1

让自己习惯 C++

Accustoming Yourself to C++

不论你的编程背景是什么，C++ 都可能让你觉得有点儿熟悉。它是一个威力强大的语言，带着众多特性，但是在你可以驾驭其威力并有效运用其特性之前，你必须先习惯 C++ 的办事方式。本书谈的便是这个。总有某些东西比其他更基础些，本章就是最基本的一些东西。

条款 01：视 C++ 为一个语言联邦

View C++ as a federation of languages.

一开始，C++只是 C 加上一些面向对象特性。C++ 最初的名称 C with Classes 也反映了这个血缘关系。

但是当这个语言逐渐成熟，它变得更活跃更无拘束，更大胆更冒险，开始接受不同于 C with Classes 的各种观念、特性和编程战略。Exceptions（异常）对函数的结构化带来不同的做法（见条款 29），templates（模板）将我们带到新的设计思考方式（见条款 41）， STL 则定义了一个前所未见的伸展性做法。

今天的 C++ 已经是个多重范型编程语言（multiparadigm programming language），一个同时支持过程形式（procedural）、面向对象形式（object-oriented）、函数形式（functional）、泛型形式（generic）、元编程形式（metaprogramming）的语言。这些能力和弹性使 C++ 成为一个无可匹敌的工具，但也可能引发某些迷惑：所有“适当用法”似乎都有例外。我们该如何理解这样一个语言呢？

最简单的方法是将 C++ 视为一个由相关语言组成的联邦而非单一语言。在其某个次语言（sublanguage）中，各种守则与通例都倾向简单、直观易懂、并且容易

记住。然而当你从一个次语言移往另一个次语言，守则可能改变。为了理解 C++，你必须认识其主要的次语言。幸运的是总共只有四个：

- C。说到底 C++ 仍是以 C 为基础。区块（blocks）、语句（statements）、预处理器（preprocessor）、内置数据类型（built-in data types）、数组（arrays）、指针（pointers）等统统来自 C。许多时候 C++ 对问题的解法其实不过就是较高级的 C 解法（例如条款 2 谈到预处理器之外的另一选择，条款 13 谈到以对象管理资源），但当你以 C++ 内的 C 成分工作时，高效编程守则映照出 C 语言的局限：没有模板（templates），没有异常（exceptions），没有重载（overloading）……
- Object-Oriented C++。这部分也就是 C with Classes 所诉求的：classes（包括构造函数和析构函数），封装（encapsulation）、继承（inheritance）、多态（polymorphism）、virtual 函数（动态绑定）……等等。这一部分是面向对象设计之古典守则在 C++ 上的最直接实施。
- Template C++。这是 C++ 的泛型编程（generic programming）部分，也是大多数程序员经验最少的部分。Template 相关考虑与设计已经弥漫整个 C++，良好编程守则中“惟 template 适用”的特殊条款并不罕见（例如条款 46 谈到调用 template functions 时如何协助类型转换）。实际上由于 templates 威力强大，它们带来崭新的编程范型（programming paradigm），也就是所谓的 template metaprogramming（TMP，模板元编程）。条款 48 对此提供了一份概述，但除非你是 template 激进团队的中坚骨干，大可不必太担心这些。TMP 相关规则很少与 C++ 主流编程互相影响。
- STL。STL 是个 template 程序库，看名称也知道，但它是非常特殊的一个。它对容器（containers）、迭代器（iterators）、算法（algorithms）以及函数对象（function objects）的规约有极佳的紧密配合与协调，然而 templates 及程序库也可以其他想法建置出来。STL 有自己特殊的办事方式，当你伙同 STL 一起工作，你必须遵守它的规约。

记住这四个次语言，当你从某个次语言切换到另一个，导致高效编程守则要求你改变策略时，不要感到惊讶。例如对内置（也就是 C-like）类型而言 *pass-by-value* 通常比 *pass-by-reference* 高效，但当你从 C part of C++ 移往 Object-Oriented C++，由于用户自定义（user-defined）构造函数和析构函数的存在，*pass-by-reference-to-*const 往往更好。运用 Template C++ 时尤其如此，因为彼时你

甚至不知道所处理的对象的类型。然而一旦跨入 STL 你就会了解，迭代器和函数对象都是在 C 指针之上塑造出来的，所以对 STL 的迭代器和函数对象而言，旧式的 C *pass-by-value* 守则再次适用（参数传递方式的选择细节请见条款 20）。

因此我说，C++ 并不是一个带有一组守则的一体语言；它是从四个次语言组成的联邦政府，每个次语言都有自己的规约。记住这四个次语言你就会发现 C++ 容易了解得多。

请记住

- 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。

条款 02：尽量以 const, enum, inline 替换 #define

Prefer consts,enums, and inlines to #defines.

