目录

Contents

0

导读

Introduction

学习程序语言根本大法是一回事；学习如何以某种语言设计并实现高效程序则是另一回事。这种说法对 C++ 尤其适用，因为 C++ 以拥有罕见的威力和丰富的表达能力为傲。只要适当使用，C++ 可以成为工作上的欢愉伙伴。巨大而变化多端的设计可以被直接表现出来，并且被有效实现出来。一组明智选择并精心设计的 classes, functions 和 templates 可使程序编写容易、直观、高效、并且远离错误。如果你知道怎么做，写出有效的 C++ 程序并不太困难。然而如果没有良好培训，C++ 可能会导致你的代码难以理解、不易维护、不易扩充、效率低下又错误连连。

本书的目的是告诉你如何有效运用 C++。我假设你已经知道 C++ 是个语言并且已经对它有某些使用经验。这里提供的是这个语言的使用导引，使你的软件易理解、易维护、可移植、可扩充、高效、并且有着你所预期的行为。

我所提出的忠告大致分为两类：一般性的设计策略，以及带有具体细节的特定语言特性。设计上的讨论集中于“如何在两个不同做法中择一完成某项任务”。你该选择 inheritance（继承）还是 templates（模板）？该选择 public 继承还是 private 继承？该选择 private 继承还是 composition（复合）？该选择 member 函数还是 non-member 函数？该选择 *pass-by-value* 还是 *pass-by-reference*？在这些选择点上做出正确决定很重要，因为一个不良的决定有可能不至于很快带来影响，却在发展后期才显现恶果，那时候再来矫正往往既困难又耗时间，而且代价昂贵。

即使你完全知道该做什么，完全进入正轨还是可能有点棘手。什么是 *assignment* 操作符的适当返回类型(return type)？何时该令析构函数为 virtual？当 `operator new`

无法找到足够内存时该如何行事？榨出这些细节很是重要，因为如果疏忽而不那么做，几乎总是导致未可预期的、也许神秘难解的程序行为。本书将帮助你趋吉避凶。

这并不是一本范围广泛的 C++ 参考书。这是一份 55 个特定建议（我称之为条款）的集合，谈论如何强化你的程序和设计。每个条款有相当程度的独立性，但大多数也参考其他条款。因此阅读本书的一个方式是，从你感兴趣的条款开始，然后看它逐步把你带往何方。

本书也不是一本 C++ 入门书籍。例如在第 2 章中我热切告诉你实现构造函数（constructors）、析构函数（destructors）和赋值操作符（assignment operators）的一切种种，但我假设你已经知道或有能力在其他地方学得这些函数的功能以及它们如何声明。市面上有许多 C++ 书籍内含这类信息。

本书目的是要强调那些常常被漠视的 C++ 编程方向与观点。其他书籍描述 C++ 语言的各个成分，本书告诉你如何结合那些成分以便最终获得有效程序。其他书籍告诉你如何让程序通过编译，本书告诉你如何回避编译器难以显露的问题。

在此同时，本书将范围限制在标准 C++ 上头。书内只会出现官方规范上所列的特性。本书十分重视移植性，所以如果你想找一些与平台相依的秘诀和窍门，这里没有。

另一个你不会在本书发现的是 C++ 福音书——走向完美 C++ 软件的唯一真理之路。本书每个条款都提供引导，告诉我们如何发展出更好的设计，如何避免常见的问题，或是如何达到更高的效率，但没有任何一个条款放之四海皆准、一体适用。软件设计和实现是复杂的差使，被硬件、操作系统、应用程序的约束条件涂上五颜六色，所以我能做的最好的就是提供指南，让你得以创造出更棒的程序。

如果任何时间你都遵循每一条准则，不太可能掉入 C++ 最常见的陷阱中。但是所谓准则天生就带有例外。这就是为什么每个条款都有解释与说明。这些解释与说明是本书最重要的一部分。惟有了解条款背后的基本原理，你才能够决定是否将它套用于你所开发的软件，并奉行其所昭示的独特约束。

本书的最佳用途就是彻底了解 C++ 如何行为、为什么那样行为，以及如何运用其行为形成优势。盲目应用书中条款是非常不适合的。但如果没有好理由，你或许也不该违反任何一个条款。

术语（Terminology）

下面是每一位程序员都应该了解的一份小小的 C++ 词汇。其中的术语十分重要，我们必须确认彼此都同意它们的意义。

所谓声明式（*declaration*）是告诉编译器某个东西的名称和类型（type），但略去细节。下面都是声明式：

```
extern int x;                                  //对象（object）声明式
std::size_t numDigits(int number);             //函数（function）声明式
class Widget;                                  //类（class）声明式

template<typename T>                           //模板（template）声明式
class GraphNode;                               //"typename" 的使用见条款 42
```

