

# C++黑客指南

Steve Oualline 著

刘海涛 译

版权 2008, Steve Oualline.这部作品是在附录 F 中出现的知识共享许可下的许可。你可以免费:

- 分享-复制, 分发, 展示和执行作品。
- 整合-制作衍生作品。

在下列条件下:

- 归属: 你必须通过使用像在知识共享许可下的“使用经过 Steve Oualline 的许可”来标识本书的局部, 指出本书归属。(该归属不应以任何方式表明 Steve Oualline 认可你或你对该作品的使用。)
- 对于任何重用或分发, 你必须向其他人说明此工作的许可条款。最好的方法是使用指向网页的链接:  
<http://creativecommons.org/licenses/by/3.0/us/>.
- 如果你得到 Steve Oualline 的许可, 任何上面的条件可以被抛弃。
- 除了在该许可授权的整合权利, 本许可证中的任何内容均不会损害或限制作者的精神权利。

## 目录

序言.....	8
真实世界的黑客.....	9
第 1 章：通常的程序方案.....	10
方案 1：使代码消失.....	10
方案 2：让其他人写代码.....	11
方案 3：为了最大化保护，经常使用 <code>const</code> 关键字.....	12
方案 4：将大的参数列表放进结构体.....	13
方案 5：定义位.....	15
方案 6：谨慎使用位字段.....	16
方案 7：记录位图变量.....	18
方案 8：创建一个不能被复制的类.....	19
方案 9：构建自注册类.....	20
方案 10：分离接口和实现.....	23
方案 11：从 Linux 内核的链表函数中学习.....	24
第二章：安全方案.....	26
方案 12：消除副作用.....	26
方案 13：不要将赋值语句放进任何其他的语句。.....	27
方案 14：当可能的时候使用 <code>const</code> 而不是 <code>#define</code> .....	28
方案 15：如果你必须使用 <code>#define</code> ，将括号放在值的两边。.....	29
方案 16：尽可能使用内联函数而不是参数化的宏。.....	29
方案 17：如果你必须使用宏，在参数两边放置圆括号.....	30
方案 18：不要编写含糊的代码.....	30
方案 19：不要聪明的使用优先级规则.....	31
方案 20：包含属于你的头文件.....	32
方案 21：同步头文件和代码文件名.....	32
方案 22：永不信任用户输入.....	33
方案 23：不要使用 <code>gets</code> .....	35
方案 24：刷新调试.....	36
方案 25：使用断言保护数组访问.....	37
方案 26：使用模板创建安全的数组.....	39
方案 27：当什么都不做的时候，使其变得明显。.....	40
方案 28：将 <code>break</code> 或 <code>/*Fall Through*/</code> 作为每个 <code>case</code> 的结尾.....	41
方案 29：一个简单的不可能条件的断言语句.....	41
方案 30：在 <code>switch</code> 中总是检查不可能的情况.....	42
方案 31：创建可在编译时检查的模糊类型（句柄）.....	43
方案 32：当清零数组时使用 <code>sizeof</code> .....	44
方案 33：在 <code>memset</code> 中使用 <code>sizeof(var)</code> 而不是 <code>sizeof(type)</code> .....	45
方案 34：避免重用清零的指针.....	46
方案 35：使用 <code>strncpy</code> 替换 <code>strcpy</code> 避免缓冲区溢出.....	47
方案 36：为了安全性使用 <code>strncat</code> 而不是 <code>strcat</code> .....	48
方案 37：使用 <code>snprintf</code> 构建字符串.....	49
方案 38：不要用人为了限制设计.....	49
方案 39：总是检查自赋值.....	51

方案 40: 使用哨兵保护类的完整性.....	52
方案 41: 使用 <code>valgrind</code> 解决内存问题.....	53
方案 42: 发现未初始化的变量.....	55
方案 29: <code>valgrind</code> 的发音.....	56
方案 43: 使用 <code>ElectricFence</code> 定位指针问题.....	56
方案 44: 处理复杂的函数和指针声明.....	57
第 3 章: 文件方案.....	59
方案 45: 只要可行, 创建文本文件而不是二进制文件.....	59
方案 46: 使用魔法字符串识别文件类型.....	61
方案 47: 为二进制文件使用魔法数字.....	61
方案 48: 通过魔法数进行自动字节排序.....	62
方案 49: 编写可移植的二进制文件.....	63
方案 50: 使你的二进制文件可扩展.....	64
方案 51: 使用魔法数保护二进制文件记录.....	65
方案 52: 知道什么时候使用 <code>_exit</code> .....	67
第 4 章: 调试方案.....	68
方案 53: 使用特别的字符标记临时的调试消息.....	68
方案 54: 使用编辑器分析输出日志.....	68
方案 55: 灵活的日志.....	69
方案 56: 使用信号打开和关闭调试.....	70
方案 57: 使用一个信号文件打开和关闭调试.....	71
方案 58: 发生错误时自动启动调试器.....	71
方案 59: 启动调试器使断言失败.....	77
方案 60: 在正确的地方暂停程序.....	78
第 5 章: 注释和导航方案.....	80
方案 61: 创建带有注释的标题.....	80
方案 62: 强调句子中的单词.....	81
方案 63: 在注释中放置图.....	81
方案 64: 提供用户文档.....	82
方案 65: 文档化 API.....	83
方案 66: 使用 <code>Linux</code> 交叉引用来导航大型代码项目.....	85
第 6 章: 预处理方案.....	89
方案 67: 使用预处理器产生名字列表.....	89
方案 68: 自动创建单词列表.....	90
方案 69: 保护头文件的双重包含.....	91
方案 70: 在 <code>do/while</code> 中封闭多行的宏.....	91
方案 71: 使用 <code>#if 0</code> 移除代码.....	92
方案 72: 使用 <code>#ifndef</code> <code>QQQ</code> 标识临时代码.....	93
方案 73: 在函数上, 使用 <code>#ifdef</code> , 而不是函数调用, 来消除多余的 <code>#ifdef</code> .....	93
方案 74: 创建代码从函数体中消除 <code>#ifdef</code> 语句.....	94
第 7 章: 构建方案.....	97
方案 75: 不要在没有验证的情况下使用任何“众所周知”的加速.....	97
方案 76: 在双核处理器机器上使用 <code>gmake -j</code> 加速.....	99
方案 77: 通过使用 <code>ccache</code> 编码重复编译.....	101

方案 78: 不改变你的 Makefiles 的情况下使用 ccache.....	102
方案 79: 使用 distcc 分配工作负载.....	103
第 8 章: 优化方案.....	104
方案 80: 除非你真的确实需要, 否则不要优化.....	104
方案 81: 使用性能分析器查找要优化的地方.....	104
方案 82: 避免格式化的输出函数.....	105
方案 83: 使用 ++x 而不是 x++, 因为它更快.....	106
方案 84: 通过使用 C 的 I/O API 优化 I/O 而不是 C++ 的 I/O API.....	107
方案 85: 使用一个本地缓存避免重新计算一样的结果.....	109
方案 86: 使用自定义 new/delete 来加速动态存储分配.....	110
对抗方案 87: 不必要的建立一个自定义的 new/delete.....	112
对抗方案 88: 使用移位乘以或除以 2 的倍数.....	112
方案 89: 使用静态内联而不是内联来节省空间.....	113
方案 90: 当你没有一个浮点数处理器的时候, 使用 double 加快操作而不是 float.....	114
方案 91: 告诉编译器打破标准, 当做计算时, 强制编译器将 float 作为 float.....	115
方案 92: 定点运算.....	115
方案 93: 验证优化代码与未优化代码版本.....	120
案例学习: 优化位到字节.....	120
第 9 章: g++ 方案.....	125
方案 94: 特定结构初始化.....	125
方案 95: 检查参数列表的 printf 样式.....	126
方案 96: 打包结构体.....	126
方案 97: 创建一个函数, 返回值不应该被忽略.....	126
方案 98: 建立永远不返回的函数.....	127
方案 99: 使用 GCC 堆内存检查函数定位错误.....	129
方案 100: 追踪内存使用.....	130
方案 101: 产生一个回溯.....	132
第 10 章: 对抗方案.....	135
对抗方案 102: 变量声明使用 "#define extern".....	135
对抗方案 103: 使用, (逗号) 加入语句.....	137
对抗方案 104: if (strcmp(a,b)).....	138
对抗方案 105: if (ptr).....	139
对抗方案 106: "while ((ch = getch()) != EOF)" 方案.....	140
对抗方案 107: 使用 #define 增强 C++ 语法.....	141
对抗方案 108: 使用 BEGIN 和 END 而不是 { 和 }.....	141
对抗方案 109: 变量参数列表.....	142
对抗方案 110: 不透明句柄.....	143
对抗方案 111: 微软 (匈牙利) 表示法.....	144
第 11 章: 嵌入式程序方案.....	146
方案 112: 总是验证硬件规格.....	147
方案 113: 使用可以指定整数宽度的可移植类型.....	148
方案 114: 验证结构体大小.....	148
方案 115: 当定义硬件接口的时候验证偏移量.....	150

方案 116: 打包结构体来消除隐藏的填充.....	150
方案 117: 理解关键词 <code>volatile</code> 做了什么和如何使用它.....	151
方案 118: 理解优化器可以为你做什么。.....	152
方案 119: 在嵌入式程序中, 尝试没有暂停地处理错误.....	155
方案 120: 检测饥饿.....	157
第 12 章: Vim 编辑方案.....	159
方案 121: 打开语法着色.....	160
方案 122: 使用 Vim 的内部 make 系统.....	160
方案 123: 自动缩进代码.....	162
方案 124: 缩进错在的代码块.....	163
方案 125: 使用标签导航代码.....	164
方案 126: 你需要定位你只知道部分名字的过程.....	166
方案 127: 使用 <code>:vimgrep</code> 搜索变量或函数.....	168
方案 128: 查看大型函数的逻辑.....	169
方案 129: 使用 Vim 查看日志文件.....	170
第 13 章: 聪明但是无用.....	172
方案 130: 翻转变量 1 和 2.....	172
方案 131: 不使用临时变量交换两个数.....	173
方案 132: 不用临时变量, 在一个字符串中翻转单词.....	174
方案 133: 使用单个指针实现一个双向链表.....	176
方案 134: 不使用一个锁, 访问共享内存.....	177
方案 135: 回答面向对象的挑战.....	179
附录 A: 黑客名言.....	181
Grace Hopper.....	181
Linus Torvalds.....	182
附录 B: 你知道如果.....你是一个黑客.....	183
附录 C: 黑客罪行.....	185
使用字母 O, l, I 作为变量名.....	185
不分享你的工作.....	185
没有注释.....	185
复制代码(通过剪切和粘贴编程).....	186
附录 D: 为黑客准备的开源工具.....	187
ctags - 函数索引系统.....	187
dxygen.....	187
FlawFinder.....	187
gcc - GUN C 和 C++编译器套件.....	187
lxr.....	187
Perl (对于 perldoc 和相关工具) - 文档系统.....	187
valgrind (内存检查工具).....	188
Vim (Vi 升级版).....	188
附录 E: 安全设计模式.....	189
1. 消除副作用。将++和--放在它们自己的行.....	189
2. 不要将赋值语句放在其它语句中.....	189
3. 定义常量.....	189

4.使用内联函数而不是参数化的宏.....	189
5.使用{}消除含糊的代码.....	189
6.让你做的一切变得明显，即使什么也不做.....	190
7.在一个 switch 中总是检查默认的 case.....	190
8.优先级规则.....	190
9.头文件.....	190
10.检查用户输入.....	191
11.数组访问.....	191
12.复制一个字符串.....	191
13.字符串连接.....	191
14.复制内存.....	191
15.清零内存.....	191
16.查找一个数组中的元素数量.....	191
17.使用 gets.....	191
18.使用 fgets.....	192
19.当建立不透明的类型时，使它们可以被编译器检查.....	192
20.在 delete/free 后清零指针，避免重用.....	192
21. 总是检查自复制.....	192
22. 使用 snprintf 建立字符串.....	192
Appendix F: Creative Commons License.....	193
License.....	193

## 序言

最初术语黑客指那些使用很少的资源和很多的技巧，能够做不可能事情的人。基本的定义是“用斧子制作精美家具的人”。黑客知晓计算机的内部和外在外在，可以用计算机执行酷炫、优雅和不可能的壮举。现在，这个术语已经退化为入侵计算机的人，但是，在本书中我们使用黑客本身的可敬形式。我第一次了解真正的黑客是我上大学时加入午夜计算机俱乐部。这不是一个官方的俱乐部，只是一组人午夜在 PDP-8 实验室中闲逛，编程和讨论计算机。

我记得有个同伴从 Radio Shack 那里拿走了 10 美元的零件，并制造了一个小黑盒子，他可以用示波器对齐 DECTaple 驱动。在当时 DEC 需要一个价值 35000 美元的定制机器做同样的事情。

也有一些人喜欢 PDP-8 上面编程来播放音乐。这种很难做到，因为机器没有声卡。但是有人发现，如果将收音机放在机器附近，扬声器就会听到干扰声。使用系统播放一段时间后，有人发现如何使用干扰产生音调，因此 MUSIC-8 编程系统诞生了。即使系统没有声卡，并不能阻止黑客从中获取声音。这说明了伟大黑客的一个属性，完全使用紧缺的资源完成“不可能”的事情。

当我的一些朋友使用汇编语言时，我第一次真正的黑客攻击出现了。他们的工作是编写一个函数计算矩阵乘法运算。我向他们展示了如何使用 PDP-10 的能力进行双重间接索引寻址，这减少了访问矩阵的次数，从每个乘法矩阵的元素到另外一个乘法矩阵的元素。

教汇编课程的教授认为唯一使用汇编编程的理由是速度，所以他定时布置作业，将结果与他的“最佳”方案比较，每隔一段时间就会找到一个更快一点的程序，但他是一个优秀的程序员，所以很少有人打败他。

除了我朋友的矩阵乘法分配。最慢的速度比老师的“最佳”方案快十倍。最快的方法是如此快，以至于打破了他正在使用的计时工具。（在看到这个非常奇怪的代码后，他作为一个教授做了件非同寻常的事：他把我的朋友请到课堂前面，给他们粉笔，让他们教自己。）

什么造就优秀的黑客？它涉及遍历，围绕或通过机器，编译器，管理，安全



或其他方面施加的限制条件。

真正的黑客发展技巧和技术，旨在克服他们面临的障碍，提高他们使用的系统的质量。这些才是真正的黑客。

本书包含四十多年编程经验中产生的方案集合。在这里你可以发现各种各样的方案程序，使你的程序更可靠，更可读，更容易调试。真正的黑客传统，是观察什么有效和如何有效，改进系统，然后传递信息的结果。

## 真实世界的黑客

我是一个真实世界的程序员，所以这本书处理真实世界的程序。例如，有一些 C 风格字符串（`char*`）的讨论和反馈。这激怒了一些 C++ 的纯粹主义者，他们相信你应该在程序中只使用 C++ 字符串（`std::string`）。这可能是正确的，但在真实的世界中依然有很多使用 C 风格字符串的 C++ 程序。任何正在工作的黑客必须很好的处理它们。

理想主义是好的，但我为了生存工作，这本书基于真实世界使用的程序，不是在理想世界中找到的。所以对于所有现实世界的黑客，我贡献了这本书。

## 第 1 章：通常的程序方案

C++不是一个完美的语言。因此有时你必须围绕语言强加的限制进行编程。在本章中我们将介绍一些简单，常见的方案，你可以使用这些方案，使程序更简单和易读。

### 方案 1：使代码消失

**问题：**写代码浪费时间并产生风险。

**方案：**不写代码。所有你不写的代码是最容易生成，调试和维护。零行的程序使你唯一确定没有错误的。

一个出色的黑客知道如何编写优秀的代码。一个出色的黑客算出如何一点也不写代码。

当你面对一个问题，坐下来思考它。一些大而复杂的问题，实际上只是由于迷糊的用户和雄心勃勃的要求而隐藏的小而简单的问题。你的工作是找到一个简洁的解决方案，而不是写代码来处理大的混乱。

让我给你提供一个例子：我被要求写一个证书管理器，允许拥有一个证书密钥的用户运行程序。有两种证书的类型，一种是在一个准确的日期到期，另一种是永远不过期。

通常有人会设计一些额外的逻辑代码来处理两种类型的证书。我重写了要求，删除了从未过期证书的要求。取而代之的是我们会给试用顾客一个 60-90 天的使用证书，给购买程序证书顾客的到期时间为 2038<sup>3</sup>。

从而，我们两种类型的证书变成一种。所有关于永久证书的代码消失了，从未被编写。

另一个例子是我不得不为一个公司编写一个新的汇报系统。他们已有的系统使用脚本语言，因此比较慢而且受限制。那时，他们有 37 种类型的报告。使用 37 块代码来生成那 37 种报告。

我的工作是将那 37 块代码从一种语言翻译成另一种。我坐下来研究正在做的事情，而不仅仅是做我被要求的事情。当我的老板问我为什么没有编码时，我告诉他我在思考，这是我认为不可缺少的一步。

事实证明，我能够将 37 种不同的报告提炼成 3 种报告类型。所有 37 种报告可以由这三种类型和一些参数生成。结果，完成工作所需的代码量缩减了至少 10 倍。

记住你从未编写的代码是最快的代码，也是你产生的错误最少的代码。攻击一些存在之外的东西是最高级的攻击形式。

### 产生代码行

我曾经负责更新使用 Perl 编写的基于 Web 的大型报表系统。这时，管理层决定通过测量编写的代码行数来了解程序员是如何生成的。

由于 Perl 语法的“设计”，坏程序员和好程序员的区别被放大。第一周，我清理了明显低效率的代码，并删除了大量冗余和无用的代码。那周我的得分大约是-1700 行。所以即使我添加了很多注释和一些新功能，程序变得更小了。

接下来的几周，我继续减少程序的规模。当我拿出旧的样式，“调用函数，错误检查，传递错误调用更改”逻辑，将其替换为基于异常的错误处理时，发生了大的变化。这一变化是我们失去了 5000 行代码。

我的经理问我为什么他们应该付我很多的钱，因为我的#代码行 生产/周是负的。我告诉他们，这正是他们为我付出巨大花费的原因。因为它需要一个非常出色的程序员才能在负代码行中产生新的功能。

## 方案 2：让其他人写代码

**问题：**写代码是慢的。写优秀的代码更慢，即使你编写了优秀的代码，你需要花时间调试它。

**方案：**建立在你所做的工作之上。

下一步什么也不做，让其他人做事是最简单的编程方式。那里有成千上万的工具，程序和其他软件。其中一个可能会帮助你完成你的工作。如果没有，它可

能几乎能帮助你完成工作，所以你不能不做的事情就是下载修改它，然后使用它。

一个很优秀的开源软件的源是 <http://www.freshmeat.net>。它是一个基于 Web 的数据库，包含很多软件项目的索引。

现在，如果你使用开源软件作为你工作的基础，那么你就有义务向社区回馈你创建的任何改进的功能。这可以让其他人使用你的工作作为他们程序的基础。

应该指出的是，一些不熟悉开源工作原理的人对它有点害怕。他们往往是商人，无法理解编写开源软件怎样赚钱。秘密是开源不是由想赚钱的人编写的，而是由想要有效软件的人编写的。

如果你想要一个程序工作，一个“创造”它的最简单方法就是以其他人的工作作为出发点。这就是黑客精神：你不只是复制别人做过的事情，而是推动艺术的发展。

### 方案 3：为了最大化保护，经常使用 **const** 关键字

**问题：**你传递一个字符串（`char*`）给函数，由于有人意外地改写指针，因此代码变得混淆。

**方案：**告诉编译器指针不能被改变。

这种方案使用了一个比较难理解的 C++ 语言的概念，就是 **const** 和指针。

我们以下面的声明开始：

```
const char* ptr_a;
```

问题是“**const** 修饰了什么？”它影响指针或者影响指针指向的数据？

这个例子中，**const** 告诉编译器字符数据是常量。指针本身可以被重新分配。



```
const char* ptr_a;
```


这意味着我们可以重新分配指针：

```
ptr_a = "A New Value";
```

但是你不能修改指针指向的数据：

```
*ptr_a = 'x'; // ILLEGAL
```

现在考虑另一种声明：



```
char* const ptr_b;
```

这个例子中指针被 **const** 修饰。指向的数据则没有。

所以我们可以修改被指向的数据：

```
*ptr_b = 'x'; // Legal
```

但是我们不能修改指针：

```
*ptr_b = 'A new string'; // ILLEGAL
```

当然，显而易见的声明是指针和数据都是常量：

```
const char* const ptr_c;
```

现在回到函数调用。如果我们期望常量数据，在函数参数中指定它：

```
void display_string(const char* const the_string);
```

现在，任何修改字符串的尝试会导致一个编译时错误。编译时错误相比运行时错误更容易定位和修复。

### 常量内存攻击

从语法上，**const** 关键字的位置修饰指针或字符并不明显。但是有一个简单的助记技巧可以帮助你记住哪个是哪个。

关键字 **const** 修饰离它最近的元素。例如：

```
const char* ptr_s;
```

这个例子中 **const** 相比于\*离 char 更近，因此数据（char）是常量。

另一个例子：

```
char* const ptr_t;
```

**const** 相比于 char 离\*更近，因此指针是常量，而不是数据（char）。

## 方案 4：将大的参数列表放进结构体

**问题：**带有很多参数的函数很难处理。参数可以很容易的被整合。

尽管你传递给函数的参数数量并没有限制，实践中，超过 6 个参数使得函数难以使用。考虑下面的用来绘制矩形的函数调用：

```
draw_rectangle(  
    x1, y1, x2, y2, // The corners of the rectangle  
    width;          // Width of the line for the rectangle  
    COLOR_BLUE,     // Line color  
    COLOR_PINK,     // Fill color  
    SOLID_FILL,     // Fill type  
    ABOVE_ALL,      // Stacking order  
    "Times",        // Font for label
```

```
    10,          // Point size for label
    "Start",     // Label
};
```

这段代码在等待一个意外发生。忘记一个参数，代码将不能编译。更糟糕的是，调换两个参数，代码也许可以编译，但是会绘制错误的东西。

**方案：**使用一个结构体传递一堆参数。

让我们看看对于矩形函数是如何工作的。

```
// Define how to draw the rectangle
struct draw_params my_draw_style;
my_rect.width = width;
my_rect.line_color = COLOR_BLUE;
my_rect.fill_color = COLOR_PINK;
my_rect.fill = SOLID_FILL;
my_rect.stack = ABOVE_ALL;
my_rect.label_font = "Times";
my_rect.label_size = 10;
my_rect.label = "Start";

draw_rectangle(x1, y1, x2, y2, &my_draw_style);
```

现在，不是通过位置传递参数，而是通过名称传递参数。这使代码更可靠。

例如，不再需要记住线条颜色是第一个，还是填充颜色是第一个。当把它写成：

```
my_rect.line_color = COLOR_BLUE;
my_rect.fill_color = COLOR_PINK;
```

很明显哪个是线条颜色，哪个是填充颜色。

**方案中的方案：**结构体 `draw_params` 可能不仅仅用于绘制矩形，也可以绘制其他的形状。例如：

```
draw_rectangle(x1, y1, x2, y2, &my_rect);
draw_circle(x3, y3, radius, &my_rect);
```

使所有参数值默认为零是个好主意。这种方式，可以使用下面的声明将所有参数设置为默认值：

```
memset(&my_rect, '\0', sizeof(my_rect));
```

这让你从不得不设置结构体每个项的值中解放出来。例如，使用默认宽度、填充、标签绘制一个红色的结构体，使用下面的代码：

```
memset(&my_rect, '\0', sizeof(my_rect));
my_rect.line_color = COLOR_RED;
draw_rectangle(x1, y1, x2, y2, &my_rect);
```

如果你使用 C++，结构体 `draw_params` 可以写成类。对于用户这个类可以提

供内部一致性检查。（“将标签设置为‘foo’，点大小为 0 对我来说是没有意义的。”）

方案 5: 定义位

问题：你需要为“第 5 位”定义一个常量。

经常需要程序员从一个字节中访问各种位。下面是 DLT 磁带机硬件手册的示意图：

Bit Byte	7	6	5	4	3	2	1	0
0-1	(MSB) Parameter Code (LSB)							
2	DU	DS	TSD	ETC	TMC	Rsvd	LP	
3	Parameter Length							
4-7	(MSB) Parameter Value (LSB)							

你需要定义一个常量访问选项字节（字节 2）中的每个位。一种方法是为每个组件定义一个十六进制常量。

```
// Bad Code
const int LOG_FLAG_DU = 0x80;    // Disable update
const int LOG_FLAG_DS = 0x40;    // Disable save
const int LOG_FLAG_TSD = 0x20;   // Target save disabled
const int LOG_FLAG_ETC = 0x10;   // Enable thres. comp.
```

问题在于 0x40 与第 6 位之间是不明显的。很容易使位混淆。

```
// Good code
const int LOG_FLAG_DU = 1 << 7;  // Disable update
const int LOG_FLAG_DS = 1 << 6;  // Disable save
const int LOG_FLAG_TSD = 1 << 5; // Target save disabled
const int LOG_FLAG_ETC = 1 << 4; // Enable thres. comp.
```

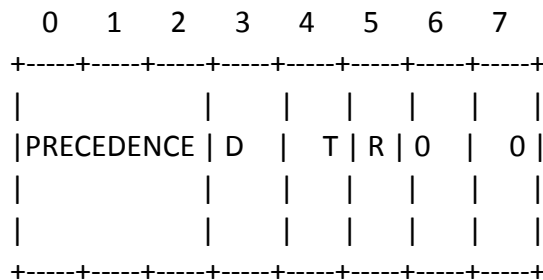
现在，很容易看出(1<<4)是第 4 位。

警告：确保你知道以什么作为结尾。在之前的例子中 0 位是最低有效位（最右边的位）。

但是一些人将第 0 位作为最高有效位（最左边的位）。例如互联网规范 RFC791 定义“服务类型”字段（最初拼写错误）：

位 0-2: 优先权

位 3: 0 = 正常延迟, 1 = 低延迟。  
 位 4: 0 = 正常吞吐量, 1 = 高吞吐量。  
 位 5: 0 = 正常可靠性, 1 = 高可靠性。  
 位 6-7: 保留供将来使用。



[原文拼写错误。]

这种方式中, 我们依然使用左移定义位。只是我们从常量 `0x80` 开始, 向右移动。

例如:

```
// Good code
// Delay flag
const unsigned int IPTOS_LOWDELAY = 0x80 >> 3;

// Throughput flag
const unsigned int IPTOS_THROUGHPUT = 0x80 >> 4;

// Reliability flag
const unsigned int IPTOS_RELIABILITY = 0x80 >> 5;
```

警告: 当定义常量时确保使用的是 **unsigned int** 而不是 **[signed] int**。有符号的整数会使得数据中的符号位被复制, 从而导致意外结果。

琐事: 下面这些是在 Linux 标准头文件 `/usr/include/netinet/ip.h` 中的定义:

```
#define IPTOS_LOWDELAY 0x10
#define IPTOS_THROUGHPUT 0x08
#define IPTOS_RELIABILITY 0x04
```

你可能注意到他们没有使用这种方案。不看前面的几页你可以告诉我 `IPTOS_LOWDELAY` 代表哪一位吗?

