Write Portable Code

跨平台软件开发

——C&C++

一本值得每一位中高级程序员阅读的图书

(美) Brian Hook 著
胡光华 郝春雨 译

清华大学出版社

NO STARCH PRESS

跨平台软件开发
——C & C++

(美) Brian Hook 著

胡光华 郝春雨 译

清华大学出版社

北 京

WRITE PORTABLE CODE
EISBN：1-59327-056-9

北京市版权局著作权合同登记号　图字：01-2005-5312

图书在版编目(CIP)数据

跨平台软件开发——C&C++/ (美) 胡克(Hook，B.) 著；胡光华，郝春雨 译.
—北京：清华大学出版社，2006.11

书名原文：Write Portable Code

ISBN 7-302-13907-5

Ⅰ. 跨…　Ⅱ. ①胡…②胡…③郝…　Ⅲ. 软件开发　Ⅳ.TP311.52

中国版本图书馆 CIP 数据核字(2006) 第 116134 号

出 版 者：清华大学出版社　　**地　　址**：北京清华大学学研大厦
http://www.tup.com.cn　　**邮　　编**：100084
社总机：010-62770175　　**客户服务**：010-62776969
组稿编辑：王　军　　**文稿编辑**：郑雪梅
封面设计：康　博　　**版式设计**：康　博
印 装 者：清华大学印刷厂
发 行 者：新华书店总店北京发行所
开　　本：185×260　　**印张**：13　　**字数**：300 千字
版　　次：2006 年 11 月第 1 版　2006 年 11 月第 1 次印刷
书　　号：ISBN 7-302-13907-5/TP•8358
印　　数：1～4000
定　　价：29.80 元

前　言

一天我与一位程序员同事进行了一场关于将程序从一种平台移植到另一种平台时所引起的棘手问题的谈话。当我们在抱怨字节存储次序，对齐限制和编译器的怪异行为时，这位朋友问了我一个天真但又很重要的问题："如果想要编写可移植代码，我应该去买本什么样的书来看？"

答案令人大吃一惊，市场上根本没有这一类图书。尽管关于 Java、C#、.NET、游戏编程、DirectX、极端编程和敏捷开发方面的图书有好几百本，但就是没有一本关于跨平台软件开发的现代书籍！这使我感到震惊，特别是考虑到近来运行在服务器、桌面电脑、手提式电脑，甚至移动电话上的各种新的操作系统的大量通现。是否至少应该有一本书讨论有关可移植软件的开发原理呢？当然应该，但是事实上并没有。

这就是我撰写本书的原由。

在我的一生中只有几次感到了自己需要做些什么——其推动力是如此强烈，使我用将近一年的时间去研究和编写关于跨越平台软件的开发。

本书读者对象

在我第一次提出编写可移植程序这个概念时，并不确切明白本书预期的读者对象。但是当完成本书时，就非常确定本书的典型读者了：对编写不同平台软件感兴趣的中高级程序员。下面是那些将会从本书受益的读者群中的几个例子：

(1) Windows 程序员，他希望在家里获得 Linux 的经验；

(2) Mac OS X 软件开发人员，他需要将软件移植到 Windows 上去，以便获得更大的市场；

(3) 索尼 PlayStation 2 的游戏开发人员，他必须为其游戏的多用户组件编写 Solaris 服务器应用程序；

(4) Palm OS 软件开发者，他需要将他的软件移植到 Windows Mobile 上；

(5) 大的纵向集成供应商刚刚发现他的传统开发平台被中止了，现在必须将他的产品移植到新系统上。

对于软件开发者为什么会需要或者想要将软件移植到新平台上去，存在着数不清的理由。但是值得庆幸的是，相同的原则在大多数的情况都适用，而与具体的细节无关。本书讨论那些可移植软件开发的普遍原则。