这个条款或许改为“宁可以编译器替换预处理器”比较好，因为或许 #define 不被视为语言的一部分。那正是它的问题所在。当你做出这样的事情：

```
#define ASPECT_RATIO 1.653
```

记号名称 ASPECT_RATIO 也许从未被编译器看见；也许在编译器开始处理源码之前它就被预处理器移走了。于是记号名称 ASPECT_RATIO 有可能没进入记号表（symbol table）内。于是当你运用此常量但获得一个编译错误信息时，可能会带来困惑，因为这个错误信息也许会提到 1.653 而不是 ASPECT_RATIO。如果 ASPECT_RATIO 被定义在一个非你所写的头文件内，你肯定对 1.653 以及它来自何处毫无概念，于是你将因为追踪它而浪费时间。这个问题也可能出现在记号式调试器（symbolic debugger）中，原因相同：你所使用的名称可能并未进入记号表（symbol table）。

解决之道是以一个常量替换上述的宏（#define）：

```
const double AspectRatio = 1.653;      //大写名称通常用于宏，
                                       //因此这里改变名称写法。
```

作为一个语言常量，AspectRatio 肯定会被编译器看到，当然就会进入记号表内。此外对浮点常量（floating point constant，就像本例）而言，使用常量可能比使用#define 导致较小量的码，因为预处理器“盲目地将宏名称 ASPECT_RATIO 替换为 1.653”可能导致目标码（object code）出现多份 1.653，若改用常量 AspectRatio 绝不会出现相同情况。

当我们以常量替换#defines，有两种特殊情况值得说说。第一是定义常量指针（constant pointers）。由于常量定义式通常被放在头文件内（以便被不同的源码含入），因此有必要将指针（而不只是指针所指之物）声明为 const。例如若要在头文件内定义一个常量的（不变的）char*-based 字符串，你必须写 const 两次：

```
const char* const authorName = "Scott Meyers";
```

关于 const 的意义和使用（特别是当它与指针结合时），条款 3 有完整的讨论。这里值得先提醒你的是，string 对象通常比其前辈 char*-based 合宜，所以上述的 authorName 往往定义成这样更好些：

```
const std::string authorName("Scott Meyers");
```

第二个值得注意的是 class 专属常量。为了将常量的作用域（scope）限制于 class 内，你必须让它成为 class 的一个成员（member）；而为确保此常量至多只有一份实体，你必须让它成为一个 static 成员：

```
class GamePlayer {
private:
    static const int NumTurns = 5;      //常量声明式
    int scores[NumTurns];                //使用该常量
    ...
};
```

然而你所看到的是 NumTurns 的声明式而非定义式。通常 C++ 要求你对你所使用的任何东西提供一个定义式，但如果它是个 class 专属常量又是 static 且为整数类型（integral type，例如 ints, chars, bools），则需特殊处理。只要不取它们的地址，你可以声明并使用它们而无须提供定义式。但如果你取某个 class 专属常量的地址，或纵使你不取其地址而你的编译器却（不正确地）坚持要看到一个定义式，你就必须另外提供定义式如下：

```
const int GamePlayer::NumTurns;  //NumTurns 的定义；
                                 //下面告诉你为什么没有给予数值
```

请把这个式子放进一个实现文件而非头文件。由于 class 常量已在声明时获得初值（例如先前声明 NumTurns 时为它设初值 5），因此定义时不可以再设初值。

顺带一提，请注意，我们无法利用#define 创建一个 class 专属常量，因为 #defines 并不重视作用域（scope）。一旦宏被定义，它就在其后的编译过程中有

效（除非在某处被#undef）。这意味#defines 不仅不能够用来定义 class 专属常量，也不能够提供任何封装性，也就是说没有所谓 private #define 这样的东西。而当然 const 成员变量是可以被封装的，NumTurns 就是。

旧式编译器也许不支持上述语法，它们不允许 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。如果你的编译器不支持上述语法，你可以将初值放在定义式：

```
class CostEstimate {
private:
   static const double FudgeFactor;   //static class 常量声明
   ...                                 //位于头文件内
};
const double                           //static class 常量定义
   CostEstimate::FudgeFactor = 1.35;   //位于实现文件内
```

这几乎是你在任何时候唯一需要做的事。唯一例外是当你在 class 编译期间需要一个 class 常量值，例如在上述的 GamePlayer::scores 的数组声明式中（是的，编译器坚持必须在编译期间知道数组的大小）。这时候万一你的编译器（错误地）不允许“static 整数型 class 常量”完成“in class 初值设定”，可改用所谓的 "the enum hack" 补偿做法。其理论基础是：“一个属于枚举类型（enumerated type）的数值可权充 ints 被使用”，于是 GamePlayer 可定义如下：

```
class GamePlayer {
private:
   enum { NumTurns = 5 };          //"the enum hack" — 令 NumTurns
                                   //成为 5 的一个记号名称.
   int scores[NumTurns];           //这就没问题了.
   ...
};
```

基于数个理由 enum hack 值得我们认识。第一，enum hack 的行为某方面说比较像 #define 而不像 const，有时候这正是你想要的。例如取一个 const 的地址是合法的，但取一个 enum 的地址就不合法，而取一个#define 的地址通常也不合法。

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。（条款 18 对于“通过撰码时的决定实施设计上的约束条件”谈得更多。）此外虽然优秀的编译器不会为“整数型 const 对象”设定另外的存储空间（除非你创建一个 pointer 或 reference 指向该对象），不够优秀的编译器却可能如此，而这可能是你不想要的。Enums 和#defines 一样绝不会导致非必要的内存分配。

认识 enum hack 的第二个理由纯粹是为了实用主义。许多代码用了它，所以看到它时你必须认识它。事实上 "enum hack" 是 template metaprogramming（模板元编程，见条款 48）的基础技术。

