

WRITE PORTABLE CODE

A GUIDE TO DEVELOPING SOFTWARE FOR
MULTIPLE PLATFORMS



Brian Hook



NO STARCH
PRESS

WRITE PORTABLE CODE

An Introduction to Developing Software for Multiple Platforms

by Brian Hook

NO STARCH PRESS

San Francisco

Translated by: Kevin

本资料仅供学习所用，请于下载后 24 小时内删除，否则引起的任何后果均由您自己承担。
本书版权归原作者所有，如果您喜欢本书，请购买正版支持作者。

目录

WRITE PORTABLE CODE	2
目录	3
前言	9
本书的读者.....	9
在线资源.....	11
导论 可移植软件开发的艺术.....	12
可移植性的益处.....	12
平台的组成.....	13
假设的问题.....	14
编码标准.....	14
可移植编程框架.....	14
第 1 章 可移植性概念.....	16
可移植是一种思想状态.....	16
养成良好的可移植习惯.....	16
良好的习惯胜过了了解 Bug 和标准的特定知识.....	17
尽早迁移、经常迁移(Port Early and Port Often)	17
在多样的环境下开发.....	17
使用各种编译器.....	18
在多个平台上测试.....	18
支持多个库.....	18
为新项目计划可移植性.....	19
使可移植更加容易.....	19
选择一个合理的可移植等级.....	19
不要把你的项目绑定在私有产品上.....	21
移植旧代码.....	22
在代码成功迁移之前假设代码是不可移植的	22
仅仅修改最基本的部分.....	23
计划你的修改.....	23
在版本控制系统中记录所有事情.....	24
第 2 章 ANSI C 和 C++.....	25
为什么不是其它语言?	25
C 和 C++提供底层访问	25
C 和 C++编译成本地代码	25
C 和 C++方言	25
可移植性和 C/C++.....	26
第 3 章 可移植的技术.....	28
避免新特性.....	28
处理不同平台可用特性的差异.....	28
使用安全的序列化和反序列化.....	31
集成测试.....	33
使用编译选项.....	34

编译期断言	34
Strict 编译	35
分离平台相关和可移植的文件	35
编写直截了当的代码	36
使用唯一的名字	36
实现抽象	39
调度抽象	39
抽象数据类型	44
使用 C 预处理器	46
为意外做好准备	47
与系统相关信息交互	48
桥接函数	49
底层 (Low-Level) 编程	53
避免自修改/动态产生的代码	53
保留一个高级的 Fallback	54
register 关键字	56
外部 vs 内嵌 asm 文件	56
第 4 章 编辑与源码控制	58
文本文件的行结束符差异	58
可移植的文件名	59
源码控制	60
源码控制系统	60
通过代理 Checkout	62
构建工具	63
平台特定的构建工具	63
可移植构建工具	64
编辑器	67
小结	67
第 5 章 处理器差异	68
对齐	68
字节序和大小端	71
大端 vs 小端	71
标准存储格式	72
固定网络字节序	74
带符号整型表示	76
本地类型的大小	77
地址空间	79
小结	79
第 6 章 浮点	80
浮点的历史	80
标准 C 和 C++ 的浮点支持	80
浮点的问题	81
求值不一致	81
浮点和网络应用	82

转换.....	83
定点整数 Math	84
从浮点提取整数位.....	85
实现相关.....	88
异常结果.....	89
特殊值.....	90
异常.....	91
访问浮点环境.....	91
存储格式.....	92
小结.....	92
第 7 章 预处理器.....	93
预定义符号.....	93
头文件.....	94
头文件路径规范.....	95
头文件名.....	96
配置宏.....	96
条件编译.....	97
Pragmas	98
小结.....	98
第 8 章 编译器.....	99
结构体大小、填充、对齐.....	99
内存管理特性.....	101
Free 的效果	101
对齐内存分配.....	102
堆栈.....	102
堆栈大小.....	102
alloca() 的问题.....	103
printf 函数.....	103
类型大小和行为.....	104
64 位整数类型.....	104
基本类型的大小.....	105
有符号和无符号字符类型.....	107
enum 用作 int	108
数值常量.....	109
带符号和无符号右移.....	109
调用约定.....	109
名字装饰.....	110
函数指针和回调.....	111
可移植性.....	111
返回结构体.....	112
位域.....	113
注释.....	114
小结.....	115
第 9 章 用户交互.....	116

用户界面的演化.....	116
命令行.....	116
窗口系统.....	116
本地 GUI 还是应用 GUI.....	117
底层图形.....	117
数字音频.....	118
输入.....	119
键盘.....	119
鼠标.....	119
操纵杆和游戏控制器.....	119
跨平台工具集.....	120
小结.....	120
第 10 章 网络.....	121
网络协议的演化.....	121
编程接口.....	121
Sockets	121
RPC 和 RMI.....	124
分布式对象.....	124
小结.....	124
第 11 章 操作系统.....	125
操作系统的演化.....	125
Hosted 和 Freestanding 环境	125
操作系统可移植性悖论.....	126
内存.....	126
内存限制.....	126
内存映射.....	127
内存保护	127
进程和线程.....	128
进程控制和通讯函数.....	128
进程间通讯 (IPC)	129
多线程.....	129
环境变量.....	129
异常处理.....	129
C 异常处理	129
C++异常处理	129
用户数据存储.....	129
Microsoft Windows 注册表	129
Linux 用户数据	129
OS X Preferences	129
安全和权限.....	129
应用安装.....	129
特权目录和数据.....	129
底层访问.....	129
小结.....	129

第 12 章 动态库.....	130
动态链接.....	130
动态装载.....	130
共享库的问题（DLL Hell）	130
版本问题.....	130
激增.....	130
Gnu LGPL	130
Windows DLL.....	130
Linux 共享对象	130
Mac OS X 框架、插件和 Bundle	130
框架.....	130
Bundle	130
插件.....	130
小结.....	130
第 13 章 文件系统.....	131
符号链接、快捷方式、别名.....	131
Windows LNK 文件	131
Unix 链接	131
目录规范.....	131
磁盘驱动器和卷标.....	131
路径分隔符和其它特殊字符.....	131
当前目录.....	131
路径长度.....	131
大小写.....	131
安全和访问权限.....	131
Macintosh 的古怪之处.....	131
文件属性.....	131
特殊目录.....	131
文本处理.....	131
C 运行时库和可移植文件访问.....	132
小结.....	132
第 14 章 伸缩性.....	133
好的算法=好的伸缩性.....	133
伸缩性的局限.....	133
小结.....	133
第 15 章 可移植性和数据.....	134
应用数据和资源文件.....	134
二进制文件.....	134
文本文件.....	134
XML	134
脚本语言作为数据文件.....	134
可移植图形.....	134
可移植声音.....	134
小结.....	134

第 16 章 国际化和本地化.....	135
字符串和 Unicode	135
货币.....	135
日期和时间.....	135
界面元素.....	135
键盘.....	135
小结.....	135
第 17 章 脚本语言.....	136
脚本语言的缺点.....	136
JavaScript/ECMAScript	136
Python	136
Lua	136
Ruby	136
小结.....	136
第 18 章 跨平台库和工具集.....	137
库	137
应用框架.....	137
QT	137
GTK+	137
FLTK	137
wxWidgets.....	137
小结.....	137
附录 A POSH	138
POSH 预定义符号.....	138
POSH 固定大小类型.....	138
POSH 工具函数和宏.....	138
附录 B 可移植性的规则	139
参考书目.....	140
索引	141

前言

有一天我和一位程序员同事谈论把软件从一个平台移植到另一个平台的痛苦，抱怨字节序、对齐限制、以及编译器怪癖等等头痛的事。这个朋友问了一个无意但却很重要的问题：“如果要编写可移植代码，我应该看什么书呢？”

答案很令人震惊：没有。

Java、C#、.NET、游戏编程、DirectX、极限编程、以及敏捷开发有着成百上千的书，但却没有一本书讲解跨平台软件开发！这使我很吃惊，特别是在各种不同操作系统运行在服务器、桌面、手持设备、甚至手机上的今天。难道不应该至少有一本书是讨论可移植软件开发的原则？当然应该有，但是实际上却没有。

这就是本书产生的缘由。

在我的一生中有几次我觉得我应该去做某件事——强烈的冲动使我花整整一年时间来研究和编写跨平台软件开发。我有非常坚定的信念来完成本书，你手上拿着的这本书就是最终结果。

本书的读者

我最初涉及编写可移植代码这个概念时，不太确定本书的期望读者，但是在完成本书之后，“典型”的读者变得非常清晰：对不同平台编写软件感兴趣的中到高级程序员。下面是能从本书获益的读者的一些例子：

- Windows 程序员，希望在家里体验 Linux 开发。
- Mac OS X 开发者，需要移植软件到 Windows 平台，以获得更广的市场。
- Sony PlayStation 2 游戏开发者，必须为游戏的多人组件编写 Solaris 服务器应用。
- Palm OS 开发者，要把软件移植到 Windows Mobile 平台。
- 大集成商发现自己的传统开发平台已经中断，必须移植自己的产品到新系统。

除此之外，还有无数的理由导致开发者发现自己或者希望切换到新平台，幸运的是除了特定的细节，多数原则都是通用的。本书讨论并教你这些可移植软件开发的通用原则。

《Write Portable Code》这本书为中到高级程序员编写。但是我猜测本书的很多读者可能是程序员新手，仅仅听过可移植性的抽象概念，并不确定可移植性会怎样影响到自己。实际上，这些读者甚至都不理解程序是什么，因为他们还没有运行过程序。同样，对于始终工作于同一系统的有经验的程序员，也可能并不怎么了解可移植性，因为他们不需要处理可移植性的问题。

为了帮助那些尚未陷入可移植性困境的程序员理解可移植性的重要性，我将列出许多实际的程序员和他们的状况。如果你可以从中发现自己的影子，你就应该引起足够的警惕，因为可移植性将成为你“必须关注的事情”列表中的一项。

Bob, Java 程序员

Bob 在过去的三年里一直使用 Borland 的 JBuilder 环境开发 Windows 应用。他使用 Sun 提供的 Java 运行时环境(JRE)高效并快乐地编写高质量的代码。他对于自己作为一名 Java 程序员非常有信心。

然后有一天，他被通知雇主将要采用 IBM AIX。他原本认为 Java 是“高级”、“可移

植”的，而且是“编写一次，到处运行”，因此这种转换应该没有任何实际的问题——把代码拷到新机器并运行就 OK 了，不是吗？

很快 Bob 就发现 AIX 上并没有 JBuilder，Windows JRE 上的一些特性在 AIX JRE 上的行为不一致，Windows 存在的一些包在 AIX 上也没有。他匆忙地列出两个平台的特性、性能参数、bug、以及包的区别列表，然后必须学习新的完整开发工具(例如 Eclipse)。原本确保非常容易的工作迅速变成他的一个梦魇。

Janice, Visual Basic 程序员

Janice 已经编写了许多年的 Visual Basic(VB)程序，提供与 Microsoft Access 数据库交互的 Form 应用。可移植性对于她来说甚至都没有概念，因为她从未考虑过 Microsoft Windows 之外的世界(或者 Microsoft 之外的产品)。

然后她被要求把软件迁移到 Mac OS X。不幸的是，她已经作为 Microsoft 程序员生活在完美的绝缘世界中太久了。她震惊地发现 Mac OS X 没有 VB 和 Access，她完全不知道怎样才能让软件在新平台上运行。不必说，接下来的几个月对于 Janice 是非常困难的，她用最困难的方式来学习跨平台开发。

Reese, 用户界面程序员

Reese 已经使用 Visual C++和 MFC 为 Microsoft Windows 设计和实现了许多用户界面。他使用这些工具和框架为几乎所有类型的应用快速搭建原型。

然后他公司最大的客户之一决定采用 Linux 以节约成本，因此 Reese 被要求移植他已经开发的应用到 Linux 平台。Reese 此前从未在 Windows 之外的世界中工作，因此他猜想在 Linux 中存在与 MFC 兼容的克隆物，毕竟 MFC 是如此的流行。经过一些基本的研究后，他发现这是错误的。现在他必须学习新的开发环境、操作系统、以及 C++语言的变体。当然，他还必须寻找一个替代物，并把 MFC 移植到 Linux。

Jordan, Linux/PPC 开发者

Jordan 专门从事服务器软件开发。她被要求移植服务器应用到 Windows，最初她认为这应该很简单，因为她的服务器应用没有用户界面。

然后她发现几乎自己使用的所有主要 API——socket、文件 I/O、安全、内存映射、线程等等——在 Windows 上都完全不一样。她目瞪口呆地认识到这些，而她原本想的是她的应用很简单，并且使用 GCC(在 Windows 上也有)，迁移应该只需要一到两天。

她还发现 Windows 和 Linux/PPC 甚至内存中描述数据的方式也不一样，大量依赖于内存映射文件的代码在 Windows 运行失败。原本认为非常简单的事情，现在看起来需要对代码进行一行一行的修正。

在上面每个情况中，有能力和才干的程序员，瞬间发现自己处于完全不胜任的处境，主要原因是他们缺乏跨多个平台的经验。一个常见的错误，就是低估了管理部门要求的“简单”任务需要的工作时间。

在本书中，我不讨论 Java 或 VB 的特定细节，但是本书的许多概念对于任何语言都是正确的。这本书暴露移植软件相关的问题和阻碍。有了这些信息，那些快乐地工作于单一平台的人，在被强制迁移到另一个平台时，将不会再感到震惊。

在线资源

下面是本书相关的可用资源：

- 关于本书的信息可以在我的网站 <http://www.writeportablecode.com> 找到。
- 简单音频库(Simple Audio Library SAL)可以在 <http://www.bookofhook.com/sal> 找到。
- Portable Open Source Harness(POSH)的网址是 <http://www.poshlib.org>。

导论 可移植软件开发的艺术

在软件工业非常强调尽可能快完成项目(尽管实际上很少有项目能够按时完成)。强调生产率和产品交付当然是好事情，但是花费稍微多一点点时间在可移植性概念上，也有相当的优点。

当我们谈到编写可移植软件时，我们说的是在不同平台迁移软件的能力。这是一个有挑战的任务，因为与计算机科学其它方面不同，可移植软件开发依赖于经验和轶事。我的目标是帮助读者，提供一个单一的资源，覆盖可移植软件开发的艺术。

可移植性的益处

开发可移植软件可能听上去需要许多工作，它也可能确实需要。那么为什么我们需要可移植软件呢？下面是主要的原因：

可移植性拓宽你的市场

如果你能够使你的代码同时运行在 Linux 和 Windows 上，那么可以明显地增加你的销售百分比。不仅如此，有时候客户拥有各种各样的计算需求，支持客户所有系统的能力，比起那些仅仅支持部分系统的竞争者，能够为你提供更强的竞争力。例如，如果一个公司运行 Macintosh 和 Windows 计算机，则该公司会选择能够同时运行在这两个系统上的软件，而不是只能运行在其中一个系统的软件。

可移植软件更加健壮

支持新平台除了销售和市场的优点之外，可移植代码还有一个主要的技术优点：它能产生更好的软件。通过暴露边缘假设和延迟编码习惯，潜伏的 bug 能够更早地暴露出来。通常在移植到新平台的过程中能够很早期地发现严重的 bug。

可移植软件更加自由

如果你的公司拥有切换到新平台的能力，则施加在各个平台的外部压力对你自己的机会有很大的作用。如果必须迁移到新平台，或者使用新工具集，可移植编码习惯能够使这个迁移过程更加平滑。

可移植性有时候是必须的

有时候可移植性不是一种奢侈品，而是必须的。随着计算领域的变化，新技术源源不断地产生，都需要提供支持和关注。考虑从 DOS 迁移到 Windows、Mac OS 到 System X、Macintosh 到 PC，特别是今天从 32 位处理器迁移到 64 位的情况。后者是一个缓慢但却稳定的运动，大多数软件开发者如果想保持竞争力，必须在不久的将来掌握这种变化。

可移植性提供更多选择

你可能觉得某个特定的工具在开发过程中是绝对不可或缺的，但是它在你偏爱的平台却不存在。我还记得当年使用 OS/2 为 DOS 开发的情景，由于我需要使用 Emacs 作为我的文本编辑器，而 DOS 版的 Emacs 不如 OS/2 版那么全功能和健壮。目前有一些开发者发现转到 Linux 是值得的，因为 Linux 上有优秀的代码分析工具。

有些程序员喜欢编写可移植软件，因为它能体现我们身上那种优雅的极客(geek)基因，但是还有其它许多实际的理由使我们应该这么做。很少有花费了精力开发可移植软件，而导致开发者后悔的例子。更大的市场、更好的软件质量、以及更多的选择对跨平台软件是很强大的激励。

两个公司的故事

Joe 的公司已经使用 **Uberwhack C/C++** 整整一年了，开发者最终厌倦了编译器的无能：产生不正确的代码、糟糕的标准支持。由于他们平时最小化对 **Uberwhack** 专有库、以及编译器特定扩展的依赖，迁移到 **Muchgood C++** 开发系统的过程很容易，也很少有痛苦。结果 Joe 公司的开发者节省了数月时间，否则如果不这样的话，他们将失去大量开发时间，用来寻找不属于自己的 Bug。

Jane 的公司很不幸地使用了大量的 **Uberwhack** 专有类库、**StraitJacket** 基础类库。开发者同时还使用了一些 **Uberwhack** 创建的 C++ 语言扩展，而这些扩展是不推荐使用的。现在他们的软件牢牢地捆绑在 **Uberwhack** 系统上，短期内迁移到不同的开发系统是不可行的。这花费了他们数月的开发时间，因为他们寻找 Bug 的时间都浪费在了 **Uberwhack** 编译器错误的代码生成器上。

Joe 工作于一个硬件设备生产厂商，当前基于 **Motorola 68000 CPU** 和 **OS 9** 操作系统。由于他的代码以可移植、抽象的方式编写，当管理层问他需要多长时间迁移现有代码到 **Intel StrongArm** 处理器和 **Microsoft Pocket PC 2003** 操作系统上时，Joe 可以诚实地说：只需要几个月。

Jane 的公司听说 Joe 公司要切换到 **StrongArm/Pocket PC 2003** 平台的传闻后，也同样决定迁移。不幸的是，Jane 告诉管理层迁移现有代码将会极度困难，因为他们的代码完全捆绑在当前平台上。她估计需要花费将近一年的时间，才能使软件运行在新的目标系统。

最后，在硬件工作组尚未准备好新的工作系统时，Joe 就被要求立即开始开发。由于 Joe 的代码在较高层次进行了抽象，他可以使用目标系统模拟器。甚至在硬件工作组交付第一个原型之前，Joe 的代码就已经准备好进行测试和调试了，这节省了好几个月的开发时间。

虽然看上去这像是一个过度虚构的例子，但类似的情况在整个工业却屡见不鲜。可移植软件避免把自己绑定在单一平台，给予软件开发者的敏捷性，在现实条件要求迁移时允许快速切换到新平台。

平台的组成

可移植性就是在平台之间迁移软件的能力，平台指的是构建和运行代码需要的环境。

在平台之间迁移涉及以下平台成分的组合：

- 构建工具，例如编译器、链接器、构建/项目管理工具。
- 支持工具，例如源代码控制、分析、调试工具
- 目标处理器
- 目标操作系统
- 库和 API

每一个因素都对软件在新平台上运行和构建有重要影响。编译器厂商可能有私有类库和语言扩展；构建工具可能在不同平台上不能工作；操作系统可能有完全不同的能力；库并不是在所有系统上都存在。

假设的问题

除了少数例外，迁移软件时遇到的多数问题的根本原因是不合理的隐式假设。这些对某个特定平台的假设，一旦软件移植到另一个平台将变得不再有效。假设包括以下：

- 编译期假设(所有整数都是 32 位的)。
- 资源假设(我能够在堆栈上分配 8MB 的数组)。
- 性能假设(我希望能有一个 3GHz 的处理器)。
- 特性假设(我能够使用多线程)。
- 实现假设(我的目标操作系统使用 `main()` 作为程序入口)。

你可以做许多方面的假设，而这些假设在以后将会回来咬你一口。

有了这个认识，我主张可移植软件开发是转换隐式假设为显式需求的行为，结合使用抽象来把通用从特定隔离开。你的目标应该是标识非法的假设，转换它们为显式的需求。然后在可能时，引入抽象从特定实现中提取出通用架构。

编码标准

在本书中我使用 ANSI C 和 C++ 作为演示语言，第二章将说明具体的理由。C 语言的可移植性问题，特别是不同实现间的问题，早已为大家所知。在 1989 年，ANSI/ISO 委员会批准了一个标准(C89)来解决这些问题。在经过十年的使用和评估后，标准修订为 C99，但是它的完全采用很缓慢。尽管 C99 不是一个流行的标准，我仍然采用它的一些惯例来简化问题。出于一致的考虑，我使用 C99 对固定大小类型的支持。表 1 展示了 C99 的数据类型。

表 1: C99 标准数据类型

类型	描述
<code>int8_t</code>	带符号 8 位整数
<code>uint8_t</code>	无符号 8 位整数
<code>int16_t</code>	无符号 16 位整数
<code>uint16_t</code>	无符号 16 位整数
<code>int32_t</code>	无符号 32 位整数
<code>uint32_t</code>	无符号 32 位整数
<code>int64_t</code>	无符号 64 位整数
<code>uint64_t</code>	无符号 64 位整数

我开发了一个包，叫做 Portable Open Source Harness(POSH)(参见附录 A)，它提供一组类似 C99 的类型定义(以及其它一些东西)。本书的源代码例子有时会使用 POSH 的特性。产品应用如果需要跨多个平台，需要使用类似 POSH 的方法(或者直接使用 POSH，因为它是开源而且自由使用的)。

可移植编程框架

看起来大多数计算机书籍都属于两个阵营之一：理论派或实践派。理论派书籍(例如学

院教材)就像你父母给你建议,在解决特定问题时作用很小。通常一本三维计算图形的书在你试图理解整个三维计算时很有用,但是当你要编写软件时,你通常希望能有适合你情况的特定例子。

而实践派的书籍(例如市场上的)则通常对立即解决问题很有用,但是却只有短暂的稳定。它现在看起来可能很好,但是在几年之后,它就会变得完全不合适。今天可能不太需要关于 IBM OS/2 和 PC BIOS 的书,但在 1989 年,它们却非常流行。

本书通过为可移植编程提供一个概念框架,同时给出足够的真实细节来提供相关的上下文,试图缩小这两个阵营的差别。

适时和不适时有时候很奇怪。由于编译器 Bug、可移植性问题、以及系统的差别,不太可能枚举出程序员面对的所有可移植性“gotchas”。实际上本书只解决你实践中可能遇到的部分问题。但是本书讨论的概念和策略将帮助你解决遇到的多数可移植问题。

我的目标是分离可移植的理论和特定的真实实例。当注意力应该指向高层问题时,一个简单的问题就会使我们陷入到细节的恐惧中。由于这个理由,当对具体问题的讨论很长时,解决办法将放在单独的小节。它们提供对上下文的更高层面讨论,但并不是本书的关注点。

除了上一节提到的 POSH,我还编写了一个随书的可移植软件库, Simple Audio Library(SAL),它说明了本书讨论的许多概念。适当的时候我会包含一些说明 SAL 或 POSH 实现的例子,以及它们如何与本书讨论的原则相关联。通过提供跨平台软件的基础部分的注释,我希望能够吸引那些通过例子比通过阅读学习得更快的人。

第1章 可移植性概念

在进入迁移的细节之前，我们需要退一步先关注可移植性本身的概念。确实，很容易就可以深入进去并开始展示一个特定的转换 Linux 软件到 Windows 机器的例子，但是那样我就只是解决了一个热点问题而已。

本章讨论可移植开发的许多方面——原则、技术、和可移植使用的过程。它们允许你编写的程序很容易就从一个平台迁移到另一个平台，而不管这两个环境的特定差异。在搞定这些基础知识后，我会解决特定的问题并讨论相应的解决办法。

可移植是一种思想状态

通常编程是一种充满各种状态的行为。你的大脑不停在编辑、编译、调试、优化、文档、和测试之间切换。你可能想把移植当作一个独立的阶段，就像编辑或调试那样，但是“可移植的思想”却不是一个步骤——它是思想的一种全封装状态，应该在程序员执行的每个步骤中执行。在你头脑的某个地方，在“使变量名更加有意义”和“不要硬编码这些常量”规则之间，“可移植思想”应该在任何时候都是激活的。

就像杂草慢慢占领整个花园一样，可移植性是一种可以影响开发过程中所有方面的习惯。如果编码是通过一种计算机懂得的语言强制计算机执行一组动作的过程，可移植软件开发就是避免对特定计算机产生任何依赖和假设，两者之间有间接但却明显的互相作用。在当前平台能够工作的需求，与让它也能工作在另一个平台是互相竞争的。

认识到迁移代码和编写可移植代码之间的区别是非常重要的。前者是挽救；后者是预防。如果让我选择的话，我宁愿程序员为了预防而不使用某个实践，而不是以后再来修正这个实践带来的副作用。这个“疫苗”就是强有力地执行可移植编码的实践习惯——深深嵌入到程序员的思想中，让程序员完全理解。

养成良好的可移植习惯

程序员最初开始进行可移植软件开发时，经常过于担心特定的技术或问题细节，但是经验告诉我们，通过习惯和哲学来鼓励，或者说强制可移植实践，能够让程序员更加容易地编写可移植代码。要养成良好的可移植习惯，你首先必须抵抗诱惑而不过于强调细节，例如字节序、或者对齐问题。

不管在理论上你拥有多少关于可移植的知识，通常迁移的实践都会证明你依然缺乏某些理论知识。按理说编写遵循标准的代码应该更加可移植，但是不经测试就做出这种假设是危险的，它很可能导致你遇到许多问题。例如，尽管 ANSI C 规范规定了特定的行为，但是很多不合作或充满 Bug 的编译器都不遵循这个标准。当你使用非标准的工具时，遵循标准的代码实际上并没有太大帮助。

一个典型的例子就是 Microsoft Visual C++ 6.0，它没有正确地支持 C++ 规范中 for 循环语句的变量作用域：

```
for (int i = 0; i < 100; i++)
;
i = 10; /* 在 MS VC++ 6.0 中，这个变量仍然存在...而在遵守标准的编译器中，
```


一旦 `for` 循环退出，这个变量就超出作用域，因此这一行会产生一个错误 `*/`

Microsoft 的开发人员在他们 C++ 编译器的 7.x 版本中“修正”了这个行为；但是这将导致与版本 6 代码的向后兼容问题，所以新版本编译器默认仍然不遵循标准。这意味着程序员在 Microsoft Visual C++ 上编写的可移植代码，只有当使用 GNU 编译器集(GCC)来编译时才会发现已经被破坏。

但是如果你实践了良好的可移植习惯，例如经常测试、在多个平台开发，类似的问题就能很早捕捉到。这能节省你很多麻烦，不用记住可能会遇到的所有特定 bug 和标准怪癖。

良好的习惯胜过了了解 Bug 和标准的特定知识

让我们来看看这些好习惯是什么。

尽早迁移、经常迁移(Port Early and Port Often)

没有代码在真正移植之前就是可移植的，因此尽早且经常迁移代码是非常有意义的。这避免了开发过程中一个经常犯的错误：先编写“可移植”代码，然后再对它进行测试，一次性找出所有可移植性相关的问题。通过尽早测试代码的可移植性，你可以在还有时间修正时捕获问题。

在多样的环境下开发

通过在多样的环境下进行开发，可以避免“编写然后迁移”这样的两步过程。这个习惯同时还能最小化基本代码衰退的危险，避免你项目的某部分得到发展，而其它部分却因为开发者的忽视而越来越糟糕。

例如你的项目最早可以运行在 Linux、Mac OS X、和 Windows 三个系统上，但是由于时间的压力，Linux 版本被弃用。六个月后，你需要重新使软件在 Linux 下工作，但是你发现已经做的许多改变无法应用到那个平台（例如条件编译指令）。

在多样环境中开发的第一步，是确保开发者在实践中使用尽可能多不同的系统和工具。如果你的项目最终会发布到 Sun Sparc 上的 Solaris、Intel 上的 Microsoft Windows、以及 Macintosh 上的 Mac OS X，那么确保团队成员使用这些混合系统作为主要的开发系统。如果不强制要求使用所有系统作为开发系统，则“先让它能够工作，我们稍后再来迁移”的思想就会悄悄生根。

即使是工作在类似的系统（例如都使用 Windows PC），使用不同的硬件（视频卡、CPU、声卡、网络适配器等）和软件也是一个好主意，这样可以更早地发现更多 Bug。这种方法可以防止报告 Bug 时开发者无奈的“它在我的机器上可以工作”的喊叫。

SAL 的例子：混合开发

我同时在以下平台开发 Simple Audio Library(SAL)：Windows XP 笔记本上使用 Microsoft Visual C++ 6.0、运行 OS X10.3 的 Apple G4 Power Mac(使用 XCode/GCC)、AMD Athlon XP 机器上的 Linux(ArkLinux，基于 Red Hat 的发行版)工作站(使用 GCC)、偶尔还使用 Pocket PC 上的 Microsoft 嵌入式 Visual C++。主要的开发工作在 Windows XP 机器上完成，每过几天我都会

对项目迁移并在其它平台进行验证。

偶尔我会使用其它编译器(例如 Metrowerks CodeWarrior 和 Visual C++ 7.x)来执行快速的验证和测试,有时候也会发现问题甚至 Bug。

SAL 的代码从未发生过太大的分歧和衰退。但是 SAL 相当晚才引入对 Pocket PC 的支持,由于 SAL 的许多假设对这个平台都是无效的,这带来了许多麻烦。例如测试程序依赖于 main() 函数就产生了一个问题,因为 Pocket PC 并没有控制台应用,因此必须提供一个 WinMain() 方法。其它问题则由于 Pocket PC 对宽字符串(国际化)的强调而产生。

我使用不同的底层 API 实现关键的特性,例如线程同步。我发现提取通用的抽象,然后把它们放到独立的抽象层,然后每个平台的实际实现进行分层。这意味着从 Win32 互斥量移植到 POSIX 线程(pthread)的互斥量是很容易实现的,因为 SAL 并没有被 Win32 特定的代码所捆绑。

使用各种编译器

你还应该尽可能多地使用不同的编译器。通常不同的宿主系统会有不同的编译器,但是有时候你可以在不同的系统上使用相同的编译器;例如 GCC 编译器在很多平台上都普遍存在。

通过在多个编译器中成功编译,你可以避免首选编译器厂商忽然消失时的进退两难。它同时还确保基本代码不会依赖于语言的新特性,以及编译器特定的特性。

在多个平台上测试

多数项目都被市场动态决定了一组完善的部署目标,这极大地简化了测试和质量保证,但是它同样也开始让你做出危险的隐式假设。即使你知道项目会运行在单一的目标系统,让它能够运行在替代平台也没有坏处——处理器、RAM、存储设备、操作系统等等——通过严格的测试。

然后如果你的目标系统由于市场的需求或者业务关系而发生了变化,你会因为你的软件没有硬编码到单一平台而高兴。

支持多个库

今天的软件开发大多是组合现有的代码,而不是重新编写新代码。如果你依赖于一组专有库或 API,则移植到新平台就会很困难。但是如果你早一点花时间支持多个备选库,来完成相同的任务。当某个库厂商破产或在另一个平台不可用时,你会拥有更多的选择。另外还有一个额外的好处,是可以对你的代码进行授权或者开源,而不用担心依赖于封闭源代码的第三方库。

一个典型的例子是支持 OpenGL 还是 Direct3D,这是当今两个主流的 3D 图形 API。OpenGL 跨平台并且在多个平台可用,包括所有主流 PC 操作系统;反过来 Direct3D 是 Windows 的官方 3D 图形 API,而且只能用于 Windows。这使得开发人员处于一个难办的处境:是专门为 Windows 进行优化,以支持这个世界上最大的用户市场;还是使用 OpenGL 一次性支持多个平台呢?

