL)，在抛出异常之前，它会先调用一个客户指定的错误处理函数即 new-handler

```
namespace std {
typedef void (*new_handler)();
new_handler set_new_handler(new_handler p) throw ();
}
```

2) Widget* pw = new (std::nothrow) Widget; operator new 失败时返回 NULL，但如果 Widget 构造函数本身需要使用 new，但没有使用 nothrow，那么异常还是会被传播的。所以没有必要使用 nothrow new

✧ **条款 50：了解 new 和 delete 的合理替换时机**

1) 使用 boost Pool 或其它成熟的内存管理/分配库，尽量不要自己重定义 operator new(内存对齐，多线程安全)

✧ **条款 51：编写 new 和 delete 时需要固守常规**

1) operator new 应该内涵一个无穷循环，并在其中尝试分配内存，如果它无法满足内存需求，就该调用 new-handler。它也应该有能力处理 0 bytes 申请。Class 专属版本则应该处理“比正确大小更大的(错误)申请”(class 被继承了，则 derived class 可能不适合专属于 base class 的 operator new)
2) operator delete 应该收到 null 指针时不做任何事。Class 专属版本则应该处理“比正确大小更大的(错误)申请”
3) 案例研究

```
void * opearator new(size_t size) throw bad_alloc)
{
   using namespace std;
   if (size == 0) {
      size = 1;
   }
   while (true) {
      try to allocate size bytes;
      if (success)
         return pointer;

      new_handler globalHandler = set_new_handler(0);
      set_new_handler(globalHandler);

      if (globalHandler) (*globalHandler)();
```

```
        else throw bad_alloc;
    }
}

class Base {
public:
    static void* operator new (size_t size) throw (bad_alloc);
    static void operator delete (size_t size) throw ();
};

void * Base::operator new (size_t size) throw (bad_alloc)
{
    if (size != sizeof(Base))
        return ::operator new(size)
    …
}

void operator delete (void* rawMemory, size_t size) throw ()
{
    if (rawMemory == 0) return
    if (size != sizeof(Base)) {
        ::operator delete (rawMemory);
        return
    }
    …
}
```

✧ 条款 52：写了 placement new 也要写 placement delete

1) 如果 operator new 接受的参数除了一定会有的那个 size_t 之外还有其它，这便是个 placement new
2) 如果内存分配成功，但构造函数抛出异常，运行期系统有责任取消 operator new 的分配并恢复旧观。运行期系统寻找“参数个数和类型都与 operator new 相同的”某个 operator delete。如果找到，那就是它的调用对象。如果没有找到，那么什么也不做。placement delete 只有在“伴随 placement new 调用而触发构造函数”出现异常时才会被调用。对一个指针实施 delete 绝不会导致调用 placement delete.所以我们必须提供一个正常的 operator delete (用于构造期间没有异常抛出)和一个 placement 版本(用于构造期间有异常抛出),后者的参数必须和 placement operator new 一样
3) C++标准库定义了一下形式的 operator new：

```
void operator new (size_t) throw (bad_alloc);
void operator new (size_t, void*) throw ();
void operator new (size_t, const nothrow_t&) throw ();
```

4) 案例研究

```
class StandardNewDeleteForms {
```

```
public:
   // normal new/delete
   static void* operator new (size_t size) throw (bad_alloc)
   { return ::operator new (size);}
   static void operator delete (void* pMemory) throw ()
   { ::operator delete (pMemory); }

   // placement new/delete
   static void* operator new (size_t size, void* ptr) throw ()
   { return ::operator new (size, ptr); }
   static void operator delete (void* pMemory, void* ptr) throw ()
   { return ::operator delete (pMemory, ptr); }

   // nothrow new/delete
   static void* operator new (size_t size, const nothrow_t& nt) throw ()
   { return ::operator new (size, nt); }
   static void operator delete (void* pMemory, const nothrow_t&) throw ()
   { return ::operator delete (pMemory); }
};
class Widget: public StandardNewDeleteForms {
public:
   using StandardNewDeleteForms::operator new;
   using StandardNewDeleteForms::operator delete;
   static void* operator new(size_t size, ostream& logStream) throw (bad_alloc);
   static void* operator delete(void* ptr, ostream& logStream) throw ();
};
```

（九） 杂项讨论

- ✧ **条款 53：**不要轻忽编译器的警告
- ✧ **条款 54：**让自己熟悉包括 TR1 在内的标准程序库
- ✧ **条款 55：**让自己熟悉 boost