## 方案 6: 谨慎使用位字段

问题: 你想使用位字段, 以便你不需要以困难的方式测试, 设置和清除位。

例如:

```
struct timestamp {
    unsigned int flags:4;
```



```
    unsigned int overflow:4;
};
```

**方案：**一个优秀的黑客谨慎对待位字段。使用它们有几个问题。包括：

1. 顺序不能保证。
2. 打包不能保证。
3. 你真的知道什么可以，什么不可以放在一个字段中。当处理一个位宽的字段时尤其如此，正如下面看到的那样。

C++标准不保证位字段的位最终位于何处。在前面的例子中，编译器可能：

1. 分配字段 `flags` 到高位，`overflow` 到低位。
2. 分配字段 `overflow` 到高位，`flags` 到低位。
3. 忽略位字段规范，将 `overflow` 和 `flags` 分配到不同的字节。

Linux 操作系统假设你正在使用 GCC 编译器，将多个字段打包成一个字节。

但是这些字段的顺序取决于机器的字节顺序。

这导致头文件有些奇怪：

```
struct timestamp {
    #if __BYTE_ORDER == __LITTLE_ENDIAN
        unsigned int flags:4;
        unsigned int overflow:4;
    #elif __BYTE_ORDER == __BIG_ENDIAN
        unsigned int overflow:4;
        unsigned int flags:4;
    #else
        #error "Please fix <bits/endian.h>"
    #endif
};
```

还有一个问题，你不得不注意位字段。看下面的代码：

```
struct flag_set {
    int big:1;
    int bigger:1;
    int biggest:1;
};
// ...
flag_set the_set;
the_set.big = 1;
std::cout << "big flag is " << the_set.big << std::endl;
```

这个程序的输出不是“big flag is 1”。怎么回事？

问题是我们有一个有符号整数的位。在有符号整数中第一位是符号位。如果

第一位数字是负数。

那么，包含单个位的有符号数只能取两个值，0 和-1。

所以表达式：

```
the_set.big = 1;
```

将符号位设置为 1，使得数为负数，设置字段为-1。

## 方案 7：记录位图变量

**问题：**位图数据很常见，但是比较复杂，难以使用。例如数据声明确实应该被注释，但是，不幸的是，对于一个程序员来说，在程序内部绘制数字或表格是没有办法的。可以使用外部文档进行记录，但是它们很容易过时或丢失。

理想地，文档应该作为注释嵌入在程序中。毕竟，很难丢失 1/2 的程序文件。然而，在编写程序时，编写文档习惯使用的所有漂亮的文字处理器绘图功能都会丢失。相反，你必须使用用于编写程序的等宽单字体来创作。

这是一个使用 ASCII 艺术字，通过绘制线条来描述位：

```
/*
 * LOG_SELECT parameters byte
 *
 *  +----- DU  (Disable Update)
 *  |+----- DS  (Disable Save)
 *  ||+----- TSD (Target Save Disable)
 *  |||+----- ETC (Enable Threshold Compression)
 *  ||||+----- TMC (Threshold Met Criteria)
 *  |||||+---- Rsvd (Reserved)
 *  |||||+--- LP  (List Parameter)
 *  76543210
 */
```

另外一种方法是使用 RFC 文档中所使用的。下面是从 RFC 791 中摘录的注释：

```
/  *bits 0-2:  Precedence.
 *Bit   3:   0 = Normal Delay,          1 = Low Delay.
 *Bit   4:   0 = Normal Throughput,     1 = High Throughput.
 *Bit   5:   0 = Normal Reliability,     1 = High Reliability.
 *Bit 6-7:  Reserved for Future Use.

 *0   1   2   3   4   5   6   7
 *+---+---+---+---+---+---+---+
 *|           |   |   |   |   |
 *|PRECEDENCE | D | T | R | O | 0 |
```

```
* |           |           |           |           |
* |           |           |           |           |
* +-----+-----+-----+-----+-----+-----+
*/
```

[原文拼写错误。]

像这样从标准中复制，具有忠实再现标准中信息的额外优点，从而产生良好的文档。由于你做的只是复制粘贴，因此工作量很小。

复制和粘贴的唯一缺陷是原文中的任何缺点，例如上面的拼写错误也会被复制。

## 方案 8：创建一个不能被复制的类

**问题：**你已经创建了一个复杂的类，但是不想为这个类构建拷贝构造或赋值运算符。另外没有人应该拷贝这个类的任何实例。

一种解决方法是构建一个拷贝构造函数，当它被调用的时候就会退出程序。

```
// Works, but not optimal
class no_copy {
    // ...
public:
    no_copy(const no_copy&) {
        std::cerr <<
            "ERROR: no_copy(no_copy) called" <<
            std::endl;
        abort();
    }
};
```

这种有效，但是最好在编译时检测出问题，而不是运行时检测。

**方案：**声明拷贝构造和赋值运算符是私有的。

```
class no_copy {
    // ...
private:
    no_copy(const no_copy&);
    no_copy& operator = (no_copy&);
};
```

现在，当你试着使用拷贝构造时会发生什么：

```
no_copy a_var;

// This will result in a compile error
no_copy b_var(a_var);
```

结果是错误消息：

```
no_copy.cc:5: error: `no_copy::no_copy(const no_copy&)' is private
no_copy.cc:10: error: within this context
```

现在，实际上，在实践中调用拷贝构造函数可能没有这么明显。可能发生的是通过参数传递或一些其他隐藏的代码意外的调用拷贝构造。但是，好的事情是现在你调用拷贝构造，你会在编译时发现问题而不是运行时发现。

## 方案 9：构建自注册类

问题：你正在编写一个有着数百个命令的，例如图像编辑器的程序。你如何构建一个管理命令列表。

方案：自注册类。

解决方法是使得类的每个实例自己注册自己。

在本例中，我们将定义一个 `cmd` 类，它为编辑器定义一个命令。使用它作为每个命令的基类，生成派生类。派生类负责定义执行实际工作的 `do_it` 函数。

当一个命令被构建，类 `cmd` 会注册它。

```
cmd(const char* const i_name):name(i_name) {
    do_register();
}
```

自然的，当它被销毁时应该注销：

```
virtual ~cmd() {
    unregister();
}
```

记住这是一个基类，所以我们让析构函数是虚的。

作为黑客，我们希望从用户那里隐藏尽可能多的信息。这种条件下，我们将完全保留命令列表以及 `do_register` 和 `unregister` 函数。

首先，命令列表作为静态成员变量声明：

```
private:
    static std::set<class cmd*> cmd_set;
```

很多人不理解声明一个成员变量为静态的意味着什么。当一个类的实例被创建时，一个正常成员变量的实例也被创建。换句话说 `a_var.member` 与 `b_var.member` 是明显不同的变量。

但是静态成员是不同的。对于静态成员变量，一个时期内只有一个变量的实例被创建。它被类的所有实例共享。所以换句话说 `x_cmd.cmd_set` 与

y\_cmd.cmd\_set 是相同的。你也可以在没有类实例的情况下引用变量：cmd::cmd\_set。（如果我们没有将其声明为**私有的**。）

现在，让我们看看 do\_register 函数：

```
void do_register() {  
    cmd_set.insert(this);  
}
```

这只是将当前类的指针插入到列表中。

注销函数同样简单：

```
void unregister() {  
    cmd_set.erase(this);  
}
```

现在有趣的来了，我们调用函数执行一个命令。它被声明为**静态**成员函数，所以我们可以没有 cmd 变量的情况下调用它。

```
static void do_cmd(const char* const cmd_name) {
```

因为它是**静态**的，所以可以使用像这样的语句调用它：

```
cmd::do_cmd("copy");
```

**注意：**静态成员函数只能访问**静态**成员变量和全局变量。

函数的主体非常简单，只需遍历命令集，直至找到匹配的那个，然后调用 do\_it 成员函数。

```
static void do_cmd(const char* const cmd_name) {  
    std::set<class cmd*>::iterator cur_cmd;  
  
    for (cur_cmd = cmd_set.begin();  
         cur_cmd != cmd_set.end();  
         ++cur_cmd) {  
        if (strcmp((*cur_cmd)->name,  
                  cmd_name) == 0) {  
            (*cur_cmd)->do_it();  
            return;  
        }  
    }  
    throw(unknown_cmd(cmd_name));  
}
```

关于这个系统的一个更有趣的事情是，当你使用类 cmd 来声明一个全局变量时会发生什么。这种情况下，命令在 main 调用之前注册。

换句话说，C++在启动每个程序之前会通过程序查找全局变量，调用它们的

构造函数。当这样做的时候，你必须谨慎。无法保证类的初始化顺序。

基本上，你不想创建任何全局变量，其构造函数取决于正在注册的命令。

完整的类列在下面：

```
#include <set>
class cmd {
private:
    static std::set<class cmd*> cmd_set;

    const char* const name;
private:
    void do_register() {
        cmd_set.erase(this);
    }
    virtual void do_it(void) = 0;
public:
    cmd(const char* const i_name):name(i_name) {
        do_register();
    }
    virtual ~cmd() {
        unregister();
    }
public:
    static void do_cmd(const char* const cmd_name) {
        std::set<class cmd*>::iterator cur_cmd;

        for (cur_cmd = cmd_set.begin();
             cur_cmd != cmd_set.end();
             ++cur_cmd) {
            if (strcmp((*cur_cmd)->name,
                      cmd_name) == 0) {
                (*cur_cmd)->do_it();
                return;
            }
        }
        throw(unknown_cmd(cmd_name));
    }
};
```

聪明的黑客可能注意到，可以使用 `std::map` 来存放命令列表。毕竟 `std::map` 采用关键字和值对，消除 `do_cmd` 中的查找循环。

这种方案也可以使用 **`std::map`**，但是需要额外的语法来使其有效。所以虽然类不是最有效的代码，但是它是书中最明智的。

## 方案 10：分离接口和实现

**问题：**当你定义类时，C++强制你暴露实现细节。

让我们查看一个字典类的典型类定义。该类定义键和值的单词对列表。然后可以使用键查找值。

```
class dictionary {
private:
    // Lots of stuff that the user doesn't need to
    // worry about.
    // (And this is the problem)
public:
    // ... usual constructor / destructor stuff
    void add_pair(const std::string& key,
                  const std::string& value);
    const std::string& lookup(const std::string& key);
};
```

问题是，这样使得任何使用这个类的人都可以访问该类的一些私有实现细节。

**方案：**通过使用一个实现类隐藏实现细节。

```
class dictionary {
private:
    dictionary_implementation* implementation;
public:
```

由于我们不需要在头文件中定义 `dictionary_implementation`，这样有效的隐藏了实现，分离了接口和实现。

**方案的方案：**实现字典有几种方案。例如，如果我们处理大约 10-100 个单词，可以使用数组实现字典。

对于 100-1000 个单词，可以使用动态链表。对于超过 100000 的，代码可以使用外部数据库。

但是，以这种方式定义字典的好处是，类可以随着条件的变化而改变实现细节。例如，类可以以数组开始。当条目增加到超过 100 时可以切换到基于链表的实现。

```
void dictionary:: add_pair(const std::string& key,
                           const std::string& value) {
    if ((implementation->type() == ARRAY_IMPLEMENTATION) &&
        (implementation->size() >= MAX_ARRAY)) {
```

```
dictionary_implementation* new_implementation =  
    new dictionary_as_list(implementation);  
delete implementation;  
implementation = new_implementation;  
}  
// ... same thing for list -> database  
implementation->add_pair(key, value);
```

因此，我们不仅隐藏了字典的实现，而且使得它可以根据数据负载动态地重新配置自身。

## 方案 11：从 Linux 内核的链表函数中学习

**问题：**有很多创建链表的方法。只有一小部分是优秀的。

**方案：**Linux 内核中链表的实现。（在任何内核源代码树中参考文件 `/usr/include/linux/list.h`。）

从这个简单的模板中可以学到很多。

首先，由于链表是一个简单而通用的数据结构，创建一个链表模型是行得通的。毕竟，如果每个人都实现自己的链表，事情会变得混淆。尤其是所有的实现只是略有不同。

**课程 1：**代码重用。

链表的实现方式是非常高效和零活的。实际上你可以在多重链表中存放项目。

**课程 2：**零活的设计。

链表函数有很好的文档。头文件使用 **Doxygen** 文档约定的大规模注释（参见方案 65）。

**课程 3：**分享你的工作。文档化便于别人使用。

有一种机制可以促进人们使用这种实现，而避免自己编写属于自己的。如果你提交一个新的链表实现内核补丁，你会被“礼貌”的告知使用标准实现。此外，在你做这件事之前，你的代码不会进入内核。

**课程 4：**执行标准策略。主要来自同伴的压力。

因此，通过查看这个简单链表的实现可以学习一些东西。这是给我们留下的最后一课：

**课程 5：**优秀的黑客通过阅读某些懂得这种编程类型更多的人写的代码来学





## 第二章：安全方案

一些程序员先编码，然后考虑安全性。他们花费前六个月编写代码，然后，随后五年的时间用来创建安全补丁。

假设你站在一个很高的悬崖边缘。你需要到达底部。你会

1. 立即开始移动，并跳下悬崖，因为那是最快的方式到达底部。（在你开始跳之后你会担心安全。）
2. 分析地形，找到最适合你的方法到达底部。

如果你回答是#1，你编程就像在做大多数编程的苦差事。如果你一点也不考虑安全性，那这是你能给的最快的答案。

真正黑客的答案是“我想一想，算出最快的安全路线。当我下去帮助那些跟随我的人，我会记下这些。”

C语言从来没有为安全设计。C++在这样的基础上，然后建立一个更复杂和不安全的语言。然而，还是有黑客可以使你的程序更安全。换句话说，你的程序会很少崩溃。即使它们崩溃了，问题会更容易发现。

### 方案 12：消除副作用

**问题：**C++允许你使用像++和--的操作符，使你的代码非常紧凑。它还可以创建混淆的代码，同时变得不可读。

**方案：**编写仅执行一个操作的语句。除非作为独立的语句，否则不要用++和--。

例如，下面代码的结果：

```
i = 0;
// Bad code
array[i++] = i;
```

包含 i 的有两个子表达式。它们是：

1. i++
2. i

编译器可以按任何想要的顺序，自由的执行它们。所以，你使用的编译器是什么，甚至使用的编译选项可以影响结果。

没有理由试着将所有内容放在同一行。你的电脑有很多空间，一些额外的行不会损害什么。简单、有效的程序总是好于简短、紧凑和破碎的程序。

所以，避免副作用，将++和--分别独自放在一行上。如果重写前面的例子：

```
i++;  
array[i] = i;
```

不仅对于编译器，甚至任何正在读代码的人，操作的顺序是明确的。

### 方案 13：不要将赋值语句放进任何其他语句。

**问题：**一个经典的错误：使用=而不是==。

像这样的代码：

```
if (i = getch()) {  
    // Do something  
}
```

**方案：**从不将赋值语句放在任何其他语句中。（实际上从不把任何语句放进其他任何语句，但是这种最常见，因为它取决于自己的方案。）

原因是很简单的：你想正确地同时做两件简单的事情。在一个复杂、无效的语句中同时做两件事是不明智的。

这种方案应用于一些常见的设计模式。例如：

```
// Don't code like this  
while ((ch = getch()) != EOF) {  
    putchar(ch);  
}
```

遵循这种安全规则，代码是这样的：

```
// Code like this  
while (true) {  
    ch = getch();  
    if (ch == EOF)  
        break;  
    putchar(ch);  
}
```

现在很多人会指出第一个版本是非常紧凑的。那又如何？你想要紧凑的代码还是安全的代码？你想要紧凑的代码还是标准的代码？你想要紧凑的代码还是有效的代码？

如果我们不考虑代码有效的条件，我可以使代码更紧凑。因为大多数人重视的是，代码是安全和有效的，使用多个简单的语句替代单个紧凑的语句是明智的。

这种方案被用于保持事情简单和恒定。作为黑客我们知道，你是一个聪明的程序员。但是，它会使一个非常聪明的程序员知道什么时候是不明智的。

## 方案 14：当可能的时候使用 **const** 而不是**#define**

**问题：****#define** 指令定义文本替换。预处理器是它自己的语言，不遵循通常的 C++ 规则。这可能会带来一些意外。

例如：

```
// Code has lots of problems (don't code like this)
#define WIDTH 8 - 1
void print_info() {
    // Width in points
    int points = WIDTH * 72;

    std::cout << "Width in points is " <<
        points << std::endl;
```

提问：**WIDTH** 是多少？如果你回答 7 那就错了。预处理器是一个非常字面上的程序。**WIDTH** 是字面上的 **8 - 1**。

但是，每个人都知道 **8-1** 是 7，对吗？不是每个人。C++ 不是。尤其当用在表达式中的时候。这行代码：

```
int points = WIDTH * 72;
```

被预处理器转换成：

```
int points = 8 - 1 * 72;
```

结果，**points** 的值并不是程序员期待的。

**方案：**任何时候有可能，就使用 **const** 而不是**#define**。

如果将 **width** 这样定义：

```
static const int WIDTH = 8 - 1;    // Width of page - margin
```

这样计算就会是正确的。那是因为我们定义 **WIDTH** 使用的是 C++ 的语法，而不是预处理器的语法。

使用 **const** 有另外的一个好处。如果在**#define** 语句中犯错，问题可能不会出现，除非你实际上使用常量。C++ 编译器对 **const** 语句进行语法检查。这些语句的任何语法问题会立即展现，你不必去猜问题出在哪里。

**方案 15：如果你必须使用#define，将括号放在值的两边。**

**问题：**有些时候你不能使用 **const**。怎么避免前面展示的问题？

你可能会问为什么不能只使用 **const**？答案是，有时你需要建立一个头文件，共享给另一种语言的一个程序。例如使用 Perl 的 h2ph 程序识别 **#define**，但是不识别 **const**。

**方案：**总是将 **#define** 定义的值使用()封闭。

例如：

```
#define WIDTH (8 - 1)    // Width of page - margin
```

现在，当你像下面这样使用语句：

```
int points = WIDTH * 72;
```

会得到正确的值。

**方案 16：尽可能使用内联函数而不是参数化的宏。**

**问题：**参数化的宏可能导致意外发生。

考虑下面的例子：

```
#define SQUARE(X) ((X) * (X))

int i = 5;
int j = SQUARE(i++);
```

**j** 的值是多少？答案取决于编译器。**i** 的值确定不是 6。为什么？看看宏展开后的代码：

```
int j = ((i++) * (i++));
```

从这里看出 **i** 被增加了两次。同样因为操作顺序在 C++ 标准中没有明确，实际上 **j** 的值取决于编译器。

注意：这个例子中违反了方案 12。这是另一个为什么方案如此重要的例子。

**方案：**尽可能使用 **inline** 函数而不是 **#define**。

带有 **inline** 函数的代码看起来像这样：

```
static inline int square(int x) {
    Return (x * x);
}
```

现在，当这个函数被调用时有两件事情发生：**j** 得到了正确的值，**i** 增加一次。换句话说，代码的行为只像写的那样。拥有看起来和表现相同的东西是一个完美

的方案。

**注意：**我确信有人指出对任何类型宏都有效，而 **inline** 函数只对整型有效。通过使 **inline** 版本变成函数 **template**，问题很容易解决。

### 方案 17：如果你必须使用宏，在参数两边放置圆括号

**问题：**预处理器处理参数化宏的方式有时会导致错误的代码。例如：

```
// Don't code like this
#define SQUARE(x) (x * x)
```

所以，在下面的语句中 **i** 的值是多少：

```
int i = square(1 + 2);
```

应该是 9。但是语句展开成：

```
int i = (1 + 2 * 1 + 2);
```

结果是 5 而不是 9。

**方案：**在参数化宏中对使用的每个参数都用圆括号。例如：

```
#define SQUARE(x) ((x) * (x))
```

现在扩展的赋值语句如下：

```
int i = ((1 + 2) * (1 + 2));
```

我们得到了正确的答案。

**注意：**在方案 16 中，这样并不能解决自增的问题。这种方案应该只用在你绝对必须使用参数化宏，不能使用 **inline** 函数。（参见方案 12，同样可以帮助避免自增问题。）

### 方案 18：不要编写含糊的代码

**问题：**考虑下面的代码：

```
if (a)
    if (b)
        do_something();
    else    // Indentation off
        do_something_else();
```

**else** 匹配哪个 **if**？

1. 它匹配第一个 **if**。
2. 它匹配第二个 **if**。
3. 如果你不这样写代码就不必担心愚蠢的问题。

**方案：**黑客的答案很明显是第三个。黑客知道如何在开始之前避免麻烦。所以，如果我们从来不接近讨厌的代码，就不必担心它们是怎么工作的。（除非我们必须处理不是黑客编写的遗留代码。）

当代码中有任何的混淆，总是使用{}包含。先前的例子应该这样写：

```
if (a) {  
    if (b) {  
        do_something();  
    } else {  
        do_something_else();  
    }  
}
```

这个代码中很容易看出 **else** 匹配哪个 **if**。

显而易见是安全编码重要的一部分。即使没有人利用 C++ 语法的模糊元素添加混淆，也已经存在很多的混淆和杂乱。

出色的黑客知道如何使事情保持简单、明显和有效。

## 方案 19：不要聪明的使用优先级规则

**问题：**下面的表达式的值是多少？

```
i = 1 | 3 & 5 << 2;
```

大多数人会回答“我不知道。”黑客会回答“我不知道，”然后写一个简短的程序验证答案。（我并不是将答案放在这里来取笑你。）

问题是，除非你真正熟悉 C++ 标准，记住 17 个操作符的优先级，否则你不能回答刚才的代码含义。

**方案：**限制自己使用两个优先级规则：

1. 乘法和除法优于加法和减法。
2. 在其他任何两边加上圆括号。

一致性和简洁性是安全编程的关键。思考和做决定越少，你做的错误决定就越少。一个优秀的黑客会生成规则和步骤的集合，以便做一致性和正确的事情。另一种说法是，优秀的黑客做对事情的伟大思考，所以他不必做太多的思考。

简化的程序规则是一个例子。几乎没有人记住官方的 17 个操作符，但是记住简化的两个是容易的。

使用我们的方案很容易计算出下面表达式的结果：

```
i = (1 | 3) & (5 << 2);
```

（我知道结果不一样，但是这是程序员最倾向的一种。）

## 方案 20：包含属于你的头文件

**问题：**在一个头文件中定义一个函数，其他的部分在代码中，这样是可行的。

例如：

square.h

```
extern long int square(int value);
```

square.cpp

```
int square(int value) {  
    return (value * value);  
}
```

**注意：**C++只是部分类型安全。函数的参数通过模块检查，而返回值不是。

所以当这个函数被调用的时候会发生什么？square 函数计算一个数，返回结果，一个 32 位的整数。调用者知道函数返回的是一个 64 位的整数。

因为返回 32 位的值和 64 位的值是返回不同的寄存器，调用程序变得垃圾。程序员的维护糟糕在于，思考像 square 这样的函数是如何工作的，很容易失败，确实是失败的。

**方案：**确保每个模块包含自己的头文件。如果 square.cpp 以此开始：

```
#include "square.h"
```

编译器会注意到问题，然后阻止编译代码。

专业的程序员知道在编译时捕获一个明显的问题比运行时定位一个随机值错误容易 10000 倍。

## 方案 21：同步头文件和代码文件名

**问题：**前面的方案告诉我们每个模块应该包含自己的头文件。如何使这变得尽可能简单。

**方案：**总是对头文件和代码文件使用相同的名字。因此，如果头文件是 square.h，那么代码文件就是 square.cpp。当你有一个这样的规则，你不必思考，避免额外的判断，这样使你的代码更可靠。

但是现在假设有 3 个文件 square.cpp，round.cpp 和 triangle.cpp。这些被编译，然后整合到一个库 libshape.a。理想的是我们应该提供给用户单个头文件，



以至于他不必要知道或理解我们的模型结构。该怎么办？

如果提供给用户一个 `shape.h` 文件，满足了简单的要求，但是违反了命名规则。

答案是两者我们都可以做到，向用户提供一个简单的接口文件，遵循我们的命名规则。以为三个模块建立 3 个头文件开始：`square.h`，`round.h` 和 `triangle.h`。然后为用户创建接口文件，`shape.h` 包括：

```
#include <shape-internal/square.h>
#include <shape-internal/round.h>
#include <shape-internal/triangle.h>
```

关于这个系统的好处是，我们的头文件镜像是对象文件。将一系列对象文件整合到一个库文件，组的头文件 `shape.h` 为用户将一系列的头文件整合到一个主头文件。

最好的方案形式之一是简化。一个优秀的黑客做了很多思考和设计，所以他不需要做很多的思考或设计。

## 方案 22：永不信任用户输入

**问题：**用户输入不好的东西。

大多数用户输入不好的东西是因为他们不理解软件或者理解软件的限制。他们可以查看下面的提示：

```
Enter a user name (5 characters only):
Do not type in more than 5 characters. It won't work.
Five character is the limit no more.
Absolutely no more than 5 characters please.
User name:
```

然后看到一个公开的邀请，键入 55 个字符。

愚蠢的用户输入坏字符。思考的比计算机多的用户输入坏字符。普通用户输入错误的内容，输入坏数据。我认为可以在这里设置模式。更糟糕的是，恶意用户为了使系统崩溃或绕过安全限制而输入恶意数据。两个经典的攻击向量是堆栈破坏攻击和 SQL 注入攻击。

在堆栈破坏攻击中，用户尝试溢出输入缓冲区。如果他输入足够多的数据，就可以覆盖栈中的返回地址，使计算机执行任意的代码。

为了对抗堆栈破坏攻击，需要总是检查用户输入的长度，确保遵循限制条件。

如果你使用 C 风格的 I/O，意味着使用 `fgets` 而不是 `gets`。（参考下面的方案 23。）

SQL 注入攻击是用户提交坏形式的数据，期望它在 SQL 查询语句中执行。例如，下面更新用户 E-Mail 地址的 SQL 代码：

```
UPDATE user_info SET email = 'fred@whatever.com'
WHERE user = 'Fred';
```

如果用户告诉你他的名字是“Fred”，那么事情进展顺利。但是一个恶意用户使用用户名来做游戏。例如，假如他提供给我们的用户名是“F’;SELECT user, password FROM user\_info;”现在的 SQL 命令是：

```
UPDATE user_info SET email = 'fred@whatever.com'
WHERE user = 'F'; SELECT user, password FROM user_info;
```

更好的写法像这样：

```
UPDATE user_info SET email = 'fred@whatever.com'
WHERE user = 'F';
SELECT user, password FROM user_info;
```

**SELECT** 语句会在数据库中返回所有用户名和密码。

### 出色的数据库安全保护

例子中的数据库模式表明了一个糟糕的数据库设计。你永远不应该让互联网可以直接访问任何数据库中的敏感信息（密码，社会保险号，信用卡号）。

这类信息应该保存在专用安全电脑，允许从你自己的电脑进行有限的访问，不允许其他人访问。它应该被紧锁。数据本身应该使用密钥加密，密钥在机器重启的时候必须在控制台输入。只有在非常有限的情况下进行未加密的数据传输。

客户端/服务器连接应该也这样被锁定。客户端应该永远不能向数据库询问密码。唯一它能做的应该是问“密码是否正确？”为了增加安全性问题应该通过加密连接传输。

这种方式查询数据并非万无一失，但是它能阻止大多数的坏人。

将敏感数据存放在笔记本或便携式驱动器，尤其

是信用卡号码和社会宝箱账号，是真的愚蠢。不要将敏感信息放在便携式设备中，不要将设备遗留在酒店房间这样的地方，很容易被偷。

同样，任何在你笔记本上的敏感数据应该通过一个好的加密系统保护。

为了防止 SQL 注入攻击，应该验证用户提供的所有字符。这是一个不要那样做的例子。

```
// Bad code
bool validate_name(const char* const name) {
    for (int i = 0; name[i] != '\0'; ++i) {
        if (name[i] == '\\')
            return (false);
    }
    return (true);
}
```

为什么这样的代码是差的？因为它只检查坏字符。实际上在 SQL 中有很多的坏字符。你不应该确保输入中不包含坏字符，而是应该确保所有的都是好的。

```
// Good code
bool validate_name(const char* const name) {
    for (int i = 0; name[i] != '\0'; ++i) {
        if (!isalnum(name[i]))
            return (false);
    }
    return (ture);
};
```

只有好的字符比排除坏字符要安全的多。因为你犯了一个错误，将“好东西”定义得很严格不会在程序中引起安全漏洞。如果在定义“糟糕的东西”上面犯了错误，坏的东西有可能通过。

记住，仅仅因为你是偏执狂，并不意味着他们不会攻击你。

## 方案 23：不要使用 gets

目前，几乎每个人都知道使用 gets 会产生安全性和可靠性问题。但是由于历史原因，把它包括在这里，因为它是差的编程的一个很好的例子。

查看下面代码的所有问题：

```
// Really bad code
```

```
char line[100];
gets(line);
```

因为 `gets` 不进行边界检查，因此一个长度超过 100 的字符串会覆盖内存。如果你足够幸运，程序会崩溃。或者它会表现出奇怪的行为。

但是，这个代码还有一个安全问题。黑客可以创建一个精心构造的字符串，覆盖栈，然后执行恶意用户想执行的任意代码。

`gets` 函数是如此的不好，以至于无论什么时候使用它，`GUN gcc` 链接器会提示一个警告。

```
/tmp/cc15WJ5m.o(.text+0x24): In function `main':
```

```
: warning: the `gets' function is dangerous and should not be used.
```

方案：替换为 `fgets`。

```
// Good code
char line[100];
fgets(line, sizeof(line), stdin);
```

`fgets` 调用不会获取超过变量所能保存的数据。这样防止黑客执行一个堆栈破坏攻击。

## 方案 24：刷新调试

问题：输出缓冲可以引起异常的结果。

例如：

```
std::cout << "Doing unrelated stuff" << std::endl;
do_stuff();

std::cout << "Doing divide... ";
i = 1/0; // Divide by zero
std::cout << "complete\n";
```

当运行程序打印（在一些系统上）：

```
Doing unrelated stuff
Floating point exception
```

从输出可以看出问题“明显”在 `do_stuff` 函数中。发生了什么？

问题是输出会被缓存。所以字符串“**Doing divide ...**”进入了缓存，然后由于一个除零错误程序崩溃，输出永远不会显示。这就是迷惑的来源。

方案：确保调试时每次输出之后缓冲区被刷新。

有几种方法可以做到。第一种是显式地刷新每一条语句。

```
std::cout << "Doing divide ... " << std::flush;
```

这样有效，但是你要记住对每条语句做刷新操作。

另一种方式是设置 **unitbuf** 标志，告诉 C++ 在每条输出操作后进行刷新。这样只需要在你程序的头部做一次。

```
std::cout << std::unitbuf;
// From now on everything is automatically flushed
```

在 C 语言中可以通过使用 `setvbuf` 函数，设置 `_IONBV` 标志来完成同样的事情：

```
static char buf[512];    // Buffer for standard out
setvbuf(stdout, buf, _IONBV, sizeof(buf));
```

对于黑客来说，意识到程序内部发生了什么是很有用的。有时候编译器，库或机器会做一些奇怪的事情。但了解内部只是成功的一半。了解如何绕过系统的内部限制是优秀黑客的标志。

## 方案 25：使用断言保护数组访问

**问题：** C++ 不做边界检查

例如，下面的代码编译和执行都是正常的。它也会破坏内存。

```
// Bad code
int data[10];
// ...
int i = 11;
data[i] = 5; // Memory corruption
```

**方案：** 使用 `assert` 检查数组访问。

例如：

```
// Better code
#include <cassert>
int data[10];
// ...
int i = 11;
assert((i >= 0) && (i < 10)); // Good example
                                // Rotten implementation
data[i] = 5;
```

这样有效，但是存在一个问题。我们在两个地方使用了常量 10。（声明和断言。）对某些人来说很容易改变其中一个，而另一个却没有发生变化。

一种解决方法是使用名字常量。

```
// Better code
#include <cassert>
const int DATA_SIZE = 10;
```

```
int data[DATA_SIZE];
// ...
int i = 11;
assert((i >= 0) && (i < DATA_SIZE)); // Better, but not best
data[i] = 5;
```

理想的情况是我们甚至不需要使用名字常量。单独使用 `data` 变量进行 `assert` 边界检查是可能的：

```
assert((i >= 0) && (i < sizeof(data) / sizeof(data[0])));
```

所以，在这里发生了什么？表达式 `sizeof(data)` 返回数组 `data` 中字节的数量。但是我们需要的是变量中元素的数量，而不是字节的数量。

解决方法是将数组中字节数除以第一个元素的字节数。结果就是一个告诉我们元素数量的表达式。

```
sizeof(data) / sizeof(data[0])
```

现在成为真正的黑客，我们不想一次又一次的写同样的表达式，所以建立一个宏让生活更容易。

```
/*
 * assert_bounds(var, index) - Make sure an index is in bounds.
 *
 * Warning: This only works if var is a array variable and not a pointer
 */
#define assert_bounds(var, index) \
    Assert((index >= 0) && \
           (index < (sizeof(var) / sizeof(var[0]))));
```

因为我们是如此好的程序员，所以我们注释宏，甚至包含一个描述自身限制的警告信息。

数组越界是最常见的编程错误之一，尝试和定位是极度令人沮丧的。这样的代码并没有消除它们，但是它确实使错误的代码退出，使问题很容易被发现。

**警告：**断言不保证使你的程序退出。例如，考虑下面的语句：

```
assert(false);
```

这个语句“明显的”会导致程序退出，因为断言是一直错误的。如果程序是带有 `NDEBUG` 编译的，所有的断言不会编译。换句话说，如果 `NDEBUG` 在之前的语句中声明，那么断言无效。

### 一个真实的系统崩溃

你应该只在程序可接受退出的地方使用 `assert`。

大多数情况下，程序崩溃对用户来说是个烦恼，但不是灾难。并不是一直是这种情况。

1996 年一些人开始在升级过的硬件平台上运行程序。由于硬件升级，程序运行时间超过预期，一个计数器溢出。因为这是 ADA 代码，所以触发了一个异常。

异常没有被捕获，因此系统执行了默认异常处理程序，并暂停了处理器。这不是一件好事。

已经升级的硬件平台是 Ariane4 号火箭到 Ariane5 号火箭。有问题的计算机的任务是使火箭指向太空。技术上，火箭没有崩溃。发射导演在开始前往地面前将其炸毁。

这个错误的成本估计大约为 500,000,000 美元（美国）。

## 方案 26：使用模板创建安全的数组

**问题：** 你不想每次访问数组时都添加断言。

**方案：** 将代码隐藏在模板中。然后工作自动完成。

```
#include <cassert>

template<typename array_type,
        unsigned int size> class array {
private:
    static const unsigned int ARRAY_SIZE = size;
    array_type data[ARRAY_SIZE];
public:
    array(void) {};
    array(const array& other_array) {
        memcpy(data, other_array.data, sizeof(data));
    };
    ~array() {};
    array& operator = (const array& other_array) {
        memcpy(data, other_array.data, sizeof(data));
        return (*this);
    };
};
```

```

public:
    array_type& operator[](int index) {
        assert(index >= 0);
        assert(index < ARRAY_SIZE);
        return (&data[index]);
    }
};

```

这个类有几个好的属性。第一个是拷贝构造和赋值操作符以一种方式实现，允许复制数组。

但是这个代码最大的优点是在数组中处理访问元素的函数。它包含 `assert` 语句，以免数组溢出。

```

    array_type& operator[](int index) {
        assert(index >= 0);
        assert(index < ARRAY_SIZE);
        return (&data[index]);
    }

```

另一个模板的属性是它不用提供一种从数组到指针的转换方式。是否认为这是一个属性或错误，取决于你自己。

## 方案 27：当什么都不做的时候，使其变得明显。

**问题：**下面的代码令人困惑。

```

// Very bad coding style
int i;
for ( i = 0; foo[i] != '\0'; ++i);
std::cout << "Result " << i << std::endl;