本书主要是为中高级程序员而写，但是本书很多读者可能是编程新手，他们只是抽象地听说过移植性，但是并不完全确切了解这将会如何影响他们。事实上，他们之所以不理解问题是什么，只是因为他们尚未遇见这个问题。有的具有更多经验的程序员也可

能未遇见这个问题，因为他们长时间内工作于同一系统，不需要处理可移植性问题。

为了帮助那些尚未在与可移植性快速碰撞中“弄伤脚趾”的人，我罗列出了几个假设的程序员和他们的情况。如果您能够和他们之中的任何一个人对上号，那就应该警惕了，将可移植性上升到让自己“必须要关心的事情”的清单之中去。

Bob，Java 程序员

Bob 在过去 3 年中使用 Borland 公司的 JBuilder 工作平台开发 Windows 应用程序。他一直在使用由 Sun 公司提供的 Java 运行时环境(JRE)，并且很高兴能够编写有效、高质量的程序。作为一名 Java 程序员他对自己的能力充满了自信。

但是有一天他被告知，他的雇主希望使用 IBM AIX。起初他以为 Java 语言是“高层次的”、“可移植的”，并且是 “编写一次，到处运行”(这一点众所周知)，转变不应该存在任何实际问题——只要将代码复制到新机器上，然后运行就行了。

Bob 很快就发现在 AIX 上不能够使用 JBuilder，在 Windows JRE 上的某些特性与 AIX JRE 上的特性有所不同。而且某些程序包在 Windows 上可用，而在 AIX 上则不可用。他需要确认它们在特征、性能特点、Bug 和程序包方面的区别，而且还必须学习像 Eclipse 这样一整套新型开发工具。原本认为是理所当然“很容易的”事情，就这样迅速地变成了一场噩梦。

Janice，VB 程序员

Janice 多年来一直从事编写 VB 软件的工作，提供与 Microsoft 公司的 Access 数据库交互的窗体。她还没有意识到可移植性这个概念，因为她从来不用去考虑 Microsoft Windows(或者是 Microsoft 公司产品)以外的世界。

她被要求将她的软件移植到 Mac OS X 上，不幸的是她作为一位以 Microsoft 为中心的开发者过着“与世隔绝”的生活。她惊慌地发现在 Mac OS X 上没有 VB 和 Access。她不知道如何在这个工作平台上安装和运行软件。毋庸置疑，未来几个月对于 Janice 来说将是非常困难的，因为她需要学习跨平台软件开发的原委始末。

Reese，用户界面程序员

使用 Visual C++和 Microsoft Foundation Classes(MFC)作为工具，Reese 在 Microsoft Windows 上设计并且实现了许多用户接口。他已经使用这些工具和框架迅速地开发了所要求的几乎任何类型的应用程序原形。

他公司的一位最大客户非常重视软件开发的经济性，因而偏爱低成本的 Linux。所以 Reese 需要将他所开发的成套应用程序移植到该工作平台上去。Reese 从未在 Windows 领域之外工作过，而且认为 MFC 很流行，在 Linux 上也会存在着兼容的版本。了解之后，他发现他的想法是不正确的。他现在必须学习新的开发环境、操作系统和 C++变量。当然他还必须找到一个能够在 Linux 上取代 MFC 的替代品并移植它。

Jordan，Linux/PPC 开发者

Jordan 的专长是服务器软件开发。她被要求将她的成套服务器应用软件移植到 Windows 上，起初她认为这应该是一件很容易的事，因为她的应用程序没有任何用户界面。

她发现几乎她所使用的每一个主要的 API(套接字、文件 I/O、安全性、内存映射、线程等)外观以及行为与在 Windows 上的都完全不相同。她原想，她的程序很简单，而且使用了 GCC(它在 Windows 上也可以使用)，移植只需要用一两天的时间，但实际情况让她不知所措。

接着她发现 Windows 和 Linux/PPC 甚至在内存中表示数据的方式都是不相同的，她的程序中的大部分代码是依靠内存映射文件的，它们在 Windows 上无法执行。原本以为是很简单的事，现在看起来则要逐行地检查整个程序了。