理想情况下,你应该抽象出图形层,这样它可以很好的支持这两组 API。这可能需要大

量的工作，因此在你开始动手前，最好仔细地考虑抽象可能导致的结果。但是当移植你的软件到新平台时，这个抽象带来的回报会是付出努力的许多倍。

为新项目计划可移植性

开始一个新项目的工作既令人乏味，又有一种纯粹的乐趣。当你创建一个新目录时会有一种类似于“新车异味”的感觉，新项目等待着你根据以前数年的经验来填充完美的源代码。

当你发现自己非常难得地处于一个清晰的起点，你就有机会来计划怎样使你的项目可移植。如果你开始动手之前能够稍微考虑多一点，就能省下你许多时间和麻烦。

使可移植更加容易

和其它许多好习惯一样，能够坚持好的可移植习惯与使用这些习惯的简单性成正比。如果开发方法学使可移植软件开发非常乏味和低效，那么在你到达里程碑之前就会抛弃它。

建立过程、库、和机制来帮助编写可移植代码是很重要的，这样可以使可移植软件开发更加自然，而不是一项持续艰苦的任务。例如程序员不应该亲自去处理字节序问题，除非他正在做的事情就是处于这个层次。

选择一个合理的可移植等级

尽管可以倾尽所能试图编写 100%可移植的代码，但从实践的角度来讲，如果不对软件特性集做出重大的牺牲，几乎是不可能达到这个目标的。

对于可移植性不可以教条主义！软件的可移植性应该刚好符合实践要求，而不需要更多。牺牲时间和精力来保证不重要工具的可移植性，就如同花费一个星期来优化一个仅仅被调用一次的例程。这不是一个花费时间的有效途径。

这就是为什么建立一个切实可行的基础原则（一组定义合理可移植性的规则）对项目是至关重要的。没有这些原则，项目就注定难逃一死。设计代码能够运行在任何地方会很糟糕。

每个平台都有自己机器相关的特性：编译器、工具、预处理器、硬件、操作系统等等。把程序从一个平台移植到另一个平台时，有无数的方式可以使程序崩溃。幸运的是，平台的许多特性是共享的，这减轻了编写可移植软件的难度。定义基本的可移植底线，是设计和编写可移植代码的第一步。

正如我在第 14 章将要说的那样，可移植性的很大一部分都与伸缩性（系统运行时体现很大的性能和特性差异）有关。伸缩性是重要的，但是它必须被很好地定义，来保留表面的相关性。

除了原始的特性，你还必须对平台的底层性能做出假设。完全有可能编写出在 8MHz Atmel AVR 微控制器和 3.2GHz Intel Pentium 4 处理器都能编译通过的软件，但是在这两个平台上这样做是没有意义的。PC 工作站所使用的算法和数据结构，和嵌入式微控制器是完全不同的，而且限制高性能的机器运行在完全不同的架构上也是低效的。

案例学习：浮点运算

ANSI C 语言支持浮点数，分别使用 `float` 和 `double` 关键字，以及相关的数学操作符，来

操作单精度和双精度浮点数。大多数程序员都假设系统能够提供支持。不幸的是，今天的某些设备，以及不久之前的很多设备，都缺乏对浮点运算的支持。例如，多数个人数字终端(PDA)所使用的处理器都不能在本地执行浮点指令，因此必须使用极其缓慢的模拟库。

对于某些特定的项目，即使是非常缓慢的浮点运算也是完全可以接受的，因为它可能很少使用浮点运算（尽管在这种情况下，如果浮点运算模拟库必须链接进来，可执行文件会变得更大）。但是对于那些要求强劲浮点运算性能的项目，如果移植到一个没有内部浮点支持的系统，事情将会马上变得很糟糕。

解决这个问题常用的办法是使用特殊的宏，来编写所有数学操作，在没有本地浮点支持的设备上，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 */
```

然后三元点乘就可以像下面这样编写：

```
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);
}
```

不过纯净的浮点版本更加容易阅读和理解：

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

如果你必须支持没有浮点能力的系统，或者你觉得将来很有可能需要支持，则使用上面宏就是一个好主意。但是如果你有能力要求系统必须支持本地浮点，你就能提高代码的可读性，编写的代码也更加简洁。

可移植性是一个好主意，编写可移植代码也是好的实践，但是如果你走极端或者编写过分移植的代码，仅仅为了符合教条主义，则你的代码最终会失败。可移植性是达到目的的手段，而不是一个单纯的目的。

一个网络应用如果断定自己的架构是低延迟、高带宽通信，则当它面对一个调制解调器

时将灾难般地失败。由于这个应用对网络的基本假设，即使这个应用可以在任何平台编译和运行，实际上它并不能移植到某些网络。

建立一个底线是可移植软件开发的关键元素，因为它创建了一些完全合理的假设，允许实际开发的软件只在有限的平台中有效。

分散可移植和足够可移植之间是有区别的。如果你的项目只针对一个目标平台，但是你知道你可能今后需要改变编译器，然后关注怎样保持代码在不同编译器之间可移植，而不需要过于担心那些你根本不会支持的目标系统。

不要把你的项目绑定在私有产品上

现代软件开发是非常复杂的，即使是简单的产品也可能包含数万行源代码。这种复杂性通常需要使用良好测试和文档化的第三方组件，例如库和工具。使用已经存在的组件能够节省时间，但是它同时也引入了许多新的可移植问题。确保你自己软件的可移植性已经很困难很费时间了，当你再引入外部影响时，它可能会成为一件彻底艰难的任务。每次外部组件整合到项目时，灵活性和控制力都大大降低了。

即使是在最好的情况下（开源库），你也必须确认这些代码能够编译和运行在你需要运行的所有平台。如果你需要支持的某个平台，是由开源库的贡献者提供支持的，则你需要自己完成迁移工作（由于开源的本性，这也是一种选择）。

不幸的是，使用闭源私有库将失去这一选择。在这种情况下，如果提供者不支持你需要的平台（例如厂商关门），你可能发现自己就像是在角落里作画。最坏的情况下，你必须为新平台从头重新实现第三方组件。把你的项目绑定到第三方组件，从长远来看是非常危险的。许多经验丰富的人都能列举出许多类似的故事，项目无情地绑定到孤立的库或工具，最终这种绑定影响了项目的整个开发过程。

例如许多 PC 开发者都使用过 Microsoft 的 DirectPlay 网络库，因为它是免费可用的，并且它声称提供大量需要长时间才能开发出来的特性。The low-hanging fruit of free and easy technology beckons，但是当试图迁移到非 Microsoft 平台，例如 Macintosh 或者游戏控制台时，整个过程就会变得一团糟。人们经常发现自己需要重新编写整个网络层，来弥补自己犯下的绑定私有技术的过错。

SAL 例子：底线和使用私有 API

SAL 拥有适度的特性和性能底线，它使用 ANSI C89 编写（除了一个文件是用 Objective-C 编写，只使用在 Mac OS X 平台），这使得 SAL 拥有最广泛平台支持的可能性。两个最关键的技术组件是混合器和假想的线程模型。

混合器基于整数并假设 32 位整数操作，特别是乘法，相对来说会比较快速。由于这个原因，它可能在 16 位平台(例如 Palm OS 4)将不能很好地工作。

然而两个关键参数（同步声音的最大数目和缓冲区长度）都是运行时由用户定义的，这使得 SAL 拥有跨越很多系统的能力。对于特别缓慢的系统，可以增大缓冲区长度，代价则是更大的延迟。此外，可以降低活动声音的数目来最小化内部混合循环的工作量；高性能系统可以轻松处理 128 声道，但是在低性能机器上你也可以按低配模式操作 SAL。

SAL 的基本实现模型创建了独立的线程负责从活动声道获取数据。这是一个概念模型，用在 SAL 的多数平台中。然而，至少有一个平台（OS X）使用基于 CoreAudio Sound API 的回调机制。（回调是在另一个线程中调用，所以技术上讲也使用了另一个线程，但是 SAL 没有

创建线程)。无论 audio 系统怎样产生混合数据，底层假设仍然是异步发生的，因此需要提供同步保护（使用 mutex）。简单的单线程操作系统（例如 Mac OS 9 或 Microsoft MS-DOS）理论上也能够支持，但是这需要非常小心地计划，因为这些平台架构使用中断来驱动声音系统。

SAL 的某些部分需要常驻的可用内存，尽管需要的内存并不过分（相对 PC 来说）——大概几百 KB。pulse code modulation(PCM)示例回放的实现假设 PCM 数据是常驻的；但是也可以使用声音流代替，因此大量地降低了内存要求。这需要应用程序员更多的工作，不过提供了另一个选择。

SAL 的核心实现需要 C 标准运行库(free, malloc, vsnprintf, memset, fprintf 等等)。但是它也可以很好地运行在没有 C 运行库或操作系统的环境下，只需要很小的修改（主要是提供 vsnprintf 和 memset 的替代物）。

SAL 的核心代码不使用任何私有 API，不过它确实使用了几个平台特定的 API（Win32, pthreads, CoreAudio, Cocoa, OSS, ALSA 等等）来实现部分架构。内部的 SAL 特定 API 架构在这些 API 的上层。例如，_SAL_lock_mutex()在 Win32 上调用 WaitForSingleObject()，在 Linux 上调用 pthread_mutex_lock()。

在这样的底线上，SAL 能够移植到所有平台，不过这也因为库通常都比应用程序更容易移植。

如果你发现自己在核心模块使用了第三方组件，你应该至少把它抽象出一个层次，这样替换或修改它对项目的其它部分的影响就不会那么大。如果你发现自己的项目必须使用私有工具或者库，那么就获得完全源代码授权，或者至少确保厂商把源代码提交到第三方中介，保证你不会被厂商倒闭所影响。

移植旧代码

不幸地是，现实很少让我们能够工作在全新的项目。在许多情况下，我们必须处理不是我们自己的错的可移植性问题，因为我们需要迁移别人的不可移植代码到另一个平台。

这种情况下，有一些通用的指导方针和规则可以帮助你管理迁移过程。

在代码成功迁移之前假设代码是不可移植的

许多程序员认为自己很擅长编写可移植代码，他们中的许多人也确实是这样。但问题是程序员经常声称自己的代码是可移植的，能够在新平台“编译即可运行”。遗憾的是事实很少如此。无论对代码的可移植性做出多少保证，你总是应该假设它不可移植。直到代码被移植到新平台并且通过测试，必须十分小心地对待代码。

在 SAL 的开发过程中，每次我在另一个系统或者切换编译器来重新编译代码，我都会遇到可移植问题。有太多的细节需要我们记住。迁移代码的过程是代码可移植性的试金石。

当然，你可以声称你的代码是“对可移植友好的”，这是对那些编写时考虑了可移植性，但尚未被移植到特定平台的代码的合理描述。被证明不可移植的代码（你尝试移植它但失败了），和不确定是否可移植的代码（尚未尝试移植），以及被证明可移植的代码（已经被移植到新平台），这三者是有很大区别的。

仅仅修改最基本的部分

移植软件需要大量的编辑和重构，因此有引入新 **bug**，以及在原有平台上破坏软件的可能性。各种因素可能诱惑程序员清理与移植无关的代码，你应该尽可能避免这样。保持一个清洁的代码库，为回归测试提供了可靠的基础。

修改一个已经在某个平台可用的大型代码库，需要相当的技巧。每次你编辑一个文件，都有可能破坏其它地方。由于这个原因，我们把“别碰你不需要碰的任何东西”原则扩展为包含“找到最小阻力的途径”。

这意味着理想情况下，软件实现应该会有一些的逻辑划分，允许你清晰地区分新代码和原有旧代码。如果你能够找到这条线，将会使移植容易许多，因为你能够在新代码和“处女”代码之间自由地来回切换。

计划你的修改

在你编写或者更改任何一行代码之前，你必须先完全理解自己正在做什么。移植软件和编写新软件是完全不同的，你使用的方法也完全不一样。识别最有可能的可移植性热点，这样你就清楚地知道移植软件到新平台需要做什么工作了。一旦你有了这样的列表，就可以坐下来好好计划移植软件的过程。

例如，把一个应用从 **Windows** 移植到 **Linux**，可能会有下面这样一个列表（非常笼统）：

- 移除所有 **Windows** 特定的头文件引用。
- 更新 **Windows** 文件处理为 **Unix** 风格的文件函数。
- 隔离并更新所有的 **CreateWindow** 路径。
- 更新资源装载为 **Unix** 风格的文件访问例程。
- 使用基于本地文件的参数配置替换原有的注册表代码。

通过预先准备一份完善的列表，你可以预测在移植过程中可能会遇到的问题，并相应地做出修改计划。

很多时候我们的第一反应是把软件拿到新平台，马上开始编译代码，并修改编译器和链接器捕获的错误。一旦程序通过编译和链接，你就开始在调试器中运行程序，直到它能够正常工作。但这并不是一种处理移植过程的高效方式。你需要识别出所有需要首先处理的区域，以避免过于深入而妨碍那些必须推迟执行的工作。

例如，在稍后另一个可移植性问题出现之前，你可能不会意识到对先前那个可移植性问题的修复方法是不切实际的。现在你必须退回原来的地方，撤销所有修改，并且想出一个同时解决这两个可移植性问题的办法。

在版本控制系统中记录所有事情

在这里我重申一遍：你做出的任何修改都可能是毁灭性的。因此你必须记录所有修改。

任何需要很长开发时间的复杂软件，都必须使用版本控制系统。然而在移植过程中，使用版本控制系统甚至更加重要，因为每个修改都可能微妙地破坏与当前工作无关的部分，如果你拥有清晰的修订日志，识别出什么时候产生破坏就会容易许多。

当开始移植时，开发者非常强烈地希望软件能够尽可能快地工作于新平台。但是如果你毫无计划地跳进移植任务，你可能会浪费大量时间走向死胡同，并不得不撤销之前的工作。

第2章 ANSI C 和 C++

影响可移植性最基本的选择之一就是编程语言。本书使用 **ANSI C** 和 **C++** 作为演示语言，因为它们使用广泛且促成了许多可移植的问题（没有它们本书就不可能会有市场）。**ANSI C** 和 **C++** 可能是最不可移植的语言，而又仍然被要求可移植。

为什么不是其它语言？

既然 **C** 和 **C++** 有如此多的可移植问题（我们后面会具体讲解），看起来我们应该选择更高级的语言，这样就能避免许多跨平台问题。确实，今天的许多应用可以也应该使用更高级的语言来编写。但是仍然有很多情况下 **C** 和 **C++** 明显是最佳（或者唯一）的开发选择。为什么呢？

C 和 **C++** 提供底层访问

C 编程语言最早设计是为了使 **Unix** 操作系统脱离特定的硬件体系，因此 **C** 被认为是第一个重要的可移植计算机编程语言。再加上标准组织（如 **ANSI**、**ISO** 和 **IEC**）的认可，以及广泛可用的文档、库、编译器、和其它开发工具，使得 **C**（以及后来的 **C++**）成为最流行的跨平台、高性能应用的开发语言。

当你需要处理硬件级的问题，而又不想求助于汇编语言时，**C** 和 **C++** 是通常是标准的实现语言。程序员使用更高级的语言，很难直接处理内存地址、端口、或者 **DMA** 缓冲区；事实上，这些语言通常都反对并阻止这些操作。

C 和 **C++** 编译成本地代码

多数 **C** 和 **C++** 实现都生成本地二进制代码，这对于嵌入式系统，或者对代码大小和性能要求很高的场合是必须的。

今天的许多语言会编译成通用的中间代码形式（例如 **Java** 字节码或之前的 **Pascal p-codes**），这是一种解决可移植问题的绝佳方法。不幸的是，这是以严重增加代码大小和执行时间为代价的。此外，这种通用的代码还必须被虚拟机或运行时环境执行。每个虚拟机或运行时环境的实现，都可能会有微妙的 **bug** 或者实现差别，导致可移植程序仍然面临平台依赖的问题。

非常高级的语言在适当的问题领域，可以使编写可移植软件比 **C** 或 **C++** 容易许多。但是提供硬件访问和高性能的低级语言，永远都会有自己的一席之地。这些低级语言也需要支持可移植性。

C 和 **C++** 方言

C 编程语言的种子在 1970 年代早期萌芽于贝尔实验室。那个时候，贝尔实验室正在开发一个叫做 **Unix** 的新操作系统，当时他们需要一种高级语言来开发 **Unix**。

包括 Ken Thompson、Brian Kernighan 和 Dennis Ritchie 在内的研究人员习惯于使用 B、BCPL、PL/I 和 Algol-60 语言。但是这些语言由于各种技术原因，最终都不适合他们当时使用的目标机器（DEC PDP-7 和 PDP-11）。

于是研究人员以 B 语言为起点，并扩展 B 语言提供额外的特性，以适应他们工作的需要，很快 C 语言就产生了。然后 Unix 内核使用 C 语言进行了重写。

到 1978 年，C 语言已经成长得非常流行，Kernighan 和 Ritchie 出版了他们著名的《C 编程语言》一书。这本书成为了 C 语言当时的事实标准。

在接下来的十年中，C 语言继续进化，使得 Kernighan 和 Ritchie 建立的标准迅速过时。很明显此时 C 语言需要一个正式的标准，来统一不同的实现。另外一个不太明显的理由是很多组织和政府拒绝采用一个缺乏正式标准的计算机语言。

由于这些原因，在 1980 年代早期，ANSI X3J11 委员会正式起草和成立一个标准。这个工作最终在 1989 年 ANSI X3.159-1989 C Programming Language 中完成。这份文档也被 ISO 采纳为 ISO/IEC 9899:1990，ANSI C 实际上等同于 ISO C。

又过了十年，标准被修正和更新来反映最近十年语言的变化，C99，也就是 ISO/IEC 9899:1999 诞生了。

当然，在 C 进化的过程中，其它语言（通常都源自 C）也开始出现，其中最引人注目的是 C++。C++ 开始于 Bjarne Stroustrup 在 1980 早期的实验，当时他对 Simula 等面向对象语言的能力，以及这些语言怎样应用到系统软件开发非常好奇。很 C 语言一样，C++ 语言的标准化的过程也有一点随意。

在 1985 年，Stroustrup 出版了《C++ 编程语言》一书，类似于 Kernighan 和 Ritchie 所做的工作，这本书提供了 AT&T 最初商业实现的基本介绍。随着 C++ 越来越流行，Stroustrup 和 Margaret Elis 在 1990 年发表了 Annotated Reference Manual(ARM)，作为 Stroustrup 早期工作的必要更新，更重要的是作为 C++ 语言的一本完全参考手册，而不仅仅是某个特定实现。

ARM 成为 ANSI 委员会的基础文档，正式修订成为 C++ 标准 XJ316。标准最终修订为 ISO/IEC 14822:1998。（注意这刚好在 C99 标准之前一年，这意味着在 C99 和 C++98 的某些地方，存在微小但繁琐的区别）。

本书关注于 C 语言的 1989 ISO/IEC/ANSI 9899:1999 规范，后面我将引用为 C89，因为这是今天最广泛的支持标准。多数特定于 C99 规范的细节将在附栏中讲解。当说明特定的语言无关的问题时，我可能也偶尔会涉及到 C++98。我倾向于 C 是因为它使用更广泛，更清晰和简洁。此外，C++ 仍然是一种快速发展和进化的语言，对于各种实现之间不统一的讨论，会使本书成为书架上的荒废物。

本书不关注 ANSI C 之前(K&R C)的可移植性问题。由于缺乏正式标准，各种问题使得对于它的讨论不可能完成。

可移植性和 C/C++

理论上的可移植性和实际的可移植性有巨大的差距。使用 C/C++ 时很容易就会执著于语言细节，而不是如何编写可移植软件。前者强调语言本身，关注某个特定的结构是否可移植、是否合法、是否未定义、或者是否实现相关。但是当使用某种语言编写可移植软件时，需要关注体系架构、设计、以及各种权衡。理解特定语言的细节只是整个过程的一部分。

这里并不是对深入理解语言进行诋毁，但是意识到你是在编写可以工作的软件，而不是完美可移植的软件，这一点非常重要。可移植性是一个目标，而不是用来套牢你自己。

我着手于编写一本关于使用 C 和 C++ 语言编写可移植软件的书，而不是如何编写可移植 C/C++ 代码的书，这是一个微妙却重要的区别。很多时候开发者可能会编写技术上不可移植

的代码，但这在现实世界是不可避免的。只要你知道并记录下所有不可移植的假设，这也是可以接受的。事实上由于那些不在开发者控制范围内的 bug 存在，即使是“100%可移植”的程序（完全遵循语言标准），在不同的平台也仍然可能会失败。这也是为什么强调结构、设计和过程，比关注语言细节更为重要的原因。

举个例子，把一个函数指针强制转换为其它类型，例如对象或整型指针，是被 ANSI 标准严格禁止的，因为其它类型可能不够大，无法安全地存储一个函数指针。

下面是代码：

```
#include <stdio.h>

void foo(void)
{
    printf("Hello world!\n");
}

int main(int argc, char *argv[])
{
    uint32_t f = (uint32_t) foo;    /* not guaranteed to work! */
    void (*fnc)() = (void (*)( )) f; /* also not guaranteed to work */

    /* the following works as you expect on most platforms, even
       though it's technically "bad" */
    fnc();

    return 0;
}
```

但是这种类型转换在很多实际的代码中都存在（甚至是很多可移植的代码）。许多事件或消息系统都使用通用的参数域，由指针数据填充。例如 Win32，WM_TIMER 消息指定 MSG 结构体的 LPARAM 域指向用户定义的回调函数（由 SetTimer() 指定）。它确实是 Win32 API 的一部分，而且即使是在 Windows 特定平台，技术上讲也是不可移植的。（“正确”的实现方式应该是使用联合作为 MSG 结构体的域，确保结构体所有域有正确的大小和对齐）。

dlsym API 返回一个 void 指针，根据不同的调用，它可能指向数据或者代码。从技术上讲这也可能失败，但是 X/Open 系统接口(XSI)（它管理着 dlfcn API），警告说任何指向数据的指针都必须同时能够包含一个代码指针，整洁地避免了这个问题。这里就使用了我们的第一个可移植规则：把隐式假设转换为显式需求。

因此，技术上讲语言标准可能规定你不应该做什么，实用主义可能却需要你使用某些不可移植的特性，即使你知道这可能会破坏你的代码。

虽然 C 语言的根本是为了提供可移植性，但是 C 语言却充斥着各种可移植问题。其中有一些是为了突破严格的标准限制，另外一些则是为了使 C 语言更适合编写系统软件。不过即使是这样，ANSI C（以及 ANSI C++）不可否认是编写高性能、紧凑、可移植代码的最佳选择，因此本书将关注于 ANSI C 和 C++ 语言。

第3章 可移植的技术

编写本书的目的之一，是我遇到许多程序员都使用非常类似，但却没有文档化的技术来解决可移植性问题。这些开发者要么各自发现了这些方法，要么通过口头相传，因为没有书籍关注可移植软件开发主题——这是计算机文献的显著遗漏。

本章组成了本书的骨架，它列出了当今可移植软件开发通常使用的各种实践和模式。

避免新特性

无论你使用什么语言，应该避免使用新的或者实验性的语言特性和库。新特性的支持不确定且很可能有 bug。即使有广泛的支持，经常也会有尚未正确测试或精确定义的死角。

典型的例子包括 C++ 模板、命名空间、以及异常处理的早期实现。在这之前，从早期 C 到 ANSI C 的发展也经历了很长时间，因为许多平台在很长一段时间内都没有完全遵循 ANSI C 标准的编译器。另外一些情况下，特性的讨论到广泛实现，时间甚至可能需要五年以上。

经验告诉我们一条准则：如果一个新特性已经被实现五年以上了，依赖于它可能是安全的。即使到我写这本书的今天，C99 规范已经出来将近五年了，仍然还没有完全符合 C99 标准的编译器存在。

处理不同平台可用特性的差异

在软件开发的世界里，每次需要编写在各种环境下可移植的库时，开发者都不可避免地面临怎样处理不同平台的特性差异问题。例如 Mac OS X/Cocoa 的 column-view 数据浏览器、Windows 的树形控件和递归 Mutex、DOS 对线程支持的缺乏、从及早期 Mac OS 对透明虚拟内存支持的缺乏。这些都是特定平台上某些特性可用或缺失的典型例子，也是跨平台库和应用需要解决的问题。

一个办法是将抽象直接映射到每个可用平台的具体实现上去。例如，平台 A 的特性 X 就直接使用平台 A 对特性 X 的具体实现。

但是当平台 A 不支持特性 X 时怎么办呢？应用需要查询特定的实现是否可用，并且在不可用时避免使用该特性。然而这种办法有无数的缺点，例如导致应用代码的回旋条件判断、当依赖的特性突然不可用时很差的健壮性等等。下面代码片断说明了这种方法：

```
api_feature features;
api_get_features(&features);
if (features.feature_x_present)
{
    /* Do things one way */
}
else if (features.feature_y_present)
{
    /* Do things a different way */
}
else if (features.feature_z_present)
{
```

```
        /* Do things yet another way */
    }
    else
    {
        /* Don't do anything at all, possibly warning user */
    }
}
```

这个例子说明了当软件需要处理大量可变特性时，将会遇到条件执行语句的梦魇。

SAL 的例子：仿真与必需特性

SAL 在两个关键领域对底层实现做了假设：递归 Mutex 和音频混合。

递归 Mutex 是一种特殊的 mutex（线程同步原语），它可以被同一个线程锁定多次而不死锁。Windows 默认就有递归 Mutex，Linux 的 pthreads 把它作为一种 mutex 来实现。OS X 则通过高级的 Cocoa NSRecursiveLock 类来提供这一机制。

我可以选择另外一种方式：在非递归 Mutex 之上实现我自己的递归 Mutex。这样可以获得一点通用性，更直接一点的好处是可以在 OS X 和 Linux 上共用同一个实现。但是这需要额外的测试，SAL 本身也需要提供一个比较大的实现，所以我没有这样做。如果 SAL 需要支持没有递归 mutex 的平台，我可能最终会在模拟层来实现它们，而仅仅依赖底层平台的原始 mutex。

音频混合有时候以操作系统服务来执行（waveOut 或 DirectSound）；或者在应用层使用独立的音频库（例如 SDL_Mixer）。由于混合作为底层数字音频子系统的本地组成，并不是在所有平台都可用，我的直觉告诉我在 SAL 内部实现它是至关重要的，因为它是一个非常重要的特性。

重复发明这个轮子的缺点实际上并不是代码膨胀，SAL 如果要抽象和支持每个平台的混合机制，它的代码量可能会更多。真正的问题是 SAL 无法利用加速混合，有一些操作系统和库以 Single Instruction Multiple Data（SIMD）的形式提供加速混合过程；一些音频设备也拥有硬件加速混合，能够完全解除 CPU 的负担。但是抽象平台相关的混合所获得的加速，和相应带来的复杂度相比，可能完全不值得，特别是考虑到许多实现本身并不可靠。许多 DirectSound 设备在使用硬件加速混合时都非常地不可靠。

SAL 选择了简单的方法，只要它能够把某种格式的单一缓冲区发送给硬件，就万事 OK，这样就完全屏蔽了混合的平台相关性。

案例学习：DIRECT3D 和 OPENGL

在 1990 年代末，在两个相互竞争的 3D 图形 API 之间曾爆发了一场激烈的对抗：OpenGL 和 Direct3D。OpenGL 在 Unix 工作站市场上已经有很长的历史，并且被很好的设计和文档化。Direct3D 则更加地专一和目光短浅，基于 PC 的理解和对支持用户级 3D 图形需要的一种反应，它天生就比为工作站设计的 OpenGL 更局限。

OpenGL 对特性有一个非常严格和清晰的政策：任何平台必须完全遵守 OpenGL 标准，即使这个平台不能提供所有特性的最佳支持。例如，如果一个特定的视频卡不能支持 texture-mapped 图形，应用请求 texture-mapped 输出时它仍然必须提供——意味着它必须完全在软件上模拟该特性。

OpenGL 把这个设计哲学扩展得更广，它拒绝暴露平台的底层特性集。理论上这强制程序

员编写“能够工作”的高度可移植代码，将兼容性和性能的责任转移到了每个 OpenGL 实现者的身上。

正如你所猜测，OpenGL 采用的方法在实践中要比理论上差很远。开发者一旦启用诸如 texture-mapping 的特性，然后就发现应用损失了 95% 的性能，这种情况比比皆是。（这并没有夸张：帧速从 60Hz 一下子掉到 1Hz 并不少见。）用户可能会大声抱怨，然后开发者在某台特定的机器上花费一两天的时间跟踪问题。最后，在程序中插入某种检测代码，查询 OpenGL 实现的 GL_RENDERER_STRING，来禁用特定平台的所有缓慢特性，像下面这样：

```
if (!strcmp(glGetString(GL_RENDERER),
    "That Bad Hardware Device Rev 3.0"))
{
    feature.use_texture_mapping = 0;
}
```

微软的 Direct3D 采用了完全不同的策略来解决这个问题，它通过一个被称为 capability bits 或者 cap bits 的查询机制暴露每个硬件实现的底层能力。程序在对可用特性集做出任何假设之前，必须查询平台的每个单独能力。

不幸的是，Direct3D 的方法在实践中也不能很好的工作。例如，驱动程序经常公布一些它们并没有很好地执行加速的能力，因此即使一个视频卡说“嗯，我支持 texture mapping”，也不能保证它的 texture-mapping 性能就是足够的。有时候能力是相互排斥的，这是一种 capability 位不能转移的概念。有些硬件声称同时支持 trilinear texture mapping 和 dual texture mapping，但实际上你同时只能启用其中一个。最令人沮丧的问题可能是硬件提供了独特的功能，而 DirectX API 并不能显示出来（OpenGL 通过扩展机制明确地支持额外功能），这让开发者和硬件厂商都感到挫折。改革和创新是没有任何奖励的，因为微软完全控制了哪些特性对开发者是可见的。

OpenGL 从确保的基准上允许扩展，而 DirectX 只在有限的特性集上允许变化。因此 OpenGL 被认为比 DirectX 更大地富于改革和创新，DirectX 则完全取决于微软。

今天，这两个 API 平静的共存着。当数十种不同性能和特性的硬件相互竞争了将近 10 年之后，现代的 3D 加速硬件已经不再充满了各种混乱的实现。Direct3D API 凭借着稳定的市场逐渐成为标准，但是它离完美还差得太远。微软的态度是，随着每次硬件的更新换代，完全地重写 Direct3D。这是一种权宜之计的现实做法，当然每个新一代的 3D 图形编程，开发者都必须重新学习。

哪种方式更好完全是哲学问题，而且双方阵营都有反对和诋毁者。在本书写作的时候，DirectX 已经更新到版本 9，拥有了非常良好和稳定的市场。开发者依然需要处理硬件加速器能力，以及驱动的 bug 和限制，但是 API 本身已经相对稳定。OpenGL 在 PC 上正在缓慢的死亡，但这主要是政策而不是技术原因。因此 DirectX 是 Windows 平台上的主流支配地位的 API，而 OpenGL 仍然是跨平台 3D 图形 API 的一个选择（它用在 Mac OS X 和几乎所有主流的 Unix 变种）。

另外一种方法是完全拒绝使用不是在所有平台都可用的特性，这能极大地简化开发，但是当该特性是某个平台的关键技术时，你的软件在那个平台将变得不够高效（更不用说无法支持某些平台独有的特色功能的杯具了）。

再另外一种方法则是尽可能多地划分出各个平台特色的区别并完全自己实现，这能有效地替换某个特性的任何本地实现。例如，跨平台图形用户界面（GUI）系统可以避开所有本地“widget”功能，并在内部处理所有的窗口管理。这样就能够解脱用户对各个平台特性差异的担心，不过这种方法同样也有缺点。

采用完全模拟来替换本地实现的第一个缺点是性能。如果可移植软件在内部处理了所有的细节，则丧失了使用软件或硬件加速的机会。另外一个缺点是开发者需要在各个平台重新实现大量的现有代码，这导致了重复劳动，需要调试更多的代码，一般来说还会有更多的工作需要完全。最后，模拟实现在特定平台可能会缺乏期望特性和 look and feel。

折中的办法是仅仅当特性缺乏时才模拟，其它情况下则使用本地实现。这看上去在多数情况下都是最佳选择（例如，要在 X Windows 系统下模拟 Windows 树形控件并不需要太多的工作），但是有时候这种方法非常地不现实。例如启用 OpenGL 的一个完全软件实现的非加速特性，而跳过任何可用的硬件加速机制，通常都会导致极度的缓慢。

使用安全的序列化和反序列化

跨平台最常见的问题是如何以一种安全、有效的方式来保存（序列化）和装载（反序列化）数据。当你工作在单一的编译器和目标环境中时，可以简单地使用 `fwrite/fread`，但是在跨平台的世界里，这却是完全错误的，特别是当存储目标不是文件的时候，例如网络缓冲区。

一个可移植的实现总是将序列化划分为两个部分：第一部分是底层平台的内存数据转换成中间格式。这种转换过程必须是 100% 可移植的。例如，假设你要序列化一个用户记录：

```
#define MAX_USER_NAME 30
typedef struct user_record
{
    char name[MAX_USER_NAME];
    int16_t priv;
} user_record;

void serialize_user_record(const user_record *ur, void *dst_bytes)
{
    /* production code should have a size tied to
       dst_bytes to prevent buffer overruns */
    /* this section has been omitted for brevity */
    uint8_t *dst = (uint8_t *) dst_bytes;
    memcpy(dst, ur->name, MAX_USER_NAME);
    dst += MAX_USER_NAME;
    dst[0] = (uint8_t) ((ur->priv >> 8) & 0xFF);
    dst[1] = (uint8_t) (ur->priv & 0xFF);
}
```

NOTE 用户记录序列化的完全实现应该包含缓冲区溢出检查，以及序列化为文本格式便于打印和人工编辑。

`serialize_user_record` 的代码在所有的平台下，都以统一的方式将用户记录复制到 `dst_bytes` 中。在 PowerPC 上的 Yellow Dog Linux 系统中序列化的 `user_record`，和 Intel xScale 上的 Windows Pocket PC 2003 系统中序列化的结果都是一样的。

如果采用以下方法实现序列化，那结果将不再正确：

```

void serialize_user_record(const user_record *ur, void *dst_bytes)
{
    /* DNAGER! DANGER! */
    memcpy(dst_bytes, ur, sizeof(*ur));
}

```

正如我将在后面章节中讲到的那样，内存中的原始结构可能会有不同的对齐、字节序、和填充属性，因此上面这个实现在不同的编译器和执行平台上，有时候能够正常工作，有时候又会失败。不幸的是用 `memcpy` 来实现的简单性在你只想尽快完成工作时太具有诱惑力了。

实现序列化接口有许多不同的方法，例如一个常见的方法就是多重继承，类继承自输入和输出基类，这是一种侵入式设计，需要修改类的基本继承体系，因此许多程序员并不愿意这样做。

一旦数据被可移植地格式化到缓冲区中，你就可以将它保存到最终目标去。例如要存储到磁盘中，你可能会使用 `fwrite`；如果要在网络中广播，你可能会调用 `send` 或 `sendto`。在最终交付数据之前也可以进行更多的处理步骤，例如压缩和加密。

反序列化以同样但相反的方式进行：

```

void deserialize_user_record(user_record **ur,
                             const void *src_bytes)
{
    /* production code should have a size tied to src_bytes
       to prevent buffer overruns */
    /* this has been omitted for brevity */
    const uint8_t *src = (const uint8_t *) src_bytes;
    *ur = (user_record *) malloc(sizeof(user_record));
    memcpy((*ur)->name, src, MAX_USER_NAME);
    src += MAX_USER_NAME;
    (*ur)->priv = (((uint16_t) src[0] << 8) | src[1]);
}

```

同样捷径也总是在诱惑着你，那就是通过指针和 `memcpy` 直接进行结构体复制：

```

void deserialize_user_record(user_record **ur,
                             const void *src_bytes)
{
    /* production code should have a size tied to src_bytes
       to prevent buffer overruns */
    /* this has been omitted for brevity */
    *ur = (user_record *) malloc(sizeof(user_record));
    **ur = *(user_record *) src_bytes; /* ouch, bad! */
}

```

和序列化的例子一样，上面代码假设 `user_record` 和 `src_bytes` 的内存格式相同，由于对齐、大小、和填充等因素，这种假设不能保证总是正确的。

当装载操作的性能要求很高时，“我将保证档案的格式和内存格式所有字节都相同”，并直接装载之前存储的内存结构体这种做法也是合理的。如果你能强制执行适当的预处理（数据是平台相关的）并加以适当的运行时检查，则可移植性将是数据驱动而不是代码驱动的，这样就能达到更高的性能而不损失实际的可移植性。

集成测试

移植软件涉及到修改和编写大量代码，这些代码在相当长的一段时间内并不能在另一个平台中得到及时的测试，这意味着许多 bug 会有很长的潜伏期。由于这个原因，执行标准化测试来尽可能早地捕获 bug 就至关重要了。

单元测试是小块的测试代码，它用已知数据执行函数或子系统，来确保所有行为都是预期的。单元测试应该能够立即捕获你在开发过程无意中引入的 bug。它也是防止你的函数在 Windows 下能够工作而在 Mac 中失败的第一道防线。编写几乎永远不会失败的测试案例看上去有点无聊且浪费时间，但是它们仅有的几次失败，你会很乐意你编写了这些测试。

POSH 例子：字节序测试

The Portable Open Source Harness(POSH)库试图在编译时通过检验一组变量来确定目标机器的字节序，这个检测是完全不允许出错的。如果 POSH 检测的底层机器使用的字节序也实际字节序不一样，那继续执行操作很快就将出错，因为你的软件会认为自己运行在错误的机器上。

由于这个原因，POSH 对于字节序有一组快速而健全的检测：

```
/* This is taken from posh.c */
/* POSH_LITTLE_ENDIAN is defined during compilation by posh.h */
#ifdef POSH_LITTLE_ENDIAN
#define IS_BIG_ENDIAN 0
#define NATIVE16 POSH_LittleU16
#define NATIVE32 POSH_LittleU32
#define NATIVE64 POSH_LittleU64
#define FOREIGN16 POSH_BigU16
#define FOREIGN32 POSH_BigU32
#define FOREIGN64 POSH_BigU64
#else
#define IS_BIG_ENDIAN 1
#define NATIVE16 POSH_BigU16
#define NATIVE32 POSH_BigU32
#define NATIVE64 POSH_BigU64
#define FOREIGN16 POSH_LittleU16
#define FOREIGN32 POSH_LittleU32
#define FOREIGN64 POSH_LittleU64
#endif
static int s_testBigEndian(void)
{
    union
    {
        posh_byte_t c[4];
        posh_u32_t i;
    } u;
```

```
    u.i = 1;
    if (u.c[0] == 1)
        return 0;
    return 1;
}
static const char *s_testEndianness(void)
{
    /* check endianness */
    if (s_testBigEndian() != IS_BIG_ENDIAN)
    {
        return "*ERROR: POSH compile time endianness does not "
            "match tun-time endianness verification.\n";
    }
    /* make sure our endian swap routines work */
    if ((NATIVE32(0x11223344L) != 0x11223344L) ||
        (FOREIGN32(0x11223344L) != 0x44332211L) ||
        (NATIVE16(0x1234) != 0x1234) ||
        (FOREIGN16(0x1234) != 0x3412))
    {
        return "*ERROR: POSH endianness macro selection failed. "
            "Please report this to poshlib@poshlib.org!\n";
    }
    /* test serialization routines */
    return 0;
}
```

上面代码验证了编译时环境大小端的猜测，与运行时环境大小端检测是匹配的。实践中永远不应该发生不匹配的情况，但是偶尔某种特别的配置或情况发生时，这些检测就会失败，你也会得到相应的错误通知。

使用编译选项

让工具为我们做更多的工作总是更好的选择，幸运的是我们最常用的工具——编译器——通常都能够以警告和错误的方式，提供帮助来寻找问题代码特性。

编译期断言

可移植编程的艺术不是避免假设，而是避免无意义和不精确的假设。一旦你确实做出了某个假设，你必须尽早验证这些假设是合法的。幸运的是许多假设可以在编译阶段使用编译期断言进行验证。

我见过无数种实现编译期断言的方法，但是我最喜欢下面这个：

```
#define CASSERT(exp, name) typedef int dummy##name[(exp) ? 1 : -1];
```

这个语句会试图以给定的名字创建一个新的类型，这个类型声明为整型数组。如果表达式 `exp` 为 `false`，则它将定义一个大小为-1的数组，这显然是非法并且会报错的。

例如，如果你编写如下代码：

```
CASSERT(sizeof(int) == sizeof(char), int_as_char)
```

宏将扩展为：

```
typedef int dummyint_as_char[-1];
```

这样就会产生一个编译错误，正如你预期的那样。GCC 的错误如下：

```
temp.c:3: error: size of array `dummyint_as_char` is negative
```

Microsoft Visual C++报告：

```
temp.c(3) : error C2118: negative subscript or subscript is too large
```

而 Metrowerks CodeWarrior 则报告：