```

乍一看，似乎有人没有正确的缩进程序。`std::cout` 这一行应该被缩进，因为它是 **for** 语句的一部分。

但仔细检查后，发现程序缩进正确。在 **for** 循环结束时有个分号：

```

for (i = 0; foo[i] != '\0'; ++i); ←

```

分号几乎不可见。让它更明显是有用的。

**方案：**什么都不做。总是放一些东西进去表明什么都没做。

```

// Almost adequate coding style
for (i = 0; foo[i] != '\0'; ++i)
    /* Do nothing */

```

这是有好处的，因为可以改进它。**continue** 关键字告诉 C++ 再次开始循环。它可以放进空循环中，语句看起来更丰富一些：



```
// Good coding style
for (i = 0; foo[i] != '\0'; ++i)
    continue;
```

## 方案 28：将 `break` 或 `/*Fall Through*/` 作为每个 `case` 的结尾

**问题：**下面的代码，程序员是想 `STATE_ALPHA` 的情况下，继续执行或他出错了？换句话说，实际情况是 `STATE_ALPHA` 内部调用 `do_alpha` 和 `do_beta` 或者就是一个错误？

```
// Rotten code
switch (state) {
    case STATE_ALPHE:
        do_alpha();
    case STATE_BETA:
        do_beta();
        break;
// ....
```

从代码中不可能知道程序员的目的。

**方案：**让你做的事情变得显而易见。如果你的目的是从一个 `case` 到另一个 `case`，使用像这样 `// Fall Through` 的注释指出。

```
// Rotten code
switch (state) {
    case STATE_ALPHA:
        do_alpha();
        // Fall Through
    case STATE_BETA:
        do_beta();
        break;
// ....
```

毕竟成为黑客意味着你必须清楚，而不是偷偷摸摸。

## 方案 29：一个简单的不可能条件的断言语句

**问题：**当你检测一个内部错误时，你怎么办？

你可能使用 `assert(false)`；但是当程序退出时不会给你更多的信息。

**方案：**在你的 `assert` 语句中放置一条消息。例如：

```
if (this_is_not_possible) {
    assert("INTERNAL ERROR: Impossible condition" == 0);
    abort();
}
```

代码 `"string" == 0` 将字符串的地址与 0 比较。它们应该不是零，所以 `assert` 失败。因为一个失败的断言打印失败的条件，现在，当你的程序退出，你会获得一个告诉你发生了什么的友好错误消息。

但是如果当你的 `assert` 语句失败时，程序退出，为什么在 `abort` 中放进去？因为有可能使用编译时的（`-DNDEBUG`）选项转换，使得断言语句失效。

所以这段代码实际表明，带有友好错误消息的退出。如果程序依然运行，那就退出好了。

### 方案 30：在 `switch` 中总是检查不可能的情况

**问题：**一个变量可以只包括元音字母。如果值为 'q' 的话怎么办？

#### 糟糕的谜语时间

**Q：**什么时候你的时钟指针指向 13？

**A：**得到一个新时钟的时候。

我们的例子中，'q' 表明一个内部错误。毕竟对我们来说得到 'q' 是“不可能的”。但是这并不阻止我们得到一个。

上面有人吹牛。主要有两条编程安全性的规则：

1. 从不信任你没有写过的代码创建的数据。
2. 从不信任你写过的代码创建的数据。

当构建安全程序的时候，良好的偏执是有益的。毕竟仅仅因为你的偏执，并不意味着系统会靠近你。

**方案：**使 `switch` 语句考虑到所有情况 - 甚至不存在的情况。

如果 `switch` 只有有限数量的值，你应该总是添加一个 **default** 子句来捕获范围之外的值：

```
switch (vowel) {  
    case 'e':  
        ++e_count;  
        break;  
    case 'i':  
        ++i_count;  
        break;  
    case 'a':  
    case 'o':  
    case 'u':
```

```
        // Ignore these values
        break;
    default:
        assert("INTERNAL ERROR: Vowel not legal" == 0);
        abort();
}
```

注意到代码的其它部分，通过语句我们清楚的指明忽略了哪些元音：

```
// Ignore these values
```

毕竟，这个想法不仅安全而且简洁明了。

### 方案 31：创建可在编译时检查的模糊类型（句柄）

**问题：**你正在创建使用很多句柄的 API。例如，字体句柄，图形句柄，窗口句柄，颜色句柄等等。

一种方案是为每个句柄建立“不同”的类型：

```
// Dangerous code
typedef short int font_handle;
typedef short int graphic_handle;
typedef short int window_handle;
typedef short int color_handle;
```

下面的代码是在操作中的这些类型：

```
font_handle the_font = find_font("Times", 10, "Bold");
color_handle the_color = find_color("Light Red");

draw_text(the_font, the_color);
```

但是，使用这种方式会有一个问题。font\_handle 和 color\_handle 有同样的基本类型。如果你像下面这样做，编译器不会报错：

```
// Parameters backwards
draw_text(the_color, the_font);
```

理想的方案是，如果我们混淆句柄，编译器就会在哪里报错。另外的要求是句柄尽可能占用少的空间，最好两个字节。

**方案：**使用小的结构体存放句柄。

看看下面这些句柄的声明：

```
// Safe code
typedef struct {
    short int handle;
} font_handle;
```

```
typedef struct {
    short int handle;
} graphic_handle;

typedef struct {
    short int handle;
} window_handle;

typedef struct {
    short int handle;
} color_handle;
```

所以，我们使用两个字节的结构体替换了两个字节的整数。有什么大不了的？重要的是类型检查。现在句柄有不同的结构体，之间没有共同的基本类型。结果，编译器会做类型检查。换句话说下面这样是合法的：

```
draw_text(the_font, the_color); // Correct and legal
```

但是下面这些不是合法的：

```
draw_text(the_color, the_font); // Incorrect and illegal
```

注意到句柄系统的所有优点依然存在。句柄仍然是小的模糊实体，但现在它们要经过类型检查。现在的方案克服了 C++ 的弱类型，使它变成更强大的。

### 方案 32：当清零数组时使用 `sizeof`

**问题：**如果常量不同步，在 `memset` 中调用命名常量时会失败。

例如：

```
// Bad code
const int CONNECTION_INFO_SIZE = 100;
char connection_info[CONNECTION_INFO_SIZE];

const int SIMPLE_CONNECTION_INFO_SIZE = 30;
char secure_connection_info[SIMPLE_CONNECTION_INFO_SIZE];

// ...
// Zero all connection information
memset(connection_info, '\0', CONNECTION_INFO_SIZE);
// Mistake
memset(secure_connection_info, '\0', CONNECTION_INFO_SIZE);
```

最后一个语句使用了错误的常量清零整个数组，然后会破坏内存。内存破坏是最难调试的错误之一，因为它在远离最初代码的部分引起奇怪的错误。

**方案：**总是使用 `sizeof()` 来确定结构体的大小。

所以，处理使用 `memset` 调用的合适方法是：

```
memset(array, '\0', sizeof(array));
```

无论你怎么改变数组的大小和基本类型，`sizeof()`操作会返回正确的大小，`memset` 调用做正确的事情。

### 方案 33：在 `memset` 中使用 `sizeof(var)` 而不是 `sizeof(type)`

**问题：**在 `memset` 中使用 `sizeof(type)` 时不安全的。

一些程序员考虑下面的好的编程实践：

```
// Not a good idea
struct data_struct {
    int i1, i2;
};

data_struct* data_ptr;
data_ptr = new data_struct;
// ...
// Bad code
memset(data_ptr, '\0', sizeof(data_struct));
```

如果你修改了 `data_ptr` 的指向问题就会出现。

```
// This code contains a mistake
struct data_struct {
    int i1, i2;
};
struct data_struct_improved {
    int i1, i2;
    int extra_data;
};

data_struct_improved* data_ptr;
data_ptr = new data_struct_improved;
// ....
// Mistake
memset(data_ptr, '\0', sizeof(data_struct));
```

在真实世界中，`data_ptr` 的声明和清零之间会有很多代码：

```
memset(data_ptr, '\0', sizeof(data_struct));
```

当程序员修改变量 `data_ptr` 的类型时，他可能没有注意到这一行。但是错误依然会产生，这是不好的。

**方案：**使用 `sizeof(*ptr)` 确定动态数据的大小

无论怎样改变 `data_ptr` 的类型，表达式 `sizeof(*data_ptr)` 会一直保持正确的字节数。

所以这样的代码一直有效：

```
memset(data_ptr, '\0', sizeof(*data_ptr));
```

### 方案 34：避免重用清零的指针

**问题：**重用指针会破坏内存。

一个指针被删除（或释放）之后，不应该再次使用。尽管并不防止糟糕的代码使用它。有时会导致内存随机损坏，这是最难调试的问题之一。

这是我们讨论的一个例子：

```
int* data;

data = new int[10];
data[0] = 0;
// ... lots more work with data ...

delete[] data;
// ... a few thousand lines of code ...
data[2] = 2; // Memory corruption
```

**方案：**在删除指针后将其设置为 `NULL`。

例如：

```
delete[] data;
data = NULL;
```

结果是在大多数系统上如果你尝试使用指针，系统会崩溃：

```
// ... a few thousand lines of code ...
data[2] = 2; // System crashes
```

这种方案并不能阻止错误，但是可以改变它们。访问删除后的内存指针会导致数据破坏或堆破坏。错误的结果可能很长时间不会出现，当你执行完全不相关的一块代码时可能会发生。

另一方面，如果指针被设置成 `NULL`，尝试使用指针，会在使用指针的地方导致系统崩溃。

所以，通过每次使用指针后重置指针，可以将难以寻找的随机错误变成可重复和容易修复的错误。

### 方案 35: 使用 `strncpy` 替换 `strcpy` 避免缓冲区溢出

**问题:** `strcpy` 函数是不安全的。

下面的代码会数组溢出，破坏内存。

```
// Bad code
char name[10];
strcpy(name, "Steve Oualline");
```

**方案:** 使用 `strncpy`

`strncpy` 复制一个字符串，但是只是有限制的字符数。如果添加了 `sizeof()` 操作符，可以保证从不会复制超过目标可以存放的数据。

所以一个安全的字符串复制方法是：

```
// Partial solution
strncpy(name, "Steve Oualline", sizeof(name));
```

但是存在一个捕获。阅读 `strncpy` 的文档发现下面的段落：

The `strncpy()` function is similar [to `strcpy`], except that not more than `n` bytes of `src` are copied. Thus, if there is no null byte among the first `n` bytes of `src`, the result will not be null-terminated.

我们希望结果始终是 null 结尾，所以我们显式地处理异常情况，确保字符串是以 null 结尾。再一次，使用 `sizeof()` 操作符确保获得正确的大小。

```
name[sizeof(name)-1] = '\0';
```

将这行与我们得到的代码放在一起：

```
// Good code
char name[10];
strncpy(name, "Steve Oualline", sizeof(name));
name[sizeof(name)-1] = '\0';
```

**警告:** 假如是名字声明为数组，`sizeof(name)`才有效。如果声明作为指针，那么 `sizeof(name_ptr)`返回的是指针的大小，而不是指向的数据大小。

**注意:** 通过使用 C++ 的 `std::string` 字符串来避免 C 风格字符串的相关内存问题是可能的。但是很多代码依然使用旧的风格字符串，使用 C++ 字符串有一些性能问题。

**注意:** 一个叫做 `strlcpy` 的新函数已经在一些 C/C++ 库中介绍，允许用于安全字符串拷贝。然后，它不是标准的，不在所有的库中存在，Solaris 和 OpenBSD 以不同的方式实现它。但是如果你拥有它，意味着使用它。

### 方案 36：为了安全性使用 `strncat` 而不是 `strcat`

**问题：**`strcat` 是不安全的。

下面的代码数组名溢出，破坏内存。

```
// Bad code
char name[10];

strncpy(name, "Oualline", sizeof(name));
name[sizeof(name)-1] = '\0';

strcat(name, ", ");
strcat(name, "Oualline");
// Memory is now corrupt
```

**注意：**使用一个数值常量（10）而不是命名常量（`NAME_SIZE`）定义数组是坏的编程实践。但是它使样例简单，所以当解释一个方案时它是好的编码实践。

**方案：**总是使用 `strncat`。

使用安全的编程方案重写前面的例子。首先将“Oualline”安全地拷贝到 `name` 变量。

接下来使用 `strncat` 添加<逗号>，<空格>字符。问题是可以在数组中放置多少个字符？变量 `name` 最多存放 10 个字符。但是为了安全性将 10 写成 `sizeof(name)`。

我们已经使用了 `strlen(name)` 个字符，所以变量中剩余的空间通过表达式指出：

```
sizeof(name) - strlen(name)    // Incomplete
```

为了字符串的结尾需要一个字节。所以把它添加到表达式：

```
sizeof(name) - strlen(name) - 1;
```

将这些放在一起得到程序的安全版本：

```
// Good code
char name[10];

strncpy(name, "Oualline", sizeof(name));
name[sizeof(name)-1] = '\0';

// Concatenation example
strncat(name, ", ", sizeof(name) - strlen(name) - 1);
name[sizeof(name)-1] = '\0';    // This is required
```



```
strncat(name, "Oualline", sizeof(name) - strlen(name) - 1);  
name[sizeof(name)-1] = '\0';
```

**注意：**一个叫做 `strlcat` 的新函数已经在一些 C/C++ 的库中介绍，允许用于安全字符串连接。然后，它不是标准，它并不存在于所有的库中，Solaris 和 OpenBSD 以不同的方式实现它。但是如果你拥有它，意味着使用它。

### 方案 37：使用 `snprintf` 构建字符串

**问题：**当使用 `sprintf` 时，可以写入比字符串长的数据。例如：

```
char file[10];  
sprintf(file, "/var/tmp/prog/session/%d", pid);
```

**方案：**使用 `snprintf` 代替。

`snprintf` 的第二个参数是字符串的大小。为了保持我们的安全方案，尽可能的在参数中使用 `sizeof(string)`。由于 `snprintf` 知道字符串的大小，它足够聪明不让字符串溢出。

所以下面的代码不会混淆内存。文件名非常短，但是至少你不需要处理内存破坏。

```
char file[10];  
  
snprintf(file, sizeof(file), "/var/tmp/prog/session/%d", pid);
```

### 方案 38：不要用人为主限制设计

**问题：**任何时候你带有人为主限制的设计代码，有人会越过它。例如，考虑代码：

```
void err(const char* const fmt, int a = 0, int b = 0, int c = 0, int d = 0,  
int e = 0)  
{  
    fprintf(stderr, "Fatal Error:\n");  
    fprintf(stderr, fmt, a, b, c, d, e);  
    fprintf(stderr, "\n");  
    abort();  
}
```

现在，如果你希望写简单的消息，这是有效的：

```
err("Size parameter (%d) out of range", size);
```

但是，当希望更复杂的调用时会发生什么？

```
err("Point (%d, %d) outside of box (%d, %d), (%d, %d)",
```

```
Point.x, point.y, box.x1, box.y1, box.x2, box.y2);
```

我们的函数可以采用最多 5 个参数的格式。我们只给它 6 个参数。它就失效了。

现在我们通过添加另外的参数修复 `err` 函数，但是当我们需要 7 个参数的时候才有效。另一个改变需要 8 个参数等等。

**方案：**编写一般函数时，不要将自己陷入死角。如果你真的了解 C++ 语言，你会知道如何设计一个接受任意数量参数的错误函数：

```
#include <cstdarg>

void err(const char* const fmt, ...)
{
    va_list args;

    va_start(args, fmt);
    fprintf(stderr, "Fatal Error:\n");
    vfprintf(stderr, fmt, args);
    fprintf(stderr, "\n");
    abort();
}
```

这不是人为限制可能产生麻烦的唯一地方。任何时候你在系统中设计限制时，迟早会有人越过它们。

一个典型的人为限制的例子是 640K 限制。引用了比尔盖茨的一句话“对任何人来说 640K 是足够的内存”。但是在最初的 PC 出现时，人们开始发明带有特殊驱动程序的附加卡，旨在绕过 640K 的限制。

另一个例子是 `ls` 命令。最初的版本使用单个的选项（`-l`, `-R` 等等。）。现在键入 `man ls`，你会发现几乎所有的单个字符（大写和小写的情况）被这个命令使用。52（26\*2）个选项还不够。将选项限制为单个字符，`ls` 的设计者限制了它们的可扩展性。有人不得不设计一种新的语法（`--long-options`）来解决这个问题。

### 老年公民

一个州的老年人跟踪系统限制人的年龄为两位数。所以当一个人 100 岁时，年龄重置为 0。这是如此大的一个问题。

当她是 107 岁时，真正的麻烦出现了，国家派遣一位逃学官到她家，找出她的父母没有在一年级就读的原因。

### 方案 39：总是检查自赋值

**问题：**建立一个类，其中存在下面的操作是有可能的：

```
x = x;
```

在真实的生活里，自赋值可能不会像这样显而易见。相反，它可能隐藏在一对或更多层的函数调用中。

但是一旦出现，结果可能是灾难性的。

考虑下面伪代码的赋值操作：

```
a_class& operator = (const a_class& other_one) {  
    1. Delete my data  
    2. Allocate a new structure the same size as other_one's data  
    3. Copy over the data  
}
```

这个类的代码是：

```
class a_class {  
    private:  
        unsigned int size;  
        char* data;  
    public:  
        a_class(unsigned int i_size) {  
            size = i_size;  
            data = new char[size];  
        }  
        // Here's the problem code  
        a_class operator = (const& a_class other) {  
            delete[] data;  
            data = NULL;  
            size = other.size;  
            data = new char[size];  
            memcpy(data, other.data, size);  
        }  
}
```

现在当一个变量赋值给它自身，考虑会发生什么。第一步：

```
1. Delete my data
```

为目标变量(\*this)删除数据。然而，目标变量也是源变量(other\_one)，删除的数据同样也是。实际上只有数据的拷贝被删除了。

当我们到这步时，程序会失败：

```
2. Allocate a new structure the same size as other_one's data
```

因为在 other\_one 中没有数据，我们在第一步中把它删除了。

**方案：**程序显式地预防和检查自我赋值。

每个不寻常的类应该在赋值操作函数中以下面的代码开始：

```
a_class& operator = (const a_class& other_one) {  
    if (this == &other_one)  
        return (*this); // Self assignment detected  
    // ....  
}
```

这点点保险可以防止发生真正令人讨厌的问题。

## 方案 40：使用哨兵保护类的完整性

**问题：**你的一个类发生了奇怪的事情。有时数据被破坏。你怀疑有人正在覆盖随机内存，但是你怎样验证。

**方案：**在类的开始和结尾放进哨兵，经常检查它们。

例如：

```
class no_stomp {  
    private:  
        enum {START_MAGIC = 0x12345678, END_MAGIC =  
0x87654321};  
        // This must be the first constant or variable declared.  
        // It protects the class against nasty code overwriting it.  
        const long unsigned int start_sentinel;  
    // ....  
    public:  
        no_stomp(): start_sentinel(START_MAGIC),  
end_sentinel(END_MAGIC){}  
    // ....  
        // Every member function should call this to make sure that  
everything is still ok  
        check_sentinels() {  
            assert(start_sentinel == START_MAGIC);  
            assert(end_sentinel == END_MAGIC);  
        }  
    // ....  
    private:  
        // This must be the last declaration  
        const long unsigned int end_sentinel;  
};
```

该类在类的两端声明了两个常量。术语“常量”有点不正确。当类创建的时候，真正的变量被初始化，并且在类的生命周期内不能修改。

如果一些错误的数组溢出会覆盖它们，比如使用损坏的内存指针，或做一些

其它疯狂的事情。如果发生，下一步调用 `check_sentinels` 会引起断言失败，使程序崩溃。

所以，实际上我们正在做的是将难以解决的问题（内存的随机覆盖引起随机结果）变成容易解决的问题（一旦内存破坏，程序崩溃）。

一些 C 代码升级为 C++ 的代码类型帮助我找到一些相当不寻常的问题。这个例子中，第一次调用 `check_sentinels` 会导致程序崩溃。分析程序，我发现问题由调用构造函数和第一个哨兵验证调用之间引起的。

所以我在构造函数中添加一个断点，计划单步执行代码，直到我发现问题的。在构造函数调用之前程序崩溃了。所以我留下了一个疑问“一个程序如何在调用构造函数之前调用成员函数？”

答案非常简单。这个类是用 `malloc` 创建的。我曾经告诉过你使用 C。证明从 C 到 C++ 并没有完全移植。使用 `new` 替换 `malloc` 解决了这个问题。

## 方案 41：使用 `valgrind` 解决内存问题

**问题：**损坏的指针，写入已分配内存的结尾，内存泄漏。

当涉及到内存管理时，C++ 提供更多的灵活性。你被允许分配和解除分配内存，可以直接操作指针。

灵活性是有代价的。因为语言允许你分配内存，你可以搞砸分配。同样你也可以搞砸解除分配和指针的使用。

由于 C++ 中没有内置的安全检查，你怎么保护你的代码？

**方案：**使用 `valgrind`。

`valgrind` 程序在虚拟机中运行你的程序。内存访问要经过额外的检查。有可能检测明显的指针类型错误。包括：

1. 使用释放的内存。
2. 向已经分配块的结尾写入。
3. 向已分配块的开始写入。
4. 内存泄漏。

看一个小的例子：

```
1  #include <iostream>
2
```

```
3    int main()
4    {
5        int* ptr = new int[10];
6        int* ptr_b = new int[10];
7
8        *(ptr+11) = 5;
9        ptr_b = NULL;
10       exit(0);
11    }
```

**注意：**你可能注意到，我们违背了我们的安全原则，没有在每次指针访问的时候放置 **assert** 语句。但是这段代码的目的是产生问题，不是捕获他们，所以我们放弃了安全性。

这段代码的问题是什么？第 8 行访问 10 个元素数组的第 11 个元素。第 9 行清零了，只有指针指向的第 6 行分配的内存，因此造成内存泄漏。

让我们看看，当程序运行的时候 **valgrind** 做了什么。

```
$ valgrind --leak-check=full ./bad_mem
==31755== Memcheck, a memory error detector.
==31755== Copyright (C) 2002-2006, and GNU GPL'd, by Julian Seward et al.
==31755== Using LibVEX rev 1658, a library for dynamic binary translation.
==31755== Copyright (C) 2004-2006, and GNU GPL'd, by OpenWorks LLP.
==31755== Using valgrind-3.2.1, a dynamic binary instrumentation framework.
==31755== Copyright (C) 2000-2006, and GNU GPL'd, by Julian Seward et al.
==31755== For more details, rerun with: -v
==31755==
==31755== Invalid write of size 4
==31755== at 0x8048712: main (bad_mem.cpp:8)
==31755== Address 0x4253054 is 4 bytes after a block of size 40 alloc'd
==31755== at 0x4019D55: operator new[](unsigned) (vg_replace_malloc.c:195)
==31755== by 0x80486F5: main (bad_mem.cpp:5)
==31755==
==31755== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 21 from 1)
==31755== malloc/free: in use at exit: 80 bytes in 2 blocks.
==31755== malloc/free: 2 allocs, 0 frees, 80 bytes allocated.
==31755== For counts of detected errors, rerun with: -v
==31755== searching for pointers to 2 not-freed blocks.
==31755== checked 104,796 bytes.
==31755==
==31755==
==31755== 40 bytes in 1 blocks are definitely lost in loss record 1 of 2
==31755== at 0x4019D55: operator new[](unsigned) (vg_replace_malloc.c:195)
==31755== by 0x8048705: main (bad_mem.cpp:6)
```

```
==31755==
==31755== LEAK SUMMARY:
==31755== definitely lost: 40 bytes in 1 blocks.
==31755== possibly lost: 0 bytes in 0 blocks.
==31755== still reachable: 40 bytes in 1 blocks.
==31755== suppressed: 0 bytes in 0 blocks.
==31755== Reachable blocks (those to which a pointer was found) are not shown.
==31755== To see them, rerun with: --show-reachable=yes
```

喋喋不休之后，程序捕获了第一个问题，工具注意到在向一个非法的内存写入：

```
==31755== Invalid write of size 4
==31755== at 0x8048712: main (bad_mem.cpp:8)
==31755== Address 0x4253054 is 4 bytes after a block of size 40 alloc'd
==31755== at 0x4019D55: operator new[](unsigned) (vg_replace_malloc.c:195)
==31755== by 0x80486F5: main (bad_mem.cpp:5)
```

程序完成之后，**valgrind** 检查堆，看是否有内存丢失。这个例子中，它发现我们在第 6 行分配的一些内存丢失了。

```
==31755== 40 bytes in 1 blocks are definitely lost in loss record 1 of 2
==31755== at 0x4019D55: operator new[](unsigned) (vg_replace_malloc.c:195)
==31755== by 0x8048705: main (bad_mem.cpp:6)
```

这个工具是聪明的。它没有报告我们在第 5 行分配的内存也丢失了，甚至我们分配了内存，而且从未释放。因为程序退出的时候有一个指向这块内存的指针。换句话说，有人在退出时使用了它，所以它没有丢失。

**valgrind** 工具不是完美的，它不能发现局部或全局数组的问题，只能发现已分配的内存问题。但是对于发现很多难以定位的问题依然是有用的。

## 方案 42：发现未初始化的变量

**问题：**你的程序在随机的时间做了奇怪的事情。你怀疑一个初始化的变量。

考虑下面的例子：

```
1  #include <iostream>
2
3  static void print_state()
4  {
5      int state;
6
7      if (state) {
8          std::cout << "State alpha" << std::endl;
9      } else {
```

```
10         std::cout << "State beta" << std::endl;
11     }
12 }
```

第 7 行，我们使用 `state` 的值决定做 `alpha` 或 `beta` 的部分。只有一个问题，我们从来没有给 `state` 一个值。

所以，哪个状态会被执行？取决于先前的函数在栈中遗留的是什么。换句话说，`state` 会被设置成一些随机数。

你怎么检测类似的事情？使用调试器检测很难。如果你打印 `state` 的值，得到 38120349，你怎么知道它是正确的或错误的？

**方案：** `valgrind` 来拯救

`valgrind` 程序也检查是否使用了任何未初始化的数据。关于这个，它确实很聪明。如果你将一个未初始化的变量赋给另一个，它不会报错。（当使用 `memcpy` 和其它类似的操作时确实会发生。）

但是，如果你试着使用未初始化的变量做决定（例如一个 `if` 语句的内部），它会大声报错。

例如，当我们在 `valgrind` 中运行前面的程序，得到：

```
... chatter ...
==26488== Conditional jump or move depends on uninitialised value(s)
==26488== at 0x80487DA: print_state() (uinit.cpp:7)
==26488== by 0x804884C: main (uinit.cpp:16)
State alpha
... more chatter ...
```

清楚的告诉我们，第 7 行有一个未初始化数据的问题。

## 方案 29: `valgrind` 的发音

**问题：** 你如何发音 “Valgrind” ？

**方案：** 我们黑客。只要它有效，我们不在乎它是如何发音的。

## 方案 43: 使用 `ElectricFence` 定位指针问题

**问题：** 你需要找到内存分配错误。

**方案：** 不要使用 `ElectricFence`，因为它是落后的。即使它的继承者，`DUMA` 也是过时的。使用 `valgrind` 替代。



## 方案 44：处理复杂的函数和指针声明

**问题：**你需要创建一个指向函数的指针，这个函数带有一个整型参数，返回一个指向函数的指针数组，指针指向的函数带有一个字符串参数，返回一个整数。呼！这个规范你不得不阅读至少三次，否则你无法理解它。

你如何处理这么复杂的东西？

**方案：**重新设计代码，所以你不需要这种类型的函数。

让我们假设世界是疯狂的，我们不能做到。然后我们将声明分离，使用大量的 `typedef` 语句让事情简单。黑客并不意味着以一种最复杂的方式完成最复杂的事情。一个优秀的黑客用一种简单的方式完成最复杂的事情。

让我们以解析句子开始，寻找规范中最底层的元素。基本的条目是一个带有一个字符串参数，返回一个整数的函数。在 C++ 中很容易表示：

```
typedef int base_function(const char* const arg);
```

现在，我们要返回一个指向这种类型函数的指针数组。首先定义这种指针的类型：

```
typedef base_function* base_function_ptr;
```

接着定义一个指针数组：

```
typedef base_function_ptr base_array[];
```

现在，需要一个返回指向这种数组的指针的函数：

```
typedef base_array* big_fun(int value);
```

最后，我们声明一个指向这个条目的指针：

```
big_fun* the_pointer;
```

现在你可能注意到我们使用了 4 个 `typedef` 语句来定义这样的一个指针。我们也可以在一个语句中做到。

但是，这个方案的目标是以最小的花费定义这个指针，不是最少的代码。清晰度是优秀黑客重视的一种品质，将这个问题分解成多个，我们可以使用很多小而明确的语句，小的 `typedef` 语句，替换大而复杂，不可能理解的 C++ 声明。

完整代码（包括注释）如下：

```
// Type definition for a function which takes a character argument and
// returns an interger
typedef int base_function(const char* const arg);

// Pointer to a base function
```

```
typedef base_function* base_function_ptr;

// An array of pointers to our base function
typedef base_function_ptr base_array[];

// Function which returns an array of function pointers
typedef base_array* big_fun(int value);

// Finally a pointer to the thing we wished to point to
big_fun* the_pointer;
```

作为黑客，应该思考盒子之外的，看到超出原始问题的东西。我们没有处理的一个问题是“为什么你需要这样的指针？”

真实世界中，优秀的黑客不会以一种更好的方式设计声明这个指针，他会设计代码，以至于最开始始终不需要这样的指针。消除不需要的复杂性是一个真正伟大黑客的特质。

## 第 3 章：文件方案

方案不只是关于写程序。程序的转换和处理数据。但是数据本身也可以成为方案的主体。

设计好的文件格式是创建优秀程序的一个关键。文件应该容易使用，鲁棒和可扩展。这一章的方案帮助你达到这些目标。

优秀文件设计的关键是尽可能使事情简洁，但不是简单，同时不会陷入愚蠢的前提下，尽可能灵活。它还有助于尽可能透明，以至于任何文件问题可以很容易定位。

作为黑客我们希望尽可能多地使用我们的程序。这意味着设计文件格式有助于人们使用它们。XML 是一个简洁的例子，而且鲁棒的文件格式。简洁到任何人都可以阅读它，同样复杂到你可以用它表示任何东西。

优秀的程序员创造优秀的文件。优秀的黑客创造伟大的文件。

### 方案 45：只要可行，创建文本文件而不是二进制文件

**问题：**从你的程序中需要保存配置文件。你使用二进制文件或文本文件？

**方案：**使用文本。它几乎总是让事情更简单。

UNIX 是黑客设计的完美案例。其中一个关键设计特性是几乎所有的配置文件都是文本，使得系统工作。

对于文本文件有很多的优势。第一个是它们是人类可读的。当你的程序写一个文本文件时，文件的内容可以很容易的检查。你不需要创建一些理想的数据转换器来查看文件。

这表明了文本文件的一个主要优势：透明度。在一个文本文件中，你可以看到发生了什么。同时你可以使用编辑器很容易修改文件。这意味着如果你的程序需要一个配置文件，这个文件是文本格式，然后一个文本编辑器可以用于编辑配置。（我没有说它简单或极好，但是它可以做到。）

同时，如果是基于文本的配置信息，其他人可以容易地编写编辑配置文件的工具。在 Linux 中，你会发现数百个配置网络设置的程序。那是因为网络配置加载到一个文本文件，对每个人来说都很容易访问。

再一次，作为一个黑客的一个目标是使事情尽可能的简单和容易使用。文本文件帮助你做到这些。

当你正在处理大量的数据（视频，图像，数据库，等等）时二进制文件才是有用的。当你处理 100,000,000 条记录或者更多，与文本文件相关的开销会使你的程序变得非常慢。那种情况下二进制文件更好。