把焦点拉回预处理器。另一个常见的#define 误用情况是以它实现宏（macros）。宏看起来像函数，但不会招致函数调用（function call）带来的额外开销。下面这个宏夹带着宏实参，调用函数 f：

```
//以 a 和 b 的较大值调用 f
#define CALL_WITH_MAX(a, b) f((a) > (b) ? (a) : (b))
```

这般长相的宏有着太多缺点，光是想到它们就让人痛苦不堪。

无论何时当你写出这种宏，你必须记住为宏中的所有实参加上小括号，否则某些人在表达式中调用这个宏时可能会遭遇麻烦。但纵使你为所有实参加上小括号，看看下面不可思议的事情：

```
int a = 5, b = 0;
CALL_WITH_MAX(++a, b);          //a 被累加二次
CALL_WITH_MAX(++a, b+10);       //a 被累加一次
```

在这里，调用 f 之前，a 的递增次数竟然取决于“它被拿来和谁比较”！

幸运的是你不需要对这种无聊事情提供温床。你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（type safety）——只要你写出 template inline 函数（见条款 30）：

```
template<typename T>                                    //由于我们不知道
inline void callWithMax(const T& a, const T& b)        //T 是什么，所以采用
{                                                       //pass by reference-to-const.
   f(a > b ? a : b);                                    //见条款 20.
}
```

这个 template 产出一整群函数，每个函数都接受两个同型对象，并以其中较大

者调用 f。这里不需要在函数本体中为参数加上括号，也不需要操心参数被核算（求值）多次……等等。此外由于 callWithMax 是个真正的函数，它遵守作用域（scope）和访问规则。例如你绝对可以写出一个“class 内的 private inline 函数”。一般而言宏无法完成此事。

有了 consts、enums 和 inlines，我们对预处理器（特别是#define）的需求降低了，但并非完全消除。#include 仍然是必需品，而#ifdef/#ifndef 也继续扮演控制编译的重要角色。目前还不到预处理器全面引退的时候，但你应该明确地给予它更长更频繁的假期。

请记住

- 对于单纯常量，最好以 const 对象或 enums 替换#defines。
- 对于形似函数的宏（macros），最好改用 inline 函数替换#defines。

条款 03：尽可能使用 const

Use const whenever possible.

const 的一件奇妙事情是，它允许你指定一个语义约束（也就是指定一个“不该被改动”的对象），而编译器会强制实施这项约束。它允许你告诉编译器和其他程序员某值应该保持不变。只要这（某值保持不变）是事实，你就该确实说出来，因为说出来可以获得编译器的襄助，确保这条约束不被违反。

关键字 const 多才多艺。你可以用它在 classes 外部修饰 global 或 namespace（见条款 2）作用域中的常量，或修饰文件、函数、或区块作用域（block scope）中被声明为 static 的对象。你也可以用它修饰 classes 内部的 static 和 non-static 成员变量。面对指针，你也可以指出指针自身、指针所指物，或两者都（或都不）是 const：

```
char greeting[] = "Hello";
char* p = greeting;                    //non-const pointer, non-const data
const char* p = greeting;              //non-const pointer, const data
char* const p = greeting;              //const pointer, non-const data
const char* const p = greeting;        //const pointer, const data
```

const 语法虽然变化多端，但并不莫测高深。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。

如果被指物是常量，有些程序员会将关键字 const 写在类型之前，有些人会把它写在类型之后、星号之前。两种写法的意义相同，所以下列两个函数接受的参数类型是一样的：

```
void f1(const Widget* pw);          //f1 获得一个指针，指向一个
                                    //常量的（不变的）Widget 对象.
void f2(Widget const * pw);         //f2 也是
```

两种形式都有人用，你应该试着习惯它们。

STL 迭代器系以指针为根据塑模出来，所以迭代器的作用就像个 T* 指针。声明迭代器为 const 就像声明指针为 const 一样（即声明一个 T* const 指针），表示这个迭代器不得指向不同的东西，但它所指的东西的值是可以改动的。如果你希望迭代器所指的东西不可被改动（即希望 STL 模拟一个 const T* 指针），你需要的是 const_iterator：

```
std::vector<int> vec;
...
const std::vector<int>::iterator iter =     //iter 的作用像个 T* const
   vec.begin( );
*iter = 10;                                 //没问题，改变 iter 所指物
++iter;                                     //错误！ iter 是 const
std::vector<int>::const_iterator cIter =    //cIter 的作用像个 const T*
   vec.begin( );
*cIter = 10;                                //错误! *cIter 是 const
++cIter;                                    //没问题，改变 cIter。
```

const 最具威力的用法是面对函数声明时的应用。在一个函数声明式内，const 可以和函数返回值、各参数、函数自身（如果是成员函数）产生关联。

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，而又不至于放弃安全性和高效性。举个例子，考虑有理数（rational numbers，详见条款 24）的 operator* 声明式：

```
class Rational { ... };
const Rational operator* (const Rational& lhs, const Rational& rhs);
```

许多程序员第一次看到这个声明时不免斜着眼睛说，唔，为什么返回一个 const 对象？原因是如果不这样客户就能实现这样的暴行：

```
Rational a, b, c;
...
(a * b) = c;            //在 a * b 的成果上调用 operator=
```