```
### mwcc.exe Compiler:
#   File: temp.c
#   -----
#       3: (sizeof(int)==sizeof(char)) ? 1 : -1];;
#   Error:                                     ^
#   illegal constant expression
```

错误信息并没有告诉你哪个断言失败，但是当程序神秘地停止编译时，至少你有了文件名和行号来进行检查——远远好于代码成功编译而出现神秘的 `bug` 和崩溃。

`name` 参数是必需的，用于避免触发重复定义错误（重复多次定义相同类型的结果），因为在 C 语言中，重复（即使完全相同）类型定义是非法的。而在 C++ 中，这是完全合法的，因此如果你使用 C++ 来编译，那么 `name` 参数可以忽略。

Strict 编译

许多编译器提供一种 `Strict` 或者 `ANSI` 编译选项。例如 GCC 有 `-ansi` 和 `-pedantic` 参数，Microsoft Visual C++ 提供 `/Xa` 开关（不幸的是许多 Windows 应用不能通过 `/Xa` 编译，因为 `<windows.h>` 自己在无数地方违反了 ANSI 标准）。

当启用 `strict` 编译时，使用了编译器特定或编译器相关的特性的程序就会产生错误。这可以极大地帮助实现可移植。同样尽可能多地启用所有类型的警告信息也是一个好主意。

分离平台相关和可移植的文件

一项简单的技术能够使移植和编写可移植代码容易许多，就是根据平台相关性来分离文件。通过为其它平台的程序员定义清晰的界限和平台“模板”，使移植过程更加地流线化。

此外，通过强制这种分离，当不可移植的代码混入项目时，你可以快速地识别出来。如果有人把 `#include <windows.h>` 放到 `SAL_linux.c` 中，立即就可以确认下次在 Linux 中编译时将失败，并发出红色警报。

SAL 例子：分离平台相关文件

SAL 有许多平台相关的文件，均根据操作系统和后台 API 进行划分。下面是操作系统相关的文件：

- `sal_win32.c`，非音频特性的 Windows 实现（线程、睡眠等等）
- `sal_wince.c`，非音频特性的 Windows CE（Pocket PC）实现
- `sal_linux.c`，非音频特性的 Linux 实现
- `sal_osx.c`，非音频特性的 OS X 实现
- `sal_pthreads.c`，Linux 和 OS X 的 pthreads 线程创建实现
- `sal_pthread_mutex.c`，Linux 下基于 pthreads 的 mutex 实现
- `sal_nsrecursiveunlock.m`，OS X 特定的 mutex（NSRecursiveLock）实现

上面的每个文件都有 `#ifdef` 保护，阻止其在错误的平台进行编译。这样就简化了开发，因为你可以简单地把所有源文件都添加到 `makefile` 或者项目文件中，而不用管哪个文件在哪个平台下是必需的。

除了操作系统相关的文件，SAL 还有一些音频后端的文件。它们使用不同的常量进行保护，可以在编译过程中由开发者进行定义。下面是音频后端文件列表：

- `sal_alsa.c`，Linux 上的 ALSA 后端支持（`SAL_SUPPORT_ALSA` 必须定义）
- `sal_coreaudio.c`，OS X 上的 CoreAudio 后端支持（总是使用）
- `sal_dsound.c`，Windows 上的 DirectSound 后端支持（`SAL_SUPPORT_DIRECTSOUND` 必须定义）
- `sal_oss.c`，Linux 上的 OSS 后端支持（`SAL_SUPPORT_OSS` 必须定义）
- `sal_waveout.c`，Win32 和 Windows CE/Pocket PC 上的 waveOut 后端支持（`SAL_SUPPORT_WAVEOUT` 必须定义）

除了通过命名方式来指示文件类型以外，操作系统相关的文件还被存放在 `os/` 目录下，而后端相关的文件则被存放在 `backend/` 目录中。如果需要支持某个新的平台，实现者只需要看一下这两个目录下的文件，就明白他需要怎么做了。

编写直截了当的代码

程序员作为一种异类，喜欢享受聪明代码的乐趣。当某件事情既可以按清晰、简练的方式实现；也可以按稍微酷一点、但更加模糊的方式实现，程序员经常会选择后者来证明自己强大的语言能力。撇开由此导致的维护和调试问题，这种聪明代码在可移植方面通常都会让你头痛不已。

聪明的编程技巧通常都依赖于编译器相关的特性、或者编程语言中并不广为实现的某些特性。这样不但模糊了代码的意图，在别人试图移植到新平台时甚至根本无法辨别代码干了些什么。

使用唯一的名字

可移植软件经常需要运行在新的不熟悉的环境中。有时候需要进行修改以调用一些此前并不甚为了解的支持库和操作系统 API，这就意味着可移植软件必须能够与其它软件包进行良好的协作。

最常见的冲突之源就是标识符。例如，当开发者把 DOS 应用迁移到 Windows 时，他们马上发现自己的 `CreateWindow` 函数遇到各种奇怪的编译和链接错误，这个常用的函数名与 Windows API 的函数重名并产生了冲突。

理论上，Microsoft 应该在所有的 Windows API 前面加上诸如 `Win`、`GDI`、或者 `MSW` 前缀。这样 `MSW_CreateWindow` 与其它程序员的内部库发生冲突的可能性将大大降低。

但是由于开发者对库和操作系统的命名规范无能为力，避免命名冲突的责任只能由开发者自己承担。在 C 语言中，这意味着我们要尽可能多地添加前缀来保证标识符的唯一性。不过要记住有一些前缀是非常流行的（例如 `x_`、`z_`、`gr_`、以及 `Gr`），因此你的标识符越长，冲突的概率就会越低。在很多年以前，长标识符是有问题的，因为那时候的链接器会在几个字符之后就截断标识符（以前通常是 6 个）。例如 `Graphics_OpenWindow` 和 `Graphics_CloseWindow` 可能最终都被截断为 `Graphi`，导致链接出错。幸运的是今天这已经不是问题，因为 C99 规范允许外部链接时标识符最少 32 个有效字符。

在 C++ 中，你可以将所有全局函数作为静态函数封装在纯静态类里：

```
class MyLibrary
{
private:
    virtual ~MyLibrary() = 0; // prevent instantiation
    MyLibrary() {}
public:
    static void foo(void);
}
```

然后 `foo` 函数就可以通过适当的前缀进行调用：

```
void func()
{
    MyLibrary::foo();
}
```

除了看上去更加纯粹的 C++ 以外，你是不是觉得这种写法和调用 `MyLibrary_foo()` 并没有区别？（嗯，老实说，你确实得到了一个小小的好处，那就是当你在同一个静态类中调用该函数时，你可以抛开前缀，但也有人认为这不是件好事情。）

如果你想更加地 C++，那么你也可以使用命名空间：

```
namespace MyLibrary
{
    void foo(void);
}
```

不过这样做和静态类的效果完全一样，除了更加符合 C++ 时尚以外。

SAL 例子：唯一标识符

SAL 给所有标识符加上了前缀 `SAL`，这并不需要特别的想象力，但是它的功能却是显著的，你可以从下面看出来：

```
/** Standard error codes returned by most SAL functions */
typedef enum
{
    SALERR_OK                = 0x0000,
```

```

    SALERR_INVALIDPARAM      = 0x0001,
    SALERR_WRONGVERSION      = 0x0002,
    SALERR_OUTOFMEMORY       = 0x0003,
    SALERR_SYSTEMFAILURE     = 0x0004,
    SALERR_ALREADYLOCKED     = 0x0005,
    SALERR_INUSE              = 0x0006,
    SALERR_INVALIDFORMAT     = 0x0007,
    SALERR_OUTOFVOICES       = 0x0101,
    SALERR_UNIMPLEMENTED     = 0x1000,
    SALERR_UNKNOWN           = 0xFFFF
} sal_error_e;

#define SAL_INVALID_SOUND-1
#define SAL_LOOP_ALWAYS     -1
#define SAL_PAN_HARD_LEFT-32767
#define SAL_PAN_HARD_RIGHT  32767
#define SAL_VOLUME_MIN      0
#define SAL_VOLUME_MAX      65535

typedef struct SAL_DeviceInfo_s
{
    sal_i32_t    di_size;
    sal_i32_t    di_channels;
    sal_i32_t    di_bits;
    sal_i32_t    di_sample_rate;
    sal_i32_t    di_bytes_per_sample;
    sal_i32_t    di_bytes_per_frame;
    char         di_name[SAL_DEVICEINFO_MAX_NAME];
} SAL_DeviceInfo;

SAL_PUBLIC_API(sal_error_e) SAL_create_device(
    SAL_Device **pp_device,
    const SAL_Callbacks *kp_cb,
    const SAL_SystemParameters *kp_sp,
    sal_u32_t desired_channels,
    sal_u32_t desired_bits,
    sal_u32_t desired_sample_rate,
    sal_u32_t num_voices);

SAL_PUBLIC_API(sal_error_e) SAL_destroy_device(SAL_Device *p_device);
SAL_PUBLIC_API(sal_error_e) SAL_get_device_info(SAL_Device *p_device,
                                                SAL_DeviceInfo *p_info);

```

几乎所有标识符都有合适的前缀来最小化与其它库和 API 产生冲突的概率。类似于

`destroy_device` 的函数名，或者类似于 `UNKNOWN` 和 `INVALIDFORMAT` 的常量名，在其它包中被定义的机会是极度危险的。

实现抽象

实现可移植的主要工具是抽象，抽象就是隔离系统相关元素和更加通用架构的过程。抽象允许你以一种清晰、系统无关的方式编写主干代码。

抽象是功能和容易使用之间的一种权衡。抽象必须花费更大精力来实现，并实现一个最小公分母式的功能。抽象虽然能够使程序员更轻松，但由于他们不能够再访问底层实现的额外特性，程序员可能会认为自己的能力被限制了。

例如 `SAL` 音频 API 假设 8 位或 16 位 PCM 数据，因为几乎所有主流操作系统和音频驱动都是这样。但是苹果的 `OS X CoreAudio API` 支持 32 位浮点音频格式，这个功能非常强大。不幸的是，我们没有一种清晰的方式，能够让 `SAL` 提供这个特性而不严重影响到工作量和复杂度（通过在不支持此特性的系统上模拟 32 位浮点格式，或者只在 `OS X` 允许这个特性）。

在这一小节，我会用 `SAL` 和 `POSH` 的例子来说明抽象的威力。

调度抽象

程序员经常错误地使用条件编译的一个地方就是：在不同的操作系统下选择适当的函数来执行。让我们用一个简单的例子来进行说明：使计算机系统的扬声器发出一声蜂鸣。

在 `Windows` 下，我们使用 `Beep()` 来实现；在支持 `ANSI` 转码序列的系统中，我们可以通过打印 `CTRL-G` 到控制台来实现。

不幸的是，太多的程序在每次需要蜂鸣时都进行一次条件判断：

```
#ifdef _WIN32
    Beep(440, 100); /* 440Hz, 100ms */
#else
    printf("\a"); /* use ANSI "bel" character */
#endif
```

如果蜂鸣使用非常频繁，这样的代码将非常丑陋。理想的情况下，某个功能在主干代码中只需要一个单一的调用，而这个调用会自动寻找到合适的底层函数——我把这个叫做调度抽象。

调度抽象的三个常用机制是链接时决定、函数指针表、和 `C++` 虚拟函数。

链接时决定

链接时决定依赖链接器来静态实现抽象，一般是通过条件编译。在跨越多个平台时，使用同一个名字来引用某个函数，但是它的实现则由各个平台单独提供。当特定平台能够单一链接时这种方式是很高效的。

例如 `SAL` 使用这个技术来抽象设备相关数据的创建和初始化，使用的是跨平台的函数 `SAL_create_device`。

```
/* NOTE: This creates a device, but is a non-system-specific
function since the abstraction is handled by the link-resolved
```

```

call to _SAL_create_device_data */
sal_error_e
SAL_create_device(
    SAL_Device **pp_device,
    const SAL_Callbacks *kp_cb,
    const SAL_SystemParameters *kp_sp,
    sal_u32_t desired_channels,
    sal_u32_t desired_bits,
    sal_u32_t desired_sample_rate,
    sal_u32_t num_voices)
{
    sal_error_e err;
    SAL_Device *p_device = 0;
    if (pp_device == 0 || kp_sp == 0 || num_voices <= 0)
        return SALERR_INVALIDPARAM;
    if (kp_cb == 0)
    {
        p_device = (struct SAL_Device_s *)
            malloc(sizeof(struct SAL_Device_s));
        memset(p_device, 0, sizeof(*p_device));
        p_device->device_callbacks.alloc = s_alloc;
        p_device->device_callbacks.free = s_free;
        p_device->device_callbacks.warning = s_print;
        p_device->device_callbacks.error = s_error;
    }
    else
    {
        if (kp_cb->cb_size != sizeof(SAL_Callbacks))
            return SALERR_WRONGVERSION;
        if ((kp_cb->alloc == 0) != (kp_cb->free == 0))
            return SALERR_INVALIDPARAM;
        *pp_device = (struct SAL_Device_s *)
            kp_cb->alloc(sizeof(*pp_device));
        memset(*pp_device, 0, sizeof(**pp_device));
    }
    /* allocate voices */
    p_device->device_voices = (struct SAL_Voice_s *)
        p_device->device_callbacks.alloc(
            sizeof(struct SAL_Voice_s) * num_voices);
    memset(*pp_device->device_voices, 0,
        sizeof(struct SAL_Voice_s) * num_voices);
    p_device->device_max_voices = num_voices;

    /* this will properly dispatch to the _SAL_create_device_data

```



```

implemented by a given platform */
if ((err = _SAL_create_device_data(
    *pp_device,
    kp_sp,
    desired_channels,
    desired_bits,
    desired_sample_rate)) != SALERR_OK)
{
    p_device->device_callbacks.free(p_device->device_voices);
    p_device->device_callbacks.free(*pp_device);
    return err;
}
/* Dispatches through a function table */
_SAL_create_mutex(*pp_device, &(p_device->device_mutex));
p_device->device_info.di_bytes_per_sample =
    p_device->device_info.di_bits / 8;
p_device->device_info.di_bytes_per_frame =
    p_device->device_info.di_bytes_per_sample *
    p_device->device_info.di_channels;
*pp_device = p_device;
return SALERR_OK;
}

```

`_SAL_create_device_data` 调用是在链接时决定的,在任何给定平台它只会被定义一次(其它实现通过`#ifdef`来保护)。例如在 Windows 平台, `_SAL_create_device_data` 定义在 `sal_win32.c` 文件中, 并且只有 `POSH_OS_WIN32` 被定义时才能找到该函数。

链接时决定提供了良好的性能和可靠性(如果函数名不能被找到,将会在构建时而不是程序运行时被捕获到)。但是它并不能提供太多的灵活性,特别是当你需要运行时进行调度。也就是说如果你需要动态地选择不同的实现,链接时决定并不能满足要求。例如 `SAL` 在 Windows 平台支持 `waveOut` 和 `DirectSound` 音频 API,如果你要在运行时选择相应的子系统,静态链接时决定就无法实现了(当然,你可以发布两个独立的可执行文件来解决这个问题)。

函数指针表

当你需要在不同实现之间动态切换时,函数指针表比静态链接更加合适。函数指针表是包含一组函数指针的数组或结构体,在可移植和不可移植程序段中提供一层间接抽象。

`SAL` 使用这个技术作为核心抽象层。在 `SAL_Device` 结构体(定义在 `sal_private.h`)中,你可以看到:

```

typedef struct SAL_Device_s
{
    SAL_Callbacks device_callbacks;
    sal_mutex_t device_mutex;
    void *device_data;
    SAL_DeviceInfo device_info;
    struct SAL_Voice_s *device_voices;
    int device_max_voices;
}

```

```

/* implement callbacks */
sal_error_e (*device_fnc_create_mutex)
    (struct SAL_Device_s *device, sal_mutex_t *p_mtx);
sal_error_e (*device_fnc_destroy_mutex)
    (struct SAL_Device_s *device, sal_mutex_t mtx);
sal_error_e (*device_fnc_lock_mutex)
    (struct SAL_Device_s *device, sal_mutex_t mtx);
sal_error_e (*device_fnc_unlock_mutex)
    (struct SAL_Device_s *device, sal_mutex_t mtx);
sal_error_e (*device_fnc_create_thread)
    (struct SAL_Device_s *device,
     void (*fnc)(void *args), void *targs);
sal_error_e (*device_fnc_sleep)
    (struct SAL_Device_s *device, sal_u32_t duration);
void (*device_fnc_destroy)(struct SAL_Device_s *d);
} SAL_Device;

```

每个实现负责将必要的函数指向相应的实现。例如 Win32 的 `_SAL_create_device_data` 实现如下：

```

sal_error_e
_SAL_create_device_data(
    SAL_Device *device,
    const SAL_SystemParameters *kp_sp,
    sal_u32_t desired_channels,
    sal_u32_t desired_bits,
    sal_u32_t desired_sample_rate)
{
    device->device_fnc_create_thread = _SAL_create_thread_win32;
    device->device_fnc_create_mutex = _SAL_create_mutex_win32;
    device->device_fnc_lock_mutex = _SAL_lock_mutex_win32;
    device->device_fnc_unlock_mutex = _SAL_unlock_mutex_win32;
    device->device_fnc_destroy_mutex = _SAL_destroy_mutex_win32;
    device->device_fnc_sleep = _SAL_sleep_win32;
    if (kp_sp->sp_flags & SAL_SPF_WAVEOUT)
    {
#ifdef SAL_SUPPORT_WAVEOUT
        return _SAL_create_device_data_waveout(
            device,
            kp_sp,
            desired_channels,
            desired_bits,
            desired_sample_rate);
#endif
    }
}

```

```

        else
        {
#ifdef SAL_SUPPORT_DIRECTSOUND
            return _SAL_create_device_data_dsound(
                device,
                kp_sp,
                desired_channels,
                desired_bits,
                desired_sample_rate);
#endif
        }
        return SALERR_UNIMPLEMENTED;
    }

```

`_SAL_create_device_data` 函数是静态链接的，由它动态地确定剩余的实现相关函数。实际上对于 SAL 来说，这样做并没有什么特别的好处，因为这些函数大多可以通过静态链接来确定。设备析构函数是个例外，它必须被动态选择。

在上面例子中，SAL 使用函数指针表的原因是为了将来的灵活性。例如要在 OS X 上使用 pthreads 的 Mutex 来替换 NSRecursiveLocks 将是非常容易的事情。此外文档工具也可以根据不同的平台为相同函数提供各自平台的文档解释（这个工具就是 Doxygen，网址是 <http://www.doxygen.org>）。

尽管 SAL 使用了函数指针表，我还是尽可能地避免直接调用它。例如 `SAL_sleep()` 是一个我希望客户能够访问的公有函数，但是我不愿意不必要地暴露 `SAL_Device` 结构体的内部结构，因此我通过包装函数来解引用函数指针，以提供另一层间接抽象：

```

sal_error_e
SAL_sleep(SAL_Device *device, sal_u32_t duration)
{
    if (device == 0 || device->device_fnc_sleep == 0)
        return SALERR_INVALIDPARAM;

    return device->device_fnc_sleep(device, duration);
}

```

现在 SAL 库的用户和 SAL 本身都可以调用系统相关的函数，而不暴露底层实现细节以及函数指针表。

SAL 获得灵活性的同时，也损失了很小的一点性能和健壮性。间接函数调用天生就比直接调用稍微昂贵一些，但是在现代的计算机上这种区别根本无法测量出来。另一个潜在的问题是函数指针表可能会是 NULL 而处于非法状态，但是在开发过程中这个问题很容易就能测试出来。

虚拟函数

C++ 通过虚拟函数提供多态支持。多态可以通过单一接口来执行不同动作，被执行的函数是由对象类型来动态确定的。这正是你在抽象机制中想要实现的功能。

SAL 不是用 C++ 编写的，但是构思一个继承层次来实现多态是很容易的：

```

class SAL_Device
{

```

```
private:
    virtual sal_error_e create_mutex(sal_mutex_t *p_mtx) = 0;
    virtual sal_error_e destroy_mutex(sal_mutex_t mtx) = 0;
    virtual sal_error_e lock_mutex(sal_mutex_t mtx) = 0;
    virtual sal_error_e unlock_mutex(sal_mutex_t mtx) = 0;
    virtual sal_error_e create_thread(void (*fnc)(void *args),
        void *targs) = 0;
    virtual sal_error_e sleep(sal_u32_t duration) = 0;
    virtual void destroy(void) = 0;
public:
    static sal_error_e create_device(
        SAL_Device **pp_device,
        const SAL_Callbacks *kp_cb,
        const SAL_SystemParameters *kp_sp,
        sal_u32_t desired_channels,
        sal_u32_t desired_bits,
        sal_u32_t desired_sample_rate,
        sal_u32_t num_voices);
}
```

这里 SAL 在基类的 private 域中定义了纯虚函数，要实例化 SAL_Device，你需要调用静态的 SAL_Device::create_device 函数（工厂函数），由它返回指向 SAL_Device 某个子类的对象指针。

你可以通过创建一个新的具体类，继承自 SAL_Device，来提供特定的实现。比如你可能定义 SAL_DeviceWin32 来实现必须的纯虚函数接口，然后还可以再派生出 SAL_DeviceDirectSound 和 SAL_DeviceWAVEOUT 两个子类。SAL 和其它客户端只能看见 SAL_Device 定义的接口，这样就提供了一层很好的抽象。

虚拟函数调度存在几个小问题。相比于静态链接，虚拟函数有轻微的性能损失，因为必须在运行时找到正确的函数。和函数指针一样，这个性能损失在现代计算机上也不需要过多考虑。但是在嵌入式设备或者特殊场合（例如器械、游戏控制台、和手持电话）这可能是一个需要考虑的问题。

另一个问题是虚拟函数在对象创建之后很难改变，如果你需要动态地改变一个特定的函数，就得使用一种“信封”方式（增加了复杂度），重新创建对象，或者向所有函数传递必要的动态信息。而对于函数指针表来说，只需要更新一个变量就可以解决问题。

抽象数据类型

数据类型可以通过使用 typedef 得到清晰地抽象。譬如相对轻率地假设整数是 32 位，不如将这个假设抽象到平台的类型定义中去，来确保这个假设是合法的：

```
#if PLATFORM_HAS_32BIT_INT
typedef int int32;
typedef unsigned int uint32;
#endif
```

虽然 ANSI/ISO C99 规范在<inttypes.h>是引入了固定长度类型的标准定义，在本书写作的

时候许多编译器仍然无法完全支持 C99 及相关特性。

POSH 在所有平台以可移植的方式提供这些类型定义，通过 C98+标准<limits.h>头文件，或者根据编译环境来进行判断。

```
#if defined POSH_USE_LIMITS_H
#  if CHAR_BITS > 8
#    error This machine uses 9-bit characters. This is a warning, \
        you can comment this out now.
#  endif /* CHAR_BITS > 8 */
/* 16-bit */
#  if(USHRT_MAX == 65535)
      typedef unsigned short posh_u16_t;
      typedef short posh_i16_t;
#  else
      /* In theory there could still be a 16-bit character type
         and shorts are 32-bits in size */
#    error No 16-bit type found
#  endif
/* 32-bit */
#  if (INT_MAX == 2147483647)
      typedef unsigned posh_u32_t;
      typedef int posh_i32_t;
#  elif (LONG_MAX == 2147483647)
      typedef unsigned long posh_u32_t;
      typedef long posh_i32_t;
#  else
#    error No 32-bit type found
#  endif
#else /* POSH_USE_LIMITS_H */
      /* This makes fairly major assumptions */
      typedef unsigned short posh_u16_t;
      typedef short posh_i16_t;
#  if !defined POSH_OS_PALM
      typedef unsigned posh_u32_t;
      typedef int posh_i32_t;
#  else
      typedef unsigned long posh_u32_t;
      typedef long posh_i32_t;
#  endif
#endif
/* Verify we made the right guesses! */
POSH_COMPILE_TIME_ASSERT(posh_byte_t, sizeof(posh_byte_t) == 1);
POSH_COMPILE_TIME_ASSERT(posh_u8_t, sizeof(posh_u8_t) == 1);
POSH_COMPILE_TIME_ASSERT(posh_i8_t, sizeof(posh_i8_t) == 1);
POSH_COMPILE_TIME_ASSERT(posh_u16_t, sizeof(posh_u16_t) == 2);
```

```
POSH_COMPILE_TIME_ASSERT(posh_i16_t, sizeof(posh_i16_t) == 2);
POSH_COMPILE_TIME_ASSERT(posh_u32_t, sizeof(posh_u32_t) == 4);
POSH_COMPILE_TIME_ASSERT(posh_i32_t, sizeof(posh_i32_t) == 4);
```

POSH 使用<limits.h>中定义的值来确定与 POSH 固定类型相适当的系统本地类型。如果<limits.h>不可靠或者不可用，则对底层系统做出一定的假设，Palm 架构除外。在类型定义完成之后，POSH 使用编译期断言来确保假设是合法的。

使用 C 预处理器

虽然 C 预处理器被许多人诟病，但它仍然是一个简单而强大的源代码管理工具。两行预处理代码就可以使一大段代码消失，而完全不用担心嵌套注释的规则。预处理器的任意替换能力意味着不同平台的较小语法区别可以很容易地进行抽象。

例如很多平台都有大小写无关的字符串比较，一般函数名为 `strcmpi`、`stricmp`、或者 `strcasecmp`。我们可以使用多种方式来封装这个区别，但是通常直接宏替换是最简单的，如下所示：

```
#ifdef _MSC_VER
#define strcasecmp stricmp
#else
... etc. etc.
#endif
```

诚然，这是一种非常简单的情况，你完全可以使用自己编写的版本，但是这种思想是值得加以应用的。

程序员编写多行宏时，如果加入了域、局部变量、引用了全局变量、以及其它非直观的东西，那么预处理器将会出现问题。换句话说，千万不要像下面这样做：

```
#ifdef POSH_OS_WIN32
#define SYS_INIT() { \
    extern HWND g_hWnd; \
    char oldtitle[1024]; \
    char newtitle[1024]; \
    \
    GetConsoleTitle(oldtitle, sizeof(oldtitle)); \
    \
    sprintf(newtitle, "SAL Test - %d:%d", GetCurrentProcessId(), \
        GetTickCount()); \
    \
    SetConsoleTitle(newtitle); \
    \
    Sleep(100); \
    \
    g_hWnd = Findwindow(NULL, newtitle); \
    \
    if (g_hWnd == 0) \
```

```
    { \
        fprintf(stderr, "Could not find window\n"); \
        exit(1); \
    } \
    \
    p_sp->sp_buffer_length_ms = 100; \
    p_sp->sp_hWnd = g_hWnd; }
#endif /* POSH_OS_WIN32 */
.
.
.
int main(int argc, char *argv[])
{
    SYS_INIT();
}
```

为意外做好准备

在我们定义一个合理的基准时，应该尽可能地小心、细致、和具有前瞻性，因为我们的要求由于不可预见的原因，不可避免地会发生变化。这并不一定是由于不良规划或预见，而是我们不可能无所不知的一个副作用（至少这是我在过去所意识到的）。

以下是一些常见的错误假设，而许多程序员并不知道它们可能是错误的：

malloc 和 new 总是可用的

特别是 PC 开发者倾向于做出这个假设，但实际上有一些系统如早期的 Mac OS、16 位 Windows、以及 Palm OS，都不使用 malloc/new。相反它们要求你间接地使用句柄来分配和引用内存，并且在使用前锁定，使用完后解除锁定。在没有虚拟内存的系统中，这是一种常用的架构，而且堆碎片也是需要考虑的。因此这个假设可能是危险的，特别是你的目标是基于低端的微控制器嵌入式系统，这种系统的内存管理通常都是硬编码的。

stdio 文件管理是可用且完整的

文件输入/输出假定 stdio 子系统和文件系统的存在。在嵌入式设备和许多手持设备中，实际上两个都不保证存在。

图形输出总是规则的 RGB（没有调色板）

现代图形子系统已经使用 RGB（红/绿/蓝）输入好几年了，而且甚至高端的手持设备也已经支持此特性。当设计一个图形应用时，相比于较老的调色板显示格式（显示器的像素包含索引在另一个颜色表中），支持 RGB 要容易很多。如果是为 PC 设计哪怕是最简单的图形应用，你也可以安全地使用 RGB 格式，但是如果你需要移植至手持或无线格式，那么你可能就需要重构整个图形后端，来支持调色板图形输出（这样能够大大降低内存消耗，因此在低端硬件中更加流行）。

网络总是使用 TCP/IP

TCP/IP 已经成为事实上的标准至少十年了，其它网络协议如 NetBEUI、IPX/SPX、和

DECNet 已经靠边站。当编写需要网络通讯的应用时，假设 TCP/IP 支持是合理的，目前甚至很多非常低端的设备也拥有 TCP/IP 网络协议栈。但是也还有许多设备可能只支持串行或并行端口通信、或者使用自定义有线协议。由于这很少见，在为这些系统抽象网络协议的实现时你可能会觉得比较混乱。

上面每个假设都是合理的，但是一旦在某个平台不能工作而需要重新设计，则代价将非常巨大。然而犯这种错的可能性也较小，因为在开发的过程中，设计需求看上去完全合理，更重要的是，要抽象这些特殊情况下的子系统，可能会引入许多的间接层和额外代码，而大部分时候这都是毫无理由的。

有时候甚至最合理、最安全的假设也可能是错误的，你不可能预知未来，因此只需要清晰认识你所有的假设就好了。

与系统相关信息交互

设计清晰、跨平台程序最复杂的因素之一，就是你的应用代码与系统相关信息的交互。譬如 SAL 中的 DirectSound 子系统，在调用 IDirectSound::SetCooperativeLevel()时需要应用的窗口句柄。由于 SAL 并不负责创建或管理应用的窗口，它必须以某种方式将这个窗口信息传递进去。有几种不同的方法可以按半可移植的方式来处理这个问题。

NOTE 理论上获取当前线程所绑定的窗口是可能的，但是有很多边界因素会导致失败。此外使用 NULL 或 GetDesktopWindow() 作为的参数的诱惑很大，这本身是一种非常不好的实践，因为 DirectSound 确实需要绑定自己到你的应用窗口，而不是任意窗口。

第一种方法是使用单一平台可用的独立的 API 调用，来注册系统相关信息，并且使用适当的编译命令进行保护，如下：

```
#ifdef POSH_OS_WIN32
    SAL_register_HWND(hWnd); /* only on Windows */
#endif
SAL_create_device(...); /* portable portion */
```

虽然有一点点繁琐，但是这个独立的接口是可以工作的，并且避免了核心 API 与系统相关信息之间相互污染（调用程序需要使用条件编译代码）。

当然你也可以选择污染你的核心 API。通过把所有系统相关变量都加到相应的创建函数，你可以一次性解决这个问题，像下面这样：

```
void create_device(
    void *hwnd_for_win32,
    void *something_for_OSX,
    int some_flag_for_linux);
```

虽然功能得到了实现，实际上这是一种非常脆弱的策略，因为每次增加新参数 API 都要修改。例如在以后可能需要支持 Palm OS，而且需要应用提供 Palm 平台特殊的参数。这意味着又必须引入新版本的 API：

```
void create_device_ex(
    void *hwnd_for_win32,
```



```
void *something_for_OSX,  
int some_flag_for_linux,  
short palm_os_thing);
```

再然后你增加 Solaris 支持，现在你又有了 `create_device_ex_ex`，如此反复。

这个方法的另一个问题是有时候在函数调用时某些特殊参数并不可用；例如 `palm_os_thing` 的值在你调用 `create_device_ex` 之前可能并未配置。

一个稍微清晰一点的修正方案是使用结构体封装所有的系统相关信息，然后再传递给你的 API，SAL 正是使用了这个技术：

```
struct SAL_SystemParametersWin32  
{  
    sal_i32_t sp_size; /* size of this data structure */  
    sal_u32_t sp_flags; /* flags for the API call */  
    sal_i32_t sp_buffer_length_ms; /* sound buffer len */  
    void *sp_hwnd; /* HWND */  
};  
struct SAL_SystemParametersDefault  
{  
    sal_i32_t sp_size;  
    sal_u32_t sp_flags;  
    sal_i32_t sp_buffer_length_ms;  
};  
#ifdef POSH_OS_WIN32  
typedef struct SAL_SystemParametersWin32 SAL_SystemParameters;  
#else  
typedef struct SAL_SystemParametersDefault SAL_SystemParameters;  
#endif
```

`sp_size` 成员允许在将来对系统参数结构体进行修正，而不影响向后兼容性。通过在调用 API 入口之前初始化 `sp_size` 变量，API 可以推断出客户端使用的 `SAL_SystemParameters` 结构体的版本，并做出相应的反应。例如返回错误、或者向新格式进行转换。另一个好处是你可以保持单一的 API 接口，而不需要根据不同的结构体数据增加入口。

这里可能存在顺序相关的问题，API 需要的某些信息在你传进系统参数结构体之前可能不可用。要处理这种情况，你要么使用独立的入口点（像之前的 `SAL_register_HWND()` 例子），或者找到另一个通信载体。

函数调用只是程序与系统交互的方式之一，其它更加间接的机制也是可能的。例如脚本语言、注册表/参数系统、以及不时新的全局变量，都可以很好的工作，而不污染你的核心 API。

无论采用哪一种特定的机制，系统相关信息都有一定的暴露。这通常是不可避免的，你可以通过封装来最小化它对你其余代码的影响，但是它依旧以这样或那样的形式存在。

桥接函数

可移植程序的一个常见模式是将程序划分为系统相关和系统无关部分，然后创建桥接（或胶合）函数来连接二者。这种抽象类似于 GUI 里面的模型-视图-控制器（MVC）模式，

用户接口无关代码（模型）与显示代码（视图）相互隔离，并通过控制器作为桥梁来连接双方。

桥接函数的主要目的是传递系统相关数据或者格式到系统无关数据（反过来也是这样），然后再输入到可移植层（参考图 3-1）。

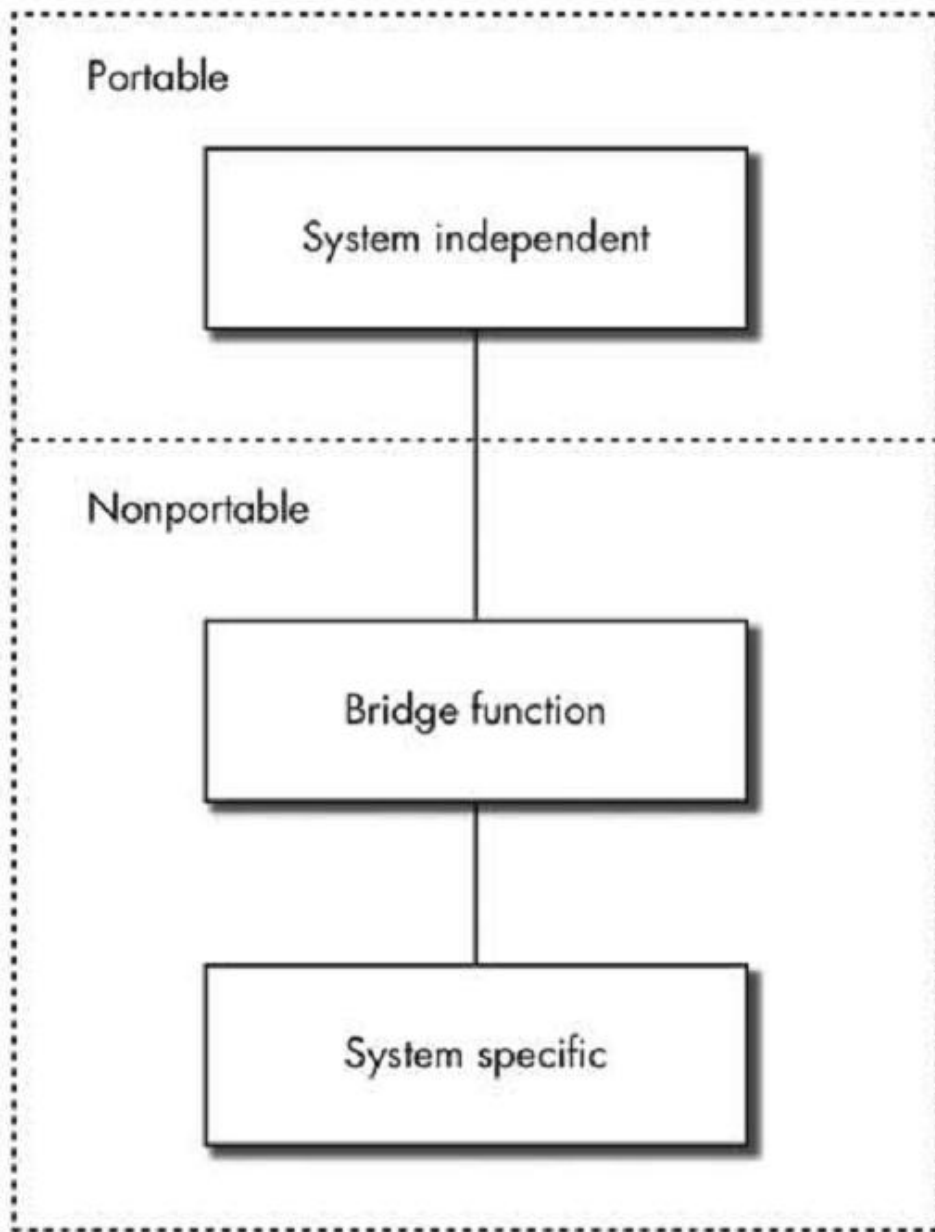


图 3-1: 桥接函数

SAL 例子: 桥接函数

在不同操作系统的多线程 API 中，创建线程时所使用的线程函数稍微有一点不同。Linux、Mac OS X、及其它 Unix 系统实现的 pthreads API 创建线程使用以下函数：