但是，对于简单的事情，比如配置文件，设置文件和基本的数据，文本文件是最好的。性能不是一个问题。谁在意启动你的程序花了 0.003 秒而不是 0.001 秒。但是人们在乎清晰度，正确性和交互性，在这些方面，文本文件是最佳的。

### 如何不去设计一个配置系统

一系列糟糕的设计也值得学习。让我们看一个非常常规的加载配置设置的系统。

首先，整个系统中每个程序的所有配置信息加载到同一个地方。这样做有几个问题：

1. 单个疯狂的程序可以意外地摧毁所有设置。（这个问题的解决方案是：不要重新设计，配置系统，提供一个精心设计的备份系统，这样当破坏确实发生时，你可以恢复。）
2. 将文件变成二进制格式，除非使用你提供的 API，否则没有人可以容易地读取。
3. 保持格式神秘，只有使用你提供的工具，让用户编辑配置数据。通过让事情神秘，你可以完全不让任何人编写一个更好的程序。
4. 因为所有的设置加载到同一个地方，因此你没法创建多个配置文件以用于不同的情况。在机器中你限制每个人是同一个配置阶段。
5. 最后，因为配置数据只能加载到本地机器的一个地方，配置分享是不可能的。

我一直在描述的系统是 Microsoft Windows 注册表。从设计的角度来看，这很有吸引力，因为它为优秀的设计师提供了许多机会来问“当他们这样做的时候，他们在想什么？”

## 方案 46：使用魔法字符串识别文件类型

**问题：**你的程序需要用户指定一个配置文件和一个设置文件。但是用户会很容易将两者混淆，对你的程序造成严重破坏。

**方案：**使用魔法字符串

在文件的每种类型开始的时候，简单使用一个著名的字符串来识别它。例如，一个典型的配置文件看起来这样：

```
# TGP Configuration File
solve = true
features = novice
...
```

关于魔法字符串的关键是识别它所属的程序和文件类型。因此，以此字符串开始的文件，并且只以此字符串开始的文件是配置文件。

因此，当需要配置文件的时候，通过强制用户提供一个配置文件可以防止错误。

## 方案 47：为二进制文件使用魔法数字

**问题：**我们的程序以二进制文件作为输入。我们如何确信获得的是正确的二进制文件？

**方案：**在文件的开始放置一个魔法数。

现在，问题是你如何确定魔法数。简单的回答是你创造出来。

做这件事情有很多种不同的方式。更倾向的方法是，想一个描述我正在创建的程序的单词。例如“HACKS”。取前 4 个字符，然后看看这 4 个字符的 ASCII 码：

H	48
A	41
C	43
K	4B

将这些放在一起，得到一个数 0x4841434B。这个数字的问题是，因为它是 4 个 ASCII 字符，文件很容易与文本文件混淆。

所以，让数加上 0x80808080 变成 0xC8C1C3C8。结果是我们的魔法数：

```
// The magic number at the beginning of each hack file
const long unsigned int HACK_MAGIC = 0xC8C1C3C8L;
```

最后一步是将新的魔法数写到文件中，看看是否 Linux 的 file 命令是否可以识别它为一个存在的数。

### XML 文件

XML 文件格式已经被设计用于解决很多关于文件格式的问题。它是结构化的，所以你可以设计相当复杂的东西，它是人为可读的，你可以很容易验证它。

## 方案 48：通过魔法数进行自动字节排序

**问题：**二进制文件不是方便的。在 Sparc 机器上写入的文件不能在 x86 机器上阅读。

**方案：**使用魔法数检测使用不同字节序写的文件，并且纠正它。

例如：

```
// The magic number at the beginning of each hack file
const long unsigned int HACK_MAGIC = 0xC8C1C3CBL;

/// The magic number on a machine with a different byte order
Const long unsigned int HACK_MAGIC_SWITCH = 0xCBC3C1C8L;

// .....
long unsigned int magic;
in_file.read(&magic, sizeof(magic));

if (magic == HACK_MAGIC) {
    process_file();
} else if (magic == HACK_MAGIC_SWITCH) {
    process_file_after_flipping_bytes();
} else {
    throw(file_type_exception("File is not a hack data file "));
}
```

思路是简单的，第一次检查就看出是否是一个正常的文件，然后处理它。

```
if (magic == HACK_MAGIC) {
    process_file();
```

注意到正常的魔法数是 0xC8C1C3CBL。在一个字节序不同的机器上魔法数是 0xC8C3C1C8L。如果检测出有这个值的机器的魔法数，就知道有一个字节翻转文件，然后处理它：

```
const long unsigned int HACK_MAGIC_SWITCH = 0xC8C3C1C8L;
// ....
} else if (magic == HACK_MAGIC_SWITCH) {
    process_file_after_flipping_bytes();
```

**注意：**尽管这个系统有效，你确实必须维护两个版本的文件读取程序。这可能是一个重要的可维护和有风险的问题。你可能想要使用下一个方案替代。

## 方案 49：编写可移植的二进制文件

**问题：**你需要创建和读取可以使用在多种机器上的二进制文件。

**方案：**总是使用“网络字节序”写数据。

为了使数据传输独立于平台，一种叫做“网络字节序”的标准字节序被创建。一些函数被添加到 C 库函数，用来将数据与网络字节序转换。

这包括：

### 函数

### 含义

htonl 转换主机格式中的长整型为网络格式。

htons 转换主机格式中的短整型为网络格式。

ntohl 转换网络格式中的长整型为主机格式。

ntohs 转换网络格式中的短整型为主机格式。

让我们看看当写文件时是如何使用的：

```
// The magic number at the beginning of each hack file
const long unsigned int HACK_MAGIC = 0xC8C1C3CBL;

short int item_count = 15; // Number of items in the file
short int field_length = 11; // Length of next field

// ...
long unsigned int magic = htonl(HACK_MAGIC);
out_file.write(&magic, sizeof(magic));

short int write_item_count = htons(item_count);
out_file.write(&write_item_count, sizeof(write_item_count));

short int write_field_length = htons(field_length);
out_file.write(&write_field_length, sizeof(write_field_length));
```

相似的代码被用于读取的一方，使这些东西可以移植。

## 方案 50：使你的二进制文件可扩展

**问题：**人们想通过添加更多的属性来扩展程序。如果程序处理二进制文件，则文件格式必须适应它。

**方案：**当设计文件时为扩展留下空间。在每条记录中使用包含记录大小的文件格式。

这是一个简单的文件格式规范：

1. 记录类型（4 字节整数）
2. 记录长度（4 字节整数）
3. 记录数据（长度-8 字节数据）

这是一种看似简单的格式。尽管简单，但仍有很大的扩展空间。

让我们从读取记录的代码开始。它必须读取前 8 个字节，获取长度，然后读取剩下的数据。整个记录返回给用于处理的读取器。非常简单。优秀的方案。

现在，当我们需要扩展和增强数据流时会发生什么。我们所要做的只是添加一个新的记录类型。读取器不需要发生变化。它仍然可以处理记录，因为它不需要知道类型。

我们所要做的就是往主程序中添加一个新的记录处理程序。例如：

```
struct record {
    int type;    // Record type
    int length;  // Length of record
    uint8 data[0]; // Data (variable length);
};

// ....
switch (record_var.type) {
    case REC_START:
        do_start(record_var);
        break;
    case REC_STOP:
        do_stop(record_var);
        break;
    case REC_PAUSE:
        do_pause(record_var);
        break;
    default:
        std::cout << "WARNING: Unknown record type " <<
```



```
        record_var.type << " Ignored" << std::endl;
        break;
    }
```

带有这样的代码，很容易为新的记录添加处理程序。新的处理程序很容易适应这个模式。

而且，这个代码对于从前或以后文件的版本也是有效的。对于以后的版本有效是因为跳过了它不知道的记录。如果我们添加一个像 `REC_SKIP` 的新的记录类型，程序依然会运行。它不会处理它不知道的记录，但它同样不会崩溃。

这种内置的弹性是真正的出色方案的标志。

## 方案 51：使用魔法数保护二进制文件记录

**问题：**文件可以被破坏。我们如何保护自己免受伤害。

**方案：**在每条记录的开始和结尾放置魔法数。例如：

```
struct record_start {
    uint32 magic_start;
    uint32 type;
    uint32 size;
};

struct record_end {
    uint32 magic_end;
};
```

在记录代码中的某处有下面的语句：

```
if (record_stream.start.magic_start != START_MAGIC)
    throw(file_corruption("Starting magic number wrong"));

// ....
if (record_stream.end.magic_end != END_MAGIC)
    throw(file_corruption("Ending magic number wrong"));
```

现在，当任何带有这种文件的程序，我们就会注意到。应该能够证明这种保护能够对抗几乎所有的由硬件错误或程序错误引起的简单数据破坏。

而且它只保护数据的结尾。如果有人讲中间部分混乱，它不会被识别。如果你想要识别，你需要在数据本身中包含校验和。但是那是偏执的层面，很少需要。在很多情况中，对我自己来说这种方案被证明是有用的。首先是一个文件系统的问题。似乎当计算机关闭和文件被写进磁盘，操作系统会在未写入的部分使用包含零的块填充。幸运的是，魔法数在那里，程序意识到它已经读取了一些东西，

是记录的一半，其他东西的一半，丢弃了数据。

这个系统引起的其他问题是真正讨厌的数据破坏错误。问题甚至可以在完全不同的模块中追溯代码，当一个错误条件出现的时候，发送 E-Mail 会发出警告。这是代码。让我们看看你如何识别问题：

```
pid_t child_pid = fork();

if (child_pid == 0) {
    // We are the child
    system("send_email_alert");
    exit(0);
}
```

如果你不能识别错误，我可以理解。它花了我一周的时间测试查明代码，然后定位问题。

问题很难发现。毕竟整个模块都没有写模块记录的代码。特别是输出文件句柄在记录编写器中根本不存在。

而且问题中的代码没有 I/O。它甚至没有操作内存，所以它不可能是内存破坏。所以发生了什么？

这个问题有两个方面。第一个是这个调用：

```
pid_t child_pid = fork();
```

这创建了一个主进程的副本。所有的文件句柄现在在两个进程之间共享。所以，在带有输出文件打开的进程之前，现在有两个。

fork 调用也复制了父进程的内存。这包括所有文件的 I/O 缓存。包括输出文件。

接下来我们到这行：

```
exit(0);
```

这样做确实关闭了程序，是吗？不准确。让我们看看这个函数的文档：

```
The exit() function causes normal program termination and the the value of status & 0377 is returned to the parent (see wait(2)). All functions registered with atexit() and on_exit() are called in the reverse order of their registration, and all open streams are flushed and closed. Files created by tmpfile() are removed.
```

**exit** 函数确实做了很多。尤其是它刷新了所有写打开文件的写缓冲。由于我

们从父进程继承了一组打开的文件及其缓冲区，因此它们会被刷新。然后这个调用将一些数据写入文件。当缓冲区满的时候主进程也会写一些数据到文件中。由于这两个写入不是一致的，所以数据会被破坏。

这是发现的一个讨厌的问题。但实际上代码是用记录保护写的，捕获问题是非常容易的。早期很明显，打开警告设置，会导致程序对记录破坏报错。唯一困难的部分是查找代码，然后尝试修复问题。

使这个问题讨厌的是，日志被破坏，但是只有在警报打开的时候。但是这些模块没有共同点。共享部分没有函数、逻辑或内存。需要花大量时间查明它们共享的东西：I/O 缓冲，但是这种共享是非常隐蔽的。

## 方案 52：知道什么时候使用\_exit

**问题：**你正在分离进程并执行一些工作，然后退出，没有在进程中做任何关于 I/O 缓冲的事情。你不能使用 `exit` 函数，因为在方案 51 中描述的问题。

**方案：**使用 `_exit`。

`_exit` 函数停止你的程序。这是它做的所有事情，停止程序。它没有通过，它不收取 200 美元。但是更重要的是，它不会刷新任何 I/O 缓冲区。调用 `_exit` 时发生的一切就是你的进程消失了。

当分离一个已经完成、需要停止的进程时，这是非常有用的。记住 `exit` 刷新东西，可能引起一些共享文件的问题。`_exit` 调用避免了这个问题。

## 第 4 章：调试方案

我已经编写了两个没有错误的程序。其中一个是三个长整型的机器指令，另一个是四个的。所有其它的都有错误。

编写代码只花费很短的时间。调试一直持续。这里展示调试方案，用于使调试更快和更有效。

### 方案 53：使用特别的字符标记临时的调试消息

**问题：**调试技术中，通过添加打印语句仍然是一个非常强大的。但是你怎么识别，用于一个特殊问题的输出信息的语句 vs 打印的就是应该在那的语句。

**方案：**在所有临时的调试输出中以字符“##”开始。例如：

```
i = find_index();  
std::cout << "## find_index returned " << i << std::endl;
```

“##”提供了几个目的。首先，它标识了一个临时调试的语句。接下来当你发现问题时，很容易删除它们。所有你需要做的就是遍历，寻找包含“##”的行，然后删除。

### 方案 54：使用编辑器分析输出日志

**问题：**当你打开显示日志时，你得到了占满我滚动屏幕的 5,000 行，我不能找到我想要查找的。

**方案：**将日志输出保存到一个文件，使用一个文本编辑器浏览输出。

日志文件的问题是，它提供给你很少或很多的信息。很少是很容易处理的-所有你需要做的就是打开显示，直到你获得很多或者带有##行的。（参考上一个方案。）

处理很多的信息是另一个问题。你怎么找到你想要的信息块。

在很多的系统中，存在一个用于搜索和操作大的文本文件的工具。叫做系统编辑器。让我们看看实际上是如何工作的。

首先，假如你已经对系统进行了详尽的测试并记录了结果。在超过 5000 次的测试中，有 3 次带有错误的失败。简单的启动你的编辑器，并搜索字符“ERROR”。你会定位第一个错误的行。

想要查看错误发生之前的事情，向上滚动查看。

编辑器不仅让你查看数据，而且还有注释。你可以添加注释，当你计算出发生了什么，你可以在日志中添加注释。不像纸质的笔记，你想要在网络上寻求帮助的情况下，可以将文件插入到 E-Mail 中。

日志文件是一个很大的信息源，文本编辑器是一个利用这些信息的好的方式。

（参考方案 127，如何使用 Vim 检查日志文件。）

## 方案 55：灵活的日志

**问题：**当你处理小程序时，记录一切是 OK 的。如果你有一个大型的程序，你需要有选择性。

**方案：**使用基于字母的日志选择。

我们想要建立一个命令行参数 `-v<letters>`，只有被 `<letters>` 指定的部分才会被调试。例如一个典型的字母集可能是：

m 记录内存分配和释放

d 例如词典入口

f 打印函数定义

r 打印常见的表达式调试信息

x 显示执行函数的调用

...等等

这个系统很容易实现。首先，我们定义一个数组保存调试选项：

```
static bool verbose_letters[256] = {0};
```

下一步，我们建立一个循环来处理命令行参数。这个例子中，我们使用 GNU 的 `getopt` 函数扫描参数。我们使用指定的参数 “`v::`” 来指出唯一的选项是 `-v`，其他的选项参数也可以这样做。（在 “`v`” 之后用 “`::`” 表示。）

```
while (1) {  
    int opt = getopt(argc, argv, "v::");
```

现在，我们处理选项。指定 `-v` 有两种可能的方式。第一种是只有 `-v`。那种情况下，我们用下面这行打开所有的调试信息：

```
memset(verbose_letters, -1, sizeof(verbose_letters));
```

如果选项有参数，然后我们只设置给出选项的字母：

```
for (unsigned int i = 0; optarg[i] != '\0'; ++i)
    Verbose_letters[static_cast<int>(optarg[i])] = true;
```

将这些放在一起，我们得到了下面处理显示选项的代码：

```
switch (opt) {
    case 'v':
        if (optarg == 0) {
            memset(verbose_letters, -1, sizeof(verbose_letters));
        } else {
            for (unsigned int i = 0; optarg[i] != '\0'; ++i)
                verbose_letters[static_cast<int>(optarg[i])] = true;
        }
        break;
    // ...process other options
```

检测是否需要发出调试消息很简单。例如检测 malloc 日志，我们使用代码：

```
if (verbose_letters['m']) {
    std::cerr << "Doing a malloc of " << size << bytes;
}
ptr = malloc(size);
```

实际的实践中，我们可以为'm'定义一个常量，但是对于这个短的例子，我们采用不好的编程实践。

这种记录什么的指定方式使用所有的小写字母，所有的大写字母，所有的数字，来选择记录什么。如果这还不够，大量的标点字符也可以使用。

（如果这还不够，那么你的程序可能太大，而无法进行调试。）

两个特性使得这是一个优秀的方案。首先它是简单的。第二它非常灵活。换句话说，它是解决复杂问题的简单的方法，而简单的解决方案总是优秀的。

## 方案 56：使用信号打开和关闭调试

**问题：**有时但不是所有时间，你想调试输出。

**方案：**使用一个信号打开和关闭调试

第一步是定义一个信号句柄，切换调试标志：

```
static void toggle_debug(int) {
    Debug != debug;
}
```

接下来我们将其与一个信号连接，这个例子中是 SIGUSR1。

```
signal(SIGUSR1, toggle_debug);
```

现在，必须做的是在 debug 标志上创建记录条件：

```
static void log_msg(const char* const msg) {  
    if (debug) {  
        std::cout << msg << std::endl;  
    }  
}
```

现在所有必须做的是使用命令打开调试：

```
$ kill -USR1 pid
```

关闭它也是一样的。

当你在一个特定的时间必须关闭调试输出时，这是有用的。当你第一次启动程序时，很容易打开它，然后让它开着。

但有时你需要检测一个程序的主循环，这个程序在运行四天之后会被神秘的挂起。因为对于日志文件，你没有几 TB 的可用磁盘空间，因此需要在程序挂起之后打开调试，以便查明正在发生的事情，这是这个方案最有用的地方。

## 方案 57：使用一个信号文件打开和关闭调试

**问题：**你需要打开和关闭调试，信号是不实际的。

**方案：**使用一个特殊的文件触发调试。

例如：

```
static void log_msg(const char* const msg) {  
    if (access("/tmp/prog.debug.on", F_OK) != 0) {  
        std::cout << msg << std::endl;  
    }  
}
```

现在，你为了打开调试所需要做的是建立一个文件/tmp/prog.debug.on。为了关闭调试，简单地移除文件。

这个方案应该只应用于，你绝对需要打开和关闭调试，而当程序正在运行时，你可以使用信号（方案 56）。访问系统调用代价很大，而且，如果经常使用，会使你的程序变慢。所以，尽管这种方案是有用的，意识到它的限制。

## 方案 58：发生错误时自动启动调试器

**问题：**你的程序已经检测到一个内部问题，需要进行调试。但是程序不在被调试。你如何让程序解决这个问题？

**方案：**定义一个可以在一个运行着的程序中启动调试器的函数。

**注意：**下面的代码是专门针对 Linux 的。如果你正在使用一个类 UNIX 的系统，它应该很容易适应你的系统。如果你正在运行 Microsoft Windows，那就做属于你自己的。

程序的基本思路是，当程序需要启动调试器（`dbg`），`debug_me` 函数被调用，然后附加在正在运行的进程。听起来很简单，但是有一些细节需要思考。

首先，让我们看看，告诉 `gdb` 启动调试进程需要什么。最初的 `gdb` 命令是：

1. `attach <pid>` - 将调试器附加在正在被调试的程序。
2. `echo "Debugger gdb started\n"` - 让用户知道发生了什么。
3. `symbol /proc/<pid>/exe` - 告诉 `gdb` 在那里寻找进程的符号表
4. `break gdb_stop` - 在一个好的停止点处停止。
5. `shell touch <flag-file>` - 告诉程序 `gdb` 被附加。（稍后会详细介绍）
6. `continue` - 继续执行，然后在正确的地方暂停。

第一个 `gdb` 命令：

```
attach <pid>
```

将调试器附加到程序中，（`<pid>` 替换为调试程序的进程 `id`。）调试器现在控制着程序。实际上，如果我们处于一个极简主义的心态，我们会停在这。

但是调试会话处在一个未准备好的状态。符号表还没有加载，不知道是否我们在一个正确的线程或一个已知的位置暂停。所以执行更多的命令让事情变得更好一点。

下一个命令：

```
echo "Debugger gdb started\n"
```

输出一个打招呼的消息。这样就开始了调试进程。

接下来，加载符号表。为了这个，需要知道程序文件名。查找的一种方式告知查看的程序名，然后对执行文件进行路径搜索。但是对于可执行文件，Linux 友好地从 `/proc/<pid>/exe` 中提供了符号链接，所以只是利用这个特性加载符号表。

```
symbol /proc/<pid>/exe
```

现在，告诉 `gdb` 暂停在一个好的位置。实际上已经定义了一个好的暂停的位置，叫做 `gdb_stop`，所以在这设置一个断点。

```
break gdb_stop
```

当调试器被附加到程序上，程序暂停。问题是不知道程序暂停的位置。可能



是通过 `debug_me` 调用的 80 层的一些函数。更糟糕的是，可能正在处理一个线程级的问题。那种情况下，甚至不能在引起错误的线程中暂停。

这个问题的方案是在已知的位置（`gdb_stop`）设置一个暂停，然后告诉调试器继续。当在 `gdb_stop` 暂停，知道位于哪里，并且确信是在正确的线程中。

`debug_me` 启动 `gdb` 之后，它等待调试器启动。实现下面的循环做到这些：

```
96:   while (access(flag_file, F_OK) != 0)
97:   {
98:       sleep(1);
99:   }
```

这个循环等待一个标志文件被创建。只要它显示，程序就知道调试器正在运行，然后程序继续。

为了建立标志文件，向 `gdb` 发出下面的命令：

```
touch <flag_file>
```

最后，告诉 `gdb` 继续。直到程序到达 `gdb_stop`，程序在这个点会持续一会儿。

所以，做这个工作的代码列在方案结尾的整个 `debug_me.c` 模块。大多数情况下，将命令输入命令文件是一个字符串处理问题。

现在讨论 `gdb` 命令的实际调用。理想情况下，应该只使用系统调用来执行命令：

```
gdb --command=<command-file>
```

这个例子中，`<command-file>` 替换为包含上面列出的命令的一个临时文件。但是并不是这么简单。永远不是。

我们的程序是在后台运行的守护程序。它没有标准的输入输出。如果我们启动 `gdb`，将没有可以键入命令的终端。

这个问题的一种解决方案是启动我们自己的终端窗口。这使用下面的命令：

```
xterm -bg red -e gdb -command=<command-file>
```

这样就启动了一个我们的调试器运行新的 `xterm` 程序。使用选项 `-bg red`，将背景设置为红色。使用红色是因为它会引起我们注意。而且死亡的红色屏幕比死亡的蓝色屏幕听起来更好。

最后，通过 `-e` 选项告诉 `xterm` 执行 `gdb` 命令。

现在，我们做的所有事情，让我们看到这个函数是如何在程序中应用的。这有一些处理无论是黑色还是白色的变量的代码（至少在正常，理智的情况下）：

```
#include "debug_me.h"
```

```
// ....
switch (black_or_white) {
    case BLACK:
        do_black();
        break;
    case WHITE:
        do_white();
        break;
    default:
        std::cerr << "INTERNAL ERROR: Impossible color" << std::endl;
        debug_me();
        break;
}
```

这种情况下，变量 `black_or_white` 经历了合理的测试。如果事情是疯狂的，我们启动调试器。

**注意：**这仅适用于内部使用的程序。如果你给顾客一个没有源代码的程序，这个系统可能不是那么有用。

随后是 `debug_me.c` 文件的完整源代码

```
1:  /*****
2:   * debug_me -- A module to start the debugger *
3:   * from a running program. *
4:   * *
5:   * Warning: This code is Linux specific. *
6:   *****/
7:  #include <stdio.h>
8:  #include <unistd.h>
9:  #include <sys/param.h>
10: #include <stdlib.h>
11:
12: #include "debug_me.h"
13:
14: static int in_gdb = 0; // True if gdb started
15:
16: /*****
17:  * gdb_stop -- A place to stop the debugger *
18:  * *
19:  * Note: This is not static so that the *
20:  * debugger can easily find it. *
21:  *****/
22: void gdb_stop(void)
23: {
24:     printf("Gdb stop\n");fflush(stdout);
```

```
25: }
26:
27: /*****
28:  * start_debugger -- Actually start *
29:  * the debugger *
30:  *****/
31: static void start_debugger(void)
32: {
33:     int pid = getpid(); // Our PID
34:
35:     // The name of the gdb file
36:     char gdb_file_name[MAXPATHLEN];
37:
38:     // File that's used as a flag
39:     // to signal that gdb is running
40:     char flag_file[MAXPATHLEN];
41:
42:     // The file with the gdb information in it
43:     FILE *gdb_file;
44:
45:     // Command to start xterm
46:     char cmd[MAXPATHLEN+100];
47:
48:     if (in_gdb)
49:         return; /* Prevent double debugs */
50:
51:     /*
52:      * Create a command file that contains
53:      * attach <pid> # Attaches to the process
54:      * echo .... # Echoes a welcome message
55:      * symbol /proc/<pid>/exe
56:      * # Loads the symbol table
57:      * break gdb_stop # Set a breakpoint in
58:      * shell touch /tmp/gdb.flag.<pid>
59:      * # Create a file that tells us
60:      * # that the debugger is running
61:      * continue # Continue the program
62:      */
63:     sprintf(gdb_file_name, "/tmp/gdb.%d", pid);
64:     gdb_file = fopen(gdb_file_name, "w");
65:     if (gdb_file == NULL)
66:     {
67:         fprintf(stderr,
68:             "ERROR: Unable to open %s\n",
```

```
69:         gdb_file_name);
70:         abort();
71:     }
72:     sprintf(flag_file, "/tmp/gdb.flag.%d", pid);
73:     fprintf(gdb_file, "attach %d\n", pid);
74:     fprintf(gdb_file, "echo "
75:         "\"Debugger gdb started\\n\\n\"");
76:
77:     fprintf(gdb_file, "symbol /proc/%d/exe\n",
78:         pid);
79:
80:     fprintf(gdb_file, "break gdb_stop\n");
81:
82:     fprintf(gdb_file, "shell touch %s\n",
83:         flag_file);
84:
85:     fprintf(gdb_file, "continue\n");
86:     fclose(gdb_file);
87:     /* Start a xterm window with the
88:      * debugger in it */
89:     sprintf(cmd, "xterm -fg red "
90:         "-e gdb --command=%s &",
91:         gdb_file_name);
92:     system(cmd);
93:
94:     /* Now sleep until the debugger starts and
95:      * creates the flag file */
96:     while (access(flag_file, F_OK) != 0)
97:     {
98:         sleep (1);
99:     }
100:     in_gdb = 1;
101:     gdb_stop();
102: }
103:
104: /*****
105:  * debug_me -- Start the debugger *
106:  *****/
107: void debug_me(void)
108: {
109:     start_debugger();
110:     gdb_stop();
111: }
```

## 方案 59：启动调试器使断言失败

**问题：**一个正常的失败断言只是打印消息和退出程序。当涉及到发现问题时，这非常无用。如果失败的断言可以启动调试器，就非常好了。

**方案：**将断言失败与 `debug_me` 结合。

我们需要做的第一件事情是，如果断言失败，确定调用的函数名字。这因系统而异。

为了查明断言是如何工作的，我们建立一个小的测试程序：

```
#include <cassert>

assert("We are failing" != 0);
```

接下来，通过预处理器运行程序，看看输出。这是使用 `gcc -E` 命令的一个例子。有趣的行是：

```
# 2 "assert.cpp" 2
(static_cast<void> ( __builtin_expect (!("We are failing" != 0), 1) ? 0 :
(__assert_fail ("\"We are failing\" != 0", "assert.cpp", 3,
__PRETTY_FUNCTION__), 0)));
```

从这我们可以看出，当一个断言被触发，函数 `__assert_fail` 被调用。更前面的输出中，编译器甚至足够友好地为我们提供了原型：

```
extern "C" {
    extern void __assert_fail (
        __const char *__assertion,
        __const char *__file,
        unsigned int __line,
        __const char *__function)
        throw() __attribute__((__noreturn__));
}
```

现在我们需要做的是，提供调用 `debug_me` 的 `__assert_fail` 版本。

```
/* *****
 * __assert_fail -- Called when an assert fails. *
 * Starts the debugger. *
 * *
 * Note: gcc specific. Different compilers use *
 * different internal routines to handle bad *
 * asserts. *
 * *****/
void __assert_fail(
    const char *const what,
```

```
const char *const file,  
const int line,  
const char *const funct  
)  
{  
  
    printf("Assert failed: %s\n", what);  
    printf("FAILURE at: %s:%d\n", file, line);  
    printf("Function is %s\n", funct);  
    debug_me();  
    abort();  
}
```

关于这个函数需要注意的一件事情是在 `debug_me` 之后调用 `abort`。这是为了防止粗心的程序员在调试会话中键入继续并键入明显失败的程序。

### **debug\_me 的历史**

`debug_me` 函数最初编写用于调试一个移动手机模拟程序。程序已经禁止很长时间，因为它是极度有问题。当我得到这个极度有问题的程序时，非常过时。

程序使用线程进行扩展，经常一个线程会崩溃。找到那个是引起问题的线程是非常困难，就算 `gdb` 线程相关的命令在这个系统上也没用。所以我发明了 `debug_me` 来帮助解决这个问题。

该程序有趣的一个方面是不寻常的失败模型。线程经常会触发断言失败，然后启动调试器。然而，当调试器启动时，另一个线程会失败，然后尝试启动调试器。

任何程序员可以编写引起断言失败和退出程序的代码。一个真正的黑客需要在单次运行中让多个断言失败。

## **方案 60：在正确的地方暂停程序**

**问题：**你知道程序在 487 次的迭代过程中会失败。如果你添加一个断点，你在得到问题之前不得不键入 586 次继续。

**方案：**构建一个调试点。

当你想要暂停程序的时候，一些调试器会很难做到，像 487 次经历循环。所以如何解决这个限制？

只是为调试器放置一些临时的代码。例如：

```
#ifndef QQQ
void debug_stop() {
    // The debugger can stop here
}
#endif /* QQQ */
// ...
for (i = 0; i < 5000; ++i) {
#ifdef QQQ
    if (i == 487) debug_stop();
#endif /* QQQ */
    // Problem code follows
}
```

现在，我们所要做的就是启动调试器，在 `debug_stop` 处设置一个断点。在问题发生之前，将调用此过程，然后将我们引入调试器。然后，我们可以单步执行程序，直到发现错误。

关于这份代码有几个事情需要注意。首先，`debug_stop` 是一个全局函数。这样的原因是有些调试器花费很长的时间茶盏静态函数。

在我们的调试代码中使用 `#ifdef QQQ` 的原因在方案 72 中描述。基本上，那是一种识别临时代码最容易的方式。

## 第 5 章：注释和导航方案

当我在大学中学习计算机科学时，接下来到来的伟大事情是排版程序的技术，然后打破了随侯我们在程序中使用的等宽字体。这是三十年前。

今天我们仍然使用等宽的 **ASCII** 字符。没有排版，没有图形-三十年前这些也没有。（语法高亮可能除外。）

然而，黑客从来不让不合适的技术阻碍我们。这章中，我们列举了一些绕过编程环境限制的方案。

### 方案 61：创建带有注释的标题

**问题：**文件格式不支持辩题，那么如何分离你的标题。

**方案：**这里有一些技巧处理你的注释。首先，标题可以加下划线或者双下划线。

```
/*
 * A Important Heading
 * -----
 *
 * Here we put a paragraph discussing the stuff we've
 * headlined above.
 *
 * A Very Important Heading
 * =====
 *
 * Headings like the one above can be used to denote
 * very important sections.
 *
 * +-----+
 * | This is almost shouting |
 * +-----+
 *
 * Now here's something that really stands out.
```

还有其他的指示主要分节的方式。例如，你可以创建一个跨越整行的标题。

```
/*  
    * <<<<<<<<<<<<<< I/O Section Follows >>>>>>>>>>>>  
*/
```



最后，我们有一个确实重要的节标题：

```

/*****
***** WARNING *****/
***** WARNING *****/
***** WARNING *****/
*****/

/*
 * Do not attempt to drive heavy equipment while reading
 * this book.
 */

```

## 方案 62：强调句子中的单词

**问题：**上面是标题的工作，但是当你想使用带有段落的排版单词时，该怎么做。

**方案：**使用“\*”创建，其他的字符可以帮助强调单词。

例如：

```

/*
 * Some people consider all capitals to be SHOUTING!
 *
 * But there are other ways to **emphasize** a word.
 * More that <<one>> in fact. After all there are a
 * number of ==fun== characters you can play around with.
 */

```

## 方案 63：在注释中放置图

**问题：**在程序编辑器中没有画图功能

在注释中放置形状和图表是不可能的。

**方案：**使用 ASCII 艺术。

只使用 ASCII 字符可以绘制原始的图形。例如，下面的注释记录了 GUI 的布局：

```

/*
 * +-----+
 * | Name: _____ |
 * | Address: _____ |
 * | City: _____ |
 * +-----+
 * |<- NAME_SIZE ->|
 * |<--- BLANK_SIZE ---->|
 */

```

```
*/
```

在这个例子中，我们不仅创建了图形，而且同时记录了用于记录维度的常量 `NAME_SIZE` 和 `BLANK_SIZE`。

## 方案 64：提供用户文档

**问题：**你需要为你的程序提供用户文档

Oualline 的文档法规定是：文档的 90% 时间是丢失的。在剩下的 10% 中，有 9% 的时间数据是完全没用的。剩下的 1% 时间，有正确的文档和正确的用日文写的文档修订版本。

**方案：**使用 Perl 的 POD 文档系统。

POD（普通旧文件）格式是简单的，但是是一种相当强大的在文档中嵌入文档的方法。在你程序开始的部分，你可以很容易地放置一个 POD 文档。

例如：

```
/*
 * The following is the documentation for the program.
 * Use the
=pod
=head1 NAME
solve_it - Solve the worlds problems
=head1 SYNOPSIS
    solve_it [-u] problem
=head1 DESCRIPTION

The l<solve_it> program solves all the worlds problem. Just input the
problem on the command line and the solution will be output.

=head1 OPTIONS
The l<solve_it> takes the following options:
=over 4
=item B<-u>
Solve the problem for the universe, not just the world.
=back
=head1 LIMITATIONS
Currently totally unimplemented.
=head1 AUTHOR
Steve Oualline, E<lt>oualline@www.oualline.comE<gt>.
=head1 COPYRIGHT
```

```
Copyright 2007 Steve Oualline.  
This program is distributed under the GPL.  
  
=cut  
*/
```

现在，你可以通过一个像 `pod2text` 的 POD 转换器运行你的源代码，获得一个文档的格式化版本。

#### NAME

`solve_it` - Solve the worlds problems

#### SYNOPSIS

`solve_it [-u] problem`

#### DESCRIPTION

The `*solve_it*` program solves all the worlds problem. Just input the problem on the command line and the solution will be output.