注意，我谈到整数 x 时称其为一个对象（object），即使它是个内置类型。某些人把“对象”一词保留给用户自定义类型（user-defined type）的变量，但我并不如此。也请注意，函数 numDigit 的返回类型是 std::size_t，这表示类型 size_t 位于命名空间 std 内。这个命名空间是几乎所有 C++ 标准程序库元素的栖身处。然而 C（正确说法是 C89）标准程序库也适用于 C++，而继承自 C 的符号（例如 size_t）有可能存在于 global 作用域或 std 内，甚或两者兼具，取决于哪个头文件被含入（#included）。本书之中我假设含入的都是 C++ 头文件，这也就是为什么我写 std::size_t 而不只是写 size_t。当我在文本中指称标准程序库内的组件时，往往略去前导的 std::，你得自己认清像 size_t, vector, cout 这类东西都在 std 内。但范例码中我总是会含入 std，因为真实程序编译时不能没有它。

顺带一提，size_t 只是一个 typedef，是 C++ 计算个数（例如 char*-based 字符串内的字符个数或 STL 容器内的元素个数等等）时用的某种不带正负号（unsigned）类型。它也是 vector, deque 和 string 内的 operator[] 函数接受的参数类型。条款 3 阐述当我们定义自己的 operator[] 函数时应该遵循的协议。

每个函数的声明揭示其签名式（*signature*），也就是参数和返回类型。一个函数

的签名等同于该函数的类型。numDigits 函数的签名是 std::size_t (int)，也就是说“这函数获得一个 int 并返回一个 std::size_t”。C++ 对签名式的官方定义并不包括函数的返回类型，不过本书把返回类型视为签名的一部分，这样比较有帮助。

定义式（*definition*）的任务是提供编译器一些声明式所遗漏的细节。对对象而言，定义式是编译器为此对象拨发内存的地点。对 function 或 function template 而言，定义式提供了代码本体。对 class 或 class template 而言，定义式列出它们的成员：

```
int x;                                          //对象的定义式
std::size_t numDigits(int number)               //函数的定义式
{                                               //此函数返回其参数的数字个数，
   std::size_t digitsSoFar = 1;                 //例如十位数返回 2，百位数返回 3.

   while ((number /= 10) != 0) ++digitsSoFar;
   return digitsSoFar;
}
class Widget {                                  //class 的定义式
public:
   Widget();
   ~Widget();
   ...
};
template<typename T>                            //template 的定义式
class GraphNode {
public:
   GraphNode();
   ~GraphNode();
   ...
};
```

初始化（*Initialization*）是“给予对象初值”的过程。对用户自定义类型的对象而言，初始化由构造函数执行。所谓 *default* 构造函数是一个可被调用而不带任何实参者。这样的构造函数要不没有参数，要不就是每个参数都有缺省值：

```
class A {
public:
   A();                                         //default 构造函数
};
class B {
public:
   explicit B(int x = 0, bool b = true);        //default 构造函数;
};                                              //关于 "explicit"，见以下信息
```

```
class C {
public:
   explicit C(int x);                          //不是 default 构造函数
};
```

上述的 classes B 和 C 的构造函数都被声明为 explicit，这可阻止它们被用来执行隐式类型转换（implicit type conversions），但它们仍可被用来进行显式类型转换（explicit type conversions）：

```
void doSomething(B bObject);      //函数，接受一个类型为 B 的对象

B bObj1;                          //一个类型为 B 的对象
doSomething(bObj1);               //没问题，传递一个 B 给 doSomething 函数
B bObj2(28);                      //没问题，根据 int 28 建立一个 B
                                  //（函数的 bool 参数缺省为 true）
doSomething(28);                  //错误! DoSomething 应该接受一个 B，
                                  //不是一个 int，而 int 至 B 之间
                                  //并没有隐式转换。
doSomething(B(28));               //没问题，使用 B 构造函数将 int 显式转换
                                  //（也就是转型, cast）为一个 B 以促成此一调用.
                                  //（条款 27 对转型谈得更多）
```

被声明为 explicit 的构造函数通常比其 non-explici 兄弟更受欢迎，因为它们禁止编译器执行非预期（往往也不被期望）的类型转换。除非我有一个好理由允许构造函数被用于隐式类型转换，否则我会把它声明为 explicit。我鼓励你遵循相同的政策。

请注意我在上述代码中以不同的颜色特别强调转型动作。我以这样的强调方式贯穿全书，让你特别注意值得注意的东西。

copy 构造函数被用来“以同型对象初始化自我对象”，*copy assignment* 操作符被用来“从另一个同型对象中拷贝其值到自我对象”：

```
class Widget {
public:
   Widget();                                  //default 构造函数
   Widget(const Widget& rhs);                 // copy 构造函数
   Widget& operator=(const Widget& rhs);      //copy assignment 操作符
   ...
};
Widget w1;                                    //调用 default 构造函数
Widget w2(w1);                                //调用 copy 构造函数
w1 = w2;                                      //调用 copy assignment 操作符
```