```
int pthread_create(  
    pthread_t *thread,  
    const pthread_attr_t *attr,  
    void *(*start_routine)(void *),
```

```
void *arg);
```

然而在标准 Win32 平台，则使用下面函数：

```
uintptr_t _beginthread(
    void(__cdecl *start_address)(void *),
    unsigned stack_size,
    void *arglist);
```

而 Windows CE/Pocket PC 则使用 Windows 的 CreateThread API：

```
typedef DWORD(WINAPI *PTHREAD_START_ROUTINE)(LPVOID lpThreadParameter);
typedef PTHREAD_START_ROUTINE LPTHREAD_START_ROUTINE;
HANDLE CreateThread(
    LPSECURITY_ATTRIBUTES lpThreadAttributes,
    SIZE_T dwStackSize,
    LPTHREAD_START_ROUTINE lpStartAddress,
    LPVOID lpParameter,
    DWORD dwCreationFlags,
    LPWORD lpThreadId);
```

在不同操作系统的线程创建过程中，总共有三种不同的线程函数：

```
/* return a void pointer, accept a void pointer,
   default calling convention */
void *(*start_routine)(void *);
/* return nothing, accept a void pointer,
   cdecl calling convention */
void (__cdecl *start_routine)(void *);
/* return a DWORD, accept a void pointer,
   WINAPI calling convention */
DWORD (WINAPI *start_routine)(void *);
```

开发者不应该为相同的线程函数提供三个变体，因为这违反了抽象原则（而且也是一个笑柄）。桥接函数可以拯救你。

每个线程函数都有一个 void 指针参数，而在不同平台则返回不同的值，因此最小公分母就是一个返回 void，接受 void 指针的函数，以及你所定义的调用约定（这样对用户来说就是一致的）。SAL 定义了传递给 _SAL_create_thread() 的线程启动函数，如下：

```
typedef void (POSH_CDECL *SAL_THREAD_FUNC)(void *args);
sal_error_e
_SAL_create_thread(
    SAL_Device *device,
    SAL_THREAD_FUNC func,
    void *targs);
```

无论在哪个平台，SAL 的任何用户（在这里就只是 SAL 自己，因为 _SAL_create_thread() 没有暴露为公有函数）都可以提供一个类型为 SAL_THREAD_FUNC 的线程启动函数，而不用担

心底层实现。每个平台则提供一个桥接的线程启动函数，调度到上面通用的函数。

我们来看一下_sal_create_thread_wince:

```
typedef struct _SAL_WinCEBridgeFunctionParameters_s
{
    SAL_THREAD_FUNC bfp_fnc; /* thread function */
    void *bfp_targs; /* pointer to thread arguments */
} _SALWinCEBridgeFunctionParameters;

static sal_error_e
_sal_create_thread_wince(
    SAL_Device *device,
    SAL_THREAD_FUNC fnc,
    void *targs)
{
    HANDLE hThread;
    _SAL_WinCEBridgeFunctionParameters bfp;
    bfp.bfp_fnc = fnc;
    bfp.bfp_targs = targs;
    if (device == 0 || fnc == 0)
        return SALERR_INVALIDPARAM;

    if ((hThread = CreateThread(
        NULL, /* lpThreadAttributes, ignored on WinCE */
        0, /* stack size, ignored on WinCE */
        s_bridge_function, /* thread start function */
        &bfp, /* lpParameter */
        0, /* creation flags */
        NULL)) == (HANDLE)-1)
    {
        return SALERR_SYSTEMFAILURE;
    }
    SetThreadPriority(hThread, THREAD_PRIORITY_HIGHEST);
    return SALERR_OK;
}
```

这里演示了两个重要的要素，第一个是调用 CreateThread 将分派到 s_bridge_function 而不是用户的线程函数，因为用户提供的线程函数签名并不匹配系统要求，而 s_bridge_function 则符合 CreateThread 的要求。第二个是传递给 s_bridge_function 的参数包含用户线程函数，以及该函数所需的参数。

s_bridge_function 函数实际上非常简单，它只是解开 bridge 参数，然后直接调用用户函数：

```
static DWORD WINAPI
s_bridge_function(LPVOID lpParameter)
{
    _SAL_WinCEBridgeFunctionParameters *bfp =
        (_SAL_WinCEBridgeFunctionParameters *) lpParameter;
    bfp->bfp_fnc(bfp->bfp_targs);
}
```

```
    return 1;
}
```

底层（Low-Level）编程

尽管高级语言已经成为应用开发的主流，程序员有时候仍然需要深入底层，用机器本地语言与机器进行交互。有时候高级语言（即使是非常接近于硬件的 C 语言）并不能暴露以下核心硬件功能：

- CPU 指令标志（进位标志、零标志）
- 新指令集，例如 Intel MMX/SSE/SSE-2、AMD 3DNow、以及 Motorola AltiVec 等
- 扩展操作结果，例如 32 位乘法，在某些体系架构将会产生 64 位结果，不能很容易且快速地被高级语言接收。

因此，在底层进行编程有着充足的理由。你也同样希望在底层编写的代码能够尽可能可移植。

避免自修改/动态产生的代码

早期微型计算机只有少量寄存器，而寄存器访问速度又要比内存快非常多。因此寄存器是宝贵而又稀缺的物品，也导致开发者使用大量技巧来提高寄存器的使用效率。

假设某个有限的 CPU 体系架构只有四个通用寄存器，你试图编写以下循环：

```
mov d, object_size    ; size of an array element in bytes
mov c, num_objects     ; number of objects in array
mov b, 0               ; initialize 'b' to 0
mov a, array           ; 'a' will point to the current object
top:
...                   ; do some operations
add a, d               ; increment 'a' by size of object
sub c, 1               ; decrement 'c'
jg top                 ; continue loop if 'c' is greater than 0
```

'b'是唯一可用作读写变量的寄存器，意味着内部循环必须访问内存来存储所有中间结果。最令人沮丧的是寄存器'd'只是存储了一个常量，如果能够预先知道，那就能直接将数值替换到指令流中，从而获得一个额外的可用寄存器。

汇编语言程序员认识到了这一点，并且使用所谓的“自修改”代码技术。程序员汇编所有指令，找到 add a, d 指令并记下地址。然后再回过头去修补代码，用适当的常量替换那个调用，从而释放一个寄存器。

这种方法除了绑定到特定 CPU 实现这一明显问题之外，还带来了其它问题。最主要的一个就是安全性。多数操作系统不允许程序写入到自身代码中，因为这在多数情况下都会导致 Bug（例如写入未初始化的指针）。当然你可以使用操作系统调用（例如 Windows 的 VirtualProtect()和 Linux 的 mprotect）绕过这个（需要足够的安全权限），但是现在你已经非常不可移植了。

这种情况的一个极端就是动态生成代码。正如名字所暗示的，动态生成代码是在运行时

通过处理器指令来装填数据缓冲区，并将内存标志改为可执行来创建和汇编代码，并赋给一个函数指针，然后再执行它。下面是一个 Windows 操作系统下的例子：

```
unsigned char *add_two;
float (*fnc_add_two)(float, float);
void test()
{
    DWORD old_protect; /* Windows-ism */
    float c;
    add_two = VirtualAlloc(NULL, 32, MEM_COMMIT, PAGE_READWRITE);
    /* initialize to NOP */
    memset(add_two, 0x90, 32);
    /* The first four instructions equate to 'fld [esp+4]' */
    add_two[0] = 0xD9; /* co-processor escape */
    add_two[1] = 0x44; /* mod r/m byte */
    add_two[2] = 0x24; /* SIB */
    add_two[3] = 0x04; /* index */
    /* the next four bytes equate to 'fadd [esp+8]' */
    add_two[4] = 0xD8; /* co-processor escape */
    add_two[5] = 0x44; /* mod r/m byte */
    add_two[6] = 0x24; /* SIB */
    add_two[7] = 0x08; /* index */
    /* this is a one-opcode RET instruction */
    add_two[8] = 0xC3;
    /* on linux you would use mprotect */
    VirtualProtect(add_two, 32, PAGE_EXECUTE, &old_protect);
    /* this should now work */
    c = fnc_add_two(1, 2);
    printf("c = %f\n", c);
}
```

这的确令人相当吃惊，同时由于几个原因也是不可移植的。

你可以使用相同的动态生成代码技术把可执行代码装载为数据。除了以常量数据装填缓冲区，你也可以从磁盘装载内容并使之可执行。这是相当危险的，不过如果你要在不支持动态库（DLL 或共享对象）的主机上装载代码，这也不失为一个实际的解决办法。

不用说这是非常不可移植的，但是有时候它也是必要的，可以在有限资源里获得最大性能。

因此虽然已经很明白了，我还是要啰嗦一遍，如果可以的话，你应该避免使用类似的技巧。

保留一个高级的 Fallback

不管是使用哪种汇编语言、动态生成或是静态链接，保留一个高级的 Fallback 版本都是至关重要的。Fallback 就是某个特性的一个默认且可以工作的实现。当你为某个特性实现另一个版本时，Fallback 很适合用做参考或者检查点。例如你正在编写一个优化的平台相关的内存复制函数，你可能会使用 memcpy 作为 Fallback，以便测试、和将你的代码移至新平台。

没有这个 Fallback，你没有办法执行回归测试，来确保底层代码没有微妙地偏离你的高层实现。

汇编语言代码片断看上去和同样功能的高级代码所执行的动作有一些不同，因为汇编代码不需要遵守高级语言的语义。例如，你可以编写一小段浮点运算的汇编代码，将两个浮点数相加并存储结果为整数：

```
; sample code to compute a+b and store the resulting
; integer version into a variable called result.
; This is Intel 80x87 FPU code.
fld a          ; load 'a' onto floating point stack
fadd b         ; add 'b' to stop of stack
fistp result   ; store result as an integer into 'result'
```

上面代码能够工作，但是与下面 C 代码并不匹配：

```
result = (int) (a + b);
```

C 对浮点到整数转换强加了自己的规则——明确地规定结果必须被截断（向零舍入）。然而汇编语言的版本，则是使用当前有效的舍入模式，通常是向最近的整数舍入，不过也可能根据 FPU 控制寄存器的当前状态而不同。

这意味着汇编语言版本 4.5 加上 1.1 可能会得到结果为 6 ($4.5 + 1.1 = 5.6$ ，舍入到最近的整数就是 6)，而 C 版本的结果由于小数部分被丢弃而总是 5。

拥有一个 C 版本的 Fallback，通过回归测试来验证汇编版本与高级语言版本的结果，你就可以确保尽可能早地捕获到所有异常情况。还是以前面代码为例，你可能会拥有下面 Fallback：

```
extern int add_two_floats_C(float a, float b);
extern int add_two_floats_asm(float a, float b);
void test_add_two_floats()
{
    int i;
    /* 使用相同种子以便重现 */
    srand(3);
    for (i = 0; i < TEST_ITERATIONS; i++)
    {
        /* test using values from -RAND_MAX/2 to +RAND_MAX/2
         * with a random fractional component thrown in */
        float a = (float) (rand() - RAND_MAX / 2) +
                  (rand() / (float) RAND_MAX);
        float b = (float) (rand() - RAND_MAX / 2) +
                  (rand() / (float) RAND_MAX);
        assert(add_two_floats_C(a, b) == add_two_floats_asm(a, b));
    }
}
```

这是一个并不怎么健壮的简单例子，但是它足够演示回归测试的意图。生产版本的实现需要更加彻底，并提供更多特性，例如日志和指定容忍的误差等功能。

选择不同实现可以通过条件编译、函数指针、或者类层次的虚拟函数来完成，正如本章前面“分配抽象”一节中所描述的。

当拥有许多底层代码的程序开始出现奇怪的 Bug 时，有一个高层的 Fallback 将会拯救你，特别是对于边界情况的验证测试不匹配时。当奇怪的行为出现时，你至少能够减少底层实现替换为 Fallback 的参考实现。如果因此而 bug 得以消失，那很有可能就是那一段底层代码所引起的。

最后当你移植到新平台时，与底层代码等同的高层参考实现会使移植容易许多。基于 Intel PC 的优化代码在移植到 PowerPC 时，如果没有一个高层的 Fallback 参考实现，将会非常地困难。

高层参考实现大大降低了移植到新平台的难度，即使你已经有了许多不可移植的底层源代码。没有它们，每次你迁移到新体系架构时都必须从头开始。

register 关键字

C 和 C++ 允许程序员提示或请求某个变量应该存储在寄存器中，前提是程序员非常确定该变量将频繁使用，如下所示：

```
register int counter;
for (counter = 0; counter < SOME_VALUE; counter++)
{
    /* do something */
}
```

register 存储类型的原理初看上去好像非常合理（“请快速访问这个变量，而不用我使用汇编来重新编写这个函数”），编译器的高级优化技术已经使多数现代软件能够自动使用这个特性。但是实际上许多编译器选择直接忽略 register（C/C++ 标准并不强制要求实现 register 关键字），因为强制将某个变量放置在寄存器中，可能会影响编译器的优化器。

register 关键字是一个落伍的事物，最好避免使用它。除非能够证明使用 register 确实提高了性能。

外部 vs 内嵌 asm 文件

当汇编代码必须整合到项目中去时，通常都将它放在单独的外部文件中，这样适合于使用独立的工具来处理它（汇编器）。这样做一般都比较讨厌和繁琐，因为其它工具如调试器和性能 profiler 不知道如何处理汇编文件。此外程序员还需要处理许多细节，例如调用和命名规范。为了解决这个问题，有一些编译器（包括 Microsoft Visual C++ 和 GCC）提供直接将汇编语言嵌入到 C/C++ 代码的功能。例如在 Microsoft Visual C++ 中你可以这样做：

```
int add_two_floats_C(float a, float b)
{
    int result;
    __asm fld a
    __asm fadd b
    __asm fistp result
    return result;
}
```

内嵌汇编指令的优点是能够与项目其余部分结合更为紧密。调试器和性能 profiler 处理这种代码要比处理单独文件更加容易，而且编译器还处理了麻烦的不同调用规范的入口和退

出代码。

例如上面的函数如果使用汇编来编写，则多数代码将是处理程序员很少关心的细节，如下：

```
PUBLIC _add_two_floats
_TEXT SEGMENT
_add_two_floats PROC NEAR
    push ebp
    mov ebp, esp
    fld DWORD PTR [ebp + 8]
    fadd DWORD PTR [ebp + 12]
    fistp DWORD PTR [ebp - 4]
    mov eax, DWORD PTR [ebp - 4]
    mov esp, ebp
    pop ebp
    ret 0
_add_two_floats ENDP
_TEXT ENDS
```

外部汇编的版本非常地啰嗦而且并未带入些许好处，而且如果汇编代码直接嵌入到 C 代码中，出错的机率也会小很多。编译器负责清除堆栈、分配变量、寻找堆栈中的变量、以及管理所有其它细节。内嵌汇编的缺点是依赖于编译器厂商正确地实现这个功能，因为内嵌汇编是编译器经常出现问题的领域之一。例如有时候内嵌汇编器并不支持较新的指令，更糟的是当遇到内嵌汇编时有些编译器会默默地禁用优化措施。

底层系统编程是现实世界里一个必须的邪恶事物，但是只要加以正确的预防措施和抽象，在接触最底层系统的同时又不牺牲平台的可移植性，也是可以做到的。

第4章 编辑与源码控制

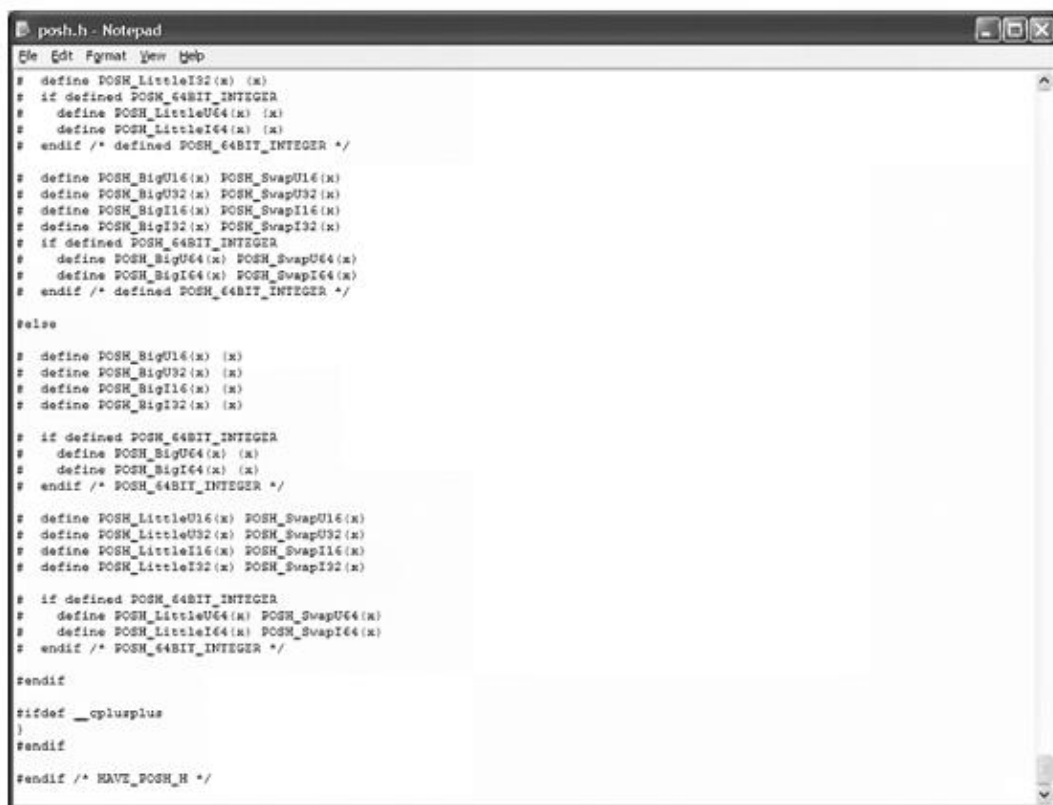
在你考虑编写的代码是否存在问题之前，你必须处理其它一些更为基础的事情：将文件获取到目标平台以便进行编辑。这项工作通常并不像你想象的那么简单，除非你非常幸运能够使用一个单一平台面向多目标的环境。本章讲解编写跨平台软件时如何编辑和管理源文件。

文本文件的行结束符差异

在不同操作系统上创建和编辑的文件文件，必须处理行结束符的不同类型（参考第十三章关于文件系统的讨论）。DOS 和 Windows 使用回车/换行组合（\r\n）来标识行结束；Unix 则使用换行（\n）；而 OS X 之前的 Macintosh 则使用回车（\r）。

在编写多个操作系统的源代码时这是一个问题。如果一个文件在 Unix 中创建，那么在 Windows 机器上就有可能无法正确地进行编辑。

例如图 4-1 显示了在 Windows 记事本中正确装载的 posh.h 文件的部分内容。



```
# posh.h - Notepad
File Edit Format View Help

/* define POSH_LittleI32(x) (x)
 * if defined POSH_64BIT_INTEGER
 *   define POSH_LittleU64(x) (x)
 *   define POSH_LittleI64(x) (x)
 * endif /* defined POSH_64BIT_INTEGER */

/* define POSH_BigU16(x) POSH_SwapU16(x)
 * define POSH_BigU32(x) POSH_SwapU32(x)
 * define POSH_BigI16(x) POSH_SwapI16(x)
 * define POSH_BigI32(x) POSH_SwapI32(x)
 * if defined POSH_64BIT_INTEGER
 *   define POSH_BigU64(x) POSH_SwapU64(x)
 *   define POSH_BigI64(x) POSH_SwapI64(x)
 * endif /* defined POSH_64BIT_INTEGER */

#else

/* Define POSH_BigU16(x) (x)
 * Define POSH_BigU32(x) (x)
 * Define POSH_BigI16(x) (x)
 * Define POSH_BigI32(x) (x)

/* if defined POSH_64BIT_INTEGER
 *   define POSH_BigU64(x) (x)
 *   define POSH_BigI64(x) (x)
 * endif /* POSH_64BIT_INTEGER */

/* Define POSH_LittleU16(x) POSH_SwapU16(x)
 * Define POSH_LittleU32(x) POSH_SwapU32(x)
 * Define POSH_LittleI16(x) POSH_SwapI16(x)
 * Define POSH_LittleI32(x) POSH_SwapI32(x)

/* if defined POSH_64BIT_INTEGER
 *   define POSH_LittleU64(x) POSH_SwapU64(x)
 *   define POSH_LittleI64(x) POSH_SwapI64(x)
 * endif /* POSH_64BIT_INTEGER */

#endif

#ifdef __cplusplus
}
#endif

#endif /* HAVE_POSH_H */
```

图 4-1：使用 Windows 风格的行结束符时记事本正确地显示了 posh.h 的内容。

图 4-2 说明了记事本试图装载使用 Unix 风格行结束符的同一文件时发生的情况。



图 4-2: Windows 记事本装载 Unix 风格行结束符的文件时出现问题。

在 Windows 中,正确处理不同的行结束符是应用的责任,包括装载和保存文件。例如 Emacs 和 Microsoft Visual Studio 在装载和显示图 4-2 的文件时都没有问题。

版本控制系统试图通过在“repository”中转换行结束符为正规格式(通常是 Unix 风格),然后在“check out”时将文件转回系统本地格式,来解决这个问题。当文件完成编辑并再次“check in”时,又再次自动转换为正规的格式。这样做还有助于查看文件差异,因为有一些 diff 程序会错误地认为使用不同行结束符的两个相同文件 100%不相同。

可移植的文件名

今天的每种文件系统对于可接受的文件名都有自己的规定。虽然大多数文件系统已经结束了 MS-DOS 8.3 FAT 文件系统的限制,但是它们之间还是有着重大的区别,因此应该避免使用“奇怪”的名字。

尽量保持文件名简短,如果你希望获得完全的安全,那么采用古老的 8.3 格式也不会有害处,不过可能会导致文件名过于隐晦。合理的限制应该是 31 个字符,对应于 Mac OS 平台上的限制;多数其它操作系统都支持长得多的文件名长度。同样大小写一致也是没有害处的,因为有一些操作系统对大小写不敏感(DOS),其它则是大小写敏感的(Unix/Linux),还有一些则是大小写不敏感且 case-retentive(Windows)。最后应该避免标点字符和空格。如果你想获得最大的可移植性,那么就只能使用字母、数字、和下划线。更多细节请参考第十三章关于文件系统的讨论。

文字扩展名是另一个头疼的东西。DOS 和 Windows 通过扩展名来确定文件类型,例如.doc、.txt、.html 等等。Unix 和 Linux 使用文件扩展名作为数据类型,但是使用文件属性

来区分程序和数据。**Mac OS 9** 忽略文件扩展名，通过检查文件元数据来确定如何运行它。源代码文件名同样也有这个问题。

有许多不同的方法来标识 **C++** 源文件：**.C**、**.cc**、**.cpp**、**.c++** 和 **.cxx**，头文件则有 **.h**、**.hh**、**.hpp** 或者 **.hxx** 扩展名。这个没有官方标准，但是今天主流的编译器都接受 **.cpp/.h** 作为标准。如果你在古老的开发环境中工作，而它有自己可接受的 **C++** 文件名规则，你可以使用脚本来批量重命名，或者如果操作系统允许的话，也可以创建合适扩展名的符号链接。

源码控制

当你在新平台进行工作时，你需要使用版本控制系统来跟踪所有的修改。每次在一个系统中编辑文件，并提交至主源码库中时，任何修改都必须被明确标识、存储、和容易获取。这样可以避免“为什么我的软件在 **Windows** 平台做了这个小小的修改，**Solaris** 的版本就会认为数据库已经损坏呢？”

源码控制系统

粗略算起来，有超过 300 种不同的源码控制系统存在，不过真正常用的只有那么几种。理想的版本控制系统，通常会提供管理修订和移动文件到新平台的功能（而不是手工复制文件），而且应该是跨平台的。幸运的是有很多源码控制系统都提供这些功能。

rcs

修订控制系统（**rcs**）最早由 **Walter Tichy** 于 1980 年代早期开发，是第一个被广泛采用的跨平台源码控制系统。当时 **AT&T** 的私有源码控制系统（**SCCS**）也移植到了 **Unix System V**，**rcs** 的价格（免费）和开源本质使它比 **SCCS** 更为流行。**rcs** 可能仍然是目前最流行的版本控制系统，主要是因为其它更高级的版本控制系统都是基于 **rcs** 的基本原理。按今天的标准来看 **rcs** 可能是相当原始的，但是它的影响和普遍性是很难抹杀的。

cvs

并发版本系统（**cvs**）最早是 **rcs** 的前端，实现了文件的同时编辑（非互斥）。**SCCS** 和 **rcs** 都要求用户锁定源文件来进行互斥访问，因此当一个文件被锁定为写时，其他人就不能再编辑它。（当然，你可以使用一些不安全的手段来打破这个规则，比如破坏本地锁，并且在其他人释放锁时立即锁定该文件）。

对于大型项目来说，互斥锁定政策成为开发的严重障碍。有时候某个程序员需要同时编辑许多源文件，因此必须锁定所有这些文件，然后才能开始工作。到五点钟时，如果这个程序员直接回家，那项目组其他成员将一直无法访问源码库里的许多文件，直到该程序员回来。这种情况太常见了，于是只能规定限制每个人最多能拥有锁的数量。

cvs 也有它的麻烦之处。随便问一个 **cvs** 的用户，她可能会五分钟连续不断地说上 **cvs** 的缺点。即使是最常见的操作，如文件重命名、删除、合并、分支、移动、以及添加，都麻烦且困难。更新到 **repository** 也不是自动完成的，这意味着备份一组修改将是一项复杂的工作。

此外二进制文件还必须被特别地标志出来，否则经常会出错。这是由于 **cvs** 解决多平台文本文件转换所采用的方式导致的一个副作用：自动将主机文件格式转换为标准文本格式。当 **cvs** 添加一个未标识的二进制文件到 **repository** 时，如果提交者忘了指定 **-kb** 标志（二进制

check in)，将会产生非常坏的后果。文件中的任何对应于回车和换行字符的二进制值都会被转换，通常都会导致下次 check out 该二进制文件时使文件损坏。（“自动识别二进制文件”作为修订控制系统的主要卖点，对于 cvs 来说无疑是个杯具）。

即使有这些缺点，对于跨平台/开源开发者来说，cvs 仍不失为一个卓越的版本控制系统。虽然粗糙、古怪、问题多多，使用困难，但 cvs 有充分的资源、书籍、前端用户界面（TortoiseCVS 是其中最流行的一个）、和许多有经验的用户，因此对于大多数团队来说 cvs 的整合和使用是比较直观的。cvs 在许多平台都可用，再加上它的开源特性，都有相当的优势。cvs 也是著名开源网站 SourceForge.net 所采用的修订控制系统，sf.net 是全世界最大的开源软件库社区。

Perforce

Perforce（也被许多用户称为 P4）是一个商业的修订控制产品，由于它的健壮性、性能、广泛的可用性、以及技术支持（Perforce 拥有非常优秀的技术支持），在商业开发领域非常流行。不像许多其它商业应用，Perforce 非常强调可移植性；它甚至在一些非常无名的平台也可使用（IBM OS/390、Amiga、OpenVMS 等等）。

Perforce 提供 cvs 缺少的一些特性。它的二进制文件处理比 cvs 要可靠许多，而且支持原子操作。一组文件在 check in 时是作为单一的单元进行的（类似于修订集）。如果任何文件提交到 repository 失败，则所有文件都会回退，因此阻止了不同步的 check-in。Perforce 还拥有自己的安全模型，而 cvs 则依赖于服务器本地操作系统来提供安全性。Perforce 这种内部的一致性能给用户提供更加统一的体验。

Perforce 是一个拥有专业支持商业工具，相反 cvs 则需要开源社区或者第三方服务（例如 Ximbiot）来进行支持。开源项目也可以免费使用 Perforce，这是值得钦佩的（尽管并没有太大的实际意义，因为开源项目都倾向于使用非商业工具）。

综上所述，Perforce 对于闭源项目是收费的，它也不是开放软件，而这两个因素在很多时候已经足够阻止很多人使用它了。

BitKeeper

BitKeeper 是另一个源码控制工具，它由于许可的原因一直饱受争议。BitMover 是 BitKeeper 的母公司，使它的这个软件“大部分免费”，这意味着 BitKeeper 可以免费使用，但是功能受限，因此受到一些开源拥挤者的讨厌。

BitKeeper 最值得一提的是它成功地赢得了 Linux 内核维护者的信赖，并用作他们的源码控制工具（译注：Linux 可能并不喜欢 BitKeeper，所以才自己又搞了个 git 出来，Linux 内核团队现在用的就是它，而且 git 势头貌似很猛，越来越流行）。技术上讲，BitKeeper 在所有方面都要比 cvs 优美。作为一个商业系统（和 Perforce 一样），它拥有很多重要的资源能够使自己成为该类工具的佼佼者。BitKeeper 支持分布式开发（而不像其它工具需要 client/server 模型），这一点使它更容易为远程开发者使用。它的点对点设计和复制数据库系统还能够确保高可靠性和高性能，这是其它传统工具无法实现的。

Subversion

Subversion 的定位是“新的 cvs”，它几乎在所有方面都优于 cvs，几乎解决了用户对于 cvs 的所有抱怨，但是 Subversion 没有采用 BitMover 式的“我们需要重新发明修订控制”的态度。Subversion 采用数据库后端（BerkeleyDB），而不像 cvs 是基于文件后端的。这样确实提供了一些好处，但是也有一个副作用：如果二进制数据库内部出现问题，你可能就无法取回它的内容。

Subversion 目前并未赢得决定性的用户优势，因为 cvs 已经占据了大部分的市场，特别

是开源项目，选择 Subversion 就有可能会疏远很大数量的贡献者（再译注：今天 Subversion 几乎已经完全取代了 cvs，不过本书出版于 2005 年，当时可能 cvs 还是有优势的）。在本书写作的时候，Subversion 刚刚发布 1.0.5 版本，希望提供动量使其能够取代 cvs。我使用的正是 Subversion，但我敢肯定 cvs 还会生存很长一段时间，就像直到今天很多遗留项目还在使用 SCCS 那样。

GNU arch

Subversion 是“新的 cvs”，而开源软件基金会则定位它自己的版本控制系统 arch 为“更新更好的 cvs”。和 Subversion 不同的是，arch 希望能够不受 cvs 固有模式的限制。

例如 arch 最大的威力是它采用了分布式 repository 模型，这不同于多数版本工具使用的单一 repository 模型。这大大地偏离了 cvs 的路线，但是却给 arch 的实现提供了很大的威力（类似于 BitKeeper）。arch 的另一个优点则更加隐秘，包括整棵树 patch、全局项目名称、以及更加“轻量级工具组合”的体系架构。不过在本书写作的时候，arch 还尚未被多数开发者采纳。

通过代理 Checkout

当你的开发平台没有你正在使用的源码控制系统，或者更糟的一种情况是你被强制使用平台相关的源码控制系统（如 Microsoft Visual SourceSafe），应该怎么办呢？虽然如果你使用主流的源码控制系统之一就不会面临这种困境，但是也无法完全避免。比如 BitKeeper 在 Mac OS 的早期版本上就不可用。

好吧！假设你非常不幸地需要移植整个项目到 Mac OS 9，而你之前一直很开心地使用 BitKeeper 作为项目的源码控制系统，那么你就得解决这个问题。你的第一个选择是迁移到另一个完全不熟悉的源码控制系统，这显然是疯狂的，有可能新系统缺乏 BitKeeper 的某些特性，而且最终还可能会不兼容，这不是一个好办法。

如果你运气好的话，你就可以让 Macintosh 来远程 mount 文件系统，使用诸如 Thursby 的 DAVE 文件共享工具。如果手头上有一台 PC，你可以把源码 check out 到这台 PC 中，然后使用 Macintosh 直接编辑这些文件，一旦所有工作都完成，再从 PC 把源码提交上去。不是所有的计算机系统都可以互相 mount 文件系统，但是你还最后有一个解决办法：通过代理 Checkout。

如果你的源码控制系统与特定主机不兼容，并且你不能 mount 目标平台的文件系统（或者 mount 其它机器），那么你就需要把文件代理到新平台。代理系统必须与你的源码控制软件相兼容，你在代理机器上进行锁定和 Checkout 操作，然后将所有文件传输至目标主机。理想情况下你可以使用 FTP 来传输文件，如果没有网络，你就需要将文件打包成某种档案格式（.zip 或 .tar.gz），然后复制该文件到目标主机（通过磁盘、软盘、CD、DVD 等）。

拷贝过去后进行解包，开发完成后，再打包并返回给代理系统，最终将所有修改提交到源码控制系统（祈祷这段时间没有其他人同时做了重大的修改）。你还必须确保你没有在代理系统和远程系统上编辑过任何文件，否则在整合修改到中心 repository 之前，还需要合并代理和远程系统的源码树。

这种通过代理 Checkout 的方法既不优雅也不容易，不过至少还能工作。确保你使用的修订控制系统能够支持你所有的平台才是最佳方法。

构建工具

现在你已经不再担心行结束符、文件名大小写、以及其它文件命名的问题，是时候继续前进并开始编译代码了。现在的问题是选择平台特定的构建工具，还是可移植的构建工具。

平台特定的构建工具

平台特定的构建工具通常都由集成开发环境（IDE）组成，负责管理文件、项目、和依赖关系。IDE 通常提供一些其它的方便特性，包括集成帮助、API 参考、调试器、profiler 等。不同操作系统拥有各自不同的 IDE：

- Windows 的开发有 Microsoft 的 Visual Studio、Metrowerks 的 CodeWarrior、Borland 的 C++builder、以及免费的 Dev-C++（基于 Cygwin 的 GCC 或 MinGW）。
- Mac OS X 上有苹果提供的免费的 ProjectBuilder 和 XCode 环境（只不过是修改版本的 jam、gdb、gcc 上包装了一个华丽的前端），以及 Metrowerks 的 CodeWarrior。
- Linux 既拥有商业开发环境，如 Borland C++Builder-X、Metrowerks CodeWarrior、CodeForge；也有免费开发环境，如 Anjuta Dev Studio、Eclipse、和 KDevelop。
- Sun 公司在 Solaris 和 Linux 上提供商业的 Sun One Studio 环境。

换句话说，不管是商业还是免费，平台特定的集成构建环境都有许多种选择。

IDE 提供友好、容易使用的界面，试图隐藏晦涩的源码依赖关系、编译器和链接器选项、以及调试指令。如图 4-3 显示了 Microsoft Visual C++ 6.0。

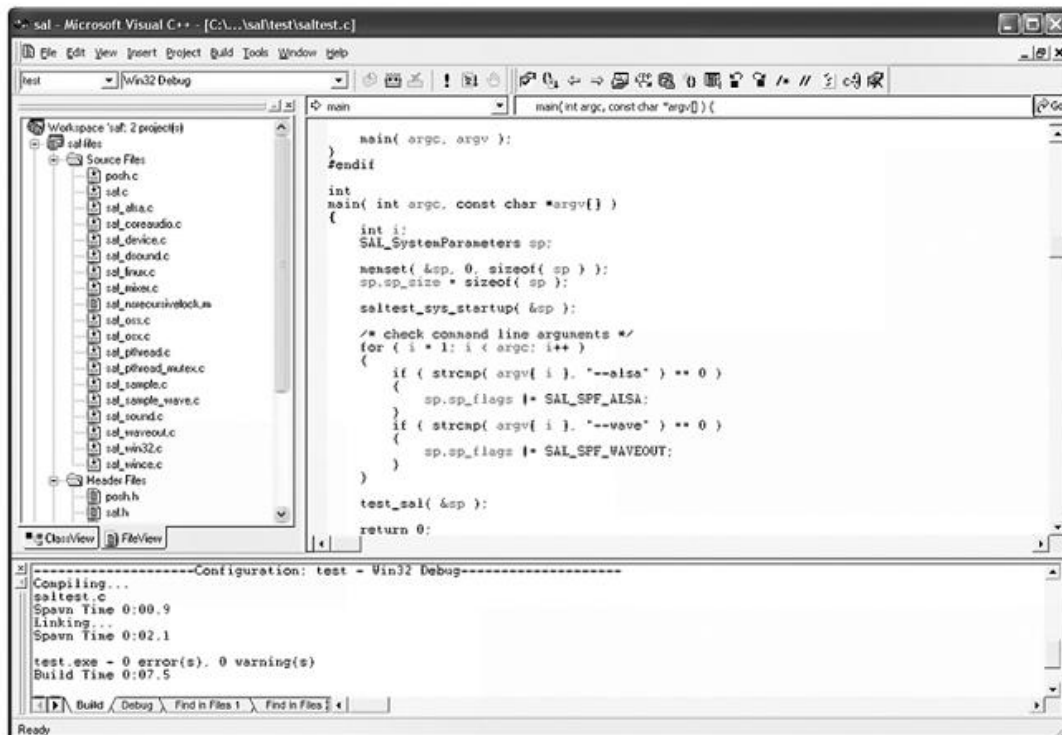


图 4-3：Microsoft Visual C++ 6.0

注意多面板提供了构建和编辑过程的可视响应。左边的工作区面板有项目所有文件的列

表，双击文件名就可以开始编辑；底下的输出窗口显示了构建的结果。重新编译只需要按一个键或者按钮。调试直接在编辑器中进行，因此可以在调试的同一个地方编辑代码。按下 F1 键就可以显示帮助。不同文件之间的依赖关系由 IDE 自动确定，无需用户关心。

对于那些不熟悉特定平台及其工具链的开发者来说，IDE 能够节省大量时间和精力，因为所有的细节都被很好的隐藏了。例如下面的编译命令多少有点复杂：