#### OPTIONS

The `*solve_it*` takes the following options:

`-u` Solve the problem for the universe, not just the world.

#### LIMITATIONS

Currently totally unimplemented.

#### AUTHOR

Steve Oualline, <oualline@www.oualline.com>.

#### COPYRIGHT

Copyright 2007 Steve Oualline. This program is distributed under the GPL.

其他的转换器包括 `pod2html`，`pod2man`，也有其它的。

## 方案 65：文档化 API

**问题：**你怎么文档化以一种很好的方式编写的库的 API。

和用户文档一样，Oualline 的文档规定也适用于 API 文档。

**方案：**使用 `doxygen` 自动地从代码的特殊注释中产生文档。

此工具允许将特殊格式的注释用于你的代码中的内部文档。这是一个简短的例子：

```
/**  
 * \brief The basic report specification  
 *  
 * This class completely describes all the parameters  
 * needed to produce a report  
 */
```

```
class ReportSpec {
    /** Name of the report */
    public:
        const char* const name;
    /**
     * Add a new report parameter
     *
     * \param name Name of the parameter
     * \param value Value of the parameter
     */
    void add_param(
        const char* const name,
        const char* const value
    );
};
```

当通过 Doxygen 运行时，会生成可以在任何浏览器查看的 HTML 文档页：

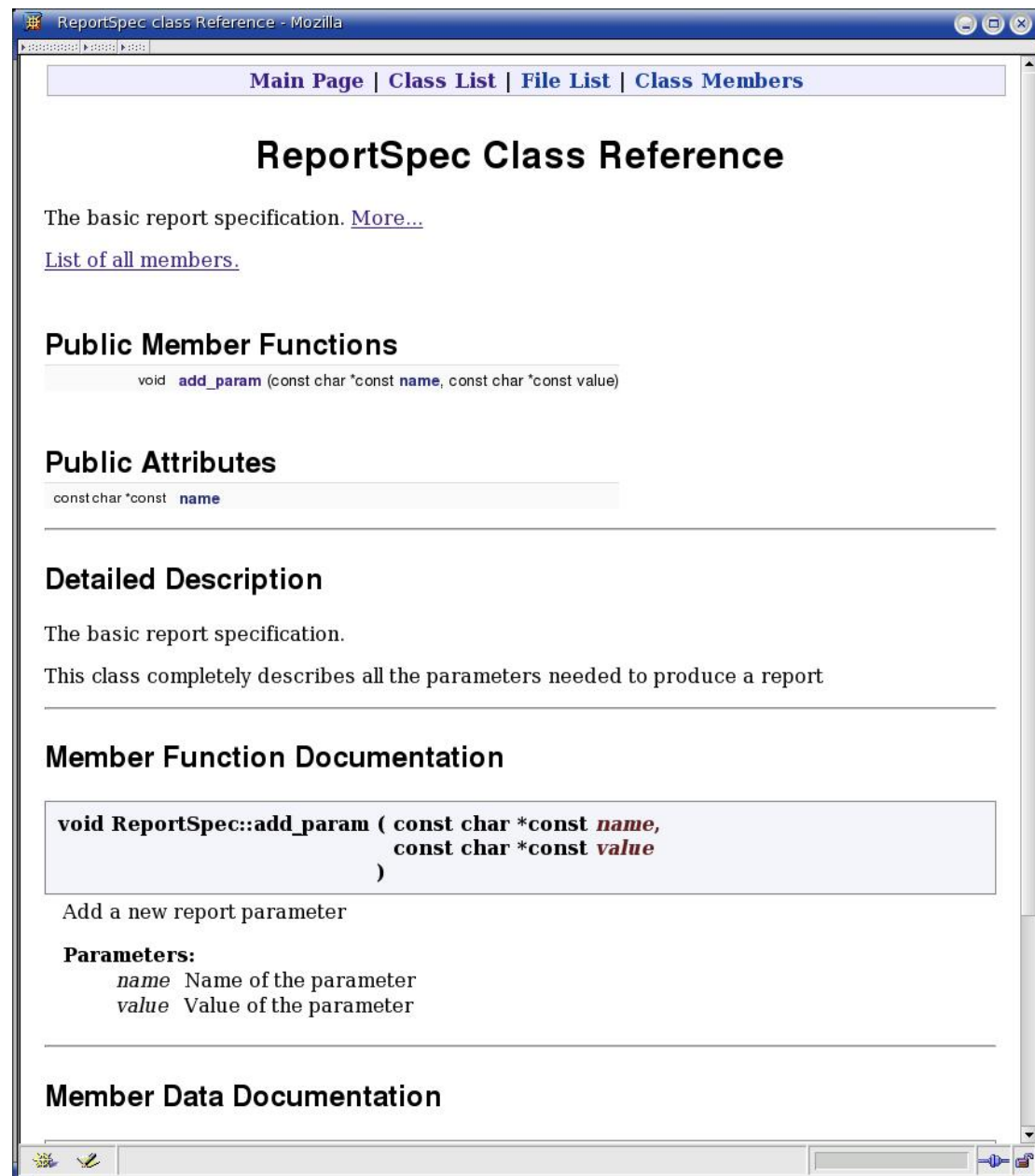


图 1: Doxygen 作为 HTML 文档

如果你想遵循惯例并正确编写注释, 这个系统确实提供了一个用于文档化程序的优秀系统。另一方面, 如果你正在处理大量的代码, 不要这样做, 查看下一个方案。

## 方案 66: 使用 Linux 交叉引用来导航大型代码项目

**问题:** 你正在处理一个具有糟糕设计的 5,000,000 行代码的项目, 你迷失了。

**方案:** 使用 Linux 交叉引用来产生你的代码的交叉引用版本。

Linux 交叉引用(lxr)是一个生成基于超文本代码标记的程序。你可以从

<http://lxr.linux.no/>获取一份拷贝。

它有一些好的特性，例如可以定位代码中的任何标记，显示使用的它的每一处地方。图 2 展示系统用于浏览文件。

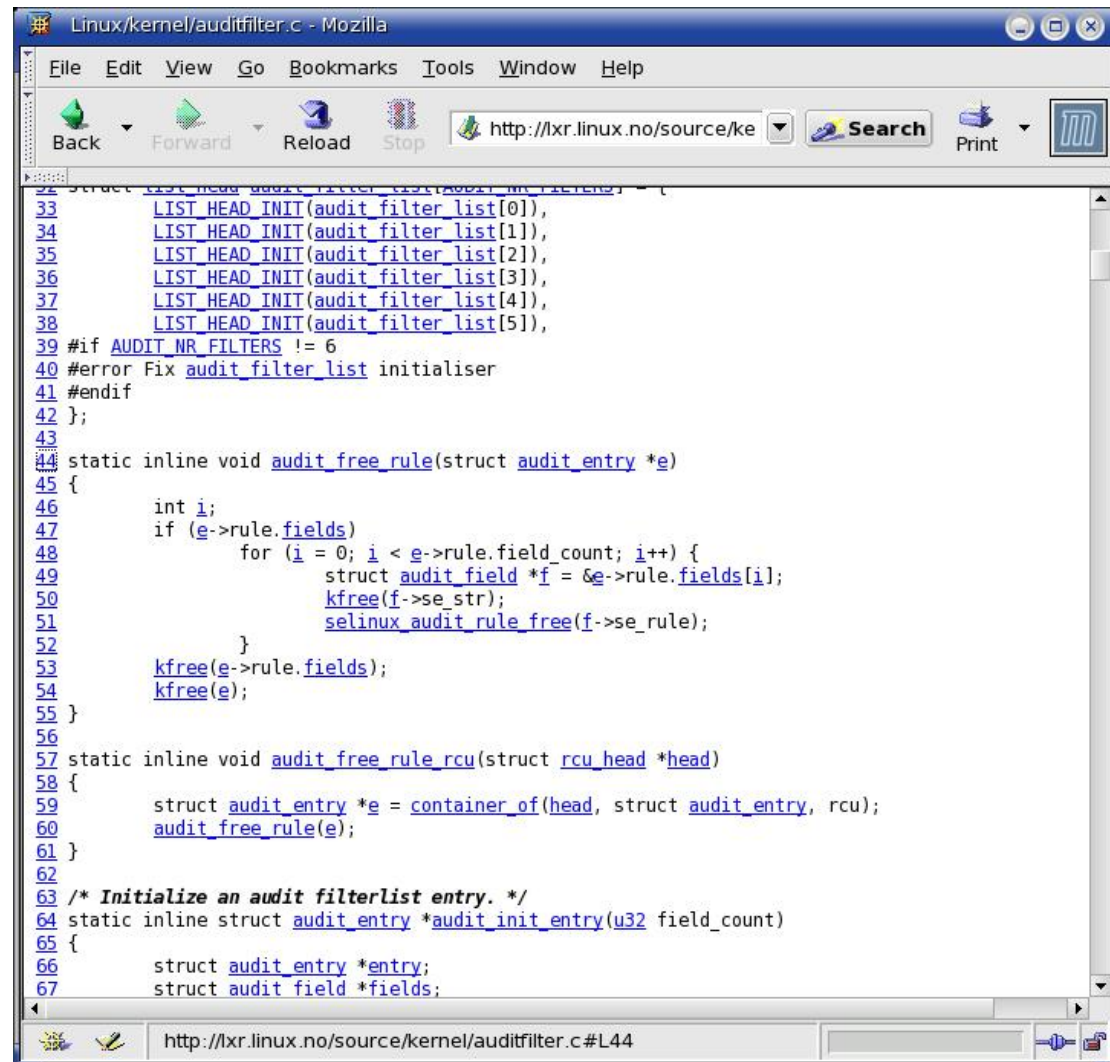


图 2: Linux 交叉引用 (文件查看)

第 44 行，有一个使用 `audit_entry` 结构的函数。我们想知道什么是 `audit_entry`。单击标识将转到一个列出使用标识的每个位置的页面。(参见图 3。)

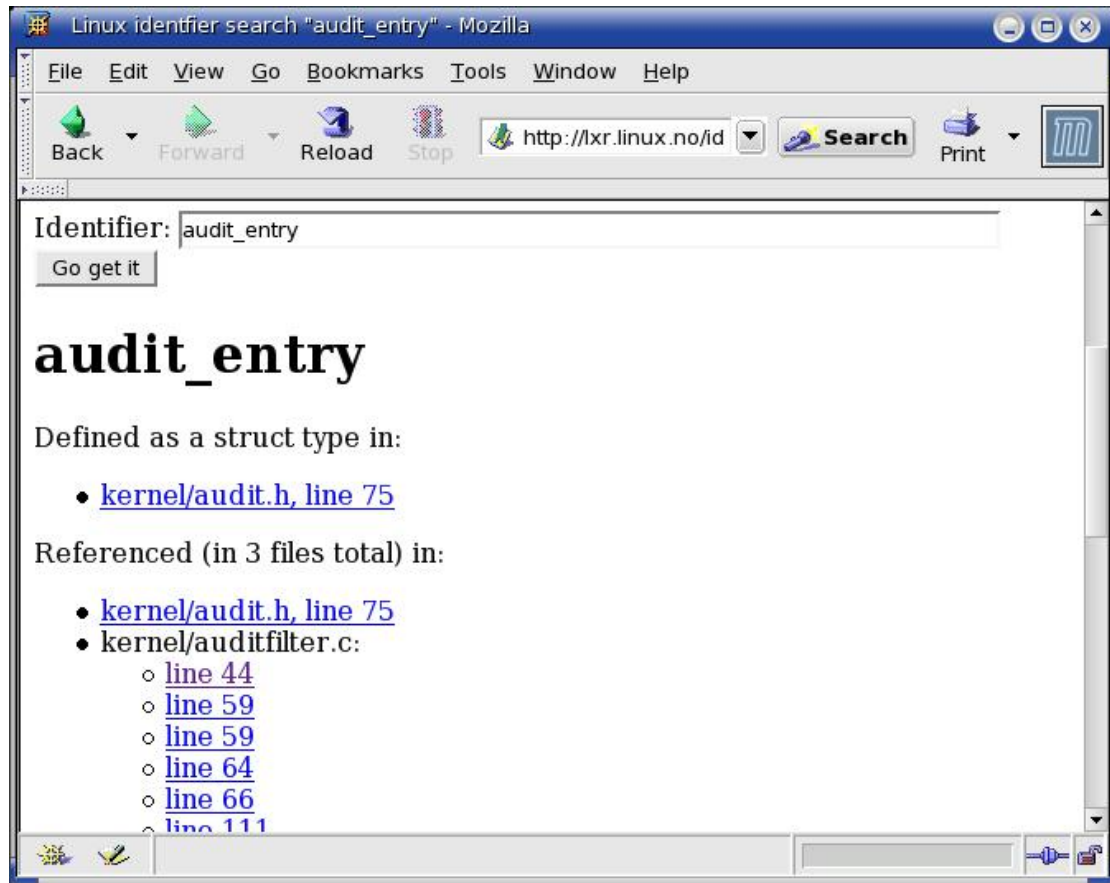


图 3：标识使用页面

这显示，使用标识被定义在文件 `kernel/audit.h`（75 行），使用在文件 `kernel/auditfilter.c` 的很多地方。点击第一个入口，给我们图 4 中展示的定义。

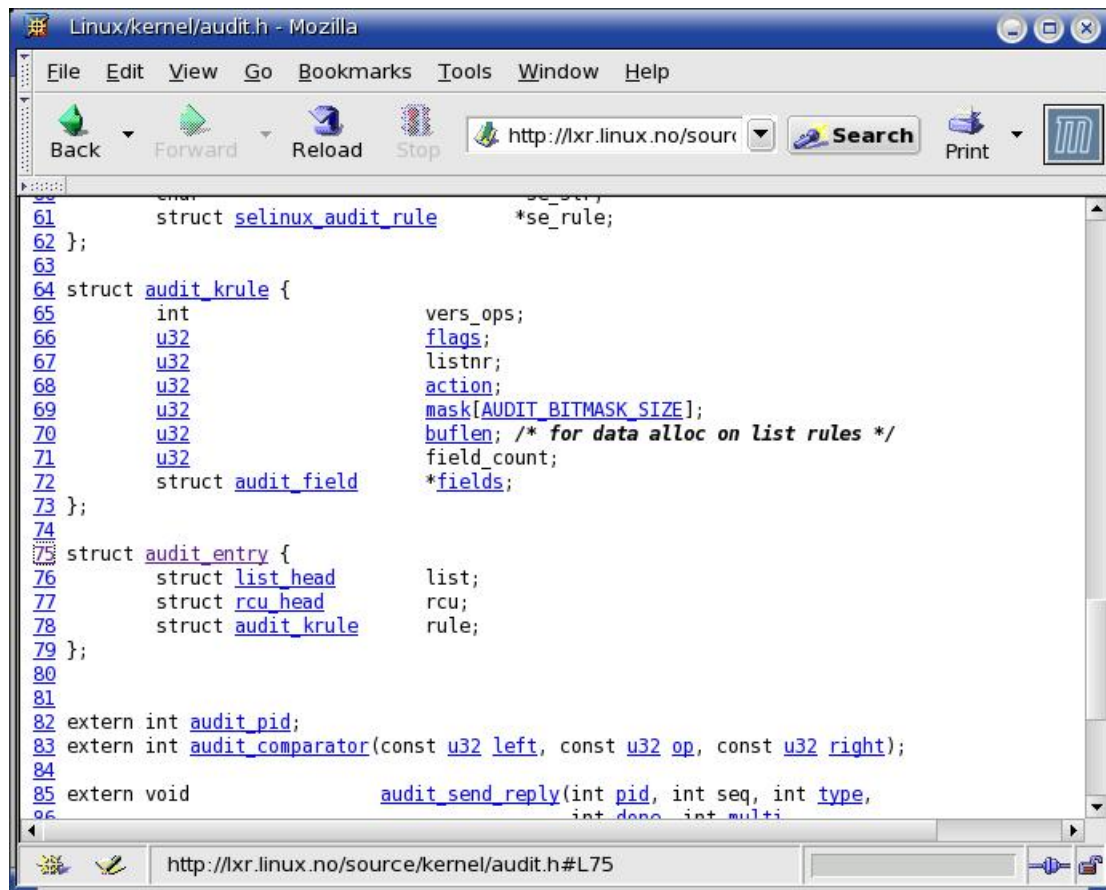


图 4: audit\_entry 的定义

LXR 是一个强大的交叉引用工具。但它也很难设置和运行。你需要一个为数据生成的工作的 web 服务器。而且，你需要很了解 Perl 的知识，用来将代码适应你的系统。（对于你的 Perl 黑客，LXR 包含一些最复杂，高级的正则表达式。）

但是，如果你正在处理大量的代码，这个工具可以成为救星。



## 第 6 章：预处理方案

C++实际上是两个语言，一个宏预处理语言和一个编译语言。所以，你不能破解的是，你可以尝试在预处理器中破解的基本语言。

### 方案 67：使用预处理器产生名字列表

**问题：**C++没有一个获得 enum 文本格式的方法。

例如：

```
enum state {STATE_GOOD, STATE_BAD, STATE_UNKNOWN};

// Doesn't work right
std::cout << "Good is " << STATE_GOOD << std::endl;
```

结果是：

```
Good is 0
```

我们想要的是：

```
Good is STATE_GOOD
```

**方案：**使用预处理器用不同的格式定义一系列条目。

首先，我们定义一个包含我们希望使用的一系列条目的宏：

```
#define STATE_LIST \
    D(STATE_GOOD), \
    D(STATE_BAD), \
    D(STATE_UNKNOWN)
```

宏 D（是我们以前没有定义的）封装条目。现在，我们使用宏 D 来以不同的方式定义列表：

```
#define D(x) x
enum state { STATE_LIST }
#undef D
```

注意，在使用后我们立即取消定义了宏 D。这防止了这个临时的宏稍后在代码中意外地被重用。

现在，让我们定义一系列名字。

```
#define D(x) #x
const char* const state_name = { STATE_LIST };
#undef D
```

现在，我们可以使用像这样的语句来输出状态的名字：

```
std::cout << "The good state is " <<
    state_name[STATE_GOOD] << std::endl;
```

**限制：**这仅适用于项目按数字顺序排列的枚举列表。如果你给项目赋值，则不会有效。

例如：

```
enum error_code {BAD=44, VERY_BAD=55, TERRIBLE=999};
```

下一个方案解决这个问题。

## 方案 68：自动创建单词列表

**问题：**你需要创建带有指定值的 enum 列表。

例如：START 是 5，PAUSE 是 7，ABORT 是 25。

**方案：**一个聪明的预处理器方案可以很容易解决这个问题。

让我们通过定义我们的列表开始：

```
#define COMMAND_LIST \
    C(COMMAND_START, 5), \
    C(COMMAND_PAUSE, 7), \
    C(COMMAND_ABORT, 25)
```

现在，让我们创建一个定义 enum 声明的宏：

```
#define C(x, y) x = y
enum COMMANDS { COMMAND_LIST };
#undef C
```

下一步是定义名字到 ID 的映射：

```
#define C(x, y) {y, #x}

struct cmd_number_to_name {
    int cmd_number;
    char* cmd_name;
} cmd_number_to_name[] = {
    COMMAND_LIST
    , {-1, NULL}
};
#undef C
```

在必须处理以多种方式表示的数据时，使用这样的宏嵌套非常有用。我们的方案专注于枚举定义，但除了这些简单的编程示例之外，这个方案还有其他的用处。

## 方案 69：保护头文件的双重包含

**问题：**所有的头文件应该包含它们需要的任何其他头文件。这很容易导致，单个文件被包含多次。但是，多次定义同样的事情可能引起更多的问题。

**方案：**在你的头文件中添加“双重包含”保护。例如：

```
// File define.h
#ifndef __DEFINE_H__
#define __DEFINE_H__
// ... rest of the header file
#endif /* __DEFINE_H__ */
```

## 方案 70：在 do/while 中封闭多行的宏

**问题：**如何创建执行两条语句的宏。

例如，我们希望创建清除的宏：

```
#define CLEAN_RETURN \
    close(in_fd);close(out_fd); return;
```

但是，如果将它放进一个 **if** 语句中，它不会有效：

```
if (done)
    CLEAN_RETURN;
```

展开它，我们得到：

```
if (done)
    close(in_fd);close(out_fd); return;
```

为了清晰，添加一些空白：

```
if (done)
    close(in_fd);
close(out_fd);
return;
```

这是我们不希望的。一种“解决方案”是在 **{}** 中封闭语句。

```
#define CLEAN_RETURN \
    { close(in_fd);close(out_fd); return; }
```

现在，我们的 **if** 语句展开为：

```
if (done)
    { close(in_fd);close(out_fd); return; };
```

这有效。有一些。问题是如果我们尝试一个 **if/else** 语句：

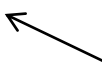
```
if (done)
    CLEAN_RETURN;
else
```

```
not_done_yet();
```

当我们尝试编译它的时候，给出了一个语法错误。为什么？

让我们看看展开后的代码：

```
if (done)
    { close(in_fd);close(out_fd); return; };
else
    not_done_yet();
```



在那一行有一个多余的分号。当没有 **else** 的时候并不影响，但是现在有一个，编译器会混淆。

所以，我们如何定义一个像一个语句一样的多个语句的宏。

**方案：**使用 **do/while** 技巧。

定义语句在一个 **do/while** 循环中的宏：

```
#define CLEAN_RETURN          \
do {                          \
    close(in_fd);             \
    close(out_fd);            \
    return;                   \
} while (0)
```

注意，在 **while(0)** 后面没有分号。

在一个 **do/while** 循环中封装多个语句为一个语句。然后，它们可以像单个语句使用在任何地方。

**注意：**这可能是你想要合法地使用 **do/while** 的唯一地方。在所有其他情况下，一个 **while** 循环比一个 **do/while** 循环可能更简单，容易理解。

## 方案 71：使用 **#if 0** 移除代码

**问题：**你需要让代码消失。注释代码是一个选择，如果你的编译器不允许嵌套的注释，它就会无效。

**方案：**将代码封装在一个 **#if 0 / #endif** 块。

例如：

```
do_it();
#if 0
    double_check_it();
#endif
finish_up();
```

这种方案让代码在你的程序之外。但是问题仍然存在，为什么不只是删掉代

码。毕竟，如果你真的想让它消失，删除它。不要将它留在程序中。我已经看过这个方案使用了很多次，每次都没有理由把死的代码留在程序中。但是，如果你曾经确实有理由将这样的代码放在程序文件中，这种方案会使代码在程序之外。

## 方案 72：使用 `#ifndef QQQ` 标识临时代码

**问题：**有时，你需要在程序中添加诊断代码，直到找到问题。当然，你不希望这些临时代码方案展示在产品代码中，因此，如果你快速删除它们会很好。

**方案：**将代码放在一个 `#ifndef QQQ / #endif` 块中。

例如：

```
read_accout();
#ifndef QQQ
    save_backup("debug.1.save");
    std::cout << "Starting sort " << std::endl;
#endif /* QQQ */
#ifndef QQQ
    save_backup("debug.2.save");
#endif /* QQQ */
```

**QQQ** 符号是可以改变的，因为很容易键入，然后没有人定义它。（没有一个合理的方式。）而且它比 **MARKER\_INDICATING\_CODE\_I\_AM\_TAKING\_OUT\_AS\_SOON\_AS\_THIS\_WORKS** 要短。

该符号使得临时代码像拇指一样突出，可以在完成后容易地取出代码。只要搜索 **QQQ** 并删除与之关联的所有条件代码块。

## 方案 73：在函数上，使用 `#ifdef`，而不是函数调用，来消除多余的 `#ifdef`

**问题：**你想只为软件的调试发行版添加日志。所以，整个代码中你有很多这样的东西：

```
#ifdef DEBUG
    debug_msg("Entering function sixth_level_of_hell");
#endif /* DEBUG */
```

这很丑而且烦人。一定有一个很好的方法。

**方案：**如果你定义了 `debug_msg`，你可以摆脱所有的 `#ifdef` 语句，是代码干净，例如：

```
#ifdef DEBUG
extern void debug_msg("const char* const msg);
```

```
#else /* DEBUG */  
static inline void debug_msg(const char* const) {}  
#endif /* DEBUG */
```

现在，在代码体中，所有我们需要做的只是调用函数：

```
debug_msg("Entering function sixth_level_of_hell");
```

如果 `DEBUG` 被定义了，真的函数被调用。如果没有，假的 `debug_msg` 被调用。因为它是 **inline** 函数，优化器将删除代码。

关于这个的好处是，你不需要使用无用的 **#ifdef** 语句使你的代码杂乱。

## 方案 74：创建代码从函数体中消除**#ifdef** 语句

**问题：**你正在为多个平台编写一个程序。结果，你的代码中散布了很多**#ifdef** 语句。所以，事实上，它们让事情看起来杂乱，代码难以阅读。

例如：

```
void send_cmd(void)  
{  
    send_cmd_start();  
  
#ifdef FE_TEXTURE  
    send_texture();  
#endif /* FE_TEXTURE */  
  
#ifdef FE_COLOR  
    send_background();  
    if (foreground != TRANSPARENT)  
        send_foreground();  
#endif /* FE_COLOR */  
  
#ifdef FE_SIZE  
    if (size != 0)  
        send_size();  
#endif /* FE_SIZE */  
  
#ifdef FE_REPLAY  
    if (prev_cmd == '\0') {  
        prev_cmd = cur_cmd;  
        prev_param = cur_param;  
    }  
#endif /* FE_REPLAY */  
  
    send_cmd_end();  
}
```

```
}
```

这段代码很难阅读。**#ifdef** 指令破坏了代码逻辑。

**方案：**只使用**#ifdef** 定义整个过程。

例如，让我们看看下面的代码：

```
#ifdef FE_TEXTURE
    send_texture();
#endif /* FE_TEXTURE */
```

这可以更简单。首先，我们使用**#ifdef** 控制函数 **send\_texture** 的定义。

```
#ifdef FE_TEXTURE
static void send_texture() {
    // body of the function
}
#else
inline static void send_texture() {}
#endif /* FE_TEXTURE */
```

现在，为了代码的主体，我们只是将没有任何**#ifdef** 的调用放进 **send\_texture**。

```
void send_cmd(void)
{
    send_cmd_start();
    send_texture();
}
```

我们可以使用 **set\_background** 和 **set\_foreground** 做同样的事情。这些函数的调用像这样：

```
send_background();
if (foreground != TRANSPARENT)
    send_foreground();
```

但是，这样不会引起额外的代码放进我们的过程？下面的语句怎么样？

```
if (foreground != TRANSPARENT)
```

实际上，如果 **send\_foreground** 的主体被定义出来，优化器会移除这个语句。

现在，让我们看看代码：

```
#ifdef FE_REPLAY
    if (prev_cmd == '\0') {
        prev_cmd = cur_cmd;
        prev_param = cur_param;
    }
#endif /* FE_REPLAY */
```

为了摆脱**#ifdef**，我们首先在一个过程中放入代码：

```
#ifdef FE_REPLAY
```

```
static inline void do_replay() {  
    if (prev_cmd == '\0') {  
        prev_cmd = cur_cmd;  
        prev_param = cur_param;  
    }  
}  
#else /* FE_REPLAY */  
static inline void do_replay() {  
    // Do nothing  
}  
#endif /* FE_REPLAY */
```

现在，我们只是在我们的代码中放入一个函数：

```
do_replay();
```

这个方案的关键是使用**#ifndef** 改变整个函数的定义。因为，函数是代码的一个简单、逻辑的单位，我们正在单位级别上定义事情的内部或外部。

在语句级别上，修改程序的旧方式会更加混淆。当我们使用这个方案时，看看函数的主体是多么的简洁：

```
void send_cmd(void)  
{  
    send_cmd_start();  
    send_texture();  
  
    send_background();  
    if (foreground != TRANSPARENT)  
        send_foreground();  
  
    if (size != 0)  
        send_size();  
  
    do_replay();  
    send_cmd_end();  
}
```



## 第 7 章：构建方案

程序的大小和复杂性呈指数增加。结果，构建过程的复杂性在增加，构建生成的时间在增加。Linux 内核有超过 13000 个文件，包含超过 650 万行代码。对一些人来说这是一个小程序。

但是，基本的构建工具仍然是 `make` 命令和一个编译器。但是，一些方案让你绕过这个系统的限制，快而合理的生成代码。

### 方案 75：不要在没有验证的情况下使用任何“众所周知”的加速

**问题：**并不缺少对于如何加快构建速度有很好想法的人。他们有各种各样的为什么他们的宠物想法会让你在 3.8 秒完成一个花费 5 天构建的理由。

例如，`gcc` 编译器有 `-pipe` 选项。通常，编译器运行编译器的第一次传递，将输出写到一个临时的文件。第二次传递会读取这个文件，写第二个临时文件。因此，需要很多传递。