我不知道为什么会有人想对两个数值的乘积再做一次赋值（assignment），但我知道许多程序员会在无意识中那么做，只因为单纯的打字错误（以及一个可被隐式转换为 bool 的类型）：

```
if (a * b = c) ...      //喔欧，其实是想做一个比较（comparison）动作！
```

如果 a 和 b 都是内置类型，这样的代码直截了当就是不合法。而一个“良好的用户自定义类型”的特征是它们避免无端地与内置类型不兼容（见条款 18），因此允许对两值乘积做赋值动作也就没什么意思了。将 operator* 的回传值声明为 const 可以预防那个“没意思的赋值动作”，这就是该那么做的原因。

至于 const 参数，没有什么特别新颖的观念，它们不过就像 local const 对象一样，你应该在必要使用它们的时候使用它们。除非你有需要改动参数或 local 对象，否则请将它们声明为 const。只不过多打 6 个字符，却可以省下恼人的错误，像是“想要键入 '==' 却意外键成 '=' ”的错误，一如稍早所述。

const 成员函数

将 const 实施于成员函数的目的，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，基于两个理由。第一，它们使 class 接口比较容易被理解。这是因为，得知哪个函数可以改动对象内容而哪个函数不行，很是重要。第二，它们使“操作 const 对象”成为可能。这对编写高效代码是个关键，因为如条款 20 所言，改善 C++ 程序效率的一个根本办法是以 ***pass by reference-to-const*** 方式传递对象，而此技术可行的前提是，我们有 const 成员函数可用来处理取得（并经修饰而成）的 const 对象。

许多人漠视一件事实：两个成员函数如果只是常量性（constness）不同，可以被重载。这实在是一个重要的 C++ 特性。考虑以下 class，用来表现一大块文字：

```
class TextBlock {
public:
  ...
  const char& operator[](std::size_t position) const //operator[] for
  { return text[position]; }                         //const 对象.
  char& operator[](std::size_t position)             //operator[] for
  { return text[position]; }                         //non-const 对象.
private:
  std::string text;
};
```

TextBlock 的 operator[]s 可被这么使用：

```
TextBlock tb("Hello");
std::cout << tb[0];              //调用 non-const TextBlock::operator[]

const TextBlock ctb("World");
std::cout << ctb[0];             //调用 const TextBlock::operator[]
```

附带一提，真实程序中 const 对象大多用于 ***passed by pointer-to-const*** 或 ***passed by reference-to-const*** 的传递结果。上述的 ctb 例子太过造作，下面这个比较真实：

```
void print(const TextBlock& ctb) //此函数中 ctb 是 const
{
   std::cout << ctb[0];          //调用 const TextBlock::operator[]
   ...
}
```

只要重载 operator[]并对不同的版本给予不同的返回类型，就可以令 const 和 non-const TextBlocks 获得不同的处理：

```
std::cout << tb[0];    //没问题 — 读一个 non-const TextBlock
tb[0] = 'x';           //没问题 — 写一个 non-const TextBlock
std::cout << ctb[0];   //没问题 — 读一个 const TextBlock
ctb[0] = 'x';          //错误！ — 写一个 const TextBlock
```

注意，上述错误只因 operator[] 的返回类型以致，至于 operator[] 调用动作自身没问题。错误起因于企图对一个“由 const 版之 operator[] 返回” 的 const char& 施行赋值动作。

也请注意，non-const operator[] 的返回类型是个 *reference to* char，不是 char。如果 operator[] 只是返回一个 char，下面这样的句子就无法通过编译：

```
tb[0] = 'x';
```

那是因为，如果函数的返回类型是个内置类型，那么改动函数返回值从来就不合法。纵使合法，C++以 *by value* 返回对象这一事实（见条款 20）意味被改动的其实是 tb.text[0]的一个副本，不是 tb.text[0]自身，那不会是你想要的行为。

让我们为哲学思辨喊一次暂停。成员函数如果是 const 意味什么？这有两个流行概念：bitwise constness（又称 physical constness）和 logical constness。

bitwise const 阵营的人相信，成员函数只有在不更改对象之任何成员变量（static 除外）时才可以说是 const。也就是说它不更改对象内的任何一个 bit。这种论点的好处是很容易侦测违反点：编译器只需寻找成员变量的赋值动作即可。bitwise constness 正是 C++ 对常量性（constness）的定义，因此 const 成员函数不可以更改对象内任何 non-static 成员变量。

不幸的是许多成员函数虽然不十足具备 const 性质却能通过 bitwise 测试。更具体地说，一个更改了“指针所指物”的成员函数虽然不能算是 const，但如果只有指针（而非其所指物）隶属于对象，那么称此函数为 bitwise const 不会引发编译器异议。这导致反直观结果。假设我们有一个 TextBlock-like class，它将数据存储为 char* 而不是 string，因为它需要和一个不认识 string 对象的 C API 沟通：

```
class CTextBlock {
public:
   ...
   char& operator[](std::size_t position) const // bitwise const 声明,
   { return pText[position]; }                  // 但其实不适当.
private:
   char* pText;
};
```

这个 class 不适当地将其 operator[] 声明为 const 成员函数，而该函数却返回一个 reference 指向对象内部值（条款 28 对此有深刻讨论）。假设暂时不管这个