```
c1 /nologo /MTd /W3 /ZI /Od /D "WIN32" /D "_DEBUG" /D "_CONSOLE"  
/D "_MBCS" /Fp"Debug/test.pch" /YX /Fo"Debug/" /Fd"Debug/" /FD /GZ /c
```

每个文件都需要这样的编译命令！多数开发者都不愿意处理这种细节，除非不得已的情况下。

不过 IDE 也不是万能的。下面是 IDE 的一些缺点：

- 由于 IDE 高度集成，它的使用方式倾向于“要么全部，要么什么也别用”。
- 用每种工具设置一个构建系统需要许多学习和维护。
- 它们通常都不怎么自动化。例如配置一个多机器的定时构建可能会比较麻烦。
- 大多数高质量的 IDE 都是私有且昂贵的，预算紧张的公司和开发者很难承担。
- 由于许多 IDE 绑定到特定的编译器，在不同编译器之间迁移对于开发者来说非常困难，因为这需要修改构建系统。

可移植构建工具

可移植构建工具通过将更多工作交给开发者，来减少工具的负担，从而解决了私有、主机特定的构建环境的许多问题。通常都是由特殊的构建工具（如 make 和 jam）来处理脚本文件（Perl、Python 或者 Shell 编写的自定义程序）。

假如你可以在所有平台执行你的脚本，那在新平台构建只需要针对那个平台的特定环境和工具，增加一个新的入口即可。所有高级的依赖检查和项目管理都可以按可移植的方式来处理。

make

make 是所有构建管理工具的鼻祖，起源于 Unix 开发的早期。那时候贝尔实验室的 Unix 开发者 Stuart Feldman 发现自己和同事总是碰到同样的问题：他们在开发过程中偶然忘记编译某些文件，因此将过时的 object 文件链接到二进制文件中。在 bug 修正和增加特性之后，这些修改竟然在最终可执行文件中神秘地不出现。

为了解决这个问题，Stuart 编写了一个叫做 make 的小程序。理论上 make 的原理是很容易理解的，makefile 文件定义了一组目标、依赖、生成规则、以及各种宏，然后由 make 对 makefile 进行处理，确保所有过时文件都得到适当地更新。

对于相对较小的 Unix 项目，make 工作得非常好。但是随着软件项目的增长，make 的作用也越有限。每次增加一个新特性，makefile 就变得更加地脆弱。实际上如果你调查使用 make 的开发者，大部分人可能都会承认除了增加文件，他们不会去碰 makefile。

为了说明 makefile 的恐怖，下面是 GNU 的 make 指南上的一个例子。这个规则用来产生.c 文件的依赖文件：

```
%.d: %.c  
@set -e; rm -f $@; \  
$(CC) -M $(CPPFLAGS) $< > $@.$$$$; \  
sed 's,\($*\)\.o[ :]*,\1.o $@ : ,g' < $@.$$$$ > $@; \  
第 64 页 / 共 141 页
```



```
rm -f $@.$$$$
```

没有什么比这个更为恐怖了！

make 在现代软件项目上的不足，导致了一些协助工作的产生，例如 GNU 的 **automake** 和 **autoconf**。这些工具执行一些生成 **makefile** 和配置脚本的工作，理论上是要使程序员的开发工作更加容易。不幸的是它们用更加复杂来取代复杂；现在创建 **makefile** 不再困难，但是程序员必须学习怎样编写 **automake** 和配置文件。

make 的流行也对其产生了影响，随着程序本身的流行，产生了过多的克隆（GNU **make**）、替代品（**jam** 和 **scons**，后面会讲到）、变种（Microsoft **NMAKE**）、以及商业版本（**Opus Make**）。

虽然 **make** 有它的缺陷和怪异，它仍然是今天使用最广泛的构建工具。它的普遍、流行、和用户基础确保了 **make** 在所有平台还会存在很久。

SAL 例子：一个简单的 makefile

下面是一个非常简单的 **makefile**，用来构建 SAL 的测试程序：

```
# Simple example makefile that uses GCC and builds the SAL test
CC=gcc
CFLAGS=-g -DSAL_SUPPORT_OSS
LDLIBS=-lpthread
saltest: src/sal.o src/posh.o src/sal_device.o src/sal_mixer.o \
    src/sal_sound.o src/sal_sample.o src/os/sal_pthread.o \
    src/os/sal_pthread_mutex.o src/backends/sal_oss.o \
    src/backends/sal_alsa.o src/os/sal_linux.o \
    src/sal_sample_wave.o \
    test/saltest.o
$(LINK.o) $^ $(LDLIBS) -o $@
```

这个 **makefile** 表达的是“构建一个名叫 **saltest** 的可执行文件，它依赖于下面这些 **object** 文件”。**\$(LINK.o) \$^ \$(LDLIBS) -o \$@** 语句告诉 **make** 怎样构建最终的可执行文件，使用了 **make** 非常晦涩的变量、宏、和隐含规则的语法定义。前三行定义了一些变量，由 **make** 自动插入到构建过程中去（**make** 隐含地理解 **CC** 是用来编译 C 源代码，**CFLAGS** 应该传递给编译器，而 **LDLIBS** 则是应该传递给链接器的库）。

生产版本的 **makefile** 要复杂得多，而且有可能产生多个 **makefile**（递归 **make**），拥有数千行定义、多重依赖、变量、条件执行、以及隐式和显式的生成文件。

jam

jam（just another make）是 Perforce 提供的一个高度可移植的开源构建管理工具，从 **jam** 这个名字你就可以猜测出来，它肯定和 **make** 有许多类似的地方。但是由于它是在 **make** 已经非常流行之后开发的，因此增加了许多的改进和修正，使得 **jam** 更容易使用，也更加强大，包括以下方面：

- **jamfile**（等同于 **make** 的 **makefile**）拥有更清晰和简单的语法。
- 编译器和平台相关的特性（如路径分隔符和编译器选项）被剥离 **jamfile**，存储在一个平台相关的 **jambase** 文件中。
- 通过扫描头文件自动检查依赖关系（不再需要 **make depend**）。
- 非常快速

- 很容易执行并发构建。

使用 jam 的唯一实际缺点是它没有 make 流行,因此使用它可能造成其它开发者的障碍。作为 Mac OS X 开发工具的组成部分, jam 免费而且越来越流行。

脚本和批处理文件

在构建管理工具出现之前, 命令行 shell 脚本 (也叫批处理文件) 已经使用很久了。如果你需要管理一个重复的构建, 可以用你喜爱的脚本语言 (如 csh, sh, DOS BATCH, VMS DCL) 来编写一个简单的脚本来完成。

虽然缓慢、效率低、不可移植, 但脚本语言几乎在任何平台都以某种形式可用。它们实现起来非常直观, 因为通常不需要处理依赖检查等高级概念。

更进一步的做法是使用高级、专门的编程语言 (如 Perl 和 Python) 来编写脚本。这样做比起 make 和 jam 的严格、预定义规则, 可以更好地支持自定义构建过程, 并提供更多的灵活性。scons 是一个用 Python 编写的构建管理工具, 并且使用 Python 作为它的配置语言。

不管是使用 Python、Perl、scons、还是其它脚本语言 (今天在多数平台你都可以使用 Python 和 Perl), 只要平台能够支持该语言, 你就可以获得巨大的威力和可移植性。

SAL 例子: SHELL 脚本

SAL 有一个非常简单的 shell 脚本, 使用 sh 编写, 虽然不那么优雅, 但也完成与 makefile 相同的工作。

好的一面是它完成的事情非常清晰, 因为它只是一个命令:

```
#!/bin/sh
gcc -g src/posh.c src/sal.c src/sal_device.c \
src/sal_mixer.c src/sal_sound.c src/sal_smaple.c \
src/os/sal_pthread.c src/os/sal_pthread_mutex.c \
src/backends/sal_oss.c src/backends/sal_alsa.c \
src/os/sal_linux.c src/sal_sample_wave.c test/saltest.c \
-lpthread -o test/test
```

当然 shell 脚本也提供条件语句、执行流控制、和变量, 因此添加命令行参数支持也是非常简单的:

```
#!/bin/sh
if ["$1" = "cpp"]; then
    CPPDEF="-x c++"
    CPPLIB=-lstdc++
else
    CPPDEF=
    CPPLIB=
fi
ALSALIB=
ALSADEF=
OSSDEF=
if ["$2" = "alsa"] || ["$3" = "alsa"] ; then
    ALSALIB=-lasound
```

```
        ALSADEF=-DSAL_SUPPORT_ALSA
    fi
    if ["$2" = "oss"] || ["$3" = "oss"] ; then
        OSSDEF=-DSAL_SUPPORT_OSS
    fi
    gcc $OSSDEF $ALSADEF $CPPDEF -g src/posh.c src/sal.c
src/sal_device.c \
        src/sal_mixer.c      src/sal_sound.c      src/sal_sample.c
src/os/sal_pthread.c \
        src/os/sal_pthread_mutex.c src/backends/sal_oss.c \
        src/backends/sal_alsa.c src/os/sal_linux.c src/sal_sample_wave.c
\
    test/saltest.c $ALSALIB -lpthread $CPPLIB -o test/test
```

上面例子根据 `c` 或 `cpp` 命令行参数来决定源码采用 `C` 或 `C++` 进行编译。同时还支持 `alsa` 和 `oss` 命令行参数，用来确定 `SAL` 应该支持哪个音频子系统。

编辑器

跨平台的编辑器不如跨平台的源码控制系统重要，但是也能提供一些帮助。跨平台编辑器的重要性实际上要看你对编辑环境有多挑剔。

有一些程序员几乎可以接受使用任何编辑器而不抱怨；其它则花费数小时甚至数天的时间对编辑环境进行精确配置。如果强迫他们在陌生的编辑环境中编辑哪怕是一点点文本，他们也会哭诉和抱怨说这工作没法完成，因为没有适当配置的工具。

如果你属于后面一种，最好是选择一个可移植的编辑器，并且能够在短时间内轻松地完成安装和配置。

一个显而易见的选择是 `Emacs`，它被开玩笑称为“好的操作系统和坏的编辑器”——这里的重点是 `Emacs` 远远不只是一个编辑器。`Emacs` 广泛可用而且安装简单，本地配置也可以轻松移到新系统，只需复制自定义的 `.emacs` 文件即可。还有很多其它的跨平台编辑器，包括简洁而普遍的 `vi` 和 `VIM`。当然大型的跨平台 IDE（如 `CodeWarrior` 和 `Eclipse`）都有内置的编辑功能。

小结

可移植软件开发很多时候都强调编程和编码习惯，而忽略了“简单”的第一个步骤：迁移源码到新平台和在新平台上进行编辑。在本章，我们讲解了许多这个步骤可以采用的方法，包括在不同平台编辑文件和使用源码控制系统。

第5章 处理器差异

各种处理器体系架构之间的区别可能是巨大的，例如桌面 RISC 芯片。在高端桌面处理器（如 IBM G5）和低端处理器（如 Intel xScale）之间成功迁移代码，所需的工作量通常都令人吃惊。

计算机处理器在存储需求（数据对齐和字节序）、数据大小和格式、还有性能等各方面的设计都有巨大差异。本章讲解在不同体系架构的处理器之间迁移，你将会遇到的一些常见问题。

注意当你从高性能系统迁移到低端系统时，有可能所有特性都可以被移植，但是你选择的算法和数据结构可能无法整洁地缩小。这个问题在第十四章讨论伸缩性时会得到解决。

对齐

多数处理器喜欢（甚至要求）访问对齐的内存。这意味着当处理器访问 n 字节长度的数据类型时，数据块的起始地址必须是 n 的倍数。例如四字节变量应该拥有四字节对齐的边界（地址为四的倍数）；二字节的变量应该在二字节对齐的边界（地址是二的倍数）；依此类推。

不同处理器通常对内存访问有不同的要求。例如 Intel x86 架构允许未对齐的内存访问，但是会有极大的性能损失；而错误对齐的内存访问在很多 RISC 处理器上则会导致处理器故障，要么引起崩溃，或者如果使用软中断处理了故障，则会产生非常缓慢的未对齐访问（完全在软件中处理访问）；而在 ARM 的嵌入式处理器中，错误对齐的内存访问会导致错误的数
据，这是最差的一种结果，因为它可能导致错误的行为被默默地接受。

NOTE 有一些带有内存管理单元的 ARM 处理器，会实现可选的对齐检查，但是这个特性在整个 ARM 家族并不普遍。

要获得最大的可移植性，对齐应该强制为可能的最高大小。任何指针操作的技巧都应该避免，因为它们可能导致错误对齐的内存访问。更常见的内存对齐错误发生在通过非法指针转换来访问内存缓冲区时。

指针转换导致的错误对齐访问

SAL 有一个 WAVE 文件解析函数：_SAL_create_sample_from_wave()，本可以直接而简单地将缓冲区转换为合适的结构体：

```
typedef struct
{
    char wh_riff[4];
    sal_u32_t wh_size;
    char wh_wave[4];
    char wh_fmt[4];
    sal_u32_t wh_chunk_header_size;
} _SAL_WaveHeader;
sal_error_e
```

```

SAL_create_sample_from_wave(
    SAL_Device *device,
    SAL_Sample **pp_sample,
    const void *kp_src,
    int src_size)
{
    _SAL_WaveHeader *pwh = (_SAL_WaveHeader *) kp_src;
    .
    .
    .
    /* verify that this is a legit WAV file
       NOTE: wf_chunk_header_size might be a misaligned access! */
    if (strncmp(pwh->wh_riff, "RIFF", 4) ||
        strncmp(pwh->wh_wave, "WAVE", 4) ||
        pwh->wh_chunk_header_size != 16)
    {
        return SALERR_INVALIDPARAM;
    }
    return SALERR_OK;
}

```

依赖于 `kp_src` 的不同对齐, 比较语句使用 `pwh->wh_chunk_header_size` 可能会导致错误对齐的访问。大多数情况下这都不会发生, 因为大部分缓冲区都以段或页边界分配。如果你编写了一个按字节边界工作的缓冲区分配/释放子系统, 这就可能真的会产生问题。

一个稍微缓慢但是安全的解决办法, 是将输入数据复制到一个结构体中, 而编译器会负责正确地对齐结构体:

```

sal_error_e
SAL_create_sample_from_wave(
    SAL_Device *device,
    SAL_Sample **pp_sample,
    const void *kp_src,
    int src_size)
{
    _SAL_WaveHeader wh;
    .
    .
    .
    /* this still makes assumptions about padding,
       byte ordering, etc. */
    memcpy(&wh, kp_src, sizeof(wh));
    /* verify that this is a legit WAV file
       NOTE: wf_chunk_header_size will be aligned correctly */
    if (strncmp(wh.wh_riff, "RIFF", 4) ||
        strncmp(wh.wh_wave, "WAVE", 4) ||
        wh.wh_chunk_header_size != 16)

```

```

    {
        return SALERR_INVALIDPARAM;
    }
    return SALERR_OK;
}

```

然而内存拷贝并不处理字节序和填充问题，因此通常你需要解析原始内存并转换为正确的格式，如下：

```

const sal_byte_t *kp_bytes = (const sal_byte_t *) kp_src;
.
.
.
/* read out wave header */
memcpy(wh.wh_riff, kp_bytes, 4);
kp_bytes += 4;
wh.wh_size = POSH_ReadU32FromLittle(kp_bytes);
kp_bytes += 4;
memcpy(wh.wh_wave, kp_bytes, 4);
kp_bytes += 4;
memcpy(wh.wh_fmt, kp_bytes, 4);
kp_bytes += 4;
wh.wh_chunk_header_size = POSH_ReadU32FromLittle(kp_bytes);
kp_bytes += 4;
/* verify that this is a legit WAV file */
if (strncmp(wh.wh_riff, "RIFF", 4) ||
    strncmp(wh.wh_wave, "WAVE", 4) ||
    wh.wh_chunk_header_size != 16)
{
    return SALERR_INVALIDPARAM;
}
return SALERR_OK;

```

union 是一个便利的机制，能够确保两种不同类型之间的对齐访问。例如 Motorola 的 SIMD AltiVec 指令集，在浮点和向量（SIMD）单元之间传输数据时，要求 16 字节对齐：

```

/* Based on code from:
   http://developer.apple.com/hardware/ve/alignment.html */
/* NOTE: "vector" is a keyword specific to the
   AltiVec enabled GCC compilers */
vector float FillVectorFloat(float f1, float f2, float f3, float f4)
{
    /* this union guarantees that the 'scalars' array will be
       aligned the same as the 'vector float v' */
    union
    {
        float scalars[vec_step(vector float)];
    }
}

```

```
        vector float v;  
    } buffer;  
    /* copy four floating point values into array of scalars */  
    buffer.scalars[0] = f1;  
    buffer.scalars[1] = f2;  
    buffer.scalars[2] = f3;  
    buffer.scalars[3] = f4;  
    /* return vec float equivalent */  
    return buffer.v;  
}
```

字节序和大小端

多字节数据类型如整数，可以被表示为两种形式：小端和大端，以指示数据类型的字节顺序。在小端（如 Intel x86）体系架构上，低位字节被存放在前面（也就是低地址）。大端（如 Motorola PowerPC）体系架构则把高位存放在前面。

也有一些机器是混合大小端和双端。例如 PDP-11 把 32 位值存储为两个大端的 short（最高位字节在低地址），但是低位的 short 存储在低地址（2-3-0-1，1 对应于最低位地址）。许多现代 CPU 和协处理器（网络处理器、图形处理单元、和音频芯片）都支持双端操作，可以按大端和小端模式工作。这有助于性能和可移植性，不幸的是，应用很少能控制这个特性。操作系统或设备驱动通常控制了特定硬件的大小端模式。

大端 vs 小端

考虑下面例子：

```
union  
{  
    long l; /* assuming sizeof(long) == 4 */  
    unsigned char c[4];  
} u;  
u.l = 0x12345678;  
printf("c[0] = 0x%x\n", (unsigned) u.c[0]);
```

上面例子小端和大端的值分别是：

地址	小端值	大端值
&c[0]	0x78	0x12
&c[1]	0x56	0x34
&c[2]	0x34	0x56
&c[3]	0x12	0x78

在小端机器上运行上面程序时，输出如下：

```
c[0] = 0x78
```

而在大端 CPU 上则输出：

```
c[0] = 0x12
```

这样就产生了一个重大的问题：多字节数据不能在不同字节序的处理器之间直接共享。例如你把一些多字节数据写到文件，然后在另一种大小端的机器上把它读回来，数据可能会出现混乱，就像这样：

```
void write_ulong(FILE *fp, unsigned long u)
{
    /* BAD! Storing to disk in 'native' format of the current CPU */
    fwrite(&u, sizeof(u), 1, fp);
}

unsigned long read_ulong(FILE *fp)
{
    unsigned long u;
    /* BAD! Blithely assuming that the format on disk matches
       the processor's byte ordering! */
    fread(&u, sizeof(u), 1, fp);
    return u;
}
```

字节序例子：PowerPC vs Intel x86

现在我们考虑一个例子，来显示字节序的影响。如果你在 PowerPC 上执行下面代码：

```
write_ulong(fp, 0x12345678);
```

然后在 Intel x86 上运行下面代码：

```
unsigned long ul = read_ulong(fp);
```

你可能会大吃一惊：变量 ul 在 Intel 处理器上将包含值 0x78563412。原因是 PowerPC 将数据存储在磁盘的格式（存储为 0x12, 0x34, 0x56, 0x78），然后被读取出来并存储到 ul 中（0x12 是最低地址，0x34 其次，依此类推）。这可能是程序员在不同平台之间进行迁移时遇到的最常见的 bug 之一，如果不是最常见的话。

标准存储格式

不同字节序问题的一个解决办法就是以某种标准字节序来存储数据。在不匹配这个标准格式的处理器上运行的软件，必须手动转换标准格式为处理器的本地格式。另一个解决办法是将数据直接存储为平台本地字节序，然后在文件头标记所使用的字节序。有一些文件格式（如 TIFF）就是通过这个方法指定大小端。

NOTE 有些文件格式（如 TIFF 图形格式）并不拥有固定大小端，程序必须检查 TIFF 头来确定字节序。

现在我们假设标准存储格式是大端。你可以如下编写本节开头的代码：


```

void write_ulong(FILE *fp, unsigned long u)
{
    unsigned char c[4];
    c[0] = (unsigned char) (u >> 24);
    c[1] = (unsigned char) (u >> 16);
    c[2] = (unsigned char) (u >> 8);
    c[3] = (unsigned char) u;
    fwrite(c, sizeof(c), 1, fp);
}

unsigned long read_ulong(FILE *fp)
{
    unsigned char c[4];
    unsigned long u = 0;
    fread(c, sizeof(c), 1, fp);
    u |= ((unsigned long) c[0]) << 24;
    u |= ((unsigned long) c[1]) << 16;
    u |= ((unsigned long) c[2]) << 8;
    u |= ((unsigned long) c[3]);
    return u;
}

```

这段代码没有对数据在内存中的组织和存储做任何假设；相反它直接通过位运算取得相关的字节。这个代码唯一的问题是即使存储格式和处理器本地格式匹配，仍然需要转换而带来一定的性能损失。

要优化这种情况，你可以检查字节序，只有需要时才执行手工构造，如下：

```

unsigned long read_ulong(FILE *fp)
{
    unsigned char c[4];
    unsigned long u = 0;
    fread(c, sizeof(c), 1, fp);
    /* this function is discussed next */
    if (is_big_endian())
    {
        /* this is fine, but only on big-endian systems */
        /* Obviously you'd move this conditional outside */
        /* this loop for performance */
        return *(unsigned long *)c;
    }
    u |= ((unsigned long) c[0]) << 24;
    u |= ((unsigned long) c[1]) << 16;
    u |= ((unsigned long) c[2]) << 8;
    u |= ((unsigned long) c[3]);
    return u;
}

```

检查大小端的 `is_big_endian()` 函数实现起来很简单，以下代码片断就可以说明问题：

```
int is_big_endian(void)
{
    union
    {
        unsigned long l;
        unsigned char c[4];
    } u;
    u.l = 0xFF000000;
    /* big-endian architectures will have the MSB
       at the lowest address */
    if (u.c[0] == 0xFF)
        return 1;
    return 0;
}
```

NOTE 如果你能够控制存储格式，那么就可以避免通过文本格式存储数据来解决字节序问题。第十五章将更详细地讨论这个问题。

固定网络字节序

TCP/IP 网络协议规定使用大端网络字节序，这意味着提供给网络层的参数（但不是指实际传输的数据）必须是大端格式。

例如 32 位 IPv4 地址和 16 位端口，包括 `sockaddr` 结构体中的所使用成员，都必须来自网络字节序。因此下面代码：

```
struct sockaddr_in svr;
/* UNPORTABLE: sin_port is expected to be in network byte order! */
svr.sin_port = PORT_NO;
```

会在小端机器上神秘地失败，因为 `PORT_NO` 使用了错误的字节序。

为了解决这个问题，BSD socket 和 Windows API 提供了几个帮助函数，用来转换网络字节序：

```
uint32_t htonl(uint32_t hostlong); /* host to network long */
uint16_t htons(uint16_t hostshort); /* host to network short */
uint32_t ntohl(uint32_t netlong); /* network to host long */
uint16_t ntohs(uint16_t netshort); /* network to host short */
```

可移植的端口赋值语句应该是这样：

```
struct sockaddr_in svr;
/* convert from host to network ordering */
svr.sin_port = htons(PORT_NO);
```

字节序不应该成为多数程序员需要关注的事情，除非他们存储/装载二进制数据、或者直接从更大的多字节数值中引用抽取字节。只要你存储时转换到标准的字节序格式，读取时从标准格式转换到本地格式；并避免直接引用抽取多字节数值，处理器大小端差异就不会是一个主要的问题。

POSH 例子：字节序能力

POSH 提供一组字节序辅助函数和宏。首先它拥有一组字节交换函数，适合用来转换大端和小端：

```
extern posh_u16_t POSH_SwapU16(posh_u16_t u);
extern posh_i16_t POSH_SwapI16(posh_i16_t u);
extern posh_u32_t POSH_SwapU32(posh_u32_t u);
extern posh_i32_t POSH_SwapI32(posh_i32_t u);
```

此外，POSH 还有序列化和反序列化函数，自动转换本地格式为用户定义的目标格式：

```
extern posh_u16_t *POSH_WriteU16ToLittle(void *dst, posh_u16_t value);
extern posh_i16_t *POSH_WriteI16ToLittle(void *dst, posh_i16_t value);
extern posh_u32_t *POSH_WriteU32ToLittle(void *dst, posh_u32_t value);
extern posh_i32_t *POSH_WriteI32ToLittle(void *dst, posh_i32_t value);
extern posh_u16_t *POSH_WriteU16ToBig(void *dst, posh_u16_t value);
extern posh_i16_t *POSH_WriteI16ToBig(void *dst, posh_i16_t value);
extern posh_u32_t *POSH_WriteU32ToBig(void *dst, posh_u32_t value);
extern posh_i32_t *POSH_WriteI32ToBig(void *dst, posh_i32_t value);
extern posh_u16_t POSH_ReadU16FromLittle(const void *src);
extern posh_i16_t POSH_ReadI16FromLittle(const void *src);
extern posh_u32_t POSH_ReadU32FromLittle(const void *src);
extern posh_i32_t POSH_ReadI32FromLittle(const void *src);
extern posh_u16_t POSH_ReadU16FromBig(const void *src);
extern posh_i16_t POSH_ReadI16FromBig(const void *src);
extern posh_u32_t POSH_ReadU32FromBig(const void *src);
extern posh_i32_t POSH_ReadI32FromBig(const void *src);
```

在这些函数的上面是转换值为本地格式的宏。这些宏根据当前平台的字节序来进行定义：

```
#if defined POSH_LITTLE_ENDIAN
# define POSH_LittleU16(x) (x)
# define POSH_LittleU32(x) (x)
# define POSH_LittleI16(x) (x)
# define POSH_LittleI32(x) (x)
# if defined POSH_64BIT_INTEGER
#   define POSH_LittleU64(x) (x)
#   define POSH_LittleI64(x) (x)
# endif /* defined POSH_64BIT_INTEGER */
# define POSH_BigU16(x) POSH_SwapU16(x)
# define POSH_BigU32(x) POSH_SwapU32(x)
# define POSH_BigI16(x) POSH_SwapI16(x)
# define POSH_BigI32(x) POSH_SwapI32(x)
# if defined POSH_64BIT_INTEGER
#   define POSH_BigU64(x) POSH_SwapU64(x)
#   define POSH_BigI64(x) POSH_SwapI64(x)
```

```
# endif /* defined POSH_64BIT_INTEGER */
#else
# define POSH_BigU16(x) (x)
# define POSH_BigU32(x) (x)
# define POSH_BigI16(x) (x)
# define POSH_BigI32(x) (x)
# if defined POSH_64BIT_INTEGER
#   define POSH_BigU64(x) (x)
#   define POSH_BigI64(x) (x)
# endif /* POSH_64BIT_INTEGER */
# define POSH_LittleU16(x) POSH_SwapU16(x)
# define POSH_LittleU32(x) POSH_SwapU32(x)
# define POSH_LittleI16(x) POSH_SwapI16(x)
# define POSH_LittleI32(x) POSH_SwapI32(x)
# if defined POSH_64BIT_INTEGER
#   define POSH_LittleU64(x) POSH_SwapU64(x)
#   define POSH_LittleI64(x) POSH_SwapI64(x)
# endif /* POSH_64BIT_INTEGER */
#endif
```

有了这些宏，应用就可以轻易地转换任何字节序，而不需要显式地检测当前平台的大小端。前面读取一个 `unsigned long` 的函数可以修改为：

```
unsigned long read_ulong(FILE *fp)
{
    unsigned char c[4];
    unsigned long u;
    fread(c, sizeof(c), 1, fp);
    return POSH_ReadU32FromBig(c);
}
或者
unsigned long read_ulong(FILE *fp)
{
    unsigned long u;
    fread(u, sizeof(u), 1, fp);
    return POSH_BigU32(u);
}
```

带符号整型表示

许多程序员假设带符号整型是按二进制补码方式来表示的，因为这是现代计算机系统上最常用的表示法；但是 `ANSI C` 和 `C++` 规范并没有强制要求带符号整型的格式。有一些处理器确实使用二进制反码甚至符号数值格式。如果你的代码可能运行在这些系统上，就不能对带符号整数的范围和位格式做出假设。

例如不能假设 16 位带符号整数的最小值就是 -32768，而应该使用 `<limits.h>` 中定义的预处理宏常量 `SHRT_MIN`。另一个常见的情况是假设 `~0 == -1`，这在二进制反码的机器上也是不正确的，在那里 `-0 == ~0`。

本地类型的大小

处理器有一个自然字大小，对应于它的内部寄存器大小，也表示了能高速处理的变量大小。最初 C 编译器使 `int` 类型对应于自然字大小，允许程序员在需要优化性能时使用 `int`（假设变量范围没有限制和要求）。很多年以来这都是正确的，很多程序也假设 `sizeof(int)==4`。

假设 `int` 大小为 4 字节，给那些目标是 64 位平台，而且需要保持向后兼容性的编译器开发者，造成了巨大的灾难。

因此无数的模型引入到 64 位体系架构，这些模型要么强调与 32 位平台的互操作性、要么强调 64 位平台的理想性能。模型命名为 LP64、ILP64、LLP64、ILP32、和 LP32 等，名字指示了核心 C 数据类型的大小，如表 5-1 所示。L 对应于 `long`、P 对应于指针大小、I 对应于 `int`、而 LL 则对应于 `long long`。（还有很多其它模型，上面这些只是最常用的几种）。

NOTE `long long` 是某些编译器的标识符，比如著名的 GCC。其它编译器如 Microsoft Visual C++，使用 `_int64` 类型。

表 5-1：一些编程模型

类型	LP64	ILP64	LLP64	ILP32	LP32
<code>char</code>	8	8	8	8	8
<code>short</code>	16	16	16	16	16
<code>int</code>	32	64	32	32	32
<code>long</code>	64	64	32	32	32
<code>long long</code>			64		
<code>pointer</code>	64	64	64	32	32

多数程序员都熟悉传统的 32 位编程模型 ILP32，整型、长整型、和指针都是 32 位大小。LP32 最早使用在 Win16 C API，是一个更简单的规范，专为 Intel 8086 家族特别设计，拥有 16 位整数寄存器，但是 20 位（8086）或者 24 位（80286）地址。（更特别地是 8086 和 80286 处理器使用分段寻址体系架构）。

由于 ILP32 模型缺乏 64 位类型，它并不适用于 64 位 CPU，它超出了 32 位系统的 4GB 地址空间限制。对于 64 位 CPU，你需要 64 位的指针，上面的其它几种模型都具备。它们的区别是如何确定以下哪个是更重要的：

- 保持 `sizeof(int)==sizeof(long)==sizeof(void *)` 假设
- 保持 `sizeof(int)`==机器字大小假设
- 保持 `sizeof(int)==4` 假设

由于前面两个假设在 64 位体系架构中是互斥的，混乱随之产生（因此导致模型数量激

增)。

遗憾的是 ANSI 标准并没有在这个问题上采取明确的态度，而是留给每个编译器厂商和用户根据情况来处理。Sun、SGI 和 Compaq/DEC 在它们的 Unix 系统中使用 LP64 模型，而 Microsoft 则使用 LLP64（更准确地说是 P64）模型来支持 64 位 Windows。

Microsoft 主要的关注是清晰、简单、和安全地迁移到 Win64。为了确保这一点，Microsoft 开发者希望尽可能多地避免打破 32 位代码的假设，同时又能获得 64 位指针。LLP64 模型只使用 `_int64` 或 `long long` 类型来创建 64 位整数，从而实现了这个目标。不含指针的结构体在 ILP32 和 LLP64 里保持完全相同的大小，这是向后兼容时很重要的一个考虑。

这使得你我这样的可移植程序员陷入困境：我们必须决定是使用 C 语言的本地类型（`short`、`int`、和 `long`）、还是使用譬如 C99 提供的那一组定长类型（`inttypes.h`），如下面表 5-2 所示。

表 5-2: C99 定长类型

类型	描述
<code>int8_t</code>	8 位带符号整数
<code>uint8_t</code>	8 位无符号整数
<code>int16_t</code>	16 位带符号整数
<code>uint16_t</code>	16 位无符号整数
<code>int32_t</code>	32 位带符号整数
<code>uint32_t</code>	32 位无符号整数
<code>int64_t</code>	64 位带符号整数
<code>uint64_t</code>	64 位无符号整数

通常如果你必须强制一个特定大小——例如创建一个精确格式化的结构体定义、或者需要确保整数的范围，那么就应该使用定长类型。如果你不需要特定的范围，例如一个只需要几千大小的索引变量，则 C 语言的本地整数类型允许编译器为你做出正确的选择，但是不幸的是这也不总是正确的。有些平台特别强调兼容性，在体系架构是 64 位时仍然只提供 32 位整数。

一个程序需要特定大小的变量（例如 32 位整数），如果使用了 C99 的 `uint32_t` 类型，那么在移植到不支持这个大小整数操作的低端平台时，可能会导致非常差的性能。例如 8086 处理器是 16 位的，因此 32 位整数操作通常需要使用一个函数调用。指定变量大小时需要非常小心，只有你确实需要它们才应该使用，例如你需要特定范围或者精确打包。

POSH 例子：定长类型

POSH 提供 C99 (`inttypes.h`) 的模拟定义，如下：

<code>posh_byte_t</code>	无符号 8 位数值
<code>posh_i8_t</code>	带符号 8 位整数
<code>posh_u8_t</code>	无符号 8 位整数
<code>posh_i16_t</code>	带符号 16 位整数
<code>posh_u16_t</code>	无符号 16 位整数

<code>posh_i32_t</code>	带符号 32 位整数
<code>posh_u32_t</code>	无符号 32 位整数
<code>posh_i64_t</code>	带符号 64 位整数
<code>posh_u64_t</code>	无符号 64 位整数

地址空间

计算机体系架构进步的一个主要标志就是地址空间，也就是计算机系统能够轻松访问的内存总数。

早期计算机由于指针大小和可用硬件的限制，只能访问很小数量的内存。

通用但并不精确的一个规则是，计算机系统不能访问超过指针大小所允许的内存数量，也就是可寻址字节数为 2 的指针大小次方。然而这里存在很多例外，例如指针大于实际地址空间的那些系统。**Motorola 68000** 只能寻址 16MB，但是它却有 32 位指针寄存器，而 **Intel 8086** 可以轻松寻址 64KB（访问单一寄存器），通过分段内存体系能够增加寻址到 1MB。今天的机器很多已经是 64 位指针，但是也只能访问少得多的内存范围，有时候仅能达到 40 位。老的计算机系统使用分页、窗口、或者 **banked** 内存访问，来访问比实际可寻址更多的内存。

使用大数组或结构体的程序在迁移到低端平台时，需要知道这些潜在的限制，这通常都是程序员没有预料到的奇怪问题。例如下面看上去没有问题的代码：