使用 `-pipe` 选项，每个传递是同时运行一个管道的输出连接下一个管道的输入。理论上，通过这样可以消除临时文件，临时文件的磁盘 I/O 和在内存中做的一切事情。

所以，`-pipe` 选项必定使事情更快。如果我们需要更方便，可以查看内核 `Makefiles`，发现这个选项在那里使用，如果 Linux 内核使用它，那么它是很好的。

**方案：**一个优秀的黑客测试“显而易见”，因为有时“显而易见”并不是明显的。

让我们考虑 `-pipe` 选项。不使用 `-pipe` 选项编译一个内核花了 57:04。带有 `-pipe` 花了 57:19。等下，带有 `-pipe` 花了更多时间。让我们做一些额外的测试：

<i>No Pipe</i>	<i>Pipe</i>
57:04	57:19
57:04	56:18
56:58	56:19
56:55	56:17

56:54

56:21

从这里我们看出**-pipe** 并不使事情更快。有时会让它们变慢。

现在，有一些这些结果的一系列原因。首先，所有的磁盘 I/O 是缓存的，Linux 使用很多缓存。可能是内存中都是临时文件，没能写进磁盘。也有可能对于管道/非管道输入流，在不同的编译器传递中是不同的逻辑。

但是最终，作为黑客，我们所关心的是结果，然而结果不支持那个理论。

在编程中有很多理论和实际不匹配的例子。很多人知道他们的代码慢的部分在哪。黑客通过使用探查来测试。

一个数据库手册说，插入记录，然后添加索引比插入索引记录快。一个黑客测试。有时，你发现手册是错误的。

毕竟，一个真正黑客的其中一个标志是，懂得编程和计算机的过去和未来。他们不只知道手册里说的有效的，而且也知道手册是正确还是错误。

### 康格里夫时钟

我进入钟表领域，我最喜爱的一种钟表是威廉·康格里夫（1772-1828）设计的。

威廉先生最著名的是他对星条旗的贡献。如果你记得“和红色炫光的火箭”这一行，他发明了火箭。他生活的时代中一件最伟大的工程挑战是创造一个最精确的时钟。威廉先生是一个多产的发明家，他把天赋用在了构建一个高度精确时钟的问题。

滴答是时钟不稳定和不准确的主要原因。齿轮的移动式一个问题，它们越少，移动得越好。解决这个问题的一种方法是使用一个非常长的钟摆。实际上，越长越好。

康格里夫想到了另一种解决方法。结果是他称作康格里夫时钟或超限擒纵时钟。球落在倾斜的平面上，并允许滚下轨道。当到达终点时，是一个杠杆挑起，而另一个方向上的平面倾斜，球向下滚动到另一端。此过程可能需要 15 到 30 秒，具体取决于时钟。

康格里夫计算出他的时钟设计师等效的或具有 60 英尺钟摆的传统时钟。因此，他的时钟应该精确在一个月中的一秒。

现在，威廉康格里夫是主意的发明者，工程师和科学家。他有很多逻辑和计算来支持他的主张。设计完美，工程不可挑剔。尽管如此，设计还是创造了超精确的康格里夫时钟。

除了它不起作用。事实证明，虽然钟摆具有固有的频率，但是沿着路径滚动的球没有。每一粒灰尘，每一次温度的变化都会导致时钟不准。我甚至没有在他的设计中找到其他问题。目标是每月错误率不到一秒的时钟。但事实上，一个好的康格里夫时钟一天只能精确到二十分钟。



图 5：一个康格里夫时钟

## 方案 76：在双核处理器机器上使用 `gmake -j` 加速

**问题：**编译大的程序花费很长时间。我们需要一些方式使事情更快。我们有

一个双核处理器。有任何方式让构建更快，而不是同时启动两个。

**方案：**使用 `gmake` 的 `-j` 标志，使得充分利用一个双核处理器系统。

今天，许多现代的计算机有超过一个的处理器。你不仅可以购买支持两个或四个 CPU 的高端主板，而且 Intel 和 AMD 都生产双核微处理器，在单个芯片上相当于两个 CPU。

GUN 的 `make` 命令（作为 `gmake` 安装在大多数系统中）有一个选项，有助于在多处理系统中作出优化。`-j2` 标志告诉 `make` 程序尝试在同一时间使用两个编译器。理想情况下，这样的结果是消减一个编译器的时间为两个。

这个系统有一些限制。首先，有可能编译错误的出现时乱序的。例如，正常情况下，如果编译文件 `alpha.cpp` 和 `beta.cpp`，你可能看到：

```
alpha.cpp:3: Syntax error
alpha.cpp:7: Another syntax error
beta.cpp:3: Syntax error
beta.cpp:7: Screwed up again
```

使用 `-j2`，你可能看到：

```
alpha.cpp:3: Syntax error
beta.cpp:3: Syntax error
alpha.cpp:7: Another syntax error
beta.cpp:7: Screwed up again
```

另一个问题是，当你使用 `-j2` 时（或更高），可能会产生错误的 `Makefiles`。

但是 `-j2` 确实有影响，或者它只是另一个 `-pipe`？测试表明，在一个双核处理器系统中 `-j2` 确实有效。这是结果：

<i>No -j</i>	<i>-j2</i>	<i>-j4</i>
56:19	29:47	29:38
56:18	29:43	29:45
56:19	29:48	29:43
56:17	29:42	29:42
56:21	29:44	29:45

现在，仅仅因为它在我的系统上有效，并不意味着在你的系统上也有效。但是，这一些小的实验，你应该能够发现优化 `make` 的设置。

极端并行处理

黑客总是想知道系统在极端条件下的行为。**-j2** 和 **-j4** 的转换确实温和。它们只会启动两个或四个工作。如果我们增加数量会发生什么。

如果不提供数字，只使用 **-j**，**gmake** 会同时启动尽可能多的作业。我曾经在一个软件的主要代码块试了这个。这是我唯一一次看到负载平均值超过 1000<sup>14</sup>。我不确定负载平均值达到多高，因为在那个点上，系统监视工具开始挂起。（它们不能得到任何 CPU 周期。）

该系统显然是疯狂的分页，确实打败了磁盘。很明显抖动发生，并正在杀死系统，所以，这个是后续我重启了系统。（不能杀死进程，因为内核不能获得足够的 CPU 来起作用。）

## 方案 77：通过使用 **ccache** 编码重复编译

**问题：**重复的编译导致很多不必要的工作。

我们都知道演习。为了建立一个清除构建，你需要执行下面的命令：

```
make clean
make
make install
```

你需要一个 **make clean** 的原因是有时有人会修改 **Makefile**，比如像标志或类似的条目。这些变化不被 **make** 命令追踪。结果，唯一保险的得到一个好的构建的方式是重新编译一切。

但是，大多数你的文件自从上次构建后很可能没有变化。所以，基本上在生成上次你编译生成的相同的文件。这种重复工作浪费时间，并且使构建的时间更长。

**方案：**使用 **ccache** 程序。

**ccache** 程序缓存已编译的输出，如果你尝试再次编译相同的源，则从缓存中获取目标文件。结果是在扩展的磁盘空间大大加快了速度。但是考虑到今天的磁盘空间成本，这是非常值得的。

使用 **ccache** 程序很简单，在你的 **Makefiles** 中，每条编译程序之前插入 **ccache**

命令。例如：

```
hello.o: hello.cc
ccache gcc -g -c hello.cc
```

当这个程序执行 `ccache` 会执行下面的操作：

1. 通过预处理器运行文件 `hello.cc`。
2. 检查结果的 `md5sum` 与任何在缓存目录中加载的校验和进行对比。
3. 如果有一个匹配，对象文件由缓存提供。
4. 否则，编译器运行，将结果加载到缓存中供以后使用。

那里有很多细节已经被覆盖，但这是基本的想法。`ccache` 程序是一种加速重复构建的简单而有效的方法。

`ccache` 程序从 <http://ccache.samba.org> 获得。

## 方案 78：不改变你的 Makefiles 的情况下使用 ccache

**问题：**你有一个退出 Makefile 的遗留应用程序。你希望使用 `ccache`，但你不想编辑 500 个 Makefiles 来完成这项工作。

**方案：**使用 `ccache` 的符号链接特性。创建 `ccache` 的黑客预计了这个程序。

创建 `ccache` 的黑客预料到了这个程序。如果你从 `ccache` 程序到编译器建立了一个符号链接。例如，如果你使用 `gcc`，你可以通过执行下面的命令来打开 `ccache`：

```
$ ln -s /usr/local/bin/ccache /home/me/bin/gcc
$ PATH=/home/me/bin:$PATH ; export PATH
```

第一个命令在当前目录中建立一个 `ccache` 和一个 `gcc` 命令之间的链接。

第二个命令确保路径中 `/home/me/bin/gcc` 是第一个 `gcc`。所以，当 Makefile 执行 `gcc` 命令，`/home/me/bin/gcc` 运行了。`ccache` 聪明到可以通过一个符号链接知道它正在被运行。

然后执行正常的操作，就像如果你编辑 Makefile，然后为每条编译规则添加一个 `ccache` 命令。

**注意：**你可能想添加 `PATH` 命令到你的 `.profile` 文件或 `.bashrc` 文件，这样你每次启动一个新的 shell，它们就会被执行。

## 方案 79：使用 `distcc` 分配工作负载

**问题：**你不得不编译一个确实很大的项目，仅仅 `ccache` 不够快。

**方案：**如果你有访问很多构建机器的权限，你可以使用 `distcc`。

`distcc` 从 <http://distcc.samba.org/> 获得。

这个工具设计用于 `gmake` 的多个命令选项。例如，当你执行命令 `gmake -j4`，`gmake` 程序会尝试一次启动四个编译工作。使用 `distcc`，四个作业的每一个都会被发送到一个不同的机器。

`distcc` 程序的操作很像 `ccache`。你通过在编译命令前面放置 `distcc` 来激活它。

例如：

```
foo.o: foo.cpp
distcc g++ -c -Wall foo.cpp
```

你也需要配置系统，通过提供带有 `distcc` 的机器列表，在构建机器中，它可以用于构建和安装 `distcc` 作为一个服务器。

`distcc` 和 `ccache` 工具被设计在一起工作。所以，如果你面对不得不做重复的大的构建，这些工具使得你的构建更快。

## 第 8 章：优化方案

### 方案 80：除非你真的确实需要，否则不要优化

**问题：**过早优化

你有一个程序，它工作很好。这个程序确实不占用很多的 CPU，在合理的时间内完成任务。但是如果你优化它，你确信你可以节省几秒运行时间。

你怎么办？

**方案：**什么都不做。

对于一个程序员来说，最困难的事情之一是不去编程。但是除非有迫切的优化需求，否则不要做。

简单有效的代码比优化过的无效的代码执行要快。而且，简单有效的代码与聪明优化的代码相比更容易维护和改进。

毕竟，谁在意启动你的程序需要花 5 秒而不是 3 秒。大多数情况下，你的程序很可能不受 CPU 限制。

如果你正在做大规模的数据操作，像视频处理，建立虚拟的世界，或大规模数据压缩，然后优化是说得过去的。但是，大多数的程序不需要这种策略。

让有效的代码独立。产生错误的最保险的方式之一是尝试优化不需要的东西。

什么也不需要做是一种最好的编程系统。对于什么也不编写不花费时间，代码不存在也是唯一保证代码中没有一个错误。

所以，对于优化，最好的建议是什么也不做，除非你真的不得不做。

### 方案 81：使用性能分析器查找要优化的地方

**问题：**你认为代码中慢的部分与它们真正位于的地方可能是两个不同的事情。

让我们假设你没有采用上一个方案的建议，已经决定你必须优化。现在，你必须决定你的程序的哪部分需要你进行尝试优化，然后使其更快。

在大多数的程序中，有一个或两个消耗 90%CPU 的函数。很多时候，一个程



程序员对程序在哪里卡住有一个很好的预感。但是，往往会发现他的感觉与现实不同。因此，如果他满足于自己的感觉，他将最终优化错误的代码，而不是加速他的程序。

**方案：**使用性能分析器。

性能分析器告诉你哪个函数花费最多的时间。

在我的经验中，一个典型的性能分析会话像这样：我被告知文件保存函数花费太长时间。我知道将参数转换为文本进行保存的逻辑可能是程序。在所有代码都没有设计为高效之后，它肯定符合设计标准。

所以，我启动了我的可靠分析器并尝试在保存逻辑中找到问题。但是这种方法发生了一些有趣的事情。事实证明，保存只占用了 1/10 的时间，另外 90%是在排序函数。

事实证明，为了保存做好准备，我们排序了参数。为了确保它们是有序的，每个级别的代码在调用下一个较低级别之前对它们进行排序。总之，对同样的数据进行了 25 种排序。第一种排序为了整理东西，其它 24 种只是浪费时间。

在这个点上，我的选择是尽可能少的排序或者使排序更高效。我决定添加一个告诉我参数列表是否排序的标志。排序函数检查标志，如果列表有序则什么也不做。问题解决了。

注意到问题并不在我预料的地方。它位于一些其它的代码，我没有使用那种方法编写的，非常低效的。如果我凭直觉优化保存代码，我就不会让程序更快。

这个加载的合理之道是使用性能分析器。它告诉你真正确实很慢的代码位置，而不是你觉得它慢的地方。

## 方案 82：避免格式化的输出函数

**问题：**C 语言的格式化输出函数是慢的。C++的格式化输出函数更慢。

考虑下面的简单代码片段：

```
for (int i=0; i < 20; ++i) {  
    printf("I is %d\n", i);  
}
```

C 和 C++都是编译型的语言，所以比解释性语言更快。但是你已经你的代码中添加了一个解释性的语言。解释是 `printf` 调用，它必须解释格式字符串 “I is %d\n”，然后向屏幕发送结果。

所以，`printf` 必须做什么来打印结果：

1. 读取字符串。任何只要不是%的都会到输出。
2. 如果遇到%，解释%规范。
  - a. 检查%后面的所有数字，指出字符串的大小。
  - b. 查找格式字符（这个例子中是 `d`）
3. 确定需要很多个字符打印这个数字。
4. 如果需要打印数字的字符数少于指定的数量，然后输出前导空格。
5. 输出这个数。

这是简化的版本。如果你想看完整的代码，下载一份 `GCC` 标准库的拷贝，自己查看代码。

C++格式化输出甚至更糟糕。库有很多灵活性和很多钟声，口哨声。所有这些通用的包袱会非常减慢库。

**方案：**自定义输出例程。

```
static void two_digits(const int i)
{
    if (i >= 10) {
        putc((i / 10) + '0');
    }
    putc((i % 10) + '0');
}
for (int i = 0; i < 20; ++i) {
    puts("I is ");
    two_digits(i);
    putc('\0');
}
```

应该注意的是，第二个，优化的版本比第一个更长，更复杂。我们所做的是用专为此特定任务设计的专用代码替换慢速的通常代码。

它更快，但是它更复杂，很可能包含错误。再一次，如果必须才去优化。（参考上面的方案 80。）

### 方案 83：使用 `++x` 而不是 `x++`，因为它更快

**问题：**你需要以尽可能快的方式实现自增一个变量。

**方案：**使用前缀自增（`++x`）而不是后缀（`x++`），因为它更快。

现在，你可能很想对你自己说的是“你在开玩笑”。无论我使用哪个版本来

递增整数，编译器都会生成准确的代码。

例如下面的代码，对于两种自增生成的代码是完全一样的。

```
int i = 1;
++i;
i++;
```

对于整数来说是正确的。但是对于用户定义的类型不是的。

让我们看看每一个操作需要执行的步骤。

前缀（++x）自增必须执行下面的步骤：

1. 自增变量。
2. 返回自增的结果。

后缀版本（x++）执行下面的操作：

1. 保存原始变量的一份拷贝。
2. 自增变量。
3. 返回你在第 1 步中保存的未自增的变量。

所以，对于后缀（x++）版本，你不得不产生一份变量的拷贝。这可能是一个有消耗的操作。

让我们看看一个典型的类如何可能实现这两个操作：

```
class fixed_point {
    // Usual stuff

    // Prefix (++x) operator
    fixed_point operator ++() {
        value += FIXED_NUMBER_ONE;
        return (*this);
    }
    // Postfix (x++) operator
    fixed_point operator ++(int) {
        // Extra work needed
        fixed_point result(this);
        value += FIXED_NUMBER_ONE;
        return (result);
    }
}
```

由于总是使用前缀版本的自增和自减操作，会使你的程序更有效率。

## 方案 84：通过使用 C 的 I/O API 优化 I/O 而不是 C++的 I/O API

应该注意到优化是取决于编译器的。然而，我使用过的每一个编译器都这样

做。

**问题：**与 C 语言相比，C++ 的 I/O 流更灵活，更不容易出错。但是，对于大多数的系统，它也更慢。

考虑一个设计用于按行读取的读文件行，然后写到标准输出的程序。

C++版本是：

```
#include <iostream>
int main()
{
    char line[5000];
    while (1) {
        if (! std::cin.getline(line, sizeof(line)))
            break;
        std::cout << line << std::endl;
    }
    return(0);
}
```

C 版本是：

```
#include <stdio.h>
int main()
{
    char line[5000];
    while (1) {
        if (fgets(line, sizeof(line), stdin) == NULL)
            break;
        fputs(line, stdout);
    }
    return (0);
}
```

C++程序拷贝一系列文件到/dev/null 花了 98 秒，而 C 程序做同样的事情花了 11 秒。

**方案：**当 I/O 速度是一个因素时，在一个 C++程序中使用 C 的 I/O。

这种方案的确加速了与 I/O 绑定的程序，但是它应该被谨慎使用。首先，你确实失去了 C++ 的 I/O 流的优势。这包括来自这个系统的增加的安全性。

而且，这里提供的时间测试来自一个计算系统的一个编译器。你应该使用你的编程环境来做相似的测试。谁知道你的编译器可能更好。

## 方案 85：使用一个本地缓存避免重新计算一样的结果

**问题：**你有一个函数，被重复调用计算相同的值。这个函数花很长的时间完成这个工作。你怎么让事情变得更快？

让我们看一个例子。下面是一个 **rectangle** 类。函数 **compute\_serial** 返回一个用于给形状分类的唯一的一系列字符串。

```
class rectangle {
private:
    int color;
    int width;
    int height;
rectangle(int i_color, int i_width, int i_height):
    color(i_color), width(i_width), height(i_height)
{}
void set_color(int the_color) {
    color = the_color;
}
char serial_string[100]; // Unique ID string
char* get_serial() {
    snprintf(serial_string, sizeof(serial_string),
        "rect-%d x %d / %d", width, height, color);
    return (serial_string);
}
```

这个类的测试显示，它使用 **snprintf** 来计算一串字符串。它的使用是一个极度消耗的函数。如果有可能，我们需要优化这个函数。

**方案：**使用一个本地缓存。

当遇到避免重复计算相同的东西时，一个小的本地缓存可能非常有用。

对于 **rectangle** 类，这里对 **get\_serial** 函数有一些简单的改变：

```
char* get_serial() {
    if (dirty) {
        snprintf(serial_string, sizeof(serial_string),
            "rect-%d x %d / %d",
            width, height, color);
        dirty = false;
    }
    return (serial_string);
}
```

现在，所有我们需要做的是确保，当任何时候用于计算一串字符串的数据改

变了，`dirty` 是被设置的。这包含构造函数：

```
rectangle(int i_color, int i_width, int i_height):
    color(i_color), width(i_width), height(i_height),
    dirty(true)
{}
```

任何改变关键成员的函数必须也发生变化：

```
void set_color(int the_color) {
    color = the_color;
    dirty = true;
}
```

这种方案用复杂性换了速度。通过标记，类变得复杂，我们能够让它更快。所以，这个方案应该被谨慎使用。只在你通过一个性能分析器运行你的程序之后，才使用它，确保这个类是瓶颈之一。很多时候，那些认为自己是黑客的人优化了一个他们认为存在速度问题的程序，只是证明他们不是真正的黑客，并且他们认为他们做的地方没有速度问题。性能分析器识别真正的速度问题，那是为什么它们被真正的黑客经常使用的原因。

使用这种方案有一个代价。一个真正的黑客知道这个代码什么时候是值得的。

## 方案 86：使用自定义 `new/delete` 来加速动态存储分配

**问题：**你有一个小的类，经常从堆中被分配和释放。动态内存分配调用确实会减慢你的程序。你如何加速？

**方案：**为你的类建立一个自定义的 `new/delete`。

我应该指出，这种方案是一个技巧，并不容易使用。如果你确信你知道你正在做的事情，才使用它，并确信对于这个类来说 `new` 和 `delete` 是一个问题。错误的实现这个可能破坏内存或引起其它更多的问题，所以，在你开始之前确信你已经阅读了方案 80 和方案 81。

为了实现属于你自己的这个类的 `new`，所有你需要做的是声明它为一个本地的操作符函数。这是一个例子：

```
// A very small class (good for local new / delete)
class symbol {
private:
    char symbol_name[8];
    // .. usual member functions
```

```
void* operator new (unsigned int size) {
    if (size == sizeof(symbol)) {
        void* ptr = local_allocate(size);
        if (ptr != NULL) {
            return (ptr);
        }
    }
    return (::new char[size]);
}
```

让我们一步步的分析。首先，我们有函数的声明：

```
void* operator new (unsigned int size) {
```

这定义一个操作符 **new**，覆盖 C++ 内置的那个。它使用一个参数，被分配的条目大小。

一个使人误解的是，他们假设当你建立一个类型为 **class symbol** 的对象时，对象的大小为 **sizeof(class symbol)**。它不是的。如果 **symbol** 是一个派生类定义（叫做 **class extended\_symbol**）的基类，然后当一个 **extended\_symbol** 被分配，**symbol** 的 **operator new** 被调用，分配 **extended\_symbol** 的大小。

如果你完全不理解我刚才说的，除非你理解，否则不要做这个优化。我建议你使用带有基类和派生类的通常的 **new/delete** 操作符做一些实验。除非你完全理解发生了什么，否则不要在产品代码中尝试这种方案。我们将假设程序分配大量的 **symbol** 类型的变量，那是我们想要优化的。但是只在决定扩展我们的类的情况下，我们检查大小：

```
if (size == sizeof(symbol)) {
```

如果大小是不正确的，我们到达程序的底部，使用系统的 **new** 去分配内存。

```
return (::new char[size]);
```

如果正确的大小被使用，那意味着我们正在分配一个新的 **symbol**，所以我们可以使用优化的内存分配：

```
if (size == sizeof(symbol)) {
    void* ptr = local_allocate(size);
```

我们将 **local\_allocate** 的实现留给你。它应该是简单和快速的。（如果这太慢，那么使用它的意义何在？）例如，分配器可能使用一个固定的内存数组来保存数据。由于你有一个固定大小的条目，知道关于期望的分配/释放用法，你应该能够建立一个非常有效的内存管理器。

无论何时，我们做一个通常的 **new**，我们也需要做一个通常的 **delete**：

```
static void delete(void* const ptr) {  
    if (local_delete(ptr))  
        return;  
    ::delete ptr  
}
```

一个最后的注意，在继续之前阅读下一个对抗方案。

## 对抗方案 87：不必要的建立一个自定义的 new/delete

**问题：**程序员认为他的程序很慢，认为一个本地的 new/delete 会加速。

**对抗方案：**通过放置你自己的 new/delete 来优化程序。

大多数时间，如果你没有通过一个性能分析器运行程序就添加了一个优化，谨慎分析它，你会在错误的地方优化。new/delete 方案是有技巧和危险的。如果你知道它为加速你的程序，否则不要想添加它。那意味着你不得不做你的家庭作业。

经常，一个人认为他是黑客，但不会执行本地的 new/delete。一系列可预测的结果通常如下。

首先，代码破坏。那是因为正确的得到函数的 new/delete 操作是一个技巧。（细节参考前面的方案。）因为程序员已经错误地实现了 new/delete，他的程序陷入一种神秘的方式。追溯这些花费时间。甚至他会放弃或最终得到一个有效的本地 new/delete。当然，在这个处理期间，项目会落后于计划。

在他最终得到一个新的内存管理系统调试后，他会发现因为他没有做家庭作业，运行一个性能分析器，他只是加速了不是时间关键程序流程的部分。然后，结果是程序没有比他刚开始的时候更快。

所以，他的努力显示的唯一事情是跳过了日程。但是他得到了什么？在过早优化中，他学到了重要的一课。一课是你不需要学习困难的方式，因为你读了这个对抗方案。

## 对抗方案 88：使用移位乘以或除以 2 的倍数

**问题：**有些人使用左移来乘以 2,4 或其他 2 的幂。他们使用右移除以同样的数。

例如，下面的两行做相同的事情：

```
i = j << 3;
```



```
i = j * 8;
```

**对抗黑客：**使用移位加速计算。

当他们想乘以 8 的时候，真正的黑客乘以 8。

回到编译器是愚蠢的和机器慢的时代，如果你乘以 8，编译器产生乘法指令。这非常慢，因为一些机器没有乘法指令，所以，为了执行这个操作，使用一个子程序调用。

人们很快发现，当使用 2 的幂（2,4,8,16.....）时，你可以使用移位代替乘法（或除法）。结果是代码更快。

但是机器已经有了改进，编译器也是如此。如果你需要做一个乘法，就做一个乘法。编译器尽可能产生最快的代码完成这个操作。

你所做的是当使用移位代替乘法或除法会使你的代码更晦涩。现在，我们已经得到太多晦涩的代码，变得清晰和简洁。使用乘法去做乘法，使用除法做除法。

### 方案 89：使用静态内联而不是内联来节省空间

**问题：**你需要节约一些字节。

我们会忽略一个问题，考虑到当前内存大小和价格，几个字节会产生什么影响。只是说你需要尽可能使用内存使用。

**方案：**总是声明 **inline** 函数为 **static**。

指令 **inline** 告诉编译器生成嵌入函数代码。这对于，在函数被声明的文件中的函数调用非常有效。

但是下面的例子会发生什么：

```
file sub.cpp
inline int twice(int i)
{
    return (i*2);
}

file body.cpp
extern int twice(int i);

int j = twice(5);
```

在这个例子中，对于文件 **body.cpp** 包含一个调用到 **twice** 是非常合法的。但是 **body.cpp** 的确不知道两次中的哪个是 **inline** 函数。仅仅像一个正常的 **extern**

函数调用它。

所以，编译器被强制不仅生成所有 **twice** 的嵌入实例，而且生成标准函数代码，只是防止有人从外部尝试调用它。

声明函数为 **static** 告诉编译器没有人从外部会调用这个函数，让它安全地消除标准函数代码。这样节省了几个字节。

### 方案 90: 当你没有一个浮点数处理器的时候, 使用 **double** 加快操作而不是 **float**

**问题:** 你在为一个嵌入式系统编写代码，你不得不使用实数。有一种简单的方案加速吗？

**方案:** 使用 **double** 而不是 **float**。它更快。

猛一看，这个建议似乎愚蠢。每个人都知道 **double** 比 **float** 包含更多的位，因此很“明显”使用 **double** 比使用 **float** 花费更多。不幸的是，单词“明显”和“编程”经常不兼容。

C 和 C++标准规范所有的实数都是用 **double** 完成。让我们看看下面代码的内容：

```
float a,b,c;  
a = 1.0; b = 2.0;  
c = a + b;
```

额外需要的步骤是：

1. 将 **a** 转换为 **double** 类型。
2. 将 **b** 转换为 **double** 类型。
3. 将 **a** 的 **double** 版本与 **b** 的 **double** 版本相加。
4. 将结果转换为 **float**。
5. 在 **c** 中加载结果。

从 **float** 转换为 **double** 和 **double** 到 **float** 并不廉价，所有的转换花费时间。现在，让我们看看将变量变为 **double** 会发生什么。

1. 将 **a** 与 **b** 相加。
2. 在 **c** 中加载结果。

三次 **float/double** 转换已经移除。由于一个浮点转换操作是昂贵的，结果是代码使用 **double** 比 **float** 版本更快。

变得警觉，这个方案不会在所有的机器上有效。尤其不会在 PC 和一些有一

个浮点处理器的机器上有效。（那就是为什么我们在描述问题的时候说，我们在写一个嵌入式程序。）

无论使用的是什么格式，硬件浮点数单元几乎总是在计算之前将任何数转换为内部格式。（有些编译器使用 **long double** 类型来表示这种格式。）

更小的，嵌入式处理器并没有丰富的浮点处理单元，所以它们为它们的浮点使用一个软件库。如果你是使用这种情况，这个方案可能有效。

在使用之前，你应该总是测试这个方案。浮点操作有自己的方式，对于毫无疑问的代码可能做一些奇怪的事情。

### 方案 91：告诉编译器打破标准，当做计算时，强制编译器将 float 作为 float

**问题：**你不得不在一个嵌入式系统中使用浮点，可以承受将所有值存储为 double 类型所需的额外空间。你如何加速？

**方案：**告诉编译器打破标准。

阅读你的编译器文档。大多数的编译器有一个选项，就是让你产生单精度的运算，甚至一个双精度的标准调用。

对于 gnu gcc 编译器这个选项是 **-fallow-single-precision**。当这个选项开启，通过添加两个数来完成添加两个单精度的数。不要转换，相加和转换回来。打开它使你的程序执行得更快，但是会有一点精度丢失。

再次，像我们在上一个方案中讨论的那样，当你不使用带有浮点数处理器的机器时，它才有效。当硬件被使用，硬件会自行处理有关转换的事情，所以编译器选项不起作用。

### 方案 92：定点运算

**问题：**你正在编写一个图形化的过滤器，通过一个过滤器处理每个图像。每个像素由三个实数（R，G，B）定义，每个值的范围在 0 到 255 之间。

为了完成他们的工作，过滤算法需要对实数有效。但是测试程序，你发现它们只需要两位精度。任何额外的数字都是矫枉过正的。

**方案：**定点运算。

定点运算让你使用实数进行计算，就像浮点。不同是你不能移除小数点。它固定在两位。

优势是可以使用整数实现定点浮点运算。整数运算比浮点运算快很多。

这个例子中,我们只需要 0.00 到 255.00 之间的值,所以我们可以 在一个 **short int** 中加载数据。

```
private:
    // The value of the number
    short int value;
```

为了给我们的定点数赋值一个浮点值,所有我们需要做的是乘以转换因子。这会 将实际值转换为整数,这个整数适合我们用来表示数。

```
fixed.value = real_number * 100;
/* 100 because 2 decimal places */
```

为了加和减一个定点数,所有我们需要做的是加或减内部值。乘法是有一点技巧的。我们需要用一个转换因子来讲隐含的小数点移回正确的位置。

```
f_result = f1 * f2 / 100;
```