事实，请注意，operator[]实现代码并不更改 pText。于是编译器很开心地为operator[]产出目标码。它是 bitwise const，所有编译器都这么认定。但是看看它允许发生什么事：

```
const CTextBlock cctb("Hello");  //声明一个常量对象。
char* pc = &cctb[0];             //调用 const operator[]取得一个指针，
                                 //  指向 cctb 的数据。
*pc = 'J';                       //cctb 现在有了 "Jello" 这样的内容。
```

这其中当然不该有任何错误：你创建一个常量对象并设以某值，而且只对它调用const 成员函数。但你终究还是改变了它的值。

这种情况导出所谓的 logical constness。这一派拥护者主张，一个 const 成员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如此。例如你的 CTextBlock class 有可能高速缓存（cache）文本区块的长度以便应付询问：

```
class CTextBlock {
public:
   ...
   std::size_t length() const;
private:
   char* pText;
   std::size_t textLength;        //最近一次计算的文本区块长度。
   bool lengthIsValid;            //目前的长度是否有效。
};
std::size_t CTextBlock::length() const
{
   if (!lengthIsValid) {
        textLength = std::strlen(pText);   //错误！在 const 成员函数内
        lengthIsValid = true;              //  不能赋值给 textLength
   }                                       //  和 lengthIsValid。
   return textLength;
}
```

length 的实现当然不是 bitwise const，因为 textLength 和 lengthIsValid 都可能被修改。这两笔数据被修改对 const CTextBlock 对象而言虽然可接受，但编译器不同意。它们坚持 bitwise constness。怎么办？

解决办法很简单：利用 C++ 的一个与 const 相关的摆动场：mutable（可变的）。mutable 释放掉 non-static 成员变量的 bitwise constness 约束：

```
class CTextBlock {
public:
   ...
   std::size_t length() const;
private:
   char* pText;
   mutable std::size_t textLength;          //这些成员变量可能总是
   mutable bool lengthIsValid;              //会被更改，即使在
};                                          //const 成员函数内。
std::size_t CTextBlock::length() const
{
   if (!lengthIsValid) {
        textLength = std::strlen(pText);    //现在，可以这样，
        lengthIsValid = true;               //也可以这样。
   }
   return textLength;
}
```

在 const 和 non-const 成员函数中避免重复

对于“bitwise-constness 非我所欲”的问题，mutable 是个解决办法，但它不能解决所有的 const 相关难题。举个例子，假设 TextBlock（和 CTextBlock）内的 operator[] 不单只是返回一个 reference 指向某字符，也执行边界检验（bounds checking）、志记访问信息（logged access info.）、甚至可能进行数据完善性检验。把所有这些同时放进 const 和 non-const operator[] 中，导致这样的怪物（暂且不管那将会成为一个“长度颇为可议”的隐喻式 inline 函数——见条款 30）：

```
class TextBlock {
public:
   ...
   const char& operator[](std::size_t position) const
   {
        ...              //边界检验（bounds checking）
        ...              //志记数据访问（log access data）
        ...              //检验数据完整性（verify data integrity）
        return text[position];
   }
   char& operator[](std::size_t position)
   {
        ...              //边界检验（bounds checking）
        ...              //志记数据访问（log access data）
        ...              //检验数据完整性（verify data integrity）
        return text[position];
   }
private:
   std::string text;
};
```

哎哟！你能说出其中发生的代码重复以及伴随的编译时间、维护、代码膨胀等令人头痛的问题吗？当然啦，将边界检验……等所有代码移到另一个成员函数（往往是个 private）并令两个版本的 operator[] 调用它，是可能的，但你还是重复了一些代码，例如函数调用、两次 return 语句等等。

你真正该做的是实现 operator[]的机能一次并使用它两次。也就是说，你必须令其中一个调用另一个。这促使我们将常量性转除（casting away constness）。

就一般守则而言，转型（casting）是一个糟糕的想法，我将贡献一整个条款来谈这码事（条款 27），告诉你不要那么做。然而代码重复也不是什么令人愉快的经验。本例中 const operator[]完全做掉了 non-const 版本该做的一切，唯一的不同是其返回类型多了一个 const 资格修饰。这种情况下如果将返回值的 const 转除是安全的，因为不论谁调用 non-const operator[]都一定首先有个 non-const 对象，否则就不能够调用 non-const 函数。所以令 non-const operator[]调用其 const 兄弟是一个避免代码重复的安全做法——即使过程中需要一个转型动作。下面是代码，稍后有更详细的解释：

```
class TextBlock {
public:
  ...
  const char& operator[](std::size_t position) const //一如既往
  {
    ...
    ...
    ...
    return text[position];
  }
  char& operator[](std::size_t position)    //现在只调用 const op[]
  {
    return
      const_cast<char&>(                    //将 op[]返回值的 const 转除
        static_cast<const TextBlock&>(*this)     //为*this 加上 const
          [position]                               //调用 const op[]
      );
  }
...
};
```