在以上这些情况中，每一位能干的程序员都突然处于一种尴尬的境况：他们感到自己不那么胜任和有才能了，这主要是由于缺乏跨平台的经验。一个共同的错误则是严重低估了完成经理要求的“简单”任务所需的工作量。

本书并没有具体涉及到 Java 或者是 VB，但无论是哪种语言，原理基本上都是相通的。本书揭示了与移植软件相关的问题和障碍。用这些信息武装起来，那些喜欢与单一工作平台打交道的人，当他们被迫走到自己专长领域之外时，就不再会感到如此震惊了。

联机信息资源

下面是与本书相关的可用资源信息：

(1) 关于本书的信息，可以在作者的网站上找到，网址为 http://www.writeportablecode.con；

(2) 简单音频程序库(SAL)可以从网址 http://www.bookofhook.com/sal 上获得；

(3) 可移植开源装置(POSH)可以从网址 http://www.poshlib.org 上获得。

目　录

1

可移植性概念

在深入讨论移植性的细节内容之前，需要首先关注可移植性概念本身。确实，如果提供一个具体的实例，比如将 Linux 软件的示例片段移植成能够在 Windows 系统上运行的程序，那我们将更容易深入理解可移植问题。但是这样却只能解决一个具体的问题。

本章讨论了可移植开发的各个方面——可移植性的原理、技术与过程，这几方面使编写可移植程序变得很容易，无需再关心具体的源环境和目标环境。一旦奠定了良好的基础知识之后，本书将会论述具体的问题以及它们的解决方法。

1.1 可移植性是一种考虑问题的方式，而不是一种状态

编程通常是一种辛苦的脑力劳动。在编辑、编译、调试、优化、记录和测试程序时，大脑需要不停地工作。人们有可能会认为移植性是一个独立的步骤，就像编辑或调试工作那样，其实“可移植性思考”并不是一个步骤，而是一种全面考虑问题的方式，它应该渗透到程序员所做的每一项具体工作中。在我们的头脑中，从“使变量名称有意义”到“不要硬编码这些常量”，任何时候都应该“考虑可移植性”。

就像杂草在花园中蔓延，可移植性应该是渗透于软件开发过程中各个方面的一种思考习惯。如果说编程是通过使用计算机能够理解的语言来迫使计算机完成一组特定动作，那么可移植性软件开发则是一个避免依赖或设定特定计算机的过程，这样，在这两个任务之间就存在着间接但很容易理解的抵触性。要求某些程序能够在当前的平台上运行就与希望它也能够在其他平台上运行的要求相抵触。

认识到移植程序和编写可移植性程序之间的区别非常重要。前者是治疗措施，而后者却是预防措施。如果可以选择的话，宁可提醒程序员提前避免不良做法，而不是以后再纠正这些不良做法所引起的副作用。这种“预防性做法”是通过非常严格地实践可移植性编程习惯获得的，从而使得该习惯成为第二天性——一个深深嵌入脑海的凭直觉就可获得的理解，它时时刻刻都会出现在程序员的思想中。

1.2 培养良好的可移植性编程习惯

在第一次接触编写可移植软件时，程序员通常会过多地考虑具体的技术细节或问题，但是经验表明，可移植性更容易通过习惯与思考方式来获得，习惯或思考方式可以鼓励(如果不是强迫的话)程序员编写可移植程序。为了培养良好的可移植性编程习惯，必须首先避免考虑类似字节排序或者排列问题这样的细节问题的诱惑。

无论在理论上对可移植性掌握有多深，通常移植性编程实践都会证实这些理论的缺陷。从理论上讲，编写与标准兼容的程序应该使得程序更加具有可移植性，但是不经过测试就做出这种假设会使您遇到许多意想不到的问题。例如，如果一个不合作或者多故障的编译器拒绝遵守标准，那么 ANSI C 规范是否限定某一种行为就无关紧要了。如果使用不兼容的工具，那么符合规范的程序实际上也无济于事。