```
static unsigned char buffer[0x20000];
```

在迁移到低端系统（如 16 位指针）上，这段代码马上就会停止构建。

小结

除了操作系统的区别，处理器是平台最基础的组成。处理器在性能、特性、和实现上存在巨大差异，而这在可移植软件开发的过程中是一个常见的问题领域。本章讲解了不同处理器体系架构上的多数关键问题。

第6章 浮点

浮点计算的性能和精度的不一致，以及浮点的位域表示，已经是困扰计算机科学家数十年的问题了，直到不同的 IEEE 标准在 1980 年代被引入，才逐渐解决这个问题。本章讨论依赖于浮点数据和运算的可移植软件将面临的问题。

浮点的历史

在计算机的早期岁月，特别是 1960 年之后，几乎所有微计算机和主机厂商都实现了自己的私有浮点格式和转换。可移植性——包括代码和数据——在这样的环境下几乎是不可能的，不过由于当时主要是私有软件，所以浮点的这种不可移植情况也是合理的。每个平台都有自己的应用，因此开发者只需要处理一个平台的特点，很少或没有需要迁移软件到竞争平台。

1970 年代末，高端迷你计算机和主机的一些特性开始稳定地向微计算机（PC）迁移，浮点也是其中之一。就在这个时候，缺少浮点运算标准成为一个严重的问题。为了解决这个问题，成立了 IEEE p754 委员会。由许多主要的微处理器和系统厂商的工程师组成（包括 Intel、Zilog、和 Motorola）。这个委员会开始定义 IEEE 754 浮点规范，这个规范也成为之后浮点格式和运算的标准。（IEEE 754 标准最后合并到了 IEC 60559 标准中，因此这两个基本上是同义的；IEC 60559 有时候也叫做 IEC 559）。

NOTE 直到现在仍然有极少数奇怪的计算机，仍然按自己的方式处理浮点，例如 Cray 和 DEC 的大型机。游戏控制台如 Sony 的 PS2 也有自己的私有格式，因为可移植性并不是这些厂商关注的事情（实际上它们希望阻止可移植）。

IEEE 754 规范花费了许多年的活动和折衷才终于看见了光明，但是回顾历史，它被证明是一个精心设计、精确、实际的标准，而且没有强加任何不合理的限制。通过定义存储格式、数学操作、舍入模型、异常、和特殊值的位表示，使用浮点的可移植软件开发变得简单许多。

标准 C 和 C++ 的浮点支持

之前 C 和 C++ 语言对浮点的支持一直是很薄弱的，以一种几乎无意义且模棱两可的术语定义了浮点的需求。基本浮点类型的大小和格式没有明确指定，而且浮点的精度和操作（+，-，*，/）和库函数都由实现定义。然而从 C++98 和 C99 开始，IEEE 754 的可选支持被加入到标准（应用可以查询是否遵循 IEEE 754 标准）。

虽然 IEEE 754 标准非常优秀，但浮点在可移植软件中的使用还是有非常大的障碍：C 和 C++ 语言规范并没有强制要求浮点。C++98 和 C99 也仅提供标准的可选支持。这是一个不方便但也可以理解的方式，因为浮点支持（特别是遵循 IEEE 754 的浮点）在很多平台都不可用，而 C/C++ 需要兼容这些平台。

这样就意味着 float 和 double 通常会对应于 IEEE 754 单精度和双精度格式，但是却无法保证这一点。对于舍入模型、异常、和 IEEE 754 标准的其它方面，也是一样的。

C++98/C99 和 IEEE 754

各种 C 实现缺乏浮点标准被认为是一个重大的瑕疵，特别是在图形、模拟、和科学计算的应用中。为了解决这个问题，C99 对语言进行了修订，并引入了名义上的 IEEE 754/IEC 60559 支持。如果某个编译器预定义了常量 `__STDC_IEC_559__`，则你可以认为它的浮点操作遵循 IEEE 754 标准。

C99 标准同时还引入一组标准 API，`<fenv.h>` 用来与平台的浮点环境进行交互。在 `<fenv.h>` 之前，每个编译器厂商都提供了自己的浮点环境头文件和接口，例如 `<float.h>` 和 `<ieeefp.h>`。（注意 `__STDC_IEC_559__` 和 `<fenv.h>` 是 C99 特定的，在 C++ 中并不可用）。

对于 C++98 的实现来说，检查 `numeric_limits<float>::is_iec559` 可以获得类似的结果。但是 C++98 的 IEEE 754 支持比 C99 要薄弱，例如缺乏 `<fenv.h>` 中的某些函数。

IEEE 754 另一个很好的特性是它对函数求值有精度要求（例如 `remainder`，`remquo`，和 `sqrt`）。没有这一点，C 实现就会按自己的方式来计算。IEEE 754 标准还对许多其它的浮点操作强加了严格的规则，包括环境控制和分类例程。

浮点的问题

现在你对浮点的问题和标准化（IEEE 和 C/C++ 标准）有了一定的认识，让我们来看一些浮点应用可能会遇到的特定问题。

求值不一致

C 和 C++ 没有严格的规则来确保浮点运算的一致性。这意味着你不能期望以下代码：

```
float add(float a, float b)
{
    return a + b;
}
```

和下面代码能够返回相同的结果：

```
float add2(float a, float b)
{
    float c = a + b;
    return c;
}
```

Intel x87 FPU 体系架构将 80 位精度的中间结果存储在浮点堆栈中，只有当值被存储到内存时才会损失精度。上面例子的第一个函数 `add()`，将会产生下面汇编代码：

```
fld a ; load 'a' onto stack
fadd b ; add 'b' onto stack
```

调用方会将结果出栈并存储在适当的变量中。如果那个变量是 `double`，则所有额外的精度都可能会保留。

第二个函数 `add2()` 在本地把结果存储到中间值，会产生以下汇编代码：

```
fld a ; load 'a' onto stack
fadd b ; add 'b' to top of stack
```

```
fstp c ; store result (and pop stack) to float variable 'c'
fld c  ; load 'c' back onto stack so that caller may retrieve result
```

`fstp/fld` 指令组合会在返回调用方之前，截断额外的精度。

因此如果在比较语句中使用这些函数：

```
if (add(x, y) != add2(x, y))
    printf("Error: this shouldn't happen!");
```

在不同的编译器中，甚至相同编译器不同的优化选项，可能会遇到未意料的行为。

在 GNU GCC 上，开启和禁用优化，函数 `add2()` 会产生不同的结果：

```
; Optimizations DISABLED
pushl %ebp
movl %esp, %ebp
subl $8, %esp
flds 8(%ebp)
fadds 12(%ebp)
fstps -4(%ebp)
movl -4(%ebp), %eax
movl %eax, -8(%ebp)
flds -8(%ebp)
leave
ret

; Optimizations ENABLED
pushl %ebp
movl %esp, %ebp
flds 12(%ebp)
fadds 8(%ebp) ; Note that the store/load has been optimized away!
popl %ebp
ret
```

应用不能依赖于浮点操作的一致性和精度，特别是在比较测试的上下文中。这在某些系统上可以工作，但是却是非常容易引起可移植问题的区域，而且要找到 `bug` 也非常困难。

浮点和网络应用

既然浮点操作的结果经常不一致，那么如果两个不同的系统直接共享浮点数据，任何坏事情都可能发生。

例如两台计算机必须运行紧密的并行模拟时，最坏的情况通常都会发生，每台机器执行内部模拟，偶尔需要接收另一台机器的输入。（点对点网络计算机游戏和模拟器通常按这种方式操作）。

一旦接收到输入，每台计算机就模拟这个离散步骤，并将当前状态广播给其它机器。

给定相同的输入、固定的更新速度、和已知的起始状态，则所有机器应该对模拟有相同的视图。当然有时候这也不一定正确，因此机器会有状态验证检查，通过检查自身的模拟状态并与其它机器进行对比。如果结果不匹配，就会产生一个同步错误。

如果是浮点运算，同步就会成为一个严重的问题，由于体系架构或编译器的差异，运算

结果也会有微妙的差异。尽管差异非常小以致于看上去可以忽略（有时候即使是最小的改变也可能使你处于“在墙后”和“在墙中”的区别），当同步检查进行按位比较时，这些差异就会成为问题。要么同步，要么不同步，即使两个浮点数有单独一位的差异，也足够导致同步错误。

一个更加健壮（但仍然不够完美）的方法是将所有浮点量化为更加受限的定点表示，通过稍微有一点不同的浮点求值方法来达到匹配的结果，后面小节“定点整数运算”我们马上会讨论。

转换

在 C 和 C++ 的标准中，浮点到整数的转换都是截断（向零舍入），浮点的小数部分将被抛弃。

```
float x = 1.33f, y = -1.33f;
/* ix will be 1 and iy will be -1 */
int ix = (int) x, iy = (int) y;
```

信不信由你，这在某些系统中会有非常糟糕的性能副作用。

多数现代 CPU 都有一个可配置的浮点舍入模型。IEEE 754 规范定义了四种不同的舍入模型：up（向无穷正数舍入）、down（向无穷负数舍入）、chop（向零舍入）、以及向最接近的可表示值舍入。多数情况下的默认舍入模型是向最接近的值舍入，但这会导致问题，因为 C 和 C++ 要求浮点到整数的转换进行截断。因此每次浮点值被转换为整数时，编译器通常都需要向 CPU 产生适当的浮点舍入模型更改指令。

下面这个简单的函数：

```
int round(float s)
{
    return (int) s;
}
```

会产生以下代码：

```
_round:
    pushl %ebp                ; save EBP
    movl %esp, %ebp          ; put stack pointer into EBP
    subl $8, %esp            ; allocate space on the stack
    fnstcw -2(%ebp)           ; store current FPU control word
    movzwl -2(%ebp), %eax     ; move the FPU control word to EAX
    orw $3072, %eax          ; or EAX with 3072 (to set rounding mode)
    movw %ax, -4(%ebp)        ; mov AX into memory EBP[-4]
    flds 8(%ebp)              ; load 's' into the FPU
    fldcw -4(%ebp)            ; load the control word from EBP[-4]
    fistpl -8(%ebp)           ; store 's' as integer back into EBP[-8]
    fldcw -2(%ebp)            ; reload the old control word
    movl -8(%ebp), %eax       ; move return value into EAX
    movl %ebp, %esp          ; restore everything
    popl %ebp
    ret
```

这样一个简单的操作也需要这么多的代码，但是 C/C++ 舍入标准还要求每次都生成这些代码。其中有些指令（如 `fildcw`）是你不希望要的，因为它们强制刷新浮点运算单元（FPU）的所有状态，导致 FPU 流水线的中断。

比较下面这个不遵循标准的实现：

```
_round:
    pushl %ebp          ; save EBP
    movl %esp, %ebp     ; put stack pointer into EBP
    subl $4, %esp       ; allocate space on the stack
    flds 4(%ebp)        ; load 's' into the FPU
    fistpl -4(%ebp)     ; store 's' as integer back into EBP[-8]
    movl -4(%ebp), %eax  ; move return value into EAX
    movl %ebp, %esp     ; restore everything
    popl %ebp
    ret
```

有一些编译器提供执行快速但非标准的浮点到整数转换（通过避免所有控制字操作，不设置 FPU 而立即调用 `fistp`）。但这些都是私有的，是对标准的非常不可移植的扩展。

定点整数 Math

既然浮点带来如此多的可移植问题，有时候转换浮点操作为整数或者定点运算会更加简单。整数表示的语义虽然并不完美，但仍然要比浮点清晰许多。

例如需要以浮点值广播位置时，你可以使用量化的定点值来代替浮点。量化过程丢弃不必要的精度，允许“不精确的”比较：

```
int snap_to_quarters(float s)
{
    s *= 4.0;
    if (s >= 0)
        s = s + 0.5;
    else
        s = s - 0.5;
    return (int) s;
}
```

这段代码将浮点值转换为 $1/4$ 精度，并返回它的定点值（按因数为 4 的比例放大）。因此多个接近于相同量化的值会计算出相同的定点数，这样就能够以定义良好的精度对浮点进行比较。

然而定点数也并不完美。定点表示法有静态的范围和精度，这不如浮点灵活。此外也没有可移植的方法来处理溢出条件，这是很重要的一点，因为在常见的定点标准操作中会经常出现溢出。例如 32 位定点乘法需要 64 位的中间存储，C 和 C++ 都没有以可移植的方式提供（至少在 C99 实现变得更加流行之前）。

从浮点提取整数位

有时候按整数来查看浮点值的原始位是很有用或者必要的。例如你可能要在调试器中按整数来显示一个浮点值的位，或者执行一个名义上不可移植的性能优化，例如直接操作符号位，或者使用整数代替浮点进行比较。

NOTE 对于 IEEE 754 标准，只要两个浮点值有相同的符号，这两个值就可以直接比较原始位来进行大小比较。

最直接的方法是转换浮点值的地址为整数指针（假设 `sizeof(int) == sizeof(float)`），然后直接对这些位进行操作。

```
uint32_t get_float_bits(float f)
{
    /* this assumes that sizeof(float) == sizeof(int); */
    /* the conversion will not result in a trap representation */
    /* for an integer, and the alignment properties of int */
    /* and float are identical */
    return *(int*)&f;
}
```

不幸的是这种方法有一个问题：技术上非法。根据 C 和 C++ 标准，int 和 float 被认为是不相容的类型，因此无法保证程序运行到这里不会崩溃。

但是在我看来这只是语言层面的问题而已。现实的情况中，在大多数流行的计算机体系架构里，这种指针转换都是正确的，因为 32 位值就是 32 位值。例如 Cygwin/x86 上的 GCC，32 位浮点值 1.0 的整数表示是 0x3f800000。如果你编写下面代码片段：

```
float foo(void)
{
    float f = 1.0;
    return f;
}
```

实际的未优化（优化的代码只是做 `fldl` 并返回，不过不能帮助我说明观点）汇编代码如下：

```
pushl %ebp          ; save EBP
movl %esp, %ebp     ; move ESP into EBP
subl $8, %esp       ; ESP = ESP - 8
movl $0x3f800000, %eax ; EAX = 0x3f800000
movl %eax, -4(%ebp) ; EBP[-4] = 0x3f800000
movl -4(%ebp), %eax ; EAX = 0x3f800000
movl %eax, -8(%ebp) ; EBP[-8] = 0x3f800000
flds -8(%ebp)       ; top of stack = 0x3f800000=1.0f
leave
ret
```

让我们先忽略性能问题，前面代码片段清晰地说明了在这个平台上，将普通的整数值解

释为 32 位浮点值是可以工作的。

不幸的是除了类型不兼容，还有另外一个问题：有些编译器通过假设没有 `aliasing`，进行侵入式的代码优化。例如下面代码：

```
/* 我们再次假设浮点到整数的转换在当前平台是可以工作的 */
/* 尽管并不符合 C 标准的要求 */
float f;
int x;
f = 1.0f;
x = * (int *) &f;
```

编译器可以自由地假设 `x` 和 `f` 完全无关（也就是对 `x` 的任何修改都不会影响到 `f`，反之亦然），因为标准认为它们是不兼容的类型。在这个假设之下，编译器可能会在 `x` 赋值之后才执行对 `f` 的赋值！

你可以通过几个方法来解决无 `aliasing` 假设：

- 使用 `union`，强制许多编译器假设 `aliasing`（GCC 有这个扩展）。技术上来讲，使用 `union` 和指针转换一样都是非法的，但是 `union` 有一些特性，可以使失败的可能性降得更小。首先 `union` 所有成员都能正确对齐；第二 `union` 足够大以容纳它最大的成员，这意味着在可能会出现对齐和溢出问题的平台（例如 `float` 是 32 位而整数为 16 位），你只会得到恰当的(pretty)错误行为，而不是真正的(really)错误行为。
- 执行中间转换为 `void *` 或者 `unsigned char *` 类型，这可以强迫编译器假设类型 `aliasing` 可能发生。（如果不允许这样做，今天的大量程序都会出错）。
- 转换浮点地址为 `unsigned char *`，并直接读取数据。这是 100%合法的，但是它只告诉你这些位是什么，而没有告诉你这些位的含义（因为原始位的含义本来就是不可移植的）。
- 使用编译器的优化开关，禁止 `aliasing` 规则或者假设没有 `aliasing`。

提取浮点的整数表示位，最安全的方法是执行一次到 `void *` 或者 `unsigned char *` 类型的中间转换，如下：

```
/* assumes that dst is at least as large as sizeof(float) */
void float_to_bits(float f, unsigned char dst[])
{
    int i;
    unsigned char *c = (unsigned char *) &f;
    for (i = 0; i < sizeof(f); i++)
        dst[i] = c[i];
}
```

这个函数提取了原始位，但并没有尝试去重新组合为不同的多字节值（例如整数）。对类型不兼容或者 `aliasing` 规则没有任何违反。

使用 `union` 则如下：

```
uint32_t float_to_bits(float f)
{
    union
```

```

    {
        int i; /* again, assuming sizeof(int)==sizeof(float) */
        float f;
    } u;
    u.f = f;
    return u.i;
}

```

根据 C 和 C++ 标准，执行上面这段代码的结果仍然是未定义的（在 union 的一个成员被写入时，不能保证其它成员会发生什么），但至少比通过指针强制转换要更好一些。

此外 C++ 还有 `reinterpret_cast` 操作符：

```

float f = 1.0f;
int i = reinterpret_cast< float & >(f);

```

虽然非常地 “C++”，但并不比直接转换更加安全。

例子：POSH 的浮点位提取

POSH 使用 union 的方法来提取浮点值的位：

```

posh_u32_t
POSH_LittleFloatBits(float f)
{
    union
    {
        float f32;
        posh_u32_t u32;
    } u;
    u.f32 = f;
    return POSH_LittleU32(u.u32);
}

posh_u32_t
POSH_BigFloatBits(float f)
{
    union
    {
        float f32;
        posh_u32_t u32;
    } u;
    u.f32 = f;
    return POSH_BigU32(u.u32);
}

```

这个代码并不完美，但到编写本书为止，在 POSH 支持的任何体系架构中，都没有出现问题。

实现相关

编写涉及浮点的代码时，可以包含<float.h>，检查表 6-1 中的预定义符号，来支持所有不同的浮点格式。

表 6-1: C 和 C++浮点格式常量

常量	含义
FLT_ROUNDS	当前舍入模型
FLT_RADIX	浮点格式的指数表示法的基数
FLT_DIG	精确表示浮点所需要的数字个数
DBL_MAX	double 可以表示的最大值

在 C++中，可以包含<numeric_limits>并检查特化模板 numeric_limits，定义如下：

```
namespace std
{
    template<class T> class numeric_limits
    {
    public:
        static const bool is_specialized = false;
        static T min() throw();
        static T max() throw();
        static const int digits = 0;
        static const int digits10 = 0;
        static const bool is_signed = false;
        static const bool is_integer = false;
        static const bool is_exact = false;
        static const int radix = 0;
        static T epsilon() throw();
        static T round_error() throw();
        static const int min_exponent = 0;
        static const int min_exponent10 = 0;
        static const int max_exponent = 0;
        static const int max_exponent10 = 0;
        static const bool has_infinity = false;
        static const bool has_quiet_NaN = false;
        static const bool has_signaling_NaN = false;
        static const float_denorm_style has_denorm = denorm_absent;
        static const bool has_denorm_loss = false;
        static T infinity() throw();
        static T quiet_NaN() throw();
        static T signaling_NaN() throw();
        static T denorm_min() throw();
```



```
static const bool is_iec559 = false;
static const bool is_bounded = false;
static const bool is_modulo = false;
static const bool traps = false;
static const bool tinyness_before = false;
static const float_round_style round_style =
round_toward_zero;
};
```

例如在 C 中可以查看 `<float.h>` 的 `FLT_ROUNDS`；在 C++ 则可以检查 `numeric_limits<float>::round_style`。

NOTE C++ 也有 `<float.h>` 的对等物 `<cfloat>` 头文件

我们还可以检查更多的常量，但在实践中，任何尝试支持所有浮点格式的行为（C 和 C++ 标准允许）都是极度烦琐的，而且非常容易出错。于是建立一个基准，仅支持那些期望的浮点格式和操作，会更加简单和清晰。当然，能够避免编写依赖于平台特定浮点实现的代码，则是更好的做法了。

异常结果

许多浮点操作和函数在不适当的输入时，会产生特殊值（无穷、not a number (NaN) 等等）。例如 90 度的 `tan`，或者负数的平方根都不能返回正确的值，有可能会抛出异常。

IEEE 754 规范定义了五种不同的异常：

下溢条件

当操作的结果值太小，不能以普通的浮点值表示时发生。

上溢条件

下溢的反面：当操作的结果值太大，不能以期望的目标格式表示时发生。

除以 0 条件

当一个合法的非 0 浮点值除以 0 时发生。注意这里特别地排除了 NaN/0 和 0/0，这两种情况下会产生非法操作异常。

非法操作异常

涵盖了其它条件下产生的多数非法操作。包括无穷减无穷，比较操作中使用 NaN，0 除以 0，以及无穷除以无穷。

不精确异常

当操作的结果不能被二进制浮点值精确表示时产生。这非常常见（例如 2.0/3.0），因此这个异常通常可以掩码。当溢出条件产生而没有溢出异常抛出时也会抛出不精确异常。

当遇到这些异常情况时，将返回一个特殊值，并且可能会抛出异常。

特殊值

在 IEEE 754 规范之前，每个编译器和平台都有自己的浮点格式，妨碍了以可移植的方式来检查异常值（你可以在调用标准库函数之后查询 `errno`，但这不能帮助你处理普通的数学运算）。每个平台提供一组函数或宏来检查这些特殊值。Microsoft Visual C++ 6.0 提供 `_isnan`, `_finite`, 和 `_fpclass` 等函数。其它编译器和平台可能提供类似的函数，如 `isnan` 和 `fpclassify`。

如果识别这些特殊值的不同类型对你的应用来说非常重要，你就需要抽象浮点识别和出错函数，除非你假设支持 C99。

为了解决不同平台浮点类别宏和函数的混乱，C99 标准在 `<math.h>` 中定义了一组分类例程，以及浮点值分类的预定义常量，如表 6-2 所示：

表 6-2: C99 浮点分类

分类例程	说明
<code>int fpclassify(x)</code>	对参数进行分类并返回它的类型，如 <code>FP_NAN</code> , <code>FP_INFINITE</code> , 或者实现定义的值
<code>int isfinite(x)</code>	如果 <code>x</code> 是有限的，返回非 0
<code>int isinf(x)</code>	如果 <code>x</code> 是 +/- 无穷，返回非 0
<code>int isnan(x)</code>	如果 <code>x</code> 是 NaN，返回非 0
<code>int isnormal(x)</code>	如果 <code>x</code> 是普通值，返回非 0
<code>int signbit(x)</code>	如果 <code>x</code> 为负返回 1，如果 <code>x</code> 为正返回 0
<code>FP_INFINITE</code>	预定义常量，表示 +/- 无穷
<code>FP_NAN</code>	预定义常量，表示 NaN
<code>FP_NORMAL</code>	预定义常量，表示普通值
<code>FP_SUBNORMAL</code>	预定义常量，表示 subnormal
<code>FP_ZERO</code>	预定义常量，表示 +/- 0

有几个非数字值必须以浮点格式描述，IEEE 754 规范特别定义了它们，如表 6-3：

表 6-3: 必须用浮点来描述的非数字值

值	表示
正 0	全 0
负 0	1 位符号，跟随 31 位 0
正无穷	1 个 0，跟随 8 个 1，再跟随 23 个 0
负无穷	1 个 1，跟随 8 个 1，再跟随 23 个 0
NaN	小数部分非 0，指数全 1，符号位确定 Signaling(0) 或者 Quiet(1)
Denormalized	小数部分非 0，指数全 0，符号位任意

NOTE IEEE 规范区分 Signaling NaN 和 Quiet NaN。在数学操作中使用 Signaling NaN 会产生异

常，而 **Quiet NaN** 可以在多数浮点操作链中传递而不会抛出异常。

通常来说，使用语言或厂商提供的分类宏，比直接检查浮点值的位表示要安全许多。

异常

浮点操作在许多情况下都可能导致错误（除以 0 是最经典的例子），在这些情况下，实现可能会产生异常（除了返回特殊值之外），但抛出异常并不是强制要求的。

许多(但不是全部)C 和 C++实现允许程序员使用 `signal()` API 来安装 SIGFPE 信号处理器；然而这种支持既不是标准要求的，也并不可靠，因为它需要平台的浮点和异常处理实现相互协作。`<fenv.h>`定义了异常 `FE_OVERFLOW`, `FE_UNDERFLOW`, `FE_DIVBYZERO`, `FE_INVALID`, 和 `FE_INEXACT`，但是仅在实现支持这些异常时才可用。

C 标准要求浮点异常在程序启动时进行掩码，应用可以通过平台特定的 API 或者 C99 的 `<fenv.h>`API 来启用特定的异常。例如 C99 允许程序通过 `feholdexcept()`函数来临时掩码，以确保操作不被中断，而 Intel 80x86 平台的 Microsoft Visual C++ 6.0 则使用私有的 `_control87()` 或 `_controlfp()`函数。

访问浮点环境

在 C99 对访问浮点环境进行标准化之前，每个编译器厂商都提供自己的一组宏、头文件、和 API 来管理相关的访问。这并不像看上去那样不好，因为实际需要的函数数量很少。例如 Microsoft Visual C++ 6 导出了 `_controlfp`, `_statusfp`, 和 `_clearfp` API 来与浮点环境交互。甚至在不支持这种 API 的系统上，使用少量汇编代码编写类似的 API 也通常是很简单的。

C99 规范在 `<fenv.h>`中提供一组宏和函数，来与浮点环境交互，如表 6-4 所示。通过这些 API，可以抛出、掩码、测试、和清除异常，也可以设置和获取浮点的舍入模型。

表 6-4: C99 浮点异常函数

函数	描述
<code>void feclearexcept(int excepts)</code>	清除指定的浮点异常
<code>void fegetexceptflag(fexcept_t *flagp, int excepts)</code>	获取实现定义的当前浮点状态描述
<code>void feraiseexcept(int excepts)</code>	抛出指定的异常
<code>void fesetexceptflag(const fexcept_t *flagp, int excepts)</code>	设置当前浮点状态
<code>int fetestexcept(int excepts)</code>	测试状态字，检查指定异常是否被抛出
<code>int fegetround(void)</code>	获取当前舍入模型
<code>int fesetround(int round)</code>	设置当前舍入模型
<code>void fegetenv(fenv_t *envp)</code>	使用信号函数调用获取完整的浮点环境（状态和控制字）
<code>int feholdexcept(fenv_t *envp)</code>	获取当前环境，并存储在指定参数中，然后设置当前状态为“不中断”，使得不抛出异常
<code>void fesetenv(const fenv_t *envp)</code>	设置当前环境，参数一般是之前 <code>fegetenv/feholdexcept</code> 获取的环境

<code>void feupdateenv(const fenv_t envp)</code>	将当前抛出的浮点异常保存在本地，安装 <code>envp</code> 指定的浮点环境，然后抛出已保存的浮点异常
--	---

存储格式

IEEE 754 规范定义了两种主要的存储格式（单精度和双精度），以及几个可选的更高精度格式，例如扩展双精度（某些编译器中的 `long double`），和四倍精度格式。

在大多数 IEEE 754 实现中，C 和 C++ 的 `float` 类型对应于 32 位值，格式如表 6-5。

表 6-5：IEEE 754 单精度格式

位	含义
31	符号位（0=正；1=负）
24-30	指数
0-23	尾数/小数（隐含地以 1 开头）

表 6-6 说明了 IEEE 双精度格式，通常在 C 和 C++ 中用 `double` 描述。

表 6-6：IEEE 754 64 位双精度格式

位	含义
63	符号位（0=正；1=负）
52-62	指数
0-51	尾数/小数

最后扩展精度格式是 80 位，如表 6-7 所示。它是一种并不常见的存储格式。有些编译器实现通过 `long double` 类型来支持它。

表 6-7：IEEE 754 80 位扩展精度格式

位	含义
79	符号位（0=正；1=负）
78-64	指数
63	总是 1
0-62	尾数/小数

小结

浮点操作的行为是许多程序员很少考虑的事情之一。他们都假设 1.0 就是 1.0，在所有计算机上都以相同的方式表示，但正如本章所说明的，通常都不是这样。ANSI C 和 C++ 标准并没有完全解决浮点操作的问题，因此任何依赖于浮点操作的精度的程序都必须非常小心地实现。

第7章 预处理器

可能 C 和 C++语言中没有哪个方面能像预处理器一样，受到计算机语言纯粹主义者如此多的诟病。预处理器几乎或完全不按照语言的语法和主义，来转换原始代码文本。如果过于放肆地使用，通常会导致难以寻找的 bug 或编译器错误。

所有后来出现的新语言都尽量避开预处理器，而这些新语言的使用者也总是抱怨缺少预处理器。事实是预处理器提供了一组强大的特性，使用条件编译和原始文本替换，这些通常都很难在语言中正确地模拟或实现。在这一章中，我们会深入到 C/C++的预处理器，让你在更容易地使用预处理器强大功能的同时，避免它的相关缺陷。

预定义符号

预定义符号在不同编译器和平台上差异很大。例如每个编译器厂商在描述“产生 80x86 目标代码”时，都会定义不同的符号。有些编译器使用 `i386`；其它的使用 `__i386__`；而 Microsoft Visual C++则使用 `_M_IX86`。

C 标准仅仅定义了几个可移植的预处理器常量。表 7-1 列出了 C89 语言标准的定义。

表 7-1：C89 预定义宏名

宏名	描述
<code>__DATE__</code>	预处理转换单元转换代码的日期：格式为 <code>mmm dd yyyy</code> 的字符串，月份的名字和 <code>asctime</code> 函数所产生的月份相同，如果日期小于 10，则 <code>dd</code> 的第一个字符为空格；如果转换日期不可用，将提供实现定义的合法日期。
<code>__FILE__</code>	当前源文件名（字符串）
<code>__LINE__</code>	当前源文件的当前行（整数常量）
<code>__STDC__</code>	整数常量 1，表示遵守标准的实现
<code>__TIME__</code>	预处理转换单元转换代码的时间：格式为 <code>hh:mm:ss</code> 的字符串，和 <code>asctime</code> 函数产生的时间格式相同。如果转换时间不可用，将提供实现定义的合法时间

C99 引入了另外几个预定义常量和标识符，如表 7-2 所示。

表 7-2：C99 预定义宏名和标识符

宏名	描述
<code>__STDC_IEC_559__</code>	数字常量 1，表示遵循附件 F 规范（IEC 60559/IEEE-754 浮点运算）
<code>__STDC_HOSTED__</code>	整数常量 1 表示实现是主机实现，否则为 0
<code>__STDC_IEC_559_COMPLEX__</code>	数字常量 1，表示遵循附件 G 规范（IEC 60559 兼容的复数运算）
<code>__STDC_VERSION__</code>	C 实现的特定版本，例如 C99 是 199901L
<code>__STDC_ISO_10646__</code>	数字常量，格式为 <code>yyyymmL</code> （例如 199712L），表示 <code>wchar_t</code>

	类型的值是 ISO/IEC 10646 所定义的字符码，以及指定年月的修订和勘误
<code>__func__</code>	预定义标识符，用来发现当前函数未装饰的名字；目的类似于 <code>__FILE__</code> 和 <code>__LINE__</code> 。要给出的一个警告是该标识符由编译器定义，而不是预处理器，因为预处理器并不知道函数名，甚至是函数的概念。

此外当编译 C++ 源文件时，还定义了 `__cplusplus`。

老的编译器可能没有定义 `__STDC__`，表示它们是在 C89 标准流行之前开发的。这实际上非常好，因为如果你发现编译器是 ANSI 标准之前的，可以立即停止（实际地讲，试图同时支持早期 C 和 ANSI C 是不值得的）。

```
#if !defined __STDC__
#error This source code requires an ANSI compliant compiler
#endif
```

这里有一个潜在的冒险，也就是不同编译器实际定义 `__STDC__` 的方式。下面是 `__STDC__` 可能的定义：

- 有定义，但没有指定值
- 定义为 1
- 如果是 `strict` 编译模式，定义为 1；否则定义为 0
- 完全不定义，除非是 `strict` 编译模式

完全不设置 `__STDC__` 是很少见的。今天的大多数流行编译器要么遵循 ANSI 规范的最小集，要么完全遵循 ANSI 并禁止扩展，但通常都会设置 `__STDC__`。

头文件

在 C 语言中只有为数不多的头文件被认为是标准的。在 C99 标准之前，你只能依赖于表 7-3 所列的头文件，即使已经很少，不完全遵循标准的实现还可能会忽略其中一些。

表 7-3: C99 之前的标准头文件

头文件	描述
<code>assert.h</code>	断言管理
<code>ctype.h</code>	字符分类
<code>errno.h</code>	错误名与处理
<code>float.h</code>	浮点环境
<code>iso646.h</code>	处理 ISO 646 字符集
<code>limits.h</code>	定义整数限制
<code>math.h</code>	标准数学函数
<code>string.h</code>	字符串处理
<code>stdarg.h</code>	可变参数列表
<code>stdio.h</code>	标准输入/输出函数

stdlib.h	标准库，各种函数
wctype.h	<ctype.h>的宽字符版本
wchar.h	宽字符字符串支持
signal.h	异常/信号处理
stddef.h	有用的宏和类型定义
time.h	时间和日期查询与转换
locale.h	本地化支持
setjmp.h	setjmp/longjmp（非本地 goto）支持

C99 支持上面表 7-3 所列的头文件，还增加了下面表 7-4 所示的头文件。

表 7-4: C99 增加的头文件

头文件	描述
complex.h	复数运算支持
fenv.h	浮点环境
inttypes.h	操作最大宽度整数的函数
stdbool.h	定义 bool 类型和 true/false
stdint.h	定义特定宽度的整数类型
tgmath.h	泛类型数学宏

对于哪些头文件是可移植的、哪些是厂商特定的、哪些是库特定的、和哪些是平台特定的，硬性地区分可能会导致混乱。标准头文件<stdio.h>, <stdlib.h>和<string.h>存在于多数支持 C 的平台，但是有许多伪标准的头文件（通常存在于特定家族的操作系统或者特定厂商的编译器实现中），如<unistd.h>, <windows.h>, <malloc.h>, <process.h>, <sys/stat.h>和<varargs.h>, 仍然是不可移植的。

头文件路径规范

在不同的平台上，头文件存放在不同的位置。假设特定头文件在绝对路径是不明智的，如下：

```
#include "/usr/include/some_project.h"
#include "../src/local_header.h"
```

只有在你确保每组源代码都安装头文件到完全相同的位置，才能使用绝对路径。通常仅通过名字来 include 是更简单的：

```
#include "some_project.h"
#include "local_header.h"
```

然后使用编译器开关来指定头文件位置，例如-I/usr/include 和-I../src。

就我所知，至少有一个编译器（Metrowerks CodeWarrior）有非常特殊的头文件查找方法。假设你有两个文件（src/foo.c 和 src/foo.h），并且 src/foo.c 包含 foo.h 如下：

```
#include "foo.h"
```

然后默认情况下 CodeWarrior 将无法正确地找到 `foo.h`，除非你是在 `src/` 目录下编译。你需要指定一个特殊的命令行开关 `-cwd source`，这样它才会在源代码所在的目录中查找头文件。

头文件名

不同的操作系统和文件系统，在大小写和路径分隔符方面有不同的行为。如果你在 Microsoft Windows 上编译下面语句：

```
#include <STDIO.H>
```

编译器成功找到并装载 `stdio.h`（注意大小写）的机会很大。但是在 Linux 上则肯定要失败，因为 Linux 文件系统是大小写敏感的。编译器或预处理器会抱怨文件无法找到。

类似地，路径分隔符在不同系统中也不一样，Windows 倾向于使用反斜杠：