如你所见，这份代码有两个转型动作，而不是一个。我们打算让 non-const operator[]调用其 const 兄弟，但 non-const operator[]内部若只是单纯调用 operator[]，会递归调用自己。那会大概……唔……进行一百万次。为了避免无穷递归，我们必须明确指出调用的是 const operator[]，但 C++ 缺乏直接的语法可以那么做。因此这里将 *this 从其原始类型 TextBlock& 转型为 const TextBlock&。是的，我们使用转型操作为它加上 const！所以这里共有两次转型：第一次用来为 *this 添加 const（这使接下来调用 operator[]时得以调用 const 版本），第二次则是从 const operator[]的返回值中移除 const。

添加 const 的那一次转型强迫进行了一次安全转型（将 non-const 对象转为 const 对象），所以我们使用 static_cast。移除 const 的那个动作只可以藉由 const_cast 完成，没有其他选择（就技术而言其实是有的；一个 C-style 转型也行得通，但一如我在条款 27 所说，那种转型很少是正确的抉择。如果你不熟悉 static_cast 或 const_cast，条款 27 提供了一份概要）。

至于其他动作，由于本例调用的是操作符，所以语法有一点点奇特，恐怕无法赢得选美大赛，但却有我们渴望的“避免代码重复”效果，因为它运用 const operator[] 实现出 non-const 版本。为了到达那个目标而写出如此难看的语法是否值得，只有你能决定，但“运用 const 成员函数实现出其 non-const 孪生兄弟”的技术是值得了解的。

更值得了解的是，反向做法——令 const 版本调用 non-const 版本以避免重复——并不是你该做的事。记住，const 成员函数承诺绝不改变其对象的逻辑状态（logical state），non-const 成员函数却没有这般承诺。如果在 const 函数内调用 non-const 函数，就是冒了这样的风险：你曾经承诺不改动的那个对象被改动了。这就是为什么“const 成员函数调用 non-const 成员函数”是一种错误行为：因为对象有可能因此被改动。实际上若要令这样的代码通过编译，你必须使用一个 const_cast 将*this 身上的 const 性质解放掉，这是乌云罩顶的清晰前兆。反向调用（也就是我们先前使用的那个）才是安全的：non-const 成员函数本来就可以对其对象做任何动作，所以在其中调用一个 const 成员函数并不会带来风险。这就是为什么本例以 static_cast 作用于*this 的原因：这里并不存在 const 相关危险。

本条款一开始就提醒你，const 是个奇妙且非比寻常的东西。在指针和迭代器身上；在指针、迭代器及 references 指涉的对象身上；在函数参数和返回类型身上；

在 local 变量身上；在成员函数身上，林林总总不一而足。const 是个威力强大的助手。尽可能使用它。你会对你的作为感到高兴。

请记住

- 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

Make sure that objects are initialized before they're used.

关于“将对象初始化”这事，C++ 似乎反复无常。如果你这么写：

```
int x;
```

在某些语境下 x 保证被初始化（为 0），但在其他语境中却不保证。如果你这么写：

```
class Point {
   int x, y;
};
...
Point p;
```

p 的成员变量有时候被初始化（为 0），有时候不会。如果你来自其他语言阵营而那儿并不存在“无初值对象”，那么请小心，因为这颇为重要。

读取未初始化的值会导致不明确的行为。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。更可能的情况是读入一些“半随机”bits，污染了正在进行读取动作的那个对象，最终导致不可测知的程序行为，以及许多令人不愉快的调试过程。

现在，我们终于有了一些规则，描述“对象的初始化动作何时一定发生，何时不一定发生”。不幸的是这些规则很复杂，我认为对记忆力而言是太繁复了些。

通常如果你使用 C part of C++（见条款 1）而且初始化可能招致运行期成本，那么就不保证发生初始化。一旦进入 non-C parts of C++，规则有些变化。这就很好地解释了为什么 array（来自 C part of C++）不保证其内容被初始化，而 vector（来自 STL part of C++）却有此保证。

表面上这似乎是个无法决定的状态，而最佳处理办法就是：永远在使用对象之前先将它初始化。对于无任何成员的内置类型，你必须手工完成此事。例如：

```
int x = 0;                                          //对 int 进行手工初始化
const char* text = "A C-style string";              //对指针进行手工初始化
                                                    //（亦见条款 3）
double d;
std::cin >> d;                          //以读取 input stream 的方式完成初始化.
```

至于内置类型以外的任何其他东西，初始化责任落在构造函数（constructors）身上。规则很简单：确保每一个构造函数都将对象的每一个成员初始化。

这个规则很容易奉行，重要的是别混淆了赋值（assignment）和初始化（initialization）。考虑一个用来表现通讯簿的 class，其构造函数如下：

```
class PhoneNumber { ... };
class ABEntry {                 //ABEntry = "Address Book Entry"
public:
   ABEntry(const std::string& name, const std::string& address,
           const std::list<PhoneNumber>& phones);
private:
   std::string theName;
   std::string theAddress;
   std::list<PhoneNumber> thePhones;
   int numTimesConsulted;
};
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;
}
```

这会导致 ABEntry 对象带有你期望（你指定）的值，但不是最佳做法。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。在 ABEntry 构造函数内，theName, theAddress 和 thePhones 都不是被初始化，而是被赋值。初始化的发生时间更早，发生于这些成员的 *default* 构造函数被自动调用之时（比进入 ABEntry 构造函数本体的时间更早）。但这对 numTimesConsulted 不为真，因为它属于内置类型，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。