Microsoft 的 Visual C++ 6.0 就是一个典型的例子，它对于 for 语句内的变量作用域不能恰当的提供 C++规范。

```
for ( int i = 0; i < 100; i++ )
   ;
i = 10; /* With MSVC++ 6.0, this variable still exists...with
           compliant compilers the variable is out of scope
           once the for loop exits and thus this line would
           generate an error */
```

<p8-1>/*对于 MSVC++ 6.0 来说，这个变量仍然存在……对于兼容的编译器来说，一旦出了 for 循环，这个变量就出了作用域。因此这一行会产生一个错误*/。

Microsoft 的开发人员在 C++编译器 7.x 版本中校正了这种错误行为，但是这引起了使用版本 6.0 编写的程序的反向兼容性问题，因此就导致了非兼容的默认行为。这意味着程序员可能开发出了在 Microsoft 的 Visual C++上安全可移植的程序，但是当使用 GCC 编译时才发现这些程序崩溃了。

但是如果养成了良好的编写可移植性程序的习惯，就可以很容易地解决这类问题，比如编程时，经常在多种环境中测试与开发代码。这样就省却了牢记所有可能遇到的具体故障和标准遁词的麻烦。

1.3 良好的习惯胜过故障或标准的具体知识

先来解释一下良好的可移植性编程习惯。

1.3.1 尽早经常移植

直到程序被移植之后，才能确定程序是否可移植，因此，尽早经常地移植程序非常重要。这样可以避免在软件开发过程中编写的是“可移植的程序”，然而在测试时却发现了很多小的有关可移植问题的常见错误。通过尽早测试程序的可移植性，就能够及时解决这些问题。

1.3.2 在不同的环境中开发

通过在不同环境中开发程序，可以避免先编写然后移植的分阶段处理方式。这种习惯同时也最大程度减少了程序失去均衡的风险，即由于软件开发人员的疏忽造成的其中一个平台程序枯萎，而程序的另一个平台版本却很强壮的情况。

例如，在项目开发初期，程序可能分别在 Linux 上，在 Mac OS X 和 Windows 上运行，但是由于时间的压力，Linux 版本被省却了。6 个月之后，需要软件能够在 Linux 上运行，但是却发现许多更改从来没有传播到这个平台上(例如，由于条件编译指令的缘故)。

在多机种环境中开发软件的第一步是，要确保软件开发人员使用尽可能多的主机系统和工具来工作。如果开发的项目将分别被安装在 Sun Sparc 的 Solaris 上，在运行 Microsoft Windows 的 Intel 上，以及在运行 Mac OS X 的 Macintosh 上面，那么就要确保开发团队成员使用这些系统作为他们的主要开发系统。如果他们没有被要求使用这些系统作为主要开发系统，那么 “先使程序能工作，以后再对它进行移植”的思维定势就会萌芽。

甚至在类似的系统上工作时，例如，在运行 Windows 的 PC 上，组合使用硬件(显卡、CPU、声卡、网卡)和软件都是好主意，因为这样能够较早发现更多的配置故障。这将有助于避免以后在现场报告程序故障时，软件开发人员惊叫“这个程序在我的机器上是工作的!”。

简单音频程序库的实例：在多机种上开发

我同时在下列不同的系统中开发简单音频程序库(SAL)，分别是在基于 Windows XP 的笔记本电脑上使用 Microsoft 的 Visual C++; 在运行 OS X 10.3 的 Apple G4 Power 上使用 XCode/GCC; 以及在基于 AMD Athlon XP 的 Linux(ArkLinux，一种基于 Red Hat 的 Linux 版本)工作站上使用 GCC。并且偶尔使用 Microsoft 的掌上电脑上的嵌入式 Visual C++。大块的程序是在 Windows XP 操作系统上开发的，每隔几天就偶尔在其他平台上移植与验证。