对于除法,乘以转换因子。

```
f_result = (f1 / f2) * 100
```

数学确实很简单。不幸的是,因为 C++ 丰富的操作集,你不得不定义很多操作符。这个方案最后的代码包含一个简单定点类的限制实现。

最后的注意,这个例子中,我们在左边两个地方使用一个定点小数点。如果我们在左边 7 个地方使用固定的二进制点,系统会快很多。换句话说,一个 128 的转换因子比 100 更快。它在稍微快速的同时给我们相同的精度。

```
#include <ostream>
class fixed_point {
private:
    // Factor that determines where
    // the fixed point is
    static const unsigned int FIXED_FACTOR = 100;
    // Define the data type we are using for
    // the implementation
    typedef short int fixed_implementation;
private:
    // The value of the number
    fixed_implementation value;
public:
    // Default value is 0 when
    // constructed with no default
    fixed_point() : value(0)
    {}
```

```
// Copy constructor
fixed_point(const fixed_point& other) :
    value(other.value)
{}

// Let the user supply us a double as well
fixed_point(double d_value) :
    value(
        static_cast<fixed_implementation>(
            d_value /
            static_cast<double>(FIXED_FACTOR))
    )
{}

// Let the user initialize with an integer
fixed_point(int i_value) :
    value(i_value * FIXED_FACTOR)
{}

fixed_point(long int i_value) :
    value(i_value * FIXED_FACTOR)
{}

// All the usual assignment operators
fixed_point& operator =
    (const fixed_point& other) {
    value= other.value;
    return *this;
}

fixed_point& operator = (float other) {
    value = fixed_point(other).value;
    return *this;
}

fixed_point& operator = (double other) {
    value= fixed_point(other).value;
    return *this;
}

fixed_point& operator = (int other) {
    value= fixed_point(other).value;
    return *this;
}

fixed_point& operator = (long other) {
    value = fixed_point(other).value;
    return *this;
}

// Conversion operators
```

```
operator double() const {
    return static_cast<double>(value) /
           static_cast<double>(FIXED_FACTOR);
}

operator short int() const {
    return value / FIXED_FACTOR;
}

operator int() const {
    return value / FIXED_FACTOR;
}

operator long() const {
    return value / FIXED_FACTOR;
}

// Unary operators
fixed_point operator +() const {
    return *this;
}
fixed_point operator -() const {
    fixed_point result(*this);
    result.value = -result.value;
    return result;
}

// Binary operators
fixed_point operator + (
    const fixed_point& other) const {
    fixed_point result(*this);
    result.value += value;
    return (*this);
}
fixed_point operator - (
    const fixed_point& other) const {
    fixed_point result(*this);
    result.value -= value;
    return (*this);
}
fixed_point operator * (
    const fixed_point& other) const {
    fixed_point result(*this);
    result.value *= value;
}
```

```
        result.value /= FIXED_FACTOR;
        return (*this);
    }

    fixed_point operator / (
        const fixed_point& other) const {
        fixed_point result(*this);
        result.value *= FIXED_FACTOR;
        result.value /= value;
        return (*this);
    }

    // Comparison operators
    bool operator == (
        const fixed_point& other) const {
        return value == other.value;
    }
    bool operator != (
        const fixed_point& other) const {
        return value != other.value;
    }
    bool operator <= (
        const fixed_point& other) const {
        return value <= other.value;
    }
    bool operator >= (
        const fixed_point& other) const {
        return value >= other.value;
    }
    bool operator < (
        const fixed_point& other) const {
        return value < other.value;
    }
    bool operator > (
        const fixed_point& other) const {
        return value > other.value;
    }
    friend std::ostream& operator << (
        std::ostream& out,
        const fixed_point& number);
};

// Quick and dirty output operator.
std::ostream& operator << (
```

```
std::ostream& out,  
const fixed_point& number) {  
    out << static_cast<const double>(number);  
    return (out);  
}
```

### 方案 93：验证优化代码与未优化代码版本

**问题：**我们已经决定，方案 92 对我们有效。但是如何说，我们是否已经正确实现了这个类？

**方案：**当使用另一个系统替换一个系统时，测试两次，然后比较结果。

在方案 92 中，我们以 **float** 表示像素，使用一个 **fixed\_point** 替换开始。两种代码应该产生相同的结果。

但是，作为黑客我们知道“应该”和“是”之间有很大不同。那是需要测试的地方。

一个优秀的测试，是运行未优化的程序并保存结果，来测试优化是否是正确的。然后，运行优化的版本，查明我们获得了同样的结果。

在真实的生活里，方案 92 被用于加速一个彩色喷墨打印机的复杂抖动算法。在输出上进行二进制比较时，结果不同。因此，尽管我们强大的数值分析专家说，两位数不足以产生相同的结果。

然而，当测试的图形打印时，优化后的结果与未经优化的版本，对于印刷委员会来说同样出色。所以，计算机可以告诉两个算法的不同，但是打印委员会不能。在喷墨业务中，印刷委员会制定了规则，因此，我们保留了更新，更快的算法。

这个故事是合理的，我很快计算出我放在这本书中的东西。

### 案例学习：优化位到字节

让我们看看一个黑客会怎么优化一个简单的函数。这个函数的工作是计算加载一个给定的位数时需要多少个字节。这是第一次出现的函数：

```
short int bits_to_bytes(short int bits)  
{  
    short int bytes = bits / 8;  
    if ((bytes % 8) != 0) {  
        bits++;  
    }
```



```
}  
    return (bytes);  
}
```

关于这个函数我们注意的第一件事是代码如此糟糕。作为黑客，我们经常面对这么糟糕的代码。

我曾经优化了一个程序，每次达到需要花费 8 秒运行的地方，需要花费 20 个小时。现在我是一个优秀的黑客，但我不是那么好。最初的程序编写得非常糟糕。为了对抗最初的程序员，这是他写过的第一个程序，尽管不了解 C 语言的许多基本功能，但他在实现复杂的加密算法方面做得非常出色。

**bits\_to\_bytes** 函数针对移动手机（ARM 处理器）。通过编译器运行代码，查看汇编代码，我们发现一些有趣的事情发生了。例如，这行的实现：

```
short int bytes = bits / 8;
```

产生的代码像这样：

```
movw r1, bits ; r1 (bytes) = bits  
asr r1, 3 ; r1 = r2 >> 3 (aka r1 = r1/8)  
lsl r1, 16 ; r1 = r1 << 16 (?)  
asr r1, 16 ; r1 = r1 >> 16 (?)
```

两个有趣的指令是什么，将结果左移 16 位，然后右移 16 位？咋一看，这是一段相当无用的代码。

处理器使用 32 位运算。在这个机器上，一个 **short int** 是 16 位。当系统做了除以 8，一个 32 位的结果产生了。所以，编译器生成两个指令，设计用于转换一个 32 位的值为一个 16 位的。

在除法和自增之后，这发生了。这给我们一共 4 条无用的指令。

使用 **int** 而不是 **short int** 来改变函数，消除这些指令，使我们的代码更快。

下一步是看看是否我们可以写一个更好的算法，消除那种条件：

```
int bits_to_bytes(int bits)  
{  
    return (bits + 7) / 8;  
}
```

这完全消除了所有的添加逻辑，节省了一些额外的指令。现在，真正的函数体只是充满指令。我们可以减少更多吗？

如何减少到 0 条指令？如果可能。所有我们需要做的是添加 **inline** 关键字：

```
inline int bits_to_bytes(int bits)  
{
```

```
    return (bits + 7) / 8;
}
```

现在，当优化器看起来是这样的一行：

```
store_size = bits_to_bytes(41);
```

它会优化为：

```
store_size = 6;
```

函数甚至没有调用。所有的计算在编译时间完成。

但是，我们没有完成优化。声明一个函数为 **inline** 和声明一个函数为 **static inline** 是有一个不同的。当一个函数被声明为 **inline** 时，编译器会嵌入它看到的函数调用，在有人从外部像调用这个函数的情况下，然后生成一个通用的非嵌入函数体。

为了解决这个问题，我们将函数声明为 **static inline**，并将其粘贴到头文件中。因此，在这个例子中，节省了几十个字节-总计-在一个 3.5MB 的程序中。

现在，作为黑客，还有一件事情我们需要考虑。当出现问题时会发生什么。毕竟我们从不心想调用者做正确的事情，我们可以用一个负数调用。我们需要回答这个问题 “-87 比特占用了多少存储空间？”

最容易做的事情是假设这永远不会发生，只是忽略它。但是如果这样做，我们需要在程序中记录这个事实。

```
/*
 * bits_to_bytes - Given a number of bits, return the
 * number of bytes needed to store them.
 *
 * WARNING: This function does no error checking so
 * if you give it a very wrong value you get a very wrong
 * result.
 */
```

假设一些糟糕的事情永远不会发生不是一个好主意。作为黑客，我们知道很多可能“永远不会发生”的事情实际上会发生。经常检查“不可能”是一个好主意。

例如，我们插入一个 **assert** 语句：

```
static inline int bits_to_bytes(int bits)
{
    assert(bits >= 0);
    return (bits + 7) / 8;
}
```

**assert** 语句的问题是它们使程序退出。在真实的生活里，这个代码存活在一个手机中，一个失败的 **assert** 会导致手机重置。这不是好的，因为手机重置发生的时候，它会播放“欢迎音乐”。终端用户考虑为什么他们的手机“一点也没有原因”地重启。

手机制造商面这个问题的“方案”是简单的。他们改变了代码，所以一个失败的 **assert** 无声地重启手机。这个代码依然非常错误，但是错误对于终端用户变得更不可见（听得见）？

而且，你应该记住断言可以被编译，所以，这段代码一点也没有提供保护。

由于这是 C++ 抛出异常是处理错误的一种方式：

```
static inline int bits_to_bytes(int bits)
{
    if (bits < 0) throw(memory_error("bits_to_bytes"));
    return (bits + 7) / 8;
}
```

这些错误检查选项是昂贵的。我们可以做的一件廉价的事情是确保错误用于不会发生。我们如何做？所有我们需要做的是使参数（和返回值）为 **unsigned**。

```
static inline unsigned int
bits_to_bytes(unsigned int bits)
{
    return (bits + 7) / 8;
}
```

最后，那有一个额外的方式处理这个错误。只是无声的改变函数，忽略它，然后返回一个默认值：

```
static inline int bits_to_bytes(int bits)
{
    if (bits < 0) return (0);
    return (bits + 7) / 8;
}
```

这通常不是一个好主意，因为在其他代码块中这样的代码趋向于隐藏错误。通常，你不想无声地修复事情。可能输出一个日志消息：

```
static inline int bits_to_bytes(int bits)
{
    if (bits < 0) {
        log_error(
            "Illegal parameter in bits_to_bytes(%d)",
            bits);
    }
}
```

```
        log_error( "Standard fixup taken" );  
        return (0);  
    }  
    return (bits + 7) / 8;  
}
```

在测试 `bits_to_bytes` 中，我们可以看到这确实是一个非常短的函数。但是它确实表明，当处理代码时，优秀黑客考虑的一些事情。这包括：

- 处理糟糕的代码。
- 知道编译器如何产生代码，设计你的代码优化信息使用。
- 最大化语言特性的使用，像 **inline** 和 **static inline**。
- 变得偏执。决定对糟糕的数据做什么。

从这个小程序中可以学到很多。作为黑客，我们一直学习。我们学习，研究，实验，然后做很多来得到关于一个程序进程的好的理解。

## 第 9 章：g++ 方案

GNU 的 *gcc* 包是最常用的 C 和 C++ 编译器之一。两件事情导致它的流行。首先，它是高质量的编译器。第二，它是免费的。

GNU 编译器以一些有用的方式已经扩展了 C 和 C++ 语言（也有一些无用的）。如果你愿意牺牲可移植性，非标准语言可能非常有用。

### 方案 94：特定结构初始化

**问题：**你有 2000 个元素的结构。为了初始化结构体你需要用准确的顺序指定 2000 个条目。很容易产生一个错误。

例如：

```
struct very_large {
    int size;
    int color;
    int font;
    // 1,997 more items
};
// Now create a instance of item
very_large something_big = {
    SIZE_BIG,
    FONT_TIMES,
    COLOR_BLACK,
    // 1,997 more items;
};
```

如果你在前面的例子中指出这个错误，恭喜。如果不是，然后你知道这个方案为什么如此必要。

**方案：**使用 gcc 特定结构初始化语法。

gcc 编译器有一个非标准扩展，允许你在初始化语句中命名字段。例如：

```
very_large something_big = {
    .size = SIZE_BIG,
    .font = FONT_TIMES,
    .color = COLOR_BLACK,
    // 1,997 more items;
};
```

通过命名东西，我们减少了以错误的顺序命名的风险。问题是我们已经建立

了一个非标准的程序。但是，除非你使用一个不平常的平台，你的系统上会有一个 gcc 编译器。

但是使用这个系统有一个问题。有可能省略一个字段。如果你执行它，得到一个默认值（0）。但是，如果你意外地省略一个值，会发生什么。它被赋值一个默认值，除非你告诉 gcc 警告你，否则没有警告。选项 `-Wmissing-field-initializer` 告诉 gcc，当你初始化的时候你想指定每个字段。如果你不这样做，一个警告会发出。

## 方案 95：检查参数列表的 printf 样式

**问题：**你写了一个叫做 `log_error` 的函数，使用 `printf` 样式参数。你如何告诉 gcc 检查像 `printf` 那样对参数的处理？

**方案：**使用 `__attribute__` 关键字告诉 gcc 检查什么。例如：

```
void log_error(const int level,
               const char* const format,
               ...) __attribute__((format(printf, 2, 3)))
```

（是的，需要双括号。）

`__attribute__((format...))` 语法告诉 gcc 像 `printf` 那样处理参数到 `log_error`。这个例子中，这个函数像 `printf`，格式字符串是函数的第二个参数，参数从第 3 个位置开始。

如果你打开了警告选项，gcc 会马上检查任何调用 `log_error` 的参数列表，来看看参数是否与格式匹配。

**注意：**如果你想同时使用 gcc 和非 gcc 编译器编译，你可能希望将下面的行放进你的代码。

```
#ifndef GCC
#define __attribute__(x) /* Nothing */
#endif /* GCC */
```

## 方案 96：打包结构体

参考方案 116。

## 方案 97：创建一个函数，返回值不应该被忽略

**问题：**一些函数返回的值永远不应该被忽略。例如，任何人想调用 `malloc`，

忽略结果是没有道理的。有什么方式强制人们使用一个函数的返回值？

**方案：**黑客不试着强制人们做事情。相反，他们使用信念。当涉及到让你们做正确的事情时，极高的信念可能成为一个极大的动力。

这个例子中，我们可以通过使用 g++ 编译器的 `__attribute__` 特性来使事情简单。设置 `warn_unused_result` 属性导致当函数的返回值被忽略的时候编译器会产生一个警告。

这是一个简短的例子：

```
#include <malloc.h>
#include <cassert>
void* my_malloc(size_t size)
    __attribute__((warn_unused_result));
void* my_malloc(size_t size) {
    void* ptr = malloc(size);
    assert(ptr != NULL);
    return (ptr);
}
```

现在，让我们使用这个函数。当然，我们错误地使用它，只是建立一个内存泄漏抛出结果。通常，这会是一个问题：

```
int main()
{
    // Memory leak
    my_malloc(55);
    return (0);
}
```

但是，因为我们使用了 `__attribute__((warn_unused_result))`，关于这个代码编译器会报错：

```
result.cpp: In function `int main()':
result.cpp:16: warning: ignoring return value of `void*
my_malloc(size_t)', declared with attribute warn_unused_result
```

## 方案 98：建立永远不返回的函数

**问题：**我们已经创建了一个叫做 **die** 的函数，它打印一个错误消息，然后退出程序。因此，当我们使用它时，编译器持续给我们警告。

例如：

```
#include <iostream>
void die(const char* const msg)
```

```
{
    std::cerr << "DIE: " << msg << std::endl;
    exit(8);
}
int compute_area(int width, int height) {
    int area = width * height;

    if (area >= 0) return (area);

    die("Impossible area");
}
```

问题是关于这个代码编译器持续报错：

```
die.cpp: In function `int compute_area(int, int)':
die.cpp:15: warning: control reaches end of non-void function
```

**方案：**使用 g++ 的 `noreturn` 属性。

例如：

```
void die(const char* const msg)
    __attribute__((noreturn));

void die(const char* const msg) {
    // ... function body
```

这告诉 g++ 函数永远不返回。**abort** 和 **exit** 函数有这个选项集。因为，我们已经告诉 g++，**die** 会永远不返回，当编译前面的例子时，编译器不再计较。

如果我们打开不可达到警告（**-Wunreachable-code**），如果我们尝试将代码放在 **die** 调用之后，编译器会计算。例如：

```
#include <iostream>
void die(const char* const msg)
    __attribute__((noreturn));
void die(const char* const msg)
{
    std::cerr << "DIE: " << msg << std::endl;
    exit(8);
}

int compute_area(int width, int height) {
    int area = width * height;

    if (area >= 0) return (area);

    die("Impossible area");
```



```
    return (0); // Return a default value
}
```

当编译时，我们得到错误：

```
g++ -Wunreachable-code -Wall -c die.cpp
die.cpp: In function 'int compute_area(int, int)':
die.cpp:18: warning: will never be executed
```

这会添加默认返回值的程序员一个想法，就是可能他的代码并没有像他期待的那样执行。当编译器对于奇怪代码警告你时，这很好。作为黑客，我们想最大化使用编译器代码检查。`__attribute__`方案让我们做那些。

## 方案 99：使用 GCC 堆内存检查函数定位错误

**问题：**内存破坏。我不想进入这些细节。如果你是一个经验丰富的程序员，你已经想起了你的糟糕内存共享问题。

**方案：**使用 GNU 的 C 库（glibc）内存检查函数帮助定位问题。

为了打开内存检查函数，你需要在你的程序中放置下面的代码：

```
#include <mcheck.h>

// ....

int main()
{
    mcheck_pedantic(NULL);
```

为了打开内存检查，`memcheck_pedantic` 调用必须在任何分配完成之前。参数是指向一个函数的指针，当一个错误被检测时这个函数被调用。如果这个参数是 `NULL`，那么一个默认的函数被使用。这个函数打印一个错误，并且退出程序。

`memcheck_pedantic` 函数引起内存分配函数在每个分配块的开始和结尾放置句子。每次你调用 `malloc`，`free` 或其他的内存分配函数，所有的这些句子会被检查。

当一个问题被检测出来时，程序会退出。

例如：

```
#include <malloc.h>
#include <mcheck.h>
int main()
{
    mcheck_pedantic(NULL);
```

```
char* ptr = (char*)malloc(10);

ptr[10] = 'X';
free(ptr);
return(0);
}
```

当运行这个程序时打印：

```
memory clobbered past end of allocated block
```

每次内存被分配或被释放的时候，**mcheck\_pedantic** 执行一次一致性检查。

每一个内存块都被检查。及结果，你的程序会比正常时运行慢。

这有一个快速的内存检查函数，**mcheck**。它打开指针检查，只为已经分配，重新分配或释放的指针。你可以通过调用 **mcheck\_check\_all** 强制系统检查所有的内存指针。

单个的指针可以被 **mprobe** 函数检查。

几乎所有的这些函数在 **glibc** 文档中记录。（在大多数 **Linux** 发行版中使用命令 **info glibc**。）一个异常是 **mcheck\_pedantic**。它一点也没再文档中提及。然而，它在头文件（**mcheck.h**）中被提及。所以，从这个方案中我们可以学到的另一课是总是查看头文件看看是否可以找到隐藏的宝藏。

## 方案 100：追踪内存使用

**问题：**内存泄漏。（并且使用 **valgrind** 时不可行的-参考方案 41。）

**方案：**使用 **mtrace** 函数追踪所有的内存分配和释放。

为了开始记录内存分配，你必须调用 **mtrace** 函数。你必须调用 **muntrace** 函数完成内存追踪。追踪信息会通过 **MALLOC\_TRACE** 变量放置在一个指定的文件中。

这是测试我们这个系统的一个小程序：

```
#include <malloc.h>
#include <mcheck.h>
int main()
{
    mtrace();

    char* data[10];

    for (int i = 0; i < 10; ++i) {
```

```
        data[i] = (char*)malloc(10);
    }
    for (int i = 1; i < 9; ++i) {
        free(data[i]);
    }
    muntrace();
    return (0);
}
```

这个程序使用下面的命令运行：

```
$ export MALLOC_TRACE=mtrace.log
$ ./leak
```

结果是提供给我们一个堆分配和释放历史的文件：

```
= Start
@ leak:[0x809b07c] + 0x80bb3b0 0xa
@ leak:[0x809b07c] + 0x80bb3c0 0xa
@ leak:[0x809b07c] + 0x80bb3d0 0xa
@ leak:[0x809b07c] + 0x80bb3e0 0xa
@ leak:[0x809b07c] + 0x80bb3f0 0xa
@ leak:[0x809b07c] + 0x80bb400 0xa
@ leak:[0x809b07c] + 0x80bb410 0xa
@ leak:[0x809b07c] + 0x80bb420 0xa
@ leak:[0x809b07c] + 0x80bb430 0xa
@ leak:[0x809b07c] + 0x80bb440 0xa
@ leak:[0x809b0a6] - 0x80bb3c0
@ leak:[0x809b0a6] - 0x80bb3d0
@ leak:[0x809b0a6] - 0x80bb3e0
@ leak:[0x809b0a6] - 0x80bb3f0
@ leak:[0x809b0a6] - 0x80bb400
@ leak:[0x809b0a6] - 0x80bb410
@ leak:[0x809b0a6] - 0x80bb420
@ leak:[0x809b0a6] - 0x80bb430
= End
```

现在，我们必须做的是配对 **malloc** 调用（通过+行来捐献）和 **free** 调用（通过-行来捐献）。任何没有匹配的 **free** 的 **malloc** 就是一个内存泄漏。

幸运的是，我们不需要手动匹配。程序 *mtrace*（与 *glibc* 一起分发）会为你做这个工作。所有你需要做的是提供给它你的执行文件的名字和日志名：

```
$ mtrace leak mtrace.log
```

```
Memory not freed:
```

```
-----
```

```
Address Size Caller
```

```
0x080bb3b0 0xa at /hack/leak.cpp:10
0x080bb440 0xa at /hack/leak.cpp:10
```

在 **mtrace** 函数上有一个限制。对于 **new** 和 **delete** 它没有用。（你得到通过使用标准 C++ 库生成的每个 **malloc** 和 **free** 的完美追踪。）所以，意识到它的这个限制。

内存跟踪和一致性检查功能不是 **glibc** 库中内置的唯一内存调试工具。有许多方式可以将诊断代码与内存函数链接起来。你可以阅读文档获得详细信息。

然而，在列出的最近几个方案中，你可以发现大多数的常见错误。

## 方案 101：产生一个回溯

**问题：****mcheck\_pedantic** 使用的默认退出函数有些微小。它打印一条错误消息，告诉你是失败的，但这就是全部。基本上你知道出了问题，现在猜猜在哪里。

**方案：**定义你自己的内存错误处理函数，然后使用 **backtrace** 得到一个调用栈，所以你知道你来自哪里。

首先，我们需要告诉 **mcheck\_pedantic** 我们想自己处理错误：

```
mcheck_pedantic(bad_mem);
```

现在，当发现一个内存问题，**bad\_mem** 函数会被调用。当然，我们需要定义函数：

```
static void bad_mem(enum mcheck_status status)
```

**backtrace** 函数将调用栈作为一个指针数组返回。为了使用这个函数，我们必须首先分配一个数组来保存这个数据，然后调用函数：

```
void* buffer[MAX_TRACE];
int stack_size = backtrace(buffer,
sizeof(buffer)/sizeof(buffer[0]));
```

这给我们一个指针集，当它涉及查找我们在哪里的时候是无用的。将这个列表转换为带有符号的可打印的东西时才是好的。幸运的是，那有一个函数来做这个，调用 **backtrace\_symbols**。它返回一个字符指针的数组，包含人们可读（大多数）的带有符号信息的我们的指针版本的。

```
char** stack_sym = backtrace_symbols(
buffer, stack_size);
```

现在，所有我们需要做的是打印结果：

```
std::cout << "Stack trace " << std::endl;
```

```
for (int i = 0; i < stack_size; ++i) {  
    std::cout << stack_sym[i] << std::endl;  
}
```

有一件额外的事情我们不得不做，就是退出程序。（由于内存破坏，我们现在无法做很多事情。）所以，我们的函数的最后一行是：

```
abort();
```

**abort** 调用是可选的，因为它创建了一个核心转储，稍后我们可以分析。

一个典型的运行这个程序产生的结果：

```
Stack trace  
trace [0x809b458]  
/lib/tls/libc.so.6 [0x401928f3]  
trace(__libc_free+0x17) [0x804e817]  
trace [0x809b510]  
trace [0x809b521]  
trace [0x809b52f]  
trace [0x809b563]  
trace(__libc_start_main+0x14c) [0x804c8dc]  
trace(__gxx_personality_v0+0x51) [0x804c701]  
Aborted (core dumped)
```

不幸的是，**backtrace\_symbols** 调用似乎不能在我们的程序中找到符号。（我们带有调试选项的编译，所以它们应该会显示。）

围绕这个问题有很多方式。最简单的是在执行和核心文件上运行调试器，使用它显示栈追踪。这使我们能够不仅可以查看栈追踪，而且可以查看核心转储时的源代码和变量的值。

```
$ gdb trace core.31957  
GNU gdb 6.3-3.1.102mdk (Mandrakelinux)  
# ... usual startup chatter ...  
Core was generated by `./trace'.  
Program terminated with signal 6, Aborted.  
#0 0xffffe410 in ?? ()  
(gdb) where  
#0 0xffffe410 in ?? ()  
#1 0xbffff08c in ?? ()  
#2 0x00000006 in ?? ()  
#3 0x00007cd5 in ?? ()  
#4 0x0808118e in raise (sig=6) at ../linux/raise.c:67  
#5 0x08057ca7 in abort () at abort.c:88  
#6 0x0809b4e5 in bad_mem (status=MCHECK_TAIL) at trace.cpp:28  
#7 0x401928f3 in mcheck_check_all () from /lib/tls/libc.so.6
```

```
#8 0x0804e817 in __libc_free (mem=0x6) at malloc.c:3505
#9 0x0809b510 in mem_problem () at trace.cpp:36
#10 0x0809b521 in do_part1 () at trace.cpp:40
#11 0x0809b52f in do_all () at trace.cpp:43
#12 0x0809b563 in main () at trace.cpp:49
(gdb)
```

现在，我们可以使用正常的 `gdb` 命令测试程序。例如，让我们看看函数中错误被发现的地方：

```
(gdb) list trace.cpp:36
31     void mem_problem()
32     {
33         char* ptr = (char*)malloc(10);
34
35         ptr[10] = 'X';
36         free(ptr);
37     }
38
39     void do_part1() {
40         mem_problem();
(gdb)
```

在这个例子中，错误非常明显，我们不需要进一步的调查。在真实的世界中，事情可能不是如此简单，我们可能需要花费超过 5 秒的时间来发现问题。但是，至少我们必须使用工具来及早的发现问题的，并且分析程序正在做什么。

另一种查明程序问题在哪里的是 `addr2line` 命令（`glibc` 包的一部分）。在程序中，它将一个地址转换为行数。例如：

```
$ addr2line -C -f -e trace 0x809b510
mem_problem()
/home/sdo/hack/trace.cpp:36
```

`-C` 标志告诉 `addr2line` 还原 C++ 符号。`-f` 选项导致函数名字按行列出。程序文件使用 `-e` 选项指定，最后，是被解码的地址。

当使用 `gdb` 不切实际时，这个函数非常有用，例如，当现场出现问题，并且客户不想向你发送一个大的核心文件时，但是愿意发送给你追溯输出。

黑客知道那有不止一种方式解决这个问题。有时，为了找到哪个有效，我们不得不使用所有的，但是在一个糟糕的工作环境，使用糟糕的工具，拥有不完整的信息，面临很多压力中修复问题，是一些黑客每天都要做的。

## 第 10 章：对抗方案

那有聪明的人，那有愚蠢的人。然后，那有自己认为是聪明的愚蠢的人。正是这种类型的人才可能造成一些真正的损害，尤其当他们认为自己是黑客时。

我早期遭遇一个人的一件事，当我不得不接管其中一个人，然后增强一个配置程序。他设计了一个非常聪明的顶级用户界面，基于一个非常聪明的第二个抽象 UI，基于另一个聪明的层次，等等，大约 8 个级别的复杂性。当然这都没有记录。

我和老板有一个有趣的交流，就是我告诉他修复这个程序需要花费我十周，而写一个新的程序只需要三周，并且这个程序可以做到旧的程序所有可以做到的，而且包括他等待的增强功能。新的程序没有 8 层抽象，但是它能完成工作。

一个好的方案是清晰的，容易理解的。简单总是更好，除非你是 Rube Goldberg 并以艰难的方式谋生。

本章专门讨论不是真正方案的方案。通过使用这些技术，制作它们的人向我们表明，他们不是真的聪明，而是真的，非常愚蠢。

### 对抗方案 102：变量声明使用“#define extern”

**问题：**为了定义一个全局变量，你不得不在一个头文件中建立一个外部声明，在一个模块中建立一个真实的声明。

例如，一个典型的头文件（debug.h）：

```
extern int debug_level;
```

和一个代码文件（debug.cpp）：

```
int debug_level;
```

相似的声明不得不放在两个地方。有没有一种方法消除这个副本？

**对抗方案：**通过聪明的使用预处理器你可以让一个头文件做两个声明。

通过创建包含变量声明的一个 debug.h 文件开始。任何使用错误模块的任何程序应该包括使用下面的语句使用头文件：

```
// Normal program file such as main.cpp
// ...
#include "debug.h"
```

然而，模块 `debug.cpp` 使用一个聪明的预处理器技巧：

```
// File debug.cpp
// ...
#define extern /* */
#include "debug.h"
#undef extern

// -- DO NOT PROGRAM LIKE THIS !!! --
```

**#define** 语句消除了 **extern** 关键字。所以现在这行：

```
extern int debug_level;
```

变成：

```
/* */ int debug_level;
```

因此，你生成了一个文件做了双重任务，节省了大量的输入。

这个代码有一个主要的问题。

首先，你已经重新定义了一个标准 C++ 关键字。现在，当有些人查看一个文件时，他们必须决定是否 **extern** 关键字真的意味着 **extern** 或它代表着其他的东西。当你未编辑时，C++ 是足够糟糕。使关键字在程序中的某个点上意味着一件事，而后来的另一件事使得程序难以调试十倍。

现在，假设一个毫无戒心的程序员遇到了 `debug.h` 文件，并决定要为其添加一个使用 I/O 流的函数。例如：

```
extern void set_output_file(std::ostream& out_file);
```

我们的 `debug.h` 现在看起来是这样的：

```
// debug.h (incomplete)
extern int debug_level;
extern void set_output_file(std::ostream& out_file);
```

因为，我们是优秀的程序员，我们想遵从这个规则，每个头文件都应该带着它所需要的任何头文件。这种情况下，我们引用 `std::ostream`，所以，我们需要 `fstream` 头文件：

```
// debug.h
#include <fstream>
extern int debug_level;
extern void set_output_file(std::ostream& out_file);
```

猜猜是什么。在 `fstream` 中的所有 **extern** 声明已经被编辑，并且他们的 **extern** 被移除。程序现在被破坏。所以，这个方案阻止程序员遵循良好的编码指南，并使头文件自给自足。



它变得糟糕。一个优秀的程序员查看 `debug.h`，看到：

```
#include <fstream>
```

是丢失的。当然，作为一个优秀的程序员他想要事情正确，所以他添加了这行。（当然不知道 `debug.cpp` 包含 `extern` 对抗方案。）现在，当这种情况发生时，优秀的程序员通过遵循良好的编程实践刚刚以奇怪和神秘的方式破坏了程序。突然，当他在调试模块中链接时，他在 `fstream` 模块中获取了重复定义的符号。

仔细检查源代码将找不到该程序。`fstream` 头文件是系统文件。他应该是正确的。此外，如果出现问题，问题将出现在很多程序中。所以问题不是 `fstream` 是“显而易见”的。但它确实如此。因为 `#define` 重新定义了 C++ 语言本身，所以 `fstream` 已经被重写，现在包含错误。

这种对抗方案用一种方式破坏有效代码，很难定位。不要使用它。

这种方案有一些其他问题：它不允许你在使用它们的代码附近定位变量定义，它不让你初始化全局变量，它很大的限制了你可以放置在一个头文件中的东西。但是与它造成的损害相比，它有一些小问题，因为它重新定义了 C++ 语言。