ABEntry 构造函数的一个较佳写法是，使用所谓的 member initialization list（成员初值列）替换赋值动作：

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

这个构造函数和上一个的最终结果相同，但通常效率较高。基于赋值的那个版本(本例第一版本)首先调用 *default* 构造函数为 theName, theAddress 和 thePhones 设初值，然后立刻再对它们赋予新值。*default* 构造函数的一切作为因此浪费了。成员初值列（member initialization list）的做法（本例第二版本）避免了这一问题，因为初值列中针对各个成员变量而设的实参，被拿去作为各成员变量之构造函数的实参。本例中的 theName 以 name 为初值进行 *copy* 构造，theAddress 以 address 为初值进行 *copy* 构造，thePhones 以 phones 为初值进行 *copy* 构造。

对大多数类型而言，比起先调用 *default* 构造函数然后再调用 *copy assignment* 操作符，单只调用一次 *copy* 构造函数是比较高效的，有时甚至高效得多。对于内置型对象如 numTimesConsulted，其初始化和赋值的成本相同，但为了一致性最好也通过成员初值列来初始化。同样道理，甚至当你想要 *default* 构造一个成员变量，你都可以使用成员初值列，只要指定无物（nothing）作为初始化实参即可。假设 ABEntry 有一个无参数构造函数，我们可将它实现如下：

```
ABEntry::ABEntry( )
   :theName(),                   //调用 theName 的 default 构造函数;
    theAddress(),                //为 theAddress 做类似动作;
    thePhones(),                 //为 thePhones 做类似动作;
    numTimesConsulted(0)         //记得将 numTimesConsulted 显式初始化为 0
{ }
```

由于编译器会为用户自定义类型（user-defined types）之成员变量自动调用 *default* 构造函数——如果那些成员变量在“成员初值列”中没有被指定初值的话，因而引发某些程序员过度夸张地采用以上写法。那是可理解的，但请立下一个规则，规定总是在初值列中列出所有成员变量，以免还得记住哪些成员变量（如果它们在初值列中被遗漏的话）可以无需初值。举个例子，由于 numTimesConsulted 属于内置类型，如果成员初值列（member initialization list）遗漏了它，它就没有初值，因而可能开启“不明确行为”的潘多拉盒子。

有些情况下即使面对的成员变量属于内置类型（那么其初始化与赋值的成本相同），也一定得使用初值列。是的，如果成员变量是 const 或 references，它们就一定需要初值，不能被赋值（见条款 5）。为避免需要记住成员变量何时必须在成员初值列中初始化，何时不需要，最简单的做法就是：总是使用成员初值列。这样做有时候绝对必要，且又往往比赋值更高效。

许多 classes 拥有多个构造函数，每个构造函数有自己的成员初值列。如果这种 classes 存在许多成员变量和/或 base classes，多份成员初值列的存在就会导致不受欢迎的重复（在初值列内）和无聊的工作（对程序员而言）。这种情况下可以合理地在初值列中遗漏那些“赋值表现像初始化一样好”的成员变量，改用它们的赋值操作，并将那些赋值操作移往某个函数（通常是 private），供所有构造函数调用。这种做法在“成员变量的初值系由文件或数据库读入”时特别有用。然而，比起经由赋值操作完成的“伪初始化”（pseudo-initialization），通过成员初值列（member initialization list）完成的“真正初始化”通常更加可取。

C++ 有着十分固定的“成员初始化次序”。是的，次序总是相同：base classes 更早于其 derived classes 被初始化（见条款 12），而 class 的成员变量总是以其声明次序被初始化。回头看看 ABEntry，其 theName 成员永远最先被初始化，然后是 theAddress，再来是 thePhones，最后是 numTimesConsulted。即使它们在成员初值列中以不同的次序出现（很不幸那是合法的），也不会有任何影响。为避免你或你的检阅者迷惑，并避免某些可能存在的晦涩错误，当你在成员初值列中条列各个成员时，最好总是以其声明次序为次序。

> 译注：上述所谓晦涩错误，指的是两个成员变量的初始化带有次序性。例如初始化 array 时需要指定大小，因此代表大小的那个成员变量必须先有初值。

一旦你已经很小心地将“内置型成员变量”明确地加以初始化，而且也确保你的构造函数运用“成员初值列”初始化 base classes 和成员变量，那就只剩唯一一

件事需要操心，就是……呃……深呼吸……“不同编译单元内定义之 non-local static 对象”的初始化次序。

让我们一点一点地探钻这一长串词组。

所谓 static 对象，其寿命从被构造出来直到程序结束为止，因此 stack 和 heap-based 对象都被排除。这种对象包括 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 作用域内被声明为 static 的对象。函数内的 static 对象称为 local static 对象（因为它们对函数而言是 local），其他 static 对象称为 non-local static 对象。程序结束时 static 对象会被自动销毁，也就是它们的析构函数会在 main() 结束时被自动调用。

所谓编译单元（translation unit）是指产出单一目标文件（single object file）的那些源码。基本上它是单一源码文件加上其所含入的头文件（#include files）。

现在，我们关心的问题涉及至少两个源码文件，每一个内含至少一个 non-local static 对象（也就是说该对象是 global 或位于 namespace 作用域内，抑或在 class 内或 file 作用域内被声明为 static）。真正的问题是：如果某编译单元内的某个 non-local static 对象的初始化动作使用了另一编译单元内的某个 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。