我偶尔也会使用像 Metrowerks CodeWarrior 和 Visual C++ 7.x 这样的编译器来完成快速测试与验证任务，而且有时其结果会出现问题，甚至出现 bug。

代码从来就没有偏离或者崩溃太多。但是，掌上电脑支持则是在 SAL 开发过程的相当晚的阶段才引入的，而且是导致工作单调乏味的原因，因为对这种平台无法运行如此众多的 SAL。具体来说，测试程序依赖于 main()的存在，这就产生一个问题。因为掌上电脑没有控制台应用程序，所以必须提供 WinMain()方法。由掌上电脑引起的其他问题还有宽字符串(便于国际化)。

我使用了几个不同的低层次 API 实现关键特性，例如线程同步。我找出公用的抽象，然后将它们放在一个单独的抽象层中，并把它放置于每一种实际实现平台上。这就意味着从一个基于 Win32 的互斥信号模式移植到一个 POSIX(Portable Operating System UNIX，可移植的 UNIX 操作系统)线程(pthread)互斥信号模式是非常容易完成的，因为 SAL 并没有充斥了针对 Win32 的代码。

1.3.3 使用不同的编译器

您也可能希望使用尽可能多的编译器。虽然不同的主机系统要求如此，但是有时也可以在不同的系统上使用相同的编译器；例如，GCC 编译器就可以在范围很广的平台上使用。

通过在大范围编译器上成功编译，一旦所偏爱的编译器的开发商突然消失，也不会陷入困境。而且还可以确保程序库不依赖于新的(和未经证明的)语言或者特定编译器的特性。

1.3.4 在多个平台上进行测试

大多数项目都有明确的既定目标，这些目标由市场动态因素所决定。这就极大地简化了测试和质量保证工作，但这也是走向危险的隐含问题的开端。即使知道程序将会在单一的目标系统上运行，使用可以替换的平台也没有坏处——处理器、RAM、存储器、操作系统等等——仅仅用于测试。

并且如果目标系统由于市场需求或者商业关系的改变而发生了变化，也会很高兴地发现软件并不是硬编码于单一平台。

1.3.5 支持多个程序库

如今在大多数的软件开发工作中，很少需要编写新的程序，更多的工作是将现成的程序模块粘合在一起。如果依赖于一组专用程序库或者 API，则移植到一个新的平台将非常困难。但是，如果尽早支持多个可以替代的程序库，这些程序库能够完成同样的任务，那么万一程序库的销售商破产或者拒绝支持他的软件在另一个平台上运行，这时将会具有更多的选择。这里还有一个小小的附带好处，即可以特许或者开源程序，而不用担心依赖于封闭源代码的第三方程序库。

典型的例子是对 OpenGL 的支持和对 Direct3D 支持的对比，这两个都是如今可以利用的优秀的三维图形 API。OpenGL 跨平台并且可以在广泛的系统上应用，包括所有主要的 PC 操作系统。而 Direct3D 则是用于 Windows 的官方三维图形 API，仅仅可以在 Windows 上使用。这就将软件开发人员置于尴尬局面之中，是优化拥有世界上最大用户市场的 Windows，还是使用 OpenGL 同时支持多个平台?

理想情况下，可以抽象图形层，使它能够用于这两种 API。这可能涉及许多工作，因此在着手之前必须非常清楚地思考抽象的细节。但是到了将软件迁移到新平台的时候，这种抽象化工作将会得到回报。