```
#include <MyLib\MyHeader.h>
```

类 Unix 系统（包括 Mac OS X），使用斜杠：

```
#include <MyLib/MyHeader.h>
```

当然，一群人里面总有那么几个另类。在这里就是 OS X 之前的 Mac OS，它使用冒号作为路径分隔符：

```
#include <MyLib:MyHeader.h>
```

幸运的是，C 标准规定反斜杠(\)在头文件包含语句是非法的。由于这个规定，以及世界上无数的 Unix 源代码，多数编译器都能正确识别斜杠(/)分隔符。我推荐在所有时候都使用斜杠作为分隔符。

此外，第 13 章将会提到的可接受文件名长度和字符，也可以应用在这里。

配置宏

项目经常会需要通过一组宏来进行全局的配置修改，例如 SAL 会查看几个符号定义来确定支持哪个音频子系统（如 `SAL_SUPPORT_OSS` 和 Linux 的 `SAL_SUPPORT_ALSA`）。

这些变量定义既可以通过命令行来完成，也可以通过在全局位置（如通用头文件）进行设置。多数编译器支持前者，但有些编译器只支持后者（Metrowerks CodeWarrior 使用 `prefix` 文件）。

通过命令行设置变量有两个主要的好处：

- 它允许你无需编辑源代码，就能够构建软件的多个版本。
- 如果你只是需要修改某些配置变量，就不需要编辑任何文件。

后面一个好处在源码控制系统中会非常方便。仅仅因为你要注释掉一行 `#define` 就要将整个项目签出是很麻烦和讨厌的事情，也很容易出错。例如程序员可能会打破文件的锁，在本地修改了某些重要的配置变量，然后做出了一些重大修改，却忘了那个文件并没有真正地签出来。

通过前缀头文件,或者通过全局包含头文件的配置段来修改配置变量的额外好处是可读性强和自说明。如果程序员正在查看某些不了解的定义,她/他可以简单地查找所有项目文件,来找到变量的定义并搞清楚它的使用方法。如果定义在源代码中没有文档或注释,则程序员也有可能不明白定义的目的。

条件编译

语言纯粹主义者不喜欢预处理器的原因之一就是预处理器很容易被滥用。对于那些必须在很多平台编译的代码,使用预处理器添加快速修复是很有诱惑力的。例如你有一些 socket 代码,在 Windows 下面运行你需要执行 `WSAStartup()`:

```
void init_sockets()
{
#ifdef _WIN32
    WSAStartup();
#endif
}
```

这本身是完全无害的,除了 `WSAStartup()` 是 Windows 函数,因此你必须包含 `<windows.h>`:

```
#ifdef _WIN32
#include <windows.h>
#endif
void init_sockets()
{
#ifdef _WIN32
    WSAStartup();
#endif
}
```

嗯,现在代码变得有些丑陋了。然后随着时间的进行,文件还在不停地增长,又增加了下面这些东西:

```
#ifdef _WIN32
{
    DWORD count, flags = 0;
    extern WSABUF wsabuffers[];
    WSAREcv(wsabuffers, 1, &count, &flags, NULL, NULL);
    ... copy out of wsabuffers into buffer ...
}
#else
    recv(s, buffer, buf_size, 0);
#endif
```

现在代码越来越难以控制。这就是滥用条件编译导致的后果,很明显你希望避免这样。

这并不是说条件编译一无是处,适度地使用条件编译是非常有用的。对于上面这个特定的例子,简单地分离出两个独立的 socket 函数实现,是更清晰和安全的做法:一个针对 Windows;另一个针对其它操作系统。这种方法在核心代码中最小化了 `#ifdef` 数量的侵略性,同时又不牺牲可移植性。

C/C++预处理器是非常简单而又强大的工具，但是它的特性太容易被滥用，导致非常难以阅读和维护的代码。如果你发现自己的代码充满了条件编译语句，重新分离你的实现可能是一个好的设计方案。

Pragmas

C 和 C++编译器提供 `pragma` 机制来直接与编译器交互，当然是以编译器实现定义的方式。例如在 Microsoft Visual C++，你可以使用 `#pragma warn(disable: xxxx)` 来禁止特定的编译器警告。

从 `pragma` 的本质上来讲，它不是特别可移植，但是 C 标准特别说明了所有编译器必须忽略那些不理解的 `pragma`。不幸地是有些编译器仍然会停止在未识别的 `pragma`，因此你需要使用条件编译来包围 `pragma`：

```
#ifdef _MSC_VER
#pragma warning(disable: 4786)
#endif /* _MSC_VER */
```

C99 标准 pragma

C99 引入了一组标准 `pragma`（有一点矛盾），格式如下：

```
#pragma STDC [name] [option]
```

下面是一个例子：

```
#pragma STDC FP_CONTRACT ON
```

在本书写作的时候，下面是三个已经被认可的标准 `pragma`：

- `FP_CONTRACT`：控制浮点异常是否被定约。
- `FENV_ACCESS`：用来包含那些修改浮点状态的操作。
- `CX_LIMITED_RANGE`：告诉编译器对于复数运算作出的假设是安全的。

小结

虽然经常被嘲笑为粗糙、原始、和不精细，C/C++预处理器对于跨平台软件开发的帮助是非常有效的。条件编译、`pragma`、原始文本替换、和预定义符号，可以极大地减小移植软件到其它系统时的负担。

第 8 章 编译器

ANSI C 和 C++ 从出现到现在已经超过 15 年，拥有许多不同的编译器。ANSI C 毫无疑问是世界上最流行的编译型编程语言，而 C++ 也正在快速地接近 C 的流行度。

既然有如此多的可用编译器，不同实现之间的古怪和特殊自然不可避免。有些编译器对语言进行扩展，来支持特殊平台的需求；其它一些则通过增加特性和库，来让开发者觉得更有用。而且当面对同一段代码时，编译器如何编译它总是拥有不同的选择。

本章讲解当你在不同编译器之间迁移时，将会遇到的编译器行为上的差异。

结构体大小、填充、对齐

C 语言的 struct 聚合数据结构，以及 C++ 的 class 类型，允许程序员以方便、容易维护的方式，来管理复杂的数据。然而在这些类型的大小、填充和对齐等方面，存在微妙的差别。

早期的计算机系统由于寻址或其它限制，经常限定结构体的总大小为一个很小的值，例如 32KB。这些限制经常在程序迁移到新体系架构下才会显示出来，导致链接错误或者非常难以寻找的运行时 bug。

C 和 C++ 标准没有规定结构体成员的对齐和填充。这意味着相同体系架构下的两个编译器，对下面结构体的格式也可能不同：

```
struct foo
{
    int i;
    char c[2];
    short s;
};
```

假设 int 为 32 位，char 为 8 位，short 为 16 位，编译器紧凑包装结构体的格式如下：

成员	字节偏移
i	0
	1
	2
	3
c	4
	5
s	6
	7

这是正确的，因为：

```
sizeof(foo) == sizeof(foo.i) + sizeof(foo.c) + sizeof(foo.s)
```

但是在许多体系架构下，访问“自然对齐”的数据元素是更快速的（甚至是强制必须的，视 CPU 设计而定，参考第五章“对齐”一节）。在这些系统中，编译器将会松散地组装结构体，插入不使用的填充字节来维持最佳对齐。因此 foo 在四字节自然对齐时的格式如下：

成员	字节偏移
i	0
	1
	2
	3
c	4
	5
填充	6
	7
s	8
	9
填充	10
	11

现在我们发现 `sizeof(foo) > sizeof(foo.i) + sizeof(foo.c) + sizeof(foo.s)`，导致许多程序员陷入编译器行为相关的黑暗森林。

这些差别可能会以很多方式伤害你的程序。如果你序列化数据到磁盘，然后使用不同编译器构建的可执行文件装载数据，填充和大小就会完全不一样，导致数据污染：

```
void read_foo(FILE *fp, struct foo *f)
{
    /* sizeof(foo) may not be consistent across platforms! */
    fread(f, sizeof(foo), 1, fp);
}
```

对布局做出的假设，当使用指针运算时也同样会导致类似的痛苦：

```
struct foo f;
/* WARNING: assumes tight packing, may fail unexpectedly! */
short *s = (short*) (f->c + sizeof(f->c));
```

第三方库希望传递给 API 的结构体按指定要求进行填充和对齐。这也就是预构建库必须显式地强制对齐/填充（通过编译器 `pragma`），或者通过插入填充字节来手工执行对齐的原因。而且即使这样做了，他们还必须小心地验证这种做法有效。

许多编译器提供命令行或者预处理器功能，来强制特定的对齐。例如 Microsoft Visual C++ 提供 `#pragma pack` 选项：

```
/* This pragma may not work with all compilers */
#pragma pack(1)
struct foo
{
    int i;
    char c[2];
    short s;
};
```

你也可以通过手工插入适当的填充字节来强制填充：

```
struct foo
{
    int i;
    char c[2];
    char pad1[2];
    short s;
    char pad2[2];
};
```

这种做法试图强制 `foo` 类型的对象进行对齐，并保证大小至少为 12 字节，但是尽管这个比前面版本要好一些，它仍然不是防弹的。例如有些编译器可能使用 8 字节对齐，或者本地类型的大小不一样（有一些 Cray 平台除了字符类型，其它的所有类型都是 8 字节）。

例子：Motorola 68000 和 PowerPC

Motorola 68K 系列处理器允许 32 位整数出现在任何地址边界，而不要求 4 字节对齐。而 Motorola PowerPC 处理器则强制 32 位整数必须在四字节边界对齐，在某些情况下需要进行填充。当程序员从 68K（用在早期 Macintosh）迁移到 PowerPC（现代 Macintosh）时，就会发现有时候他们的“可移植”代码异常地崩溃，原因就是这两种处理器对齐的区别。

不要对结构体的大小、填充、和对齐做出任何假设，除非你通过适当的编译器指令进行了显式地控制。

内存管理特性

内存管理是另一个潜在的可移植性问题源泉。许多机器表示和管理内存的方式并不一样，包括堆和栈两个方面。很容易就会习惯于某个平台的特定方式，当移植到另一个平台时就会出现。处理已释放指针和对齐内存分配是两个有差异的地方，需要特别注意。

Free 的效果

许多老的软件假设最近释放的指针还能够“短时间内”可用（也就是直到调用另一个内存分配操作之前）：

```
typedef struct node_struct
{
    struct node_struct *next, *prev;
    void *data;
} node;

void delete_node(node *n)
{
    free(n);
    n->prev->next = n->next; /* BAD! */
}
```

这在某些编译器上可能可以工作，但这个技术非常不好。因为不同的堆管理实现可能会立即覆写已释放的指针。有些编译器的调试版本会清空已释放指针指向的内存，任何使用那块内存的操作都会导致失败。

对齐内存分配

C 的 `malloc` 函数和 C++ 的 `new` 操作符都必须返回最大对齐的指针，因为内存可能被用作任意大小的类型和对象。如果你这样做：

```
char *c = (char *) malloc(sizeof(char) * 1);
```

返回的指针将会以访问任何对象所要求的最小对齐大小进行对齐，和请求分配的字节数无关。这样你通过指针使用动态分配的内存时，就永远不会出现错误的对齐。（当然，使用指针转换还是可以强制产生未对齐内存访问的）。

从实用主义讲，这样做降低了内存分配的最小粒度，因此对于相同大小的内存，多个小的分配，会比分配单一内存消耗更多的空间。

但是不可避免地有些编译器会比较糟糕，我就知道至少有一个编译器总是提供四字节边界对齐的内存数据，即使某些对象（例如 `double` 类型）要求 8 字节对齐。

如果对齐是至关重要的，或者你的对齐必须超过默认对齐，那就需要对 `malloc` 和 `new` 进行封装，填充适当的字节，并返回对齐于特定大小要求的指针。

堆栈

C/C++ 的堆栈是一个古怪的小怪兽。每个人都需要使用它，并且每个人都知道堆栈是什么，但是许多程序员也经常不了解堆栈的很多限制。

堆栈大小

通常编译器和链接器会静态地确定可用堆栈的大小，一旦堆栈溢出，程序会立即崩溃。不幸的是，当在有限的硬件中运行时，例如嵌入式和手持平台，程序很容易就会产生堆栈溢出。比如在早期的 `Palm OS` 上开发时，开发者很少能够获得几 K 字节的堆栈空间。使用 `BREW API` 的无线电话应用，只有可怜的 500 字节堆栈可以使用。

每次局部（自动）变量在作用域中使用时，都需要消耗堆栈。桌面 PC 系统的堆栈大小一般都以兆计算，这些程序员经常意识不到堆栈大小存在的问题。PC 程序员随意地声明 1KB 的数组来做临时存储，这种情况并不少见。她的程序在 `BREW` 平台立即就会崩溃，在 `Palm OS` 手持平台也只需要几个递归就会导致崩溃。

即使是桌面系统，也可能会遇到堆栈耗尽的问题，特别是那些使用大量递归的程序。例如几年前，我所在的小组正在开发一个必须由 `Windows` 迁移到 `Macintosh`（`Mac OS 9`）的应用。我们发现一个只在 `release` 版本中才会出现的神秘崩溃，我们跟踪问题到了音频子系统——至少那里是我们认为问题存在的地方。经历了一周的失望和抓狂的 `bug` 寻找之后，我们最终发现问题是堆栈贪污。编译器 `release` 版本的默认堆栈大小是 64KB，而 `debug` 版本则是 256KB，在代码的某个关键区域，我们使用了 65KB 的堆栈。这已经足够破坏一些重要的内部数据了，而问题在随后很久才最终出现（在音频子系统中）。

堆栈相关的崩溃通常都很诡异，并且很难追踪，因为产生的破坏可能会出现在离实际错误很远的位置。

alloca()的问题

与堆栈相关的一个神奇函数是 `alloca()`，它的诱惑来自于强大和简单，但却极度地危险。`alloca()`的用途和目的，实际上就是堆栈上的 `malloc()`。`alloca()`不从堆（heap）中分配数据，而是直接从堆栈中分配，而且当程序退出当前作用域时会自动释放数据。

使用 `alloca()`看上去有几个适当的理由。对于初学者来说，它更加简单，如果你的函数有许多不同的退出路径，你就不需要记住什么时候调用 `free()`。这虽然很便利但却并不是非常地吸引人；但是另一个诱惑则是 `alloca()`不会使用堆，因此可以避免碎片。它给你直接在本地堆栈上分配的便利的同时，还提供分配长度可变的强大功能（而不是依赖于硬编码的堆栈数组）。

但是 `alloca()`有太多的问题：对于初学者来说，它不属于 ANSI 标准，意味着它不能工作在所有平台（尽管它在足够多的平台都可用，导致许多人认为它是可移植的）。和 `malloc()`一样，它不能正确地工作于 C++的类。如果 `alloca()`不能找到足够的堆栈空间而失败，我们也不能保证它会返回 `NULL`（这一点和 `malloc()`不同），因此它可能会无论成功或失败，都给你一个堆栈空间（简单地通过堆栈指针调整），或者抛出一个堆栈溢出异常。反正你就是没有办法确定。

当调试器跟踪本地堆栈来检查变量时，`alloca()`还经常造成混乱。你单步跳过一个 `alloca()`调用之后，会立即发现在监视窗口中，无法查看调用堆栈和变量。

printf 函数

如果说 C 语言只剩下一个函数，那肯定是无处不在的 `printf()`，部分原因是许多程序员的“第一次”都是“Hello World!”，同时也因为 `printf` 本身就是非常强大和有用的函数。

C 标准规定了一组兼容和标准的 `printf()`格式符，例如整数值、浮点数、十六进制值、和字符串。但是在不同实现之间还是存在一些微妙的差别，例如非法值的处理、和打印编译器特定的数据类型。

有一些编译器能够正确地处理 `NULL` 参数，考虑以下例子：

```
printf("this is a string: %s\n", 0);
```

这个代码在某些系统中会崩溃，但是在其它系统中，运行时库会捕获到 `NULL` 条件，并打印出“NULL”或“null”。

由于编译器特定的类型扩展有许多，打印这些值在不同平台的结果都会不一样。一个常见的例子就是 64 位整数。Microsoft Visual C++的 64 位整数格式说明符是：

```
__int64 v = 0x0123456789ABCDEF;
printf("this is a 64-bit int: %I64i\n", v);
```

而 `glibc`（多数 GCC 实现所使用的 C 运行时库）则使用 `ll` 说明符：

```
long long v = 0x0123456789ABCDEFLL;
printf("this is a 64-bit int: %lli\n", v);
```

NOTE MinGW 是 Windows 系统的 GCC 移植，实际上使用 Microsoft 运行时库 MSVCRT.DLL，而不是 glibc。

虽然很少有现代商业应用依赖于 printf，因为图形用户界面（GUI）的流行。但是这些应用仍然需要面向缓冲的变种（sprintf()和 vsprintf()），这些函数被许多程序员大量使用。

一个解决办法是使用常量格式说明符，例如下面代码：

```
#ifdef _MSC_VER
#define PRINTF_SPEC_64BIT "%I64i"
#elif defined __GNUC__
#define PRINTF_SPEC_64BIT "%lli"
#endif
printf("this is a 64-bit int: "PRINTF_SPEC_64BIT"\n", v);
```

ANSI C99 规范引入了一组常量说明符，来处理这种情况，下一节我们会说明。

类型大小和行为

许多可移植问题都出现在编译器对类型的实现上。这包括处理 64 位类型，以及基本类型的大小和处理。

64 位整数类型

早先的 ANSI C 规范编写于 64 位 CPU 体系架构流行之前。而修订后的 C99 规范则通过 <stdint.h> 头文件和相关的 int64_t, uint64_t, long long 和 unsigned long long 类型，引入了 64 位整数的支持。

但是在 C99 标准之前，编译器厂商已经提供了不同的 64 位整数实现。在 DOS 和 Microsoft Windows 下类型是 __int64（Microsoft、Borland 和 Watcom 编译器都使用它）。Unix 系列的编译器则倾向于使用 long long 类型。当然，在支持 LP64 模型的本地 64 位编译器中，标准的 long 整数类型也可能是 64 位的，而 int 是 32 位。（参考第五章关于编程模型的细节）。

需要经常使用 64 位整数的可移植代码，要么抽象并封装 64 位类型的名字（使用 typedef）；要么如果遵循 C99，则坚持使用 <stdint.h> 和 <inttypes.h> 中的定义。C99 标准提供以下定义：

- long long 和正式的 64 位类型，如 int64_t 和 uint64_t
- 64 位常量宏定义 INT64_C() 和 UINT64_C()
- 64 位的 printf 格式说明符 PRIi64 和 PRId64

注意标准化的 64 位类型是 C99 的一部分，但在 C++ 中不可用。（到本书写作的时候，我还不知道有哪个编译器能够完全遵循 C99 规范）。

除了指定类型有不同的方式以外，定义常量的方式也不一样。使用 __int64 类型的编译器通常需要转换：

```
__int64 x = (__int64) 0xFEDCBA9876543210;
```

使用 long long 类型的编译器通常则使用 LL 后缀：

```
long long x = 0xFEDCBA9876543210LL;
```


当然，处理 64 位原生 long 整数的编译器可以直接使用常量，无需后缀。

基本类型的大小

每个编译器对基本类型和用户自定义类型的大小都有不同的实现。C 标准并没有规定特定类型的大小，它只对范围进行了要求。标准头文件<limits.h>包含了表 8-1 中所示的预定义常量。

表 8-1: C 标准头文件定义的类型大小

常量	定义	值
CHAR_BIT	char 的 bit 位数	>=8
SCHAR_MIN	带符号 char 的最小值	<=-127
SCHAR_MAX	带符号 char 的最大值	>=127
UCHAR_MAX	无符号 char 的最大值	>=255
CHAR_MIN	char 的最小值	如果 char 带符号,则必须等于 SCHAR_MIN; 否则为 0
CHAR_MAX	char 的最大值	如果 char 带符号,则必须等于 SCHAR_MAX ; 否则为 UCHAR_MAX
SHRT_MIN	short int 的最小值	<=-32767
SHRT_MAX	short int 的最大值	>=+32767
USHRT_MAX	unsigned short int 的最大值	>=65535
INT_MIN	int 的最小值	<=-32767
INT_MAX	int 的最大值	>=+32767
UINT_MAX	unsigned int 的最大值	>=65535
LONG_MIN	long int 的最小值	<=-2147483647
LONG_MAX	long int 的最大值	>=+2147483647
ULONG_MAX	unsigned long int 的最大值	>=4294967295
LLONG_MIN(C99)	long long int 的最小值	<=-9223372036854775807
LLONG_MAX(C99)	long long int 的最大值	>=+9223372036854775807
ULLONG_MAX(C99)	unsigned long long int 的最大值	>=18446744073709551615

表 8-1 中显示的值能够推断出某些大小相关的信息：
sizeof(char) <= sizeof(short int) <= sizeof(int) <= sizeof(long) <= sizeof(long long)

此外，这些值实际上也就规定了不同类型的最小大小，如表 8-2 所示：

表 8-2: 类型大小的最小值

类型	最小大小
char	8 位
short	16 位

int	16 位
long	32 位
long long	64 位

例子：POSH 和 64 位整数支持

POSH 提供一组可移植的 64 位抽象类型、常量定义、和 printf 格式定义符。

```
#if defined (__LP64__) || defined (__powerpc64__) || \
defined POSH_CPU_SPARC64
# define POSH_64BIT_INTEGER 1
typedef long posh_i64_t;
typedef unsigned long posh_u64_t;
# define POSH_I64(x) ((posh_i64_t)x)
# define POSH_U64(x) ((posh_u64_t)x)
# define POSH_I64_PRINTF_PREFIX "l"
#elif defined _MSC_VER || defined __BORLANDC__ || \
defined __WATCOMC__ || (defined __alpha && defined __DECC)
# define POSH_64BIT_INTEGER 1
typedef __int64 posh_i64_t;
typedef unsigned __int64 posh_u64_t;
# define POSH_I64(x) ((posh_i64_t)x)
# define POSH_U64(x) ((posh_u64_t)x)
# define POSH_I64_PRINTF_PREFIX "I64"
#elif defined __GNUC__ || defined __MWERKS__ || defined __SUNPRO_C \
|| defined __SUNPRO_CC || defined __APPLE_CC__ || defined \
POSH_OS_IRIX || defined _LONG_LONG || defined _CRAYC
# define POSH_64BIT_INTEGER 1
typedef long long posh_i64_t;
typedef unsigned long long posh_u64_t;
# define POSH_U64(x) ((posh_u64_t)(x##LL))
# define POSH_I64(x) ((posh_i64_t)(x##LL))
# define POSH_I64_PRINTF_PREFIX "ll"
#endif

/* hack for MinGW */
#ifdef __MINGW32__
#undef POSH_I64
#undef POSH_U64
#undef POSH_I64_PRINTF_PREFIX
#define POSH_I64(x) ((posh_i64_t)x)
#define POSH_U64(x) ((posh_u64_t)x)
#define POSH_I64_PRINTF_PREFIX "I64"
#endif
```

使用 POSH 宏，你可以可移植地指定和使用 64 位值：

```
posh_i64_t x = POSH_I64(0x1234567890ABCDEF);
printf("64-bit value = %"POSH_I64_PRINTF_PREFIX"d", x);
```

标准并没有规定所有类型的大小不能同时为 64 位，实际上有些体系架构对类型大小特别激进（例如某些 Cray 变种有 32 位 short 和 64 位 int，而没有任何 16 位类型）。非常多的程序员假设 short 是 16 位，而 long 是 32 位的，这并不能得到保证（但非常有可能，这也是这个假设存在的原因）。

由于这些原因，你永远不应该对大小做出硬编码的假设，如下：

```
int *foo = (int *) malloc(4); /* will fail if int is 8-bytes! */
```

相反，你应该使用 sizeof：

```
int *foo = (int *) malloc(sizeof(int));
```

或者：

```
int *foo = (int *) malloc(sizeof(*foo));
```

如果你需要具体的类型，使用适当的基于底层平台的类型定义，并通过编译期断言验证这些假设。C99 的<stdint.h>提供一组确保大小的类型定义，POSH 则在<posh.h>中提供了相同的东西（参考第三章“抽象数据类型 typedef”一节）。

有符号和无符号字符类型

ANSI C 标准没有规定 char 默认是有符号还是无符号的。由于隐式地类型提升规则，这可能导致一些极度混乱的行为。考虑下面例子：

```
char c; /* 未确定符号，因为没有显式地指定 */
c = 0xFF;
if (c == 0xFF) /* 注意：'c'提升为 int 以进行比较 */
{
    /* 根据不同的编译器，这里可能永远无法到达 */
    printf("Hello!");
}
```

这个代码段可能不会按你期望的方式运行，依赖于编译器对字符默认符号的选择。如果 char 是隐式带符号的，则 C 类型提升规则会把它扩展为带符号整数值 0xFFFFFFFF，然后进行比较（在 32 位整数的机器上）。因此 0xFF 和 0xFFFFFFFF 比较的结果是不相等。

但是如果字符默认是无符号的，则 c 提升后的值仍然是 0xFF，比较将会通过。

遇到这个问题的一个常见地方就是标准库函数 getchar()。

```
char c;
while ((c = getchar()) != EOF)
    processCharacter(c);
```

EOF 默认被定义为-1，因此在 char 默认为非负数的系统中，while 循环永远不会结束。

很明显这个问题很不直观，可能会引起非常难以理解的 bug。要么不对 char 的符号做出假设，要么通过全局编译器选项强制 char 的符号（多数编译器允许你设置 char 的默认符号），并配合全局编译期断言进行验证：

```
/* this ensures that characters are signed */
CASSERT((char)0xFF == (int)~0, char_signed);
/* this ensures that characters are unsigned */
CASSERT((unsigned char)0xFF == (int)0xFF, uchar_unsigned);
```

enum 用作 int

当你创建大量常量组，并希望自动产生新的值时，enum 数据类型是非常便利的工具。但是 ANSI C 规范没有规定 enum 的大小，允许编译器根据实际枚举值选择最佳的大小。例如：

```
enum Color
{
    RED, GREEN, BLUE
};
```

某个编译器可以推断出合法值的范围是 [0, 2]，因此可能决定 8 位字符比整数更加适合，因为这样可以消耗更少的空间。另一个编译器却认为最佳的实现是 32 位，因为这样在目标平台能有更好的对齐。

enum 的大小区别可能会导致许多不同和微妙的错误。首当其冲的就是本章开头讨论的结构体填充：

```
enum Color {RED, GREEN, BLUE};
struct foo
{
    enum Color a, b, c, d;
};
```

根据编译器为 Color 枚举选择的不同大小，结构体可能会有 4 字节或者 16 字节的大小。

即使是简单如传递参数，也可能会出现问题：

```
void set_color(enum Color c);
```

在多数现代计算机系统中，传递到栈中的参数会被填充和对齐为 32 位，因此出错的可能性被降低了。但是在简单的 8 位微控制器平台就可能会出现未预料的行为。就上面例子来说，如果调用方假设 enum Color 是 1 字节，但是被调用的函数却要求 4 字节，就会出现未预料的错误。

多数编译器有可选开关将 enum 值作为 int 值来处理，强制所有 enum 值的大小至少是整数大小。如果编译器不提供选项，你可以通过模拟常量来强制 enum 的最小大小：

```
enum Color
{
    RED,
    GREEN,
    BLUE,
    FORCE_INTEGER = 0xFFFFFFFF /*force Color to be at least 32-bits*/
};
```

数值常量

不能假设数值常量在所有平台都使用相同的表示方法。程序员通常假设-1 和 0xFFFFFFFF 是相同的，但只有在整数是 32 位，而且采用二进制补码表示时才是正确的。0xFFFFFFFF 在 64 位系统只是一个很大的正数值而已。

另一个常见的热点问题与位掩码有关。如果你要屏蔽一个值的低 4 位，下面做法是经常见到的：

```
unsigned long x = some_value;
x = x & 0xFFFFFFF0;
```

只有系统是 32 位整数时，上面代码才正确。我们不应该像上面代码那样使用硬编码掩码值，而应该使用反码操作，来确保除了你希望屏蔽的位为 0，其余位都是 1：

```
unsigned long x = some_value;
x = x & (~0xF);
```

同样的道理，当你实际想要的是~0 时（所有位为 1），请避免使用-1。

带符号和无符号右移

在右移一个带符号整数时，C 标准允许实现来选择是否进行符号扩展：

```
int32_t x = 0x80000000;
x >>= 31; /* 不同的实现，结果可能是 1 或 0xFFFFFFFF */
```

如果你希望无符号移位（例如你在操作屏蔽位），你应该在执行移位操作前，将值强制转换为无符号数。不要试图通过右移来完成快速地除以 2，相反你应该使用除法运算，而将产生优化的机器级移位指令的任务，交给编译器去做。

调用约定

函数调用在概念上来讲对于程序员是非常简单的，实际上却是非常复杂的操作。调用方和被调用的函数必须遵守相同的协议，该协议定义了以下几个方面：

- 参数传递的方式
- 返回值如何返回
- 哪些状态必须被保留

这些因素集合在一起被称为调用约定。根据具体的编译器和操作系统，不同的平台会有非常不同的调用约定，有时候甚至同一体系架构都会有多个调用约定。

例如 Microsoft 的 PASCAL 调用约定有以下特点：

- 从左向右传递参数
- 要求被调用函数清理堆栈
- 要求被调用函数保留方向标志和 EBX, ES, FS, GS, EBP, ESI, 和 EDI 寄存器

此外，PASCAL 函数的名字会被转换为大写，因为 Pascal 语言是大小写无关的。

在多数平台，编译器和链接器都遵守一个严格的调用约定，通常由操作系统定义并作为应用二进制接口（ABI）的一部分。遵循单一的协议避免了混合调用约定时的许多不必要的复杂性。例如一个程序的调用约定与它使用的库的调用约定不同时，就会出现错误。（根据不可兼容性，要么是链接时错误，要么是运行时错误）。

不幸的是 Intel x86 和其它复杂指令集计算机（CISC）体系架构（如 Motorola 68K），发明了太多的调用约定来解决不同的问题。有些调用约定是 C 语言所必需的（如支持可变参数列表）；其它一些约定则是设计用来最小化空间或最大化性能。由于 C 和 C++标准都没有解决调用约定的问题，编译器编写者可以自由地提供不同的调用约定。

Intel x86 体系架构至少有以下三种常见的调用约定：

- cdecl 是默认的调用约定，因为它非常灵活且允许可变参数列表（可变参数列表要求调用方清除堆栈，因为被调用函数并不知道有多少个参数被传递进来）。但是这个调用约定有轻微的性能损失。
- stdcall 约定更快更通用一些。Windows ABI 的大多数 API 入口点都使用了 stdcall 调用约定（通过 WINAPI 宏）。
- register 或 fastcall 调用约定将实现相关的数量的参数（通常是两个）传递到寄存器，其它参数则分布在堆栈中。这是理论上最快的调用约定；但是实践中有足够的因素导致这种速度优势无法得到。随着参数数量的增加，相对于其它约定的性能优势也会降低。

这些约定的特性总结如表 8-3。

表 8-3：Intel x86 常见的调用约定

约定	参数位置	参数顺序	堆栈职责
cdecl	堆栈	从右向左	调用方
stdcall	堆栈	从右向左	调用方
fastcall/register	寄存器和堆栈	从右向左	调用方

虽然这都是常用的调用约定，但请记住不同的编译器仍然可能对这些约定有稍微不同的解释。register 调用约定在不同的编译器中可能不同，并且有时候在相同编译器的不同版本中也不一样。不过编译器厂商认识到了互操作性的重要性，因此你会发现不同体系架构对调用约定的实现很多时候都是一样的。此外，操作系统都会推广自己的标准 ABI，因此如果 Windows 说 stdcall 是怎麼样的，你可以打赌编译器厂商会遵守 Windows 的定义。

名字装饰

如果程序以某种调用约定编译，却试图链接以另一种约定编译的库，就会出现潜在的问题。因此多数工具链都会根据调用约定对函数名字进行不同的装饰。

Microsoft Visual C++对函数 int test(int x)会产生以下符号名：

约定	装饰后的名字
cdecl	_test
stdcall	_test@4

fastcall	@test@4
----------	---------

如果以 `cdecl` 约定构建了一个库，而应用以 `stdcall` 来调用它的函数，API 调用会出现链接错误。这比链接成功，而由于调用约定不兼容在运行时崩溃，要好得多。

函数指针和回调

调用约定并不仅仅影响静态链接；使用函数指针时它也会出现问题。如果你向应用描述一个接受函数指针的 API（例如注册一个回调），你需要指明它的调用约定：

```
/* NOTE: use a CDECL macro of some type so that it works on
multiple compilers/platforms */
void register_callback(void (CDECL *func)(int a, int b, int c));
```

可移植性

可移植软件应该避免对调用约定做出任何假设。永远不要对堆栈中的参数格式做任何假设，正如下面例子所说明的那样：

```
void func(int a, int b)
{
    int *pa = &a; /* what if a is in a register?! */
    /* no guarantee that they're contiguous! */
    int *pb = (int*)((char*)&a + sizeof(int));
}
```

如果你开发的库以二进制的形式发布，你很有可能需要处理调用约定的问题，除非你工作的平台只有一种调用约定。最经常的情况是，你要么需要发布以不同调用约定编译的多个二进制版本，要么通过编译器私有的函数签名修饰符强制使用特定的调用约定，如下：

```
#ifdef _MSC_VER
#define API_TYPE __cdecl
#else
#define API_TYPE
#endif
void API_TYPE MyFunction(void);
```

例子：SAL 对调用约定的处理

SAL 使用编译时自动启用的调用约定，由于 SAL 是以源代码的方式整合到其它项目中，这样做没有问题。如果要发布 SAL 库的二进制版本，则打包的人就需要注意编译和链接时使用的调用约定。

SAL 显式地设置了用户自定义回调的调用约定：

```
typedef struct SAL_Callbacks
{
    sal_i32_t cb_size;
```

```

void * (POSH_CDECL *alloc)(sal_u32_t sz);
void (POSH_CDECL *free)(void *p);

void (POSH_CDECL *warning)(const char *msg);
void (POSH_CDECL *error)(const char *msg);
} SAL_Callbacks;

```

SAL 依赖于 POSH，后者根据编译器和平台适当地定义了 POSH_CDECL：

```

#if defined POSH_CPU_X86 && !defined POSH_CPU_X86_64
#  if defined __GNUC__
#    define POSH_CDECL __attribute__((cdecl))
#    define POSH_STDCALL __attribute__((stdcall))
#    define POSH_FASTCALL __attribute__((fastcall))
#  elif defined _MSC_VER || defined __WATCOMC__ || defined \
__BORLANDC__ || defined __MWERKS__
#    define POSH_CDECL __cdecl
#    define POSH_STDCALL __stdcall
#    define POSH_FASTCALL __fastcall
#  endif
#else

/* This will likely have to be expanded if running on a system with
varying calling conventions and which was not x86 based, such as
68K Macintosh systems which had C, Pascal, and OS calling conventions */

#define POSH_CDECL
#define POSH_STDCALL
#define POSH_FASTCALL
#endif

```

返回结构体

不同编译器对于返回聚合数据类型（例如 `struct`, `class`, 和 `union` 对象）的处理方式是不一样的。小的对象通常都以单独的寄存器返回，但有些编译器无论对象多大，都可能会直接返回堆栈中的对象。

按值返回大的对象（超过寄存器大小）时情况更为混乱。有些实现会传递一个隐藏的本地堆栈变量的指针；其它实现则可能返回静态变量的地址（这不是多线程安全的，但是在单线程操作系统如 MS-DOS 或 Mac OS 中是可以的）。如何返回聚合对象并没有标准，所以调用方传递一个本地变量指针，而不是试图通过值拷贝，是更好的方法：

```

struct thingy get_a_thingy_by_value(void)
{
    struct thingy result;
    /* do some stuff */
}

```



```

        return result; /* how is this returned? Implementation dependent */
    }

    void get_a_thingy_by_copy(struct thingy *t)
    {
        struct thingy s;
        /* do some stuff */
        *t = s;
    }

```

位域

C 和 C++能够在结构体中用名字来标识特定位置，称为位域：

```

struct OpCode
{
    int code: 4;
    int operand0: 4;
    int operand1: 4;
    int flags: 8;
};

```

位域存在太多的问题，以致于我不太确定从何说起。首先语言标准没有规定位域默认有无符号，因此如果某个特定编译器实现位域为带符号，实际范围 ± 7 而你试图去取 15 的值时，就会出现問題。

此外位域的顺序和填充也由实现决定。编译器为了保持对齐，很有可能会自动插入未命名的位到结构体中：