当你看到赋值符号时请小心，因为 "=" 语法也可用来调用 *copy* 构造函数：

```
Widget w3 = w2;                    //调用 copy 构造函数!
```

幸运的是“*copy* 构造”很容易和“*copy* 赋值”有所区别。如果一个新对象被定义（例如以上语句中的 w3），一定会有个构造函数被调用，不可能调用赋值操作。如果没有新对象被定义（例如前述的 "w1 = w2" 语句），就不会有构造函数被调用，那么当然就是赋值操作被调用。

copy 构造函数是一个尤其重要的函数，因为它定义一个对象如何 ***passed by value***（以值传递）。举个例子，考虑以下代码：

```
bool hasAcceptableQuality(Widget w);
...
Widget aWidget;
if (hasAcceptableQuality(aWidget)) ...
```

参数 w 是以 ***by value*** 方式传递给 hasAcceptableQuality，所以在上述调用中 aWidget 被复制到 w 体内。这个复制动作由 Widget 的 *copy* 构造函数完成。***Pass-by-value*** 意味“调用 *copy* 构造函数”。以 ***by value*** 传递用户自定义类型通常是个坏主意，***Pass-by-reference-to-***const 往往是比较好的选择；详见条款 20。

STL 是所谓标准模板库（Standard Template Library），是 C++ 标准程序库的一部分，致力于容器（如 vector, list, set, map 等等）、迭代器（如 vector<int>::iterator, set<string>::iterator 等等）、算法（如 for_each, find, sort 等等）及相关机能。许多相关机能以函数对象（***function objects***）实现，那是“行为像函数”的对象。这样的对象来自于重载 operator()（***function call*** 操作符）的 classes。如果你对 STL 陌生，阅读本书时手边可能需要摆一本最新参考读物，因为 STL 对我太有用了，我不可能不用它。一旦你也用上它，你一定会有相同的感觉。

C++ 程序员如果原先来自诸如 Java 或 C# 语言阵营，可能会对所谓“不明确行为”（***undefined behavior***）感到惊讶。由于各种因素，某些 C++ 构件的行为没有定义：你无法稳定预估运行期会发生什么事。下面两个代码片段就带有“不明确的行为”：

```
int* p = 0;                //p 是个 null 指针
std::cout << *p;           //对一个 null 指针取值（dereferencing）
                           //会导致不明确行为。
```

```
char name[] = "Darla"; //name 是个数组，大小为 6（别忘记最尾端的 null!）
char c = name[10];     //指涉一个无效的数组索引
                       //导致不明确行为。
```

我要特别强调，不明确（未定义）行为的结果是不可预期的，很可能让人不愉快。经验丰富的 C++ 程序员常说，带有不明确行为的程序会抹煞你的辛勤努力。那是真的：一个带有不明确行为的程序会抹煞你的辛勤努力。但不一定如此，更可能的是这样的程序会出现错误行为，有时执行正常，有时造成崩坏，有时更产出不正确的结果。有战斗力的 C++ 程序员都知道尽可能避开不明确行为。我会在书中指出你需要密切注意的若干地方。

对其他语言转换至 C++ 阵营的程序员而言，另一个可能造成困惑的术语是接口（*interface*）。Java 和 .NET 语言都提供 Interfaces 为语言元素，但 C++ 没有，尽管条款 31 讨论了如何近似它。当我使用术语“接口”时，我一般谈的是函数的签名（signature）或 class 的可访问元素（例如我可能会说 class 的“public 接口”或“protected 接口”或“private 接口”），或是针对某 template 类型参数需为有效的一个表达式（见条款 41）。也就是我所说的接口完全是指一般性的设计观念。

所谓客户（*client*）是指某人或某物，他（或它）使用你写的代码（通常是一些接口）。函数的客户是指其使用者，也就是程序中调用函数（或取其地址）的那一部分，也可以说是编写并维护那些代码的人。Class 或 template 的客户则是指程序中使用 class 或 template 的那一部分，也可以说是编写并维护那些代码的人。说到“客户”时通常我指的是程序员，因为程序员可能被迷惑、被误导、或因糟糕的接口而恼怒，他们所写的代码却不会有这种情绪。

或许你不习惯想到客户，但我会花费大量时间试着说服你尽可能让他们的生活轻松些。毕竟你也是其他人所开发的软件的客户。难道你不希望那些人为你把事情弄得更轻松些吗？除此之外，在某个时间点你几乎必然会发现，你就是你自己的客户（也就是使用你自己写的代码），那个时候你就会很高兴你在开发接口时把客户放在心上了。