1.4 为新项目规划可移植性

创建一个新工程往往并不令人兴奋。可当创建新目录时，并可以将那些建立在多年经验基础之上的完美源代码批量载入时，就会有一种惊喜。

当发现自己处于从零开始的少见处境时，可以规划如何使得项目具有可移植性。如果在开始新项目之前仔细考虑，就能够在今后节省许多时间与麻烦。

1.4.1 使可移植性变得容易

正如许多其他良好习惯那样，坚持良好的可移植性习惯的可能性直接与习惯的易用性成正比。如果软件开发方法使得可移植性软件的开发工作令人非常生厌或者效率低下，那么它将会比您说“错过的里程碑”更快地被丢弃。

创建过程、程序库和机制非常重要，这可以使得编写可移植性程序成为第二天性而不是一项费力、持续的任务。例如，程序员不应该处理字节排序之类的工作，除非确实涉及到了那个层次上的工作。

1.4.2 选择可移植性的合理水平

尽管可以努力编写完全移植的程序，但是现实地讲，如果不牺牲软件的功能，要达到这个目的几乎是不可能的。

对于可移植性，不宜固执己见！软件的可移植性应该实用。为了确保仅具有有限用途的实用程序的可移植性，而付出时间与精力类似于花费一周的时间去优化一个只被调用一次的例程，这并不是支配时间的好方法。

这就是说建立基本的和现实的基准——一组定义合理可移植性的基本规则——对于项目极其重要。没有这样一套基准，项目将注定是脆弱的，它将会在各种平台上运行得都很差。

每一个平台都具有它自己所涉及的计算机、编译器、工具、处理器、硬件、操作系统等特性的集合。在从一个平台迁移到另一个平台的过程中，存在着几千种可能中断程序的方式。值得庆幸的是，其中许多特性是共同的，从而减轻了编写可移植性软件的任务。定义共同的基本准则是设计与编写可移植性程序的第一步。

· 正如稍后在本书第 14 章中讨论的那样，可移植性部分内容与在基准内的可伸缩性相关(这是一种能够在性能和特性大幅度变化的系统上运行的能力)。可伸缩性非常重要，但是它必须在明确定义的参数之内，以便保持类似的相关性。

除了基本的特性之外，还必须深刻理解平台的性能。编写的软件必须能够在 8 MHz 的 Atmel AVR 微控制器上和在 3.2 GHz 的 Intel 奔腾 4 处理器上同等地编译和运行，但是两种情况的结果是否能够在这两个环境中都有意义和令人感兴趣则是个问题。用于工作站级的计算机的算法和数据结构与那些用于嵌入式微控制器的算法和数据结构完全不同。而且限制功能强大的计算机，使其运算能力与完全不同体系结构的计算机的运算能力相同，这非常浪费资源。

实例研究：浮点数学运算

ANSI C 语言分别通过 float 和 double 这两个关键词以及与它们相关的数学运算符来支持单精度浮点运算与双精度浮点运算。大多数程序员认为这种支持是理所当然的。遗憾的是，如今的某些设备，以及不久前的许多设备都具有非常差的浮点数学支持。例如，用于 PDA 的大多数处理器不能在本地执行浮点运算指令，因此就必须使用速度明显比较慢的模拟库。

现在，对某一个具体的项目来讲如果很少用缓慢的浮点运算则有可能还可以接受，(即使在这种情况下，如果必须连接浮点模拟库，一个可执行文件的大小也有可能变得很大)。但是对于那些需要强大的浮点运算性能的项目来说，当该程序必须迁移到没有内部浮点运算支持功能的系统上去时，事情有可能很快变得很糟糕。

解决这种分歧的常用方法是使用特殊的宏编写所有的数学运算，这些宏在没有本机浮点支持的设备上调用定点例程而不是浮点例程。下面是一个例子：

```
#if defined NO_FLOAT
typedef int32_t real_t;
extern real_t FixedMul( real_t a, real_t b );
extern real_t FixedAdd( real_t a, real_t b );
#define R_MUL( a, b ) FixedMul( (a),(b))
#define R_ADD( a, b ) FixedAdd( (a),(b))
#else
typedef float real_t;
#define R_MUL( a, b ) ((a)*(b))
#define R_ADD( a, b ) ((a)+(b))
#endif /* NO_FLOAT */
```