```

/* 下面结构体不太可能是 64 位的大小，除非是在拥有 64 位本地整数的系统中 */
struct RegisterBits
{
    int a : 31;
    int b : 31;
    int c : 2;
};

```

位域的一个常见用途，是更容易地访问硬件设备寄存器的特定位置范围，一般是和寄存器大小的整数一起放在 union 中：

```

/* 下面代码在多数 32 位系统中都无法得到你预期的结果 */
union RegisterSpec
{
    struct RegisterBits register_bits;
    int64_t register_direct;
};

```

位域看上去很美，但却充满了可移植问题。不要使用位域——它真的没有那么方便。

注释

即使是简单的注释，也可能产生可移植问题。嵌套注释、C 源文件中的 C++ 风格单行注释(//)、假设注释转换为空白，所有这些都可能出现。

C 和 C++ 规范没有说明是否允许嵌套多行(/*)注释。因此下面这行代码可能在某个编译器上可以工作，却在另一个编译器中失败：

```
/* I have commented this out but the following /* may cause problems */
```

如果一个编译器支持嵌套注释，上面注释就会导致一个编译错误，因为存在两个/*符号，但只有一个*/。这可能会引起很大的混乱，因为某个之前完全正常的代码，在另一个平台忽然编译失败，而且通常都是类似于“unexpected end of file found inside comment”这样的隐晦的错误信息。

有人半开玩笑地说 C++ 语言最大的贡献就是单行注释：

```
// This line is commented out
```

这是一个很小但非常方便的特性，许多现代的语言都支持，但 C 语言却没有。许多 C 编译器厂商以非标准扩展的方式增加单行注释支持。实际上，这已经流行到许多开发者并没有意识到这是非标准的（多数主流编译器甚至在 C99 标准修订之前就已经支持单行注释）。

ANSI C99 规范正式向语言增加了单行注释，但是有些编译器还没有实现这个功能（可能永远也不会实现）。如果你有大量代码使用了单行注释，而你不得不修改它们，那你将会有个长长的夜晚与文本编辑相处。

当然，如果你使用 C++，那你可以快乐地忽略这个问题。

注释的最后一个问题，是关于早期 C，空白注释通常用作字符连接的技巧：

```
/* this doesn't do what you want under ANSI C! */
#define combine(a, b) a/**/b
```

这个代码的意图是替换 combine(a,b) 为 ab。

ANSI C 提供一种更好的方法来完成这个工作，使用连接操作符：

```
/* this works with ANSI C, but not under K&R C */
#define combine(a, b) a##b
```

如果你坚持要支持老式的字符连接，你可以使用 __STDC__ 宏：

```
#ifdef __STDC__
#define combine(a, b) a##b
#else
```

```
#define combine(a, b) a/**/b  
#endif
```

小结

C 和 C++ 标准看上去很严格精确，但编译器作者对于标准的解释、和特性的实现都有相当大的自由和余地。开发可移植软件时，需要处理不同编译器实现之间的问题。只要拥有正确的知识、合理的计划和准备，是可以减少很多类似痛苦的。

第9章 用户交互

任何现代软件应用的一个关键组成是展现给用户的界面，通常依赖于操作系统的图形用户界面(GUI)的支持。常见的例子包括 Microsoft Windows GDI、Commodore Amiga Intuition、Atari ST GEM、Linux X11 及可选的桌面管理器如 GNOME 或 KDE、以及 Palm OS。

虽然许多基本的命令行工具程序（如过滤器和批处理器）可以通过最简单的机制（如 `printf()` 和 `fgets()`）来实现，现代应用则拥有复杂的用户界面，它们天生就是依赖于系统的。抽象用户交互是软件开发者架构考虑的关键因素。

用户界面的演化

在 glitzy 用户界面流行之前（根据你使用的是 Mac 还是 PC 用户，分别是 1985 或 1990 年），用户需要通过命令行或者命令提示符与计算机系统进行交互。用户界面进化主要就是从基于文本的命令行向今天几乎所有电脑使用的基于鼠标的 GUI 的迁移。

命令行

命令行界面由你输入命令，然后查看显示器滚动的输出（或者如果你足够老，可能还会记得电传打字机）。没有鼠标、没有窗口、没有对话框——只有无穷无尽的文本。类似“立即响应按键”之类的概念没有很好的支持，也完全不可移植。FTP 和 Telnet 等应用就是采用命令行方式实现。

虽然很原始和简单，文本驱动的命令用户界面却同时也是可移植的，因为它们使用了 C 标准库的标准 I/O 函数，如 `fputs()`，`printf()`，和 `fgets()`。当你能完全抛开图形概念和显示器系统时，编写一个可移植程序就简单很多了。

对于简单和原始的程序，命令行界面也是可以接受的，但通常商业应用都需要视觉美观和操作直观的 GUI。

窗口系统

Apple 在 1984 年发布的 Macintosh 计算机中使鼠标驱动的 GUI 得以商业化，这个 GUI 系统最初由 Xerox 的 Palo Alto 研究中心开发，这是个人计算机发展史上的重要里程碑事件。在随后的二十年里，GUI 已经把命令行推到了消退的边缘。

GUI 被认为天生就是私有的。许多公司对 GUI 采取了专利和版权的保护措施，例如 Apple 对自己认为独特的概念（如工作区中的窗口重叠）进行了保护；所有公司都不愿意公开摹仿竞争者的用户界面（没有人喜欢被看作追随者而不是领导者）。

结果就是到了 1980 年代末期，产生了无数互相竞争（而且私有）的 GUI 系统：

- Apple 的 Mac OS（到 System 9 为止）
- Apple 的 Mac OS X/Cocoa（基于 NeXT 公司的 NeXTStep）
- C64 的 GEOS 和 Apple II
- PC 的 GeoWorks
- Atari ST 和 PC（较少）上的 GEM

- Commodore 的 Amiga Intuition
- 微软的 Windows（已经进化过无数次了）
- 基于 X Window 的窗口管理器，如 Motif、Sun 的 OpenLook/OpenWindows、GNOME、和 KDE
- IBM OS/2 的 Presentation Manager
- 基于 Adobe PostScript 的显示系统，如 NeXT 的 NeXTStep 和 Sun 的 NEWS

而这还仅仅只是桌面计算机！PDA 和手持设备市场也有自己的用户界面派系。

当我们谈到 GUI 时，实际上指的是某个 GUI 实现的一个或多个组件。GUI 可以包含三个不同的层次：

- 窗口工具集，提供原始的描绘和事件管理能力
- 窗口管理器，展现通用的外观
- 桌面管理器，处理文件系统导航和管理

有些操作系统，如 Apple Mac OS 和 Microsoft Windows，只有一个整体的用户界面，把上面三个层次全部整合到操作系统中。其它操作系统，特别是基于 Unix 的系统，把操作系统和 GUI 独立开来，GUI 本身又分为独立的层次。例如典型的 Linux 桌面可能使用 X Window 系统作为窗口系统，Sawfish 作为窗口管理器，GNOME 作为桌面管理器。

和多数工业领域的萌芽阶段一样，用户界面的早期竞争者之间斗争非常激烈，杀得你死我活，最终只有少数幸存下来。今天桌面计算机占主导地位的用户界面是 Microsoft Windows，然后是范围小很多的 Apple Macintosh（包括老的 Mac OS 和新的 Cocoa）。Unix 和类似 Unix 的操作系统（Sun Solaris、IBM AIX、Linux 等等）使用的 X Window 占据着剩余的市场，份额非常非常小。

虽然 GUI 通常与操作系统都缠绕在一起，例如 Microsoft Windows。但在某些情况下，用户界面也能与操作系统无关。最常见的例子是 MIT 的 X Window 系统 GUI 工具集，它可以部署在任何操作系统之上，但一般只用在 Unix 系列操作系统的工作站中（也是多数 Linux 发行版的默认界面库）。

本地 GUI 还是应用 GUI

尽管操作系统提供了图形界面，有时候应用可能会希望展现自己的图形 GUI。这种方式需要完成大量的工作，缺点非常多，特别是与相同机器上的其它应用保持一致的外观非常困难。因此当用户使用你自定义界面的程序时，他们通常都会为应用非标准的行为而挣扎和生气。

好的一面是，通过解除应用和特定操作系统用户界面的耦合，你增强了软件的可移植性。至少你可以保证应用在跨平台时保持一致。你的忠实用户会感谢你的应用不管在哪个系统中运行，都能保持一致。

底层图形

用户界面和现代的计算机应用都经常需要显示高质量的图形。早期的计算机系统允许直接访问视频卡的内存（frame 缓冲区）；现代的操作系统则会拒绝用户级进程在这样底层的级别上访问任何硬件。

提供 frame 缓冲区访问的特殊 API，都是通过管理应用向“表面”内存中写入的数据，然后把结果返回到该缓冲区中。Microsoft Windows 通过 GDI 的设备无关位图，和底层的 DirectDrawSurface（DirectX API 的一部分）来支持这个特性。Macintosh 和 Linux 系统也提供类似的功能。

这种“画在表面，然后复制到视频卡”的基本概念，在多数操作系统中都是通用的，因此在它们之间移植只需要简单的抽象即可。

尽管如此，有时候还是可以直接访问视频内存，要么是在无保护的操作系统中，要么是通过设备驱动访问底层硬件。应用可能会使用一个指针指向后端缓冲区，对该缓冲区进行操作，然后在结束时，再翻转到前端缓冲区。这种“翻转”由更新显示器刷新硬件的指针组成。更新这个指针是非常快速的，而且通过“翻转”缓冲区你还避免了多余的拷贝。

今天的计算机程序对图形的需求已经完全超过了多数程序员自己编码的范围，因此直接访问 frame 缓冲区已经不像 10 年前那么重要了。流行的 3D 库，如 OpenGL 和 DirectX，执行高级、复杂的绘画，最大限度地使用底层硬件的所有特性，因此更进一步地减少了访问 frame 缓冲区的需要。

数字音频

如今声音已经渐渐成为计算机的必备体验，但在 1998 年数字音频仍然是 PC 的可选组件（尽管 Macintosh 和其它家用电脑早就支持数字音频）。

每个操作系统都有一个或多个机制来获得原始波形，通常编码为 pulse code modulated(PCM)采样数据，然后发送到扬声器。有些操作系统也提供分层的 API，增强对底层声音子系统的访问功能。例如 Microsoft Windows 就有高级的 PlaySound API，中级的 waveOut API，和非常底层的 DirectSound 缓冲区操作 API。应用可以根据自己的需求，为了方便和控制，自由选择最适合的 API。

其它操作系统通常也有类似的功能划分。例如 Mac OS X 的 Cocoa API 有 NSSound 类处理基本的声音装载和播放。下面是中级（现在已经不推荐使用）CarbonSound/SoundManager API。最底层的声音接口是 CoreAudio，提供对数字音频子系统非常原始高性能的访问。

多数现代操作系统都遵循这种基本模式。提供高级、容易使用的声音接口，用来实现简单的功能；同时又提供底层的接口，供应用进行复杂的音频处理。封装所有这些库并不困难，但非常烦琐，因为声音子系统的各种实现存在各种微妙的差异。跨平台的声音库有商业的 Miles Audio 和 FMOD，开源的 SDL、OpenAL、PortAudio，当然还有我编写的 SAL。

但是也有一些操作系统和设备完全不提供任何声音 API，要求程序员自己直接访问硬件设备。在这种情况下，你需要编写自己的声音访问、混合、和缓冲区管理引擎。

SAL 例子：处理数字音频

SAL 没有对底层声音 API 的能力进行抽象，相反它对这些能力进行了封装。SAL 使用声音 API 的底层缓冲区管理功能来实现自己的高级采样播放和混合特性。

有一些声音 API 拥有自己内建的混合功能，但通常都很薄弱和不可靠。SAL 通过实现自己的混合器，并完全依赖于后端来播放缓冲区，解决了这个问题。

输入

跨平台应用必须处理不同输入设备之间的差别。虽然几乎所有计算机都有键盘和鼠标，但键盘和鼠标配置的微小差别也可能影响移植性。

键盘

你可能认为键盘就是键盘，没什么大不了的地方。但在跨平台软件开发时，情况就不是这样。即使是相同计算机系统的键盘，按键也可能是不一样的。笔记本键盘和桌面计算机键盘在键位布局上通常就有很大的不同，因为笔记本键盘需要减小空间的使用。许多键在某些系统中并不存在。下面是一些例子：

- 早期 sun 的键盘有 15 个键不存在于 PC 键盘中。
- 早期 iMac 键盘没有数字键区。
- 早期 Macintosh 没有 PC 上常见的 ESC 和功能键。
- 现代的 Apple Pro USB 键盘没有 NUM LOCK、PRTSCRN、和 SCROLL LOCK 键。

你不能假设这些键一定存在，也不能依赖于它们的位置。另外不同语言的键盘还存在更多的区别。（参考第 16 章国际化的相关讨论）。

鼠标

多数情况下，鼠标操作在不同平台上都是一样的，但你仍然需要处理不同的配置。一个例子是 Apple Pro 的单键鼠标。另外一个极端是 Microsoft IntelliMouse Explorer，拥有五个按键和一个滚轮。还有另外一些设备，如 Wacom 数字化平板和笔记本触摸板，对应用来说它们就像是鼠标，但实际上它们却根本就不是鼠标。

需要运行在各种系统的应用，不能依赖于鼠标的按键数，适当的时候还应该支持额外的按键（以及滚轮和拇指滚轮）。

典型的桌面操作系统通过 GUI 工具集的事件系统，来提供鼠标和键盘信息的访问。例如在 Windows 中，应用在消息处理函数中处理 WM_MOUSEMOVE、WM_KEYDOWN 和 WM_KEYUP 消息。Cocoa/OS X 应用则覆盖 keyDown:、keyUp:、mouseDown:和 mouseUp:方法（直接获取 NSEvent 也是可以的）。X Window 应用则使用 XPending()和 XNextEvent() API 来获取 XEvent 事件。

操纵杆和游戏控制器

虽然鼠标和键盘可能是最流行的输入设备，但输入设备不仅仅只有这些。许多计算机系统也支持其它输入设备，如操纵杆和游戏控制器。这些设备很少与事件系统整合在一起，因为多数窗口系统都认为操纵杆不是高优先级的输入设备。

对于那些必须支持其它输入设备的应用，需要使用系统特定的 API。Windows 有标准的 Win32 API: GetJoyPosEx(), 以及完整的 DirectInput 库。Linux 则把操纵杆认为是标准 Unix 文件设备（应用打开/dev/js0 或类似的其它文件，然后执行 read()操作）。

对输入进行抽象最烦琐的部分，是每个操作系统对于事件如何传送给应用都有不同之处。在 Windows 中有标准事件循环，应用还可以增加特定消息的处理回调函数。在 X Window 系统中，应用可以直接调用事件系统，或者设置一个独立的线程来处理事件队列。

另外，轮询操纵杆设备通常还必须在标准事件循环之外进行。

跨平台工具集

由于管理不同用户界面非常困难和复杂，许多跨平台工具集被设计出来抽象这些差别。包括 C++ 的 wxWindows（后改名为 wxWidgets）、QT、和 GTK+；Java 的 SWT、AWT、和 Swing。第 18 章会更加详细地讨论。

小结

虽然命令行程序仍然有一席之地，但今天的多数应用都必须拥有图形界面，并且支持声音。这些接口在不同的操作系统中差别非常大，可移植软件必须能够在不同 GUI 中迁移。

第10章 网络

计算机通过网络连接在一起已经几十年了，但是直到 Internet 流行之后，网络才成为跨平台开发很重要的一个问题。网络通信意味着多样性，因为两个完全不同的系统也可能连接在一起。

当多样化的机器连接在一起并交换数据时，你需要处理数据存储和装载的问题。第 15 章会讨论这个问题。编写可以运行在不同网络层，能够与不同平台进行通信的软件需要很高的技巧，本章我就是来解决这个问题。

网络协议的演化

网络最早出现的时候，主要是由独立私有的硬件和软件实现组成，大家都在竞争和控制局域网（LAN）和广域网（WAN）标准的市场。数字设备公司有它的 DECnet 协议簇；IBM 有 NetBIOS 扩展用户接口（NetBEUI）；Apple 有 LocalTalk/AppleTalk 和 OpenTransport；Novell 有 Internetwork Packet Exchange（IPX）和 Sequenced Packet Exchange（SPX）；Banyan 系统提供 VINES；甚至基于 DOS 的 PC 都拥有低端的网络操作系统，如 Artisoft LANtastic。不过这些协议大多局限为 LAN 或私有 WAN。

你可以想像得到，如此多的不同协议导致编写可移植网络应用非常麻烦和困难。不过当时很少程序需要移植或工作于多样的体系架构下，所以一时也没有问题。

到 1980 年代期间，焦点逐渐转移到可互操作网络上，因此决定宣布传输控制协议/因特网协议（TCP/IP）作为 ARPAnet（因特网前身）的标准协议。这个标准化直接促使和确保了因特网的急速增长。

在今天的环境里，真正相关的网络协议就是 TCP/IP。它是今天多数计算机使用的标准网络实现；任何连接到因特网的计算机都需要使用 TCP/IP。私有的协议，如 DECnet 和 IPX/SPX 已经极少使用，通常只有在遗留应用和系统中才能见到。

编程接口

在 1980 年代早期，伯克利发布的 Unix 4.1c 包含了 BSD Socket API，如今已成为基于 Unix 的网络应用的标准编程 API。Socket API 的简单和容易使用也导致其它 Unix 变种，甚至是微型计算机操作系统采用它。如 BeOS、Amiga、OS/2、和 Windows。

如果你正在编写可移植的网络应用，通常要么你直接使用 Socket API；要么使用基于 Socket 的高级 API，例如远程过程调用（RPC）、Java 远程方法调用（RMI）、分布式对象（CORBA 和 COM/DCOM）、或者简单对象访问协议（SOAP）。

Sockets

BSD Socket API 是使用最普遍的底层 TCP/IP 编程接口，几乎所有支持网络的计算机系统都可以使用 Socket，因此也成为所有类 Unix 系统的标准接口。当然，Microsoft 再次实现了自己的 Socket API：WinSock。如果你使用底层网络通信来编写跨平台应用，一般只需要抽象 WinSock 和 BSD Socket 就可以了。

虽然 Socket 有相对稳定的 API 和标准化的概念，但是不同的 socket 实现或 TCP/IP 协议栈实现也可能会有很大的不同。这些实现的区别没有既定规则，因此通常都非常微妙（例如某些配置变量的默认值不同）。

WinSock 与 BSD Socket 标准是非常相似的——许多方面都完全一样，但又需要你处理一定的区别之处。它正好处于“足够多的区别，需要使用抽象”和“非常相似，几个条件语句就能搞定”之间。

抽象 WinSock 和 BSD Socket 非常简单，因为它们的概念和语法大多一样。它们都是对 TCP/IP 功能，特别是 UDP 和 TCP，很薄的一层封装。并以相同的语法提供打开和关闭连接、发送和接收数据等能力。

WinSock 和 BSD Socket 的区别很小，但非常令人恼火。例如最基础的 socket 句柄类型，在两个平台就不一样。BSD Socket 定义为带符号整型，符合 Unix 对文件描述符的惯例；WinSock 则使用 SOCKET 类型，定义在 winsock.h 和 winsock2.h 中，实际上是无符号类型。这意味着非法 socket 的值依赖于平台：BSD Socket 是 -1；Windows 是 0xFFFFFFFF。

要解决这个问题，你可以在 BSD Socket 中整合 SOCKET 类型和 INVALID_HANDLE 定义，使其更像 Windows 版本，并且避免污染核心代码，如下：

```
#if !defined POSH_OS_WIN32
#define INVALID_HANDLE -1
typedef int SOCKET;
#endif
```

这样在两种系统中分配 socket 时都只需要单一的代码路径就可以完成，达到拥有最多共享资源的目标，而又不需要依赖于大量的条件编译指令。下面代码：

```
SOCKET s = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP);
if (s == INVALID_HANDLE)
    return -1;
/* 上面代码可以在 BSD Socket 和 WinSock 中编译和运行 */
```

比下面这样的代码要好得多：

```
#ifdef POSH_OS_WIN32
    SOCKET s;
#else
    int s;
#endif
s = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP);
#ifdef POSH_OS_WIN32
    if (s == 0xFFFFFFFF)
#else
    if (s == -1)
#endif
    return -1;
```

WinSock 和 BSD Socket 的另一个细小差别是头文件处理。在 Windows 下，WinSock 的定义在 <winsock.h> 或 <winsock2.h> 中。BSD Socket 则要求一组头文件，包括 <sys/types.h>、<sys/socket.h>、<netinet/in.h>、<netdb.h>、<arpa/inet.h> 和 <unistd.h>。如果你有很多源文件

需要使用 `socket`，使用一个公共头文件包含所有 `socket` 头文件可以帮你保持代码整洁：

```
#ifndef MY_SOCKETS_H
#define MY_SOCKETS_H

#ifdef POSIX_OS_WIN32
#include <winsock2.h>
#else
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <arpa/inet.h>
#include <unistd.h>
#endif

#endif /* MY_SOCKETS_H */
```

还有其它一些很小的区别。例如 Unix 按传统的文件隐喻来使用 `socket`，因此 BSD Socket 使用 `close()` 函数来关闭 `socket`。但是在 Windows 中，Socket 并不是文件，你必须使用专门的 `closesocket()` API 来关闭 `socket`。类似地，BSD Socket 使用 `fcntl()` 来控制 `socket` 的属性（如阻塞或非阻塞），而 WinSock 则使用 Windows 专有的 `ioctlsocket()` 函数。

这两种 API 的错误处理也存在微妙的差别。Unix 下标准的错误处理方法是检查全局变量 `errno`。这个变量的值是标准错误常量如 `EWOULDBLOCK` 或 `EAGAIN`。而 WinSock 则使用特定的 `WSAGetLastError()` 函数，它返回一个类似于标准 Unix 的常量，值为 `WSAEWOULDBLOCK` 或 `WSAEAGAIN`。

你可以使用宏替换和类型定义来封装所有这些区别，如下面例子：

```
#if defined POSIX_OS_WIN32
#define CLOSESOCKET(s) closesocket(s)
#define SOCKERR() WSAGetLastError()
/* etc. etc. */
#else
typedef int SOCKET;
#define INVALID_SOCKET -1
#define CLOSESOCKET(s) close(s)
#define SOCKERR() errno
#define WSAEWOULDBLOCK EWOULDBLOCK
/* etc. etc. */
#endif
```

遗憾的是，你只能使用预处理来管理这些差别，这样就导致代码不够整洁，有时候还难以调试。不过聊胜于无，至少这样能工作，只需要编写最少量的平台特定代码。如果差别更大，可能就要采用更大范围的抽象了。

RPC 和 RMI

RPC 是实现于底层传输（如 BSD Socket）之上的高级函数调用抽象。远程过程调用被设计成透明的，对应用来说与标准函数调用在操作上是一样的。

特定 RPC 的实现可以采用相关的工具集，如 Sun 的 `rpcgen` 和开放软件基金会（OSF）的分布式计算环境（DCE）。这些工具集都能够处理 RPC 相关的底层细节。首先定义接口，然后生成合适的“`stub`”（空的功能实现）客户端和服务端代码，这些代码处理了 `marshal/unmarshal` 参数和广播自己的功能，并且对返回值进行相关处理。

XML-RPC（更加重量级和笨重的 SOAP 的前身）是一个 RPC 规范，使用 HTTP 作为传输层，使用 XML 作为参数编码。XML-RPC 的主要目标是通过工业标准协议（HTTP）来传输人类可读的编码格式（XML），理论上可以增强网络化应用的互操作性。

许多防火墙和路由器会阻止未知端口的通信，而“原始”RPC 实现通常都会使用这样的端口。例如典型的 RPC 实现可能需要 TCP 端口 9822 来通信（随意选择的端口），但防火墙会阻止管理员明确开放端口之外的任何通信。由于 HTTP 端口 80 被全世界用作 web 通信，这个端口的通信一般不会被阻止。XML-RPC（和 SOAP）就使用端口 80，因此通常使用 XML-RPC/SOAP 的程序会更加简单和容易，因为它们不需要对网络进行配置。

分布式对象

RPC 抽象了函数调用的概念，属于过程化编程范式。面向对象编程也提供一种类似 RPC 的能力：分布式对象。两个主要的分布式对象实现是微软的 Distributed Common Object Model (DCOM)；和 Object Management Group(OMG)的 Common Object Request Broker Architecture (CORBA)，CORBA 比 DCOM 更符合工业标准。

和 RPC 实现一样，分布式对象的实现也提供相关的工具，如接口定义语言（IDL）编译器；和代码生成器，用来生成 `stub`（客户端）和 `skeleton`（服务端）代码。

小结

现代计算机系统之间互相通信是不可避免的。网络化计算机系统面临着各种可移植的问题，特别是编程接口的区别，和交换原始数据等等。

第 11 章 操作系统

任何现代计算机系统运行的一个核心软件就是操作系统。由于操作系统既是所有计算机系统的核心，又是访问有限系统资源的中枢仲裁，它从根本上强烈地影响着软件如何编写。在本章中，我会讨论操作系统与可移植相关的基础细节和 API。

操作系统的演化

技术上来讲，操作系统是可选的。特殊系统的嵌入式软件通常就只包含原始程序，直接为 EEPROM 或闪存编写，再加上一个最小的启动引导程序，在机器启动时执行。

比无操作系统更进一步的是老式简单的操作系统，如 Microsoft MS-DOS 和 Digital CP/M。MS-DOS 只不过是对硬件很薄的一层封装。应用可以独占整个机器并直接访问许多资源。MS-DOS 不允许多个程序同时执行（多任务），除非使用高级的多任务软件进行协助，如 Quarterdesk、Microsoft Windows、或 GEM 的 DRI。

比 DOS 更高级一些的是提供某些有用特性（如简单多任务、GUI、底层设备访问控制）的操作系统。早期 Apple 的 Mac OS，Microsoft Windows 2.x（基于 MS-DOS）、Commodore 的 AmigaDos、和 Atari 的 TOS/GEM 都属于这一类操作系统。然而它们都有相同的一个特征，就是仍然允许应用直接访问任何内存和设备。

现代操作系统则整合了许多重要的特性，例如多用户支持、保护内存、和安全性。Microsoft Windows XP、Linux、FreeBSD、和 Apple Mac OS X 都拥有这些特性。

操作系统主要完成以下任务：

- 管理系统资源，如内存、文件描述符、和硬件设备。
- 实现安全协议
- 限制进程消耗资源和空间的能力
- 多任务，同时执行不同应用
- 提供简化、集中的应用编程接口，协助内存管理、输入、输出、进程控制、和其它应用需要的功能。

Hosted 和 Freestanding 环境

不是所有计算机都需要操作系统。那些没有操作系统的系统有时候也称为 freestanding 环境，程序启动和终止都是由具体的实现定义。此外，许多程序员依赖的特性（如 C/C++ 标准库）通常也不可用。嵌入式系统——如视频游戏控制台、便携式 MP3 播放器、及车载控制软件等——通常都是 freestanding 环境。

反过来 hosted 环境就是多数用户熟悉的，拥有操作系统的计算机系统。操作系统负责装载和执行程序，并提供各种系统服务。

当系统资源非常紧张，无法承受操作系统所需的开销时，系统设计师就可能选择 freestanding 环境。不过随着时间的进展，今天即使是最低端的设备也有足够的能力运行有限的操作系统。不久之前，PDA 和蜂窝电话都是 freestanding 环境，只有单一的内建应用。现在这些设备却都运行着操作系统。

操作系统可移植性悖论

操作系统的悖论之一，和可移植库（参考第 18 章）一样，操作系统在增强应用可移植性的同时，又阻碍了应用的可移植性。

第一个重要的跨平台微型计算机操作系统是 Gary Kildall 的 Control Program/Monitor (CP/M)，它的流行主要归功于其对各种微型计算机的可移植性，支持当时主流的 Intel 8080 和 Zilog Z80 芯片。

Intel 8080 芯片宣告了微型计算机时代的到来。无数厂商使用 8080 作为自己系统的核心芯片，这也导致了一个严重的问题，因为每个系统的体系架构都有稍微的区别。在某个 8080 系统中运行良好的应用，却不能在另一个类似的系统运行，可能是软盘驱动控制器或显示电路不一样等等各种原因。

CP/M 操作系统就是为了解决这些问题而编写，目的是创建统一的环境，使得不同基于 8080 微型计算机系统的应用都能运行。这项创新的关键在于要求计算机厂商提供 BIOS。CP/M 以统一的接口和方式访问每个厂商的 BIOS，由此来创建跨平台的 API。

然后应用按照 CP/M 的标准 API 编写（通过软中断），与任何特定计算机体系架构隔离开来。这样就产生了一些跨平台的杀手级应用，如 WordStar 和 dBase II，这些应用可以运行在各种类似但又存在区别的微型计算机中。如果没有 CP/M 这样的操作系统，这些应用就很难达到这样的高度。它们无法实现“兼容 CP/M 所有的机器”，只能自己解决个人计算机世界无穷无尽的差异。

获得这种可移植性的代价则是需要编写 CP/M，当然还必须使用 CP/M。因此与新操作系统（如后来的 MS-DOS）的可移植性在某种程度上变得更加困难。实际上 MS-DOS 为了降低 CP/M 向自身移植的痛苦，已经借用了 CP/M 的许多特性。

操作系统越复杂，应用就越依赖于操作系统提供的服务，如用户界面、内存分配和映射、安全和权限访问、声音、视频、和网络等等。如果应用直接使用操作系统的 API，那么迁移到新的操作系统时通常都需要重写全部代码。

当然本书的目标之一就是向你展示如何避免这种情况的发生。

内存

随着软件越来越大型和复杂，它们需要的内存经常超出计算机系统的可用物理内存。应用对于内存可以采取两种态度：假设内存容量非常小，或者假设内存容量接近无穷（仅受系统地址空间限制）。这个选择会严重影响到程序的可移植性。其它与可移植性相关的问题包括内存映射和内存保护。

内存限制

嵌入式系统和早期个人计算机的内存容量非常小，而且多数系统还将内存全部暴露给任何正在运行的程序。如果一个程序超过了可用内存，就会崩溃或运行失败，除非应用实现自己的分页系统。

现代操作系统给每个应用独立的地址空间，隐藏了讨厌的内存限制问题，使每个程序都以为自己拥有大容量内存，实际上数据被分页并实时交换到磁盘中。系统允许分配大于实际物理内存容量的内存，只要不是过度分配，一般对系统的影响不大。

内存映射

内存映射文件是另一个常见的可移植问题。内存映射把文件内容映射到内存中，使用指针操作数据，可以完全避免文件操作。这样做比起分配缓冲区然后读取文件内容，有很大的优点，包括更好的性能和更低的内存使用。实际上对于大型文件，把全部内容装载到内存中是不可行的。

例如地理信息系统数据或医学影像数据通常有几十 G 大小，超出了典型桌面 PC 完全装载的能力。

处理跨平台开发和内存映射有许多不同的方法：

需要内存映射

你可能必须使用内存映射，这时候你直接抽象它的实现。当你编写操作 16GB 文件的应用时可以采用内存映射。

假设内存映射可用

你可以假设内存映射总是可用，创建一个抽象，然后如果内存映射不可用时自己模拟一个。不过这通常并不可行，因为模拟内存映射可能会非常缓慢，并且占用大量资源。

优先考虑可移植性

你认为内存映射虽然很巧妙，但可移植性更加重要。因此你编写程序时假设内存映射不可用，这意味着你的软件访问文件会缓慢且烦琐。为了获得更好的可移植性，你的软件也增加了复杂度，因为你本来可以使用系统的内存管理机制。

内存保护

C 和 C++ 程序员新手最容易犯的错误之一就是访问越界内存，如超出数组末尾或解引用已释放指针。即使是有经验的程序员，偶尔也会出现非法内存访问。

早期 PC 操作系统不提供任何非法内存访问错误的保护，一个 bug 就能使整个系统崩溃，或者损坏重要的系统数据结构。Apple Mac OS、Commodore AmigaDOS、和 Microsoft MS-DOS 都允许应用随意访问任何内存。当时的 CPU 没有足够的逻辑进行内存保护，直到大量采用芯片内置内存管理单元（MMU）之后，保护模式的理念才真正被桌面操作系统采纳。

在这些早期操作系统中，应用不需要权限，就可以随意访问操作系统的核心内容，获取到有价值的系统信息。例如基于 MS-DOS 的 PC，应用可以查看甚至修改存放重要系统变量的原始内存。更常见的是，如果应用希望图形渲染更快，它就会直接写入到视频卡的帧缓冲区（16 位实模式 DOS 上的魔法数字内存地址 0xA000:000）。

当然，这些系统中的行为良好的程序，都非常努力不使用到不属于自己的内存。要在这些机器上安全地访问视频内存，一般通过 BIOS 中断 10h 端口，这是绝对安全有效的方法，但非常缓慢。

商业应用不能接受低劣的性能，你不能说“嘿，我们可能会慢一点，但我们以非常礼貌的方式访问你计算机的资源呢”。所以许多程序员都继续访问不属于自己的内存，尽可能地提高性能。结果就是大量程序覆盖堆栈或全局内存位置，轻易地使系统动摇或崩溃。

在今天的计算机中，这种行为是绝对不能容忍的。内存错误通常会导致一个弹出窗口，告诉你发生了不好的事情（通用保护错误、访问保护违例、总线错误、或其它名字）。但不会关闭整个系统，也不会污染内存数据。违例的程序会被终止，用户的其它程序则继续运行，不受任何干扰。

如果应用依赖于保护内存访问，那可移植性也会是一个问题，例如前面提到的直接视频

内存操作。通常我们不需要考虑保护内存，因为这种类型的访问可以很简单地进行抽象。

进程和线程

进程是操作系统的基本组成，通常对应于一个正在运行的程序（也不总是）。每个进程都封装了完整的状态，包括代码、数据、堆栈、当前指令指针、和寄存器。在受保护的操作系统中，每个进程的内存空间都是逻辑独立的，因此不会影响其它进程的状态（进程 A 不能直接修改进程 B 的变量）。

原始的操作系统只支持单个进程。例如 CP/M 和 DOS 只能理解当前进程的概念，多个进程不能同时运行。现代操作系统则允许多任务，能够同时运行多个进程。

进程控制和通讯函数

多数程序员并不太关心操作系统的进程模型。通常他们只知道自己正在运行一个进程，并且通过从 `main()` 返回，或者调用适当的函数（如 `exit()`）来退出进程。但是某些程序仍然需要执行进程控制和通讯函数，特别是进程启动。

例如网络服务器应用可能需要为每个连接用户创建一个新的进程；应用也可能需要启动另一个程序，可以使用 `system()`（ANSI C 标准）或 `spawn*()` 函数（常用但非标准的 C/C++ 扩展）来实现。不幸的是，

进程间通讯（IPC）

多线程

环境变量

异常处理

C 异常处理

C++异常处理

用户数据存储

Microsoft Windows 注册表

Linux 用户数据

OS X Preferences

安全和权限

应用安装

特权目录和数据

底层访问

小结

第 12 章 动态库

动态链接

动态装载

共享库的问题（DLL Hell）

版本问题

激增

Gnu LGPL

Windows DLL

Linux 共享对象

Mac OS X 框架、插件和 Bundle

框架

Bundle

插件

小结

第 13 章 文件系统

符号链接、快捷方式、别名

Windows LNK 文件

Unix 链接

目录规范

磁盘驱动器和卷标

路径分隔符和其它特殊字符

当前目录

路径长度

大小写

安全和访问权限

Macintosh 的古怪之处

文件属性

特殊目录

文本处理

C 运行时库和可移植文件访问

小结

第 14 章 伸缩性

好的算法=好的伸缩性

伸缩性的局限

小结

第 15 章 可移植性和数据

应用数据和资源文件

二进制文件

文本文件

XML

脚本语言作为数据文件

可移植图形

可移植声音

小结

第 16 章 国际化和本地化

字符串和 **Unicode**

货币

日期和时间

界面元素

键盘

小结

第 17 章 脚本语言

脚本语言的缺点

JavaScript/ECMAScript

Python

Lua

Ruby

小结

第 18 章 跨平台库和工具集

库

应用框架

QT

GTK+

FLTK

wxWidgets

小结

附录 A POSH

POSH 预定义符号

POSH 固定大小类型

POSH 工具函数和宏

附录 B 可移植性的规则

参考书目

索引