实例可以帮助理解。假设你有一个 FileSystem class，它让互联网上的文件看起来好像位于本机（local）。由于这个 class 使世界看起来像个单一文件系统，你可能会产出一个特殊对象，位于 global 或 namespace 作用域内，象征单一文件系统：

```
class FileSystem {                          //来自你的程序库
public:
   ...
   std::size_t numDisks() const;            //众多成员函数之一
   ...
};
extern FileSystem tfs;                      //预备给客户使用的对象;
                                            //tfs 代表 "the file system"
```

FileSystem 对象绝不是一个稀松平常无关痛痒的（trivial）对象，因此你的客户如果在 theFileSystem 对象构造完成前就使用它，会得到惨重的灾情。

现在假设某些客户建立了一个 class 用以处理文件系统内的目录（directories）。很自然他们的 class 会用上 theFileSystem 对象：

```
class Directory {                          //由程序库客户建立
public:
   Directory( params );
   ...
};
Directory::Directory( params )
{
   ...
   std::size_t disks = tfs.numDisks();     //使用 tfs 对象
   ...
}
```

进一步假设，这些客户决定创建一个 Directory 对象，用来放置临时文件：

```
Directory tempDir( params );               //为临时文件而做出的目录
```

现在，初始化次序的重要性显现出来了：除非 tfs 在 tempDir 之前先被初始化，否则 tempDir 的构造函数会用到尚未初始化的 tfs。但 tfs 和 tempDir 是不同的人在不同的时间于不同的源码文件建立起来的，它们是定义于不同编译单元内的 non-local static 对象。如何能够确定 tfs 会在 tempDir 之前先被初始化？

喔，你无法确定。再说一次，C++ 对“定义于不同的编译单元内的 non-local static 对象”的初始化相对次序并无明确定义。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便可完全消除这个问题。唯一需要做的是：将每个 non-local static 对象搬到自己的专属函数内（该对象在此函数内被声明为 static）。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。Design Patterns 迷哥迷姊们想必认出来了，这是 ***Singleton*** 模式的一个常见实现手法。

这个手法的基础在于：C++ 保证，函数内的 local static 对象会在“该函数被调用期间”“首次遇上该对象之定义式”时被初始化。所以如果你以“函数调用”（返回一个 reference 指向 local static 对象）替换“直接访问 non-local static 对象”，你

就获得了保证，保证你所获得的那个 reference 将指向一个历经初始化的对象。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；真正的 non-local static 对象可没这等便宜！

以此技术施行于 tfs 和 tempDir 身上，结果如下：

```
class FileSystem { ... };          //同前
FileSystem& tfs()                  //这个函数用来替换 tfs 对象；它在
{                                  //FileSystem class 中可能是个 static。
   static FileSystem fs;           //定义并初始化一个 local static 对象，
   return fs;                      //返回一个 reference 指向上述对象。
}
class Directory { ... };           //同前
Directory::Directory( params )     //同前，但原本的 reference to tfs
{                                  //现在改为 tfs()
   ...
   std::size_t disks = tfs().numDisks( );
   ...
}
Directory& tempDir()               //这个函数用来替换 tempDir 对象；
{                                  //它在 Directory class 中可能是个 static。
   static Directory td;            //定义并初始化 local static 对象，
   return td;                      //返回一个 reference 指向上述对象。
}
```

这么修改之后，这个系统程序的客户完全像以前一样地用它，唯一不同的是他们现在使用 tfs()和 tempDir()而不再是 tfs 和 tempDir。也就是说他们使用函数返回的“指向 static 对象”的 references，而不再使用 static 对象自身。

这种结构下的 reference-returning 函数往往十分单纯：第一行定义并初始化一个 local static 对象，第二行返回它。这样的单纯性使它们成为绝佳的 inlining 候选人，尤其如果它们被频繁调用的话（见条款 30）。但是从另一个角度看，这些函数“内含 static 对象”的事实使它们在多线程系统中带有不确定性。再说一次，任何一种 non-const static 对象，不论它是 local 或 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，这可消除与初始化有关的“竞速形势（race conditions）”。

当然啦，运用 reference-returning 函数防止“初始化次序问题”，前提是其中

有着一个对对象而言合理的初始化次序。如果你有一个系统，其中对象 A 必须在对象 B 之前先初始化，但 A 的初始化能否成功却又受制于 B 是否已初始化，这时候你就有麻烦了。坦白说你自作自受。只要避开如此病态的境况，此处描述的办法应该可以提供你良好的服务，至少在单线程程序中。

既然这样，为避免在对象初始化之前过早地使用它们，你需要做三件事。第一，手工初始化内置型 non-member 对象。第二，使用成员初值列（member initialization lists）对付对象的所有成分。最后，在“初始化次序不确定性”（这对不同编译单元所定义的 non-local static 对象是一种折磨）氛围下加强你的设计。

请记住

- 为内置型对象进行手工初始化，因为 C++不保证初始化它们。
- 构造函数最好使用成员初值列（member initialization list），而不要在构造函数本体内使用赋值操作（assignment）。初值列列出的成员变量，其排列次序应该和它们在 class 中的声明次序相同。
- 为免除“跨编译单元之初始化次序”问题，请以 local static 对象替换 non-local static 对象。