具有三个元素的点乘积可以编写成下面的代码：

```
real_t R_Dot3( const real_t a[ 3 ], const real_t b[ 3 ] )
{
real_t x = R_MUL( a[ 0 ], b[ 0 ] );
real_t y = R_MUL( a[ 1 ], b[ 1 ] );
real_t z = R_MUL( a[ 2 ], b[ 2 ] );
return R_ADD( R_ADD( x, y ), z );
}
```

但是，纯粹的浮点运算程序很明显比较容易阅读与理解：

```
float R_Dot3( const float a[ 3 ], const float b[ 3 ] )
{
return a[ 0 ] * b[ 0 ] + a[ 1 ] * b[ 1 ] + a[ 2 ] * b[ 2 ];
}
```

如果必须支持没有浮点运算功能的系统，或者认为非常有必要，那么使用宏则可能是一个好方法。但是如果有能力规定本地浮点支持为基准支持，那么就会受益于程序的可阅读性与简洁性。

可移植性是一个好主意，编写可移植性程序是一个好的做法，但是如果对它走极端或者为了满足思想教条主义而编写冗长的可移植程序，其结果会适得其反。可移植性是达到目的的手段，但不是目的本身。

体系结构基于低延迟时间、高带宽通信环境的网络应用程序在遇到调制解调器时将会产生灾难性的后果。因此尽管这个应用程序能够在任何地方编译与运行，但实事求是地说，它对于某些类型的网络类而言是不可移植的，这是由于对它所驻留的网络所做的基本假定。

确定基准是可移植软件开发的关键，因为它确立了完全合理的前提条件，使得软件在有限平台上的开发工作更加有效。

漫无目的的可移植性和充分的可移植性之间明显不同。如果项目针对单一的目标平台，而且知道在何时改变编译器，就可以专心致治解决程序在编译器之间的可移植性问题，而不用过多地考虑那些无需支持的目标系统。

1.4.3 不要将项目变成专有产品

现代软件开发工作的复杂性令人难以置信，即使一个简单的项目都可能是由成千上万行源代码组成。这种复杂性经常要求使用(指理想的情况)经过精心测试与归档记录的第三方组件，例如程序库与工具。使用已经存在的组件可以节省时间，但是它也引入了许多新的对于可移植性的担心。确保自己软件的可移植性已经是一项非常困难和花费时间的工作了，但是当引入了外来影响，那它可能是真正的令人恐怖。每一次外来的组件被集成到项目中，该项目的灵活性和控制性就会显著下降。

甚至在最佳情况中——开源程序库——也必须验证该程序能够在所有需要它运行的任何平台上编译与运行。如果所需要的平台尚未被开源程序库的编写者所支持的话，那么就需要自己处理移植工作(幸运的是，它们仍然是一种选择，这是由于开源的性质所决定的)。

遗憾的是，封闭式源代码专用程序库的使用摒除了这种选择。在这种情况中，如果提供者不愿意或者是不能够支持所要求的平台的话(例如供应商破产了)，那就会陷入困境。最糟糕情况下，是发现必须为了新的平台从新实现这个第三方组件。从长远的角度来讲，项目使用第三方组件是非常危险的。许多业界老手都能够讲述关于项目不可避免地与一种孤立的程序库或者工具组相连接，结果这种连接影响了整个软件开发过程的故事。

例如，许多 PC 软件开发人员使用 Microsoft 的 DirectPlay 网络程序库，因为它免费而且易于获取，而且它声称提供了大量的特性，如果要重新实现这些特性就需要很长时间。免费与易于使用的技术非常诱人，但是却使得那些采用的人们在试图将程序移植到像 Macintosh 和游戏控制台这样的非 Microsoft 平台时陷入了困境。他们时常发现自己需要重新编写整个联网层，这是他们轻信专有技术而付出的代价。