本书中我常常掩盖 functions 和 function templates 之间的区别，以及 classes 和 class templates 之间的区别。那是因为对其中之一为真者往往对另一方也为真。当不是这种情况的时候，我会区分 classes, functions 及它们所对应的 templates。

当我在程序批注中提到构造函数和析构函数时，有时我会使用缩写字 *ctor* 和 *dtor*。

命名习惯（Naming Conventions）

我尝试挑选有意义的名称用于 objects, classes, functions, templates 等等身上，但某些隐藏于名称背后的意义可能不是那么显而易见，例如我最喜爱的两个参数名称 lhs 和 rhs。它们分别代表 "left-hand side"（左手端）和 "right-hand side"（右手端）。我常常以它们作为二元操作符（binary operators）函数如 operator== 和 operator* 的参数名称。举个例子，如果 a 和 b 表示两个有理数对象，而如果 Rational 对象可被一个 non-member operator* 函数执行乘法（如条款 24 所言），那么下面表达式：

```
a * b
```

等价于以下的函数调用：

```
operator*(a, b)
```

在条款 24 中我声明此一 operator* 如下：

```
const Rational operator* (const Rational& lhs, const Rational& rhs);
```

如你所见，左操作数 a 变成函数内的 lhs，右操作数 b 则变成 rhs。

对于成员函数，左侧实参由 this 指针表现出来，所以有时我单独使用参数名称 rhs。你可能已经在第 5 页的若干 Widget 成员函数声明中注意到了这一点。对了，我经常以 Widget class 示例，"Widget" 并不代表任何东西，它只是当我需要一个示范用的 class 名称时偶尔采用的名称，它和 GUI toolkits 的 widgets 完全无关。

我常将“指向一个 T 型对象”的指针命名为 pt，意思是 "pointer to T"。下面是一些例子：

```
Widget* pw;                    //pw = "ptr to Widget".
class Airplane;
Airplane* pa;                  //pa = "ptr to Airplane".
```

```
class GameCharacter;
GameCharacter* pgc;          //pgc = "ptr to GameCharacter"
```

对于 references 我使用类似习惯：`rw` 可能是个 reference to `Widget`，`ra` 则是个 reference to `Airplane`。

当我讨论成员函数时，偶尔会以 `mf` 为名。

关于线程（Threading Consideration）

作为一个语言，C++ 对线程（threads）没有任何意念——事实上它对任何并发（concurrency）事物都没有意念。C++ 标准程序库也一样。当 C++ 受到全世界关注时多线程（multithreaded）程序还不存在。

但现在它们存在了。本书的焦点放在标准可移植的 C++，但我不能忽略一个事实：线程安全性（thread safety）是许多程序员面对的主题。我对“标准 C++ 和真实世界之间的这个缺口”的处理方式是，如果我所检验的 C++ 构件在多线程环境中有可能引发问题，就把它指出来。这远远无法构成一本 C++多线程编程专著，却能让一本 C++ 编程书籍尽管大量限制其自身处于单线程考虑之下仍承认多线程的存在，并指出“有线程概念的程序员”在评估我所提供的忠告时需特别谨慎的地方。

如果你不熟悉多线程或无需忧虑它，可以忽略本书的线程相关讨论。然而如果你正在编写一个与线程有关的应用程序或程序库，请记住，我的注释或许比一般“以 C++ 解决问题时需注意……”的起点还多一些些。

TR1 和 Boost

你会发现，本书处处提到 TR1 和 Boost。各有一个条款详细描述它们（条款 54 针对 TR1，条款 55 针对 Boost）。不幸的是这些条款位于全书末尾（它们被放在那儿是因为那样的安排比较好。真的，我试过其他许多摆法）。如果你喜欢，可以现在就翻过去读它们，但如果你喜欢从头读起而不颠倒次序，下面的实施摘要将助你飞渡难关：

- TR1（"Technical Report 1"）是一份规范，描述加入 C++ 标准程序库的诸多新机能。这些机能以新的 class templates 和 function templates 形式体现，针对的题目有 hash tables, reference-counting smart pointers, regular expressions，以及更多。所有 TR1 组件都被置于命名空间 `tr1` 内，后者嵌套于命名空间 `std` 内。

- Boost 是个组织，亦是一个网站（http://boost.org），提供可移植、同僚复审、源码开放的 C++ 程序库。大多数 TR1 机能是以 Boost 的工作为基础。在编译器厂商于其 C++ 程序库中含入 TR1 之前，对那些搜寻 TR1 实现品的开发人员而言，Boost 网站可能是第一个逗留点。Boost 提供比 TR1 更多的东西，所以无论如何值得了解它。