### 对抗方案 103：使用,（逗号）加入语句

**问题：**你希望代码尽可能紧凑，花括号使用太多字符。一定有一种方法可以使你的代码更紧凑。

**对抗方案：**使用,（逗号）加入语句。例如，不是这样写：

```
if (flag) {  
    do_part_a();  
    a_done = true;  
}
```

你可以不带花括号做同样的事情：

```
if (flag)  
    do_part_a(), a_done = true;
```

这节省两行，减少两个字符（花括号）。

使用 `for` 循环可以获得更多创意：

```
for (cur_ptr = first, count = 0; cur_ptr != NULL;  
     cur_ptr = cur_ptr->next, ++count);
```

我们刚才定义的 `for` 循环在一个链表中计数条目数量，通过创造性的使用,（逗号）操作符，我们已经消除了所有 `for` 之前和 `for` 主体内部的代码。

你可以发现对于使你的代码非常紧凑,（逗号）操作符是一个非常强大的工

具。但是，为什么你从没想过做这样的事情？紧凑的代码可能在计算的最初几天有用，当存储成本为每个字节几美元，但今天你可以在便利店购买 128M 的 USB 大哥霹雳碟。

使代码更紧凑会掩盖逻辑并使程序更难以阅读。因此，这样的代码更容易出错，并且当它们出现时更难以找到错误。当你使用这个方案时，你并没有表明你是一个多么聪明的程序员。经验丰富的黑客已经知道这个技巧。它们已经足够成熟到不能使用它。

### 对抗方案 104: `if (strcmp(a,b))`

**问题：**你不想在你的代码中放“不必要”的操作，尤其使用 `strcmp` 时。

**对抗方案：**使用 `strcmp` 的原始返回值。例如：

```
if (strcmp(name, key_name)) {  
    do_special_stuff();  
}
```

这段代码的逻辑是简单和容易理解的-极度误导的。

**问题：**当 `name` 时 `key_name` 或非关键字时，是否执行函数 `do_special_stuff`？记住当字符串是相同的时候，`strcmp` 返回 0。所以，如果你不是关键人，那你就很特别。

代码是误导的，因为大多数相似的函数会返回一个 `true/false` 值，`true` 是积极的例子。换句话说，一个如果 `strcmp` 遵从下面的规则，如果相等返回 `true`，不相等返回 `false`。

下面的代码是 OK 的：

```
if (strings_are_same(name, key_name)) {  
    do_key_stuff();  
}
```

这个例子中，程序的逻辑和程序员开始读取代码时会匹配并避免混淆。但是带有：

```
if (strcmp(name, key_name)) {
```

计算机的逻辑与任何人阅读预料的代码是冲突的，你会感动困惑。

一个优秀黑客的目标之一应该是清晰。毕竟，如果没有人理解你的工作，你怎么向世界展示你是多么好的黑客。`strcmp` 代码可以更好地编写为：

```
// strcmp return value for equal strings.
```

```
static const int STRCMP_SAME = 0;

// Check for average (non-key) users and process
// their special requests.
if (strcmp(name, key_name) != STRCMP_SAME) {
    do_special_stuff();
}
```

什么使这段代码出色？很容易从下面的语句中发现：

```
if (strcmp(name, key_name) != STRCMP_SAME) {
```

表示字符串是不同的。但是只是为了确保阅读此代码的人了解正在发生什么的注释，已经添加了一个注释，用英语告诉读者发生了什么。

## 对抗方案 105：if (ptr)

**问题：**很多函数返回一个指针，当错误的时候返回一个好的值和 **NULL**。我们有一个关于 **NULL** 的测试的快速方式。

**对抗方案：**使用 if (ptr)测试查看指针是否是 **NULL** 或不是。

例如：

```
char* ptr = strdup(name);
if (ptr) {
    process_ptr();
} else {
    throw(out_memory_error);
}
```

再次，我们有一个程序技术导致极度紧凑，简洁和混乱的代码。

这个代码的主要问题是像这样的指针不是一个布尔型的。因此，在 **if** 语句中使用它是没有意义的。

作为黑客，我们的目标是创造精彩和可理解的代码。省略短期的理解限制。（离开单词会导致短句，但限制却难以理解句子。）

键入

```
if (ptr != NULL)
```

而不是

```
if (ptr)
```

只需要几分之一秒的时间，但可以节省数小时或数天的调试时间，因为它可以让你更轻松地阅读你的程序。

作为黑客，我们想要人们理解我们的代码，惊叹于它，然后扩展它。加密代

码不能帮助我们实现这一目标。

## 对抗方案 106: "while ((ch = getch()) != EOF)"方案

**问题:** 你需要在一个流字符串读取或者执行一些其他相似的操作。

**对抗方案:** 使用一个通常的 C++ 设计模式:

```
// Don't code like this
while (ch = getch() != EOF) {
    // ... do something
}
```

关于这个代码有几个错误的地方。第一个是你必须非常谨慎确保变量 `ch` 不是一个 `char` 变量。它必须是一个 `int` 才能比较。

第二个问题是这个代码在单个语句中做多个操作。结果，代码是混淆和紧凑的。使代码详细和清晰比混淆和紧凑更好。

最后，行:

```
while (ch = getch() != EOF) {
```

是混淆的。你知道是否赋值操作符(=)比不相等(!=)具有更高的优先级?

例如，这两个语句中的哪一个等同于我们的 `while` 循环:

```
while ((ch = getch()) != EOF) {
while (ch = (getch() != EOF)) {
```

这是方案 19 派上用场的一个例子。

通常是这样的，紧凑的写一个 **while** 循环的方式。但是只因为它是普通的，并不意味着它不好。一个更详细的和更容易地做同样事情的方式是:

```
// Please code like this
while (true) {
    int ch = getch();

    if (ch == EOF)
        break;

    // .. do something
}
```

现在，有人告诉你这种方案是有用的。他们指出这是很通用的，因为它通用到人们认可和理解它。因为它是如此通用和可理解，人们用新的代码编写它，使它更通用和强制更多的人认可它，理解它。

但是，最终它是稀疏的模式，导致其它的稀疏代码模式。由于这个原因它应

该被避免。

## 对抗方案 107：使用#define 增强 C++语法

**问题：**C++语法并不是像它那么丰富。例如，没有单个语句来遍历链表的条目。

**对抗方案：**通过使用预处理器定义新的语言元素来增强 C++语言。例如：

```
#define FOR_EACH_ITEM(list) \  
    for (list_ptr cur_item = list.first; \  
        cur_item != NULL; \  
        cur_item = cur_item->next)
```

这些额外语法的问题是你现在从 C++语言到 C++语言进行改变。更糟糕的是，这是你自己的 C++语言版本。

对于某人管理 C++是足够困难。在语言中添加新的并未记录的特征使事情更糟。即使有人煞费苦心记录下来，维护程序的人也必须学习一种新的独特语言。

另一个问题是像这样的宏趋向于隐藏错误。例如，下面的会导致一个无线循环或一个核心转储：

```
FOR_EACH_ITEM(name_list)  
    std::cout << cur_item->name << std::endl;
```

很难去识别错误，因为误解是在 **FOR\_EACH\_ITEM** 宏中隐藏。对于这本书的读者，这个错误时容易识别的，因为所有你需要做的是找到宏的定义，查看前面的代码例子。在真实的生活里，事情会更糟糕。宏定义经常隐藏在一个头文件中，你不得不寻找它。

通常，写语法可见和标准的代码是很好的。这让追随你的人理解你的工作，所以他们可以改进和提高你的工作。

## 对抗方案 108：使用 BEGIN 和 END 而不是{ 和 }

**问题：**花括号{和}简洁而神秘。此外，它们看起来与你在 ALGOL 编程中看到的完全不同。

**对抗方案：**使用#define 定义一个宏集合，允许你生成看起来像 ALOGL-68 的 C 代码。

这是一个小的例子：

```
#define BEGIN {
```

```
#define END }

#define IF      if (
#define THEN    ){
#define ELSE    } else {
#define ENDIF   }
```

代码看起来像：

```
WHILE (system_ready())
  BEGIN
    int success = process_cmd();
    IF success THEN
      good();
    ELSE
      bad();
    ENDIF
  END
```

如果你是一个 **ALGOL-68** 程序员，这使事情简单和容易理解，如果你是一个 **C** 程序员，这它让事情接近不可能。

当大多数的程序员面对这类代码时，诅咒，然后使用预处理器移除宏，回到底层的 **C** 代码。

这个方案具有历史意义，因为最初 **Borne shell** 是用这种方式编写的。幸运的是，很快就被 **C** 版本所取代，这让所有必须维护这个程序的人感到安慰。

## 对抗方案 109：变量参数列表

**问题：**你有一个使用很多不同参数的函数。指定它们为传统的参数是不切实际的。所以你怎么办？

**对抗方案：**在很多参数中，使用 **C++** 的变量参数列表特性传递。

例如，让我们假设我们正在绘制一个盒子。我们可以指定一个宽度，高度，线的宽度，颜色和其它条目。所有这些是可选的，如果我们没有指定其中一个，则默认值被使用。

对于 **draw\_box** 函数，这是一个典型的调用：

```
draw_box(BOX_WIDTH, 5, BOX_HEIGHT, 3, NULL);
```

这个例子中，**NULL** 用于指明参数列表的末尾。

这个概念可以被进一步扩展。假设，这个盒子包含一系列单词。现在，我们可以做下面的这些：

```
draw_box(BOX_WIDTH, 5,  
         BOX_WORDS, "alpha", "beta", NULL,  
         BOX_HEIGHT, 3, NULL);
```

所以，单词列表（BOX\_WORDS）现在采用可变的单词列表作为它的参数。所以，我们在变量参数列表中有变量参数。这是极度聪明的！这也是极度危险和愚蠢的。首先，这完全违背了 C++ 类型检查机制。在这个例子中，BOX\_WIDTH 后面应该跟一个 int。但是如果我们跟在后面是一个字符串会发生什么。

```
draw_box(BOX_WIDTH, "large",  
         // ....
```

编译器没有方法知道已经提供的错误类型。运行时没有方式告知一个整数的指针。大多数系统，一个指针看起来像一个非常大的指针。所以，这个代码的结果是一个非常大的盒子和一个非常混淆的程序员。

但是，那不是最糟糕的。如果你略去一个 NULL 会发生什么？如果你忽略终止 NULL，函数没有方式告知参数列表的末尾在哪。会发生的是它会遍历参数查找一个 NULL 来停止。没有找到一个 NULL，它会继续，使用加载在栈中的任何参数，直到意外地发现一个 NULL。或意外，或一些邪恶的发生。

任何情况下，结果是不可预料和难以判断的。因为这个系统击败了许多 C++ 安全检查，所以应该避免它。

那有很多其他的方式传递大量的参数，并且提供灵活性和安全性。（例如参考方案 4。）替换使用一个。

## 对抗方案 110：不透明句柄

**问题：**你不想透露你的实现给用户。毕竟，他不需要知道你的内部代码是怎么的，只有如何调用它。

**对抗方案：**给用户一个不透明句柄使用。换句话说，将实现决定性的信息转换为无法使用的 void \* 或 short int 之类的信息。

例如：

```
typedef short int font_handle;  
typedef short int window_handle;  
  
font_handle the_font = CreateFont( "Courier" , 10);  
window_handle the_window = CreateWindow(100, 300);
```

```
DrawText(the_font, the_window, "Hello World", 30, 30);
```

现在，一些人可能问“这哪里出错了？”毕竟，用户不需要知道建立一个字形或一个窗口的细节。通过给他一个不透明类型，我们向他隐藏所有的信息。

问题是信息也向编译器隐藏。编译器完全没有方式区别一个字形句柄和一个窗口句柄的不同。

在我们的例子中，`DrawText` 函数采用一个窗口句柄和字形句柄作为它的前两个参数。在那种顺序！上面一样例子代码中，顺序被切换，编译器可能不会发现任何错误。

建立不透明句柄的类型检查，向用户隐藏实现是可能的，仍然允许运行时检查。这些在方案 31 中描述。

## 对抗方案 111：微软（匈牙利）表示法

**问题：**仅从变量名字分辨变量类型是不可能的。例如，`box_size` 是一个浮点数，一个整数或一个结构体？

**对抗方案：**微软表示法。

微软表示法（也被叫做匈牙利表示法）是一个每个变量的前缀是类型信息的命名系统。例如，整数以“i”开始，浮点变量以“f”开头。短整型以“si”开头，像 `siSize`。除非短整型是一个处理某些东西的句柄，然后加入“h”表示。例如，`hFCurrentFont` 是一个字形句柄。当然，一些人认为“h”太神秘，所以他们使用“hndl”，就是 `hndlCurrentFont`，但是，当相同的前缀用于字形句柄、窗口句柄和其他所有你不得不在 **Microsoft Windows** 中处理的其他句柄，可能导致混淆。

这个系统有很多的问题。首先，拥有编译器和高级语言的整个想法是从你那里隐藏实现的细节。将这些字根放在每个变量的开始提供给你信息，你真的不必在意。

编译器需要知道每个变量的详细类型信息，而你不需要。

第二个问题是存在非标准的一系列前缀字符。很多人发明自己的，留给你猜测 `pszName` 和 `szName` 之间的区别。答案是第一个是指向一个以一个零字符结尾的字符串的指针（通过 `char*` 实现），第二个是一个以零字符结尾的字符串（通



过 **char\***实现)。换句话说，这没有区别。

最后，存在一个灵活性和程序更新的问题。假设你已经实现了一个地址簿，地址被加载到一个结构体（**stAddress**）中。你改进程序，将结构体替换为一个类（**cAddress**）。你去遍历你的整个程序，然后更新每一个地址变量的前缀？如果你做了，它会是一个很大的工作量。（不好。）如果你不这样做，现在有使用一个错误前缀的变量。（非常不好。）

这是一个荒唐糟糕的局面。

你应该尽可能清晰地使用英语（或者无论你的母语是什么）编写变量名字。将类型信息留给编译器。不要将好的单词与随机词根混淆。没有使用符号系统的程序时模糊不清的，这些符号系统在为程序添加任何值时，使得值更加模糊。

### 对于英语的微软表示法

多年来，C 和 C++程序员已经设计出很多种方式来使他们的编写更清晰。结果，程序比以前更具有可读性。我常常思考，从编程中学到的课程是否可以应用在英语中。

采用微软表示法（又叫匈牙利表示法）的例子。在每个变量名字中放置前缀告诉我们变量的类型。如果我们使用“v”鉴别每个动词、使用“n”鉴别每个名词等等，难道英语不会更清晰吗？

毕竟，很多英语单词可以有超过一种含义。例如，“mall”同时是一个动词和一个名词。使用我们新的表示法，你可以编写像这样的，“aThere vWas aSuch aA nCrowd pAt arThe nMall aThat pnWe vWere vMalled”，人们可以很容易地理解你。

## 第 11 章：嵌入式程序方案

计算机的数量正在爆炸式增长。它们现在在手机、洗衣机、电视、烤面包机和手表中。为这些非标准平台编程确实是一个挑战。

当我在高中的时候，我得到我的第一个编程工作。老板告知，“你正在为 Camsco Waterject I 切割机创建嵌入式软件。”

“非常好，”我说。“机器在哪里？”

“你看那些盒子，”他指着房间回答说。确实，那有一个中等偏大的盒子挨着另一个。“它们包含我们将要构建的一部分。”

“非常好，”我说，“硬件文档在哪。”

“我们仍然处在设计硬件的过程中。只要我们一完成协议，我们会开始编写文档。”

“那关于 I/O 控制器的文档呢？”

“其中一部分是现成的，我可以为你提供文件。但是有一些我们正在制作它们，并且它们仍在设计中。你不得不等待那些设计还没稳定的信息。”

这个工作被证明是我所参与的一个最有趣的挑战。我学习到的部分是如何编写非常灵活的代码，使得它可以很容易配置在一个指定的特殊硬件。这主要靠自己，因为难以获得硬件规格，按时得到甚至更难，并获得正确的信息-不可能的。（参考下面的第一个方案。）

### 测试水刀

测试第一个水刀对于所有涉及的是一个挑战。这是有史以来生成的第一台水射流切割机，所以对我们所有人来说都是新的。为了得到正确的结果，我们队机器进行了超过一年的测试和调试。我们与购买机器的鞋匠达成了协议。他会提供给我们测试所用的原始材料，而我们给他在测试过程中所有的切片，以至于他可以用它们制作网球鞋-或许我们也是这样认为的。

为了得到持续的实时结果，我们使用单一尺寸进行所有的基准测试-9 对。因此，在大约 5-7 个月的时间里，我们什么也没做，只是切割了一批 9 对，调整机器，并将切好的零件云送给鞋匠。这是一家非常大的公司，很多人每年生产数十万双鞋。

在最后调整好机器并且工作良好之后，我们将其分解装箱，并运送给客户。

就在那时，我们接到了工厂领班的电话。“你是持续向我们运送 9 对的人吗？”

我们告诉他我们是。

“最后，我跟踪了你们。采购人员不知道你是谁，因为我们从未向任何人预定 9 对。直到现在我才想到查看我们的国会大厦设备采购组。”

“我们的切片有什么问题吗？”在这个点上我们有一点点担心。

“不完全是，但是你意识到你们发给我们的是 100009 对，并且没有剩余！”

## 方案 112：总是验证硬件规格

**问题：**硬件规格经常不完整，语无伦次，不可理解，过时或遗失。当我为 Camsco WaterJet 工作时，我不得不编写一个状态面板。这个设备包含 10 个警告灯。我从硬件人员那里得到了手写规范并决定测试。

灯 1 应该是“油压低”，它是“拉姆失效”。灯 2 应该是“水供应失败”，它是“定位器错误”。灯 3，“直流电源故障”被证明是“压缩机故障”。

所以，我编写了一个好的文档，描述系统哪个地方是真的奇怪，我把它交给硬件指南，告诉他“Woody，这些灯是混乱的。”我把纸递给他，“这是它们当前如何混乱的地方。”

他从我的手上接过文档，径直走向复印机，做了份拷贝。然后他递给我拷贝（不是最初的），然后说，“这是新的规范。”

**方案：**尽早测试，经常测试。

面对任何新硬件时，你应该做的第一件事是创建一个简短的诊断程序，测试它的基本功能。这个程序可以让你发现像：

- 命令不像文档记录的那样有对应的功能。
- 字节顺序问题（当然没有记录）。
- 文档包含非标准的位顺序表示。（我实际上已经读了一个硬件手册，它将最低有效位放在左边，最高有效位放在右边。）
- 文档的其它遗漏，错误，含糊之处和问题。

对于第一次处理新的硬件是一个困难和挑战的操作。许多硬件人员不谈软件，不知道如何编写合适的文档。（很多软件人员当涉及写东西的时候表现也很差。）

通过尽早测试你可以发现硬件是如何真正工作的，而不是它应该如何工作。通过这种方式，你可以编写一个真正完成工作的程序，而不是那个应该完成工作的程序。

### 方案 113：使用可以指定整数宽度的可移植类型

**问题：**C++没有定义 `int` 类型有多少位。（或任何其他类型。）在嵌入式系统中，为了与硬件交流，我们需要知道准确的位数。

**方案：**定义可移植的类型或使用已经为你定义的可移植的类型。

头文件 `<sys/types.h>` 包含为整数值准确定义的类型定义集。包括：

<code>int8_t</code>	<code>int16_t</code>	<code>int32_t</code>	<code>int64_t</code>
<code>u_int8_t</code>	<code>u_int16_t</code>	<code>u_int32_t</code>	<code>u_int64_t</code>

通过使用这些类型，你可以确保你定义的整数有你想要的准确位数。

如果你的系统在一个某处的头文件中没有这些类型，然后你可以定义你自己的。（当然在你发布它们之前验证它们。）

### 方案 114：验证结构体大小

**问题：**关于结构体的大小编译器的想法可能和你的不同。

这是为旧的 `RIO MP3` 播放器的结构体定义。因为这个结构体必须准确地匹配硬件规范，每个字段的偏移在注释中。其他需要记录的是这个块必须是准确的 512 字节长（再次，这个数来自硬件规范。）

```
// directory header
struct rio_dir_header
{
    u_int16_t entry_count;           //[0] Number of entries
    u_int16_t blocks_available;      //[2] Blocks available
    u_int16_t Count32KBlockUsed;     //[4] Blocks used
    u_int16_t block_remaining;       //[6] Blocks remaining?
    u_int16_t Count32KBlockBad;      //[8] Bad blocks
    int32_t TimeLastUpdate;          //[10] Last time updated
    u_int16_t check_sum1;            //[14] Checksum part 1
    u_int16_t check_sum2;            //[16] Checksum part 2
    char not_used_1[2];              //[18] Junk
    u_int16_t Version;               //[20] Version number
    char not_used_2[512 - 22];       //[22] Not used
};
```

当音乐被加载到 RIO 播放器，一些问题被发现。首先，每次你开始播放一首歌曲，你听到一声紧随歌曲的短促的噪音。第二，如果你按下快进按钮，播放器会跳到下一首歌。

所以这里发生了什么？这个问题尤其难以调试，因为没有 RIO 借口的官方文档，绝对没有办法知道设备内部发生了什么。

然而，你可能注意到我们只在这本书中包含了头文件定义，这可能给你一些灵感。实际问题涉及编译器放置结构体的方式。

通过测试我们可以发现每个字段真实的偏移量。结果与我们手算的有轻微的不同，放在注释里面：

```
u_int16_t block_remaining;          //[6] Blocks remaining?
u_int16_t Count32KBlockBad;         //[8] Bad blocks
int32_t TimeLastUpdate;             //[10 12] Last time updated
u_int16_t check_sum1;               //[14 16] Checksum part 1
```

所以，为什么两字节的 Count32KBlockBad 占据 4 个字节？答案是编译器想在一个 4 字节边界的上面对其 32 位的值（TimeLastUpdate）。偏移量 10 不是在一个 4 字节的边界上，所以编译器放置了一些填充字节，移动字段到下一个 4 字节边界。

所以，真正的结构体布局是：

```
u_int16_t block_remaining;          //[6] Blocks remaining?
u_int16_t Count32KBlockBad;         //[8] Bad blocks
u_char hidden_pad[2];               //[10] Added by compiler
int32_t TimeLastUpdate;             //[10 12] Last time updated
```

```
u_int16_t check_sum1;    //[14 16] Checksum part 1
```

所以,我们的 512 字节结构体实际上是 514 字节,一些字段位于错误的位置。

(一些人可能问一个问题“如果 **check\_sum1** 和 **check\_sum2** 在错误的位置,为什么 RIO 没有检测到校验和是错误的而显示一个错误消息?”好问题,但是作为一个软件黑客,你应该知道得比期望硬件设计师知道的一致性,理智性或常识更多。)

现在我们知道是什么引起了这个问题,我们可以做什么,防止在以后发生这样像这样的事情。

**方案:** 验证对比计算出的结构体大小和实际的大小。

简单地说,我们的程序需要下面的这行:

```
assert(sizeof(struct rio_dir_header) == 512);
```

对于软件和硬件共享的每个数据结构体都应该这样做。像我们已经看到的,在一个结构体上软件的想法和硬件的想法可能不同。

可以在方案 116 中找到解决此问题的可能方法。

## 方案 115: 当定义硬件接口的时候验证偏移量

**问题:** 编译器不仅填充结构体,而且移动周围的字段。

**方案:** 使用 **offsetof** 操作验证结构体中的每个字段的偏移。

回到我们先前的例子,让我们不只验证结构体的大小,而且还有每个字段的位置:

```
assert(sizeof(struct rio_dir_header) == 512);

assert(offsetof(struct rio_dir_header,
                entry_count) == 0);

assert(offsetof(struct rio_dir_header,
                blocks_available) == 2);

// ...
```

## 方案 116: 打包结构体来消除隐藏的填充

**问题:** 编译器在没有我们的允许下填充。当我们处理硬件时,这会搞砸事情。

**方案:** 使用 gcc 的打包属性告诉 gcc 使结构体尽可能紧凑。

```
struct rio_dir_header
```

```
{
    u_int16_t entry_count; //[0] Number of entries
    // ....
} __attribute__((packed)); // Hack 41
```

这是一个黑客充分使用编译器提供的特性的例子。如果你没有使用 gcc，阅读你的编译器文档。有可能它们与打包属性类似。

如果你的编译器不支持打包任何类型，你不得不将长字段 `TimeLastUpdate` 分割为两个小的 `u_int16_t` 字段，这样编译器会打包。

当遭遇像这样的问题，一个优秀的黑客会寻找一些方法，克服编译器设置的障碍。

### 方案 117：理解关键词 `volatile` 做了什么和如何使用它

**问题：**一个出色的编译器会移除无用的代码。在嵌入式编程中，一些代码并不像它看起来那么无用。

假设我们在处理一串 I/O 控制器。为了清除设备卸载三个字符的输入缓冲区。这是代码：

```
// Won't work

// The device is a memory mapped I/O device
// Initialize the pointer to the input register
char* serial_char_in = DEVICE_PTR_SERIAL_IN;

// ...
char junk_ch; // A junk character
junk_ch = *serial_char_in; // Read one useless character
junk_ch = *serial_char_in; // Clear second useless char
junk_ch = *serial_char_in; // Third character gone
                        // buffer now clear
```

所以，这个代码到底做了什么？它从设备中分配 `junk_ch` 顶部字符。这重复了三次，所以清楚了三个字符缓存。

但是编译器是聪明的。它查看代码，然后说，“第一个赋值是无用的。我们计算 `junk_ch` 的值，只是将它抛出。我可以消除那行，程序的效果一样。”

所以优化重写代码看起来像这样：

```
char junk_ch;           // A junk character
junk_ch = *serial_char_in; // Read one useless character
junk_ch = *serial_char_in; // Clear second useless char
```

```
junk_ch = *serial_char_in; // Third character gone
```

现在它查看第二行。它也可以被消除。

如果在程序的后面没有地方使用 `junk_ch`，第三行可以只被消除。在这个例子中，它并非这样，所以也被消除。最后的代码是被很好的优化：

```
char junk_ch;           // A junk character
junk_ch = *serial_char_in; // Read one useless character
junk_ch = *serial_char_in; // Clear second useless char
junk_ch = *serial_char_in; // Third character gone
```

由于 `junk_ch` 现在从未使用，编译器可能使变量消失，节省它使用的栈空间。所以，优化器通过消除一个无用的变量，减掉无用的指令和内存，节省我们的时间。这哪里不对？

答案是程序不再具备功能。设备不再被清除。这是嵌入式编程的一个问题，当处理内存映射 I/O 时，编译器不知道奇怪的边缘效应会出现。

**方案：**使用 **volatile** 关键字标识内存映射设备和其他的 **volatile** 数据。

**volatile** 关键字告诉 C++，一个变量可能在任何时间，在正常编程环境之外的控制下被强制改变。更具体地说，它告诉编译器不允许修改访问这个变量的次数或顺序。

```
volatile char* serial_char_in = DEVICE_PTR_SERIAL_IN;
```

现在，我们执行：

```
junk_ch = *serial_char_in; // Read one useless character
junk_ch = *serial_char_in; // Clear second useless char
junk_ch = *serial_char_in; // Third character gone
```

I/O 设备会被访问三次，然后我们会清除设备。

机敏的读者可能注意到我们违反了方案 3，没有在我们应该使用的地方使用 **const**。由于，我们改变指针的指向，应该添加 **const**：

```
volatile char* const serial_char_in =
    DEVICE_PTR_SERIAL_IN;
```

现在，在单个变量声明中我们有足够的高级关键字来表明我们使一个真正的黑客。

## 方案 118：理解优化器可以为你做什么。

**问题：**在我的早期生涯，我不得不使用一个慢速和有着糟糕设计的磁盘控制器。有一件事情是，它出错了，以下面的方式产生了一个中断：



1. 触发中断的行
2. 在一个状态寄存器中设置中断位，告诉系统中断为什么出现。

这是这个系统的一个问题。我工作在一个快速的处理器，这是个慢速的磁盘控制器。因此，整个中断服务程序可以在控制器执行步骤#1 和步骤#2 之间完成它的工作。

直到我发现这一点，我的代码一直在报错，存在伪中断和丢失的中断。

这个问题会启动中断，我想读取寄存器（什么也没设置-记录一个伪中断），中断跟踪会结束，然后控制器会设置指定 I/O 可用的状态位。（当然此时我已经解除了中断，因此 I/O 的中断丢失了。）

这个问题的“解决方案”是在中断服务程序开始时向代码添加一个延迟循环，以便让 I/O 控制器有时间捕获事件。

```
/*  
 * WARNING: This is highly processor dependent  
 */  
const int DELAY_COUNT = 100; // Loop this many times  
  
int result;  
  
for (int i = 0; i < DELAY_COUNT; ++i) {  
    result = 12 * 34;  
}
```

但是这段代码中有一些意外。首先，它执行多少次：

- a) 零次
- b) 一次
- c) 100 次

如果你猜是“a”，你是正确的-有时。但是“b”和“c”也是正确的-有时。

问题是这段代码优化器可能采用不同的方式。优化器可能查看代码，决定整个代码是浪费时间，所以代码被消除了。所以循环会执行零次。

如果 **result** 在代码的后面被使用，然后，通过循环一次就足够了，优化器可以删掉其他 99 次。

GNU C++编译器是聪明的。它的优化器说“这个代码看起来是无用的代码，你发现在一个延迟的循环中，所以我会产生代码遍历 100 次循环。”

第二个问题是“乘法完成多少次？”即使没有优化器，这个问题的答案也是

零。任何现代的编译器懂得如何做常量折叠，会在编译器评估常量表达式，所以它们不需要在运行时进行计算。

**方案：**实际上这里有两个方案。第一个是定期查看汇编代码，看看编译器对你做了什么。第二种使用 **volatile** 修饰符强制编译器不对代码进行优化。

让我们看看 GNU 编译器处理我们最初的延迟循环：

```
    movl $99, %eax
.L5:
    decl %eax
    jns .L5
```

所以，我们有了循环的部分，但不是乘法部分。这会产生一些延迟，但是还不够。

所以，让我们使用 **volatile** 变量重写循环。**volatile** 关键字告诉编译器这个变量时特殊的，不要假设这个变量的健全性。

这是新的，改进的循环：

```
volatile int result;
volatile int f1 = 12, f2 = 34;

for (int i = 0; i < 100; ++i) {
    result = f1 * f2;
}
```

现在，从汇编代码中我们可以发现，乘法是实际上发生了，我们的延迟循环正在引起一个延迟。

```
    movl $99, %ecx
    movl $12, -8(%ebp)
    movl $34, -12(%ebp)
.L5:
    movl -8(%ebp), %eax
    movl -12(%ebp), %edx
    imull %edx, %eax
    decl %ecx
    movl %eax, -4(%ebp)
    jns .L5
```

最后，一个优秀的黑客会设计一个短的测试程序来测试这个延迟循环的长度。延迟循环时极难得到正确的，处理器在为你加载的时候能有一些意外。

应该指出的是，黑客尽可能地避免编写像这样的延迟循环。这样的循环对于代码是危险的，不可移植，并且浪费 CPU 资源。只有在没有其它方法让程序员

运行时才应该创建它们。但现在至少如果我们被迫使用它们，我们可以让它们正确。

### 方案 119：在嵌入式程序中，尝试没有暂停地处理错误

**问题：**嵌入式程序运行在它们自己的环境下。用户通常很难（如果不是不可能）启动和停止服务。引起一个程序停止或崩溃的错误可以很容易地破坏整个设备。

如果你需要一个提醒，有关不良的错误处理可能会对系统造成多大损坏，请参考第 44 页的侧栏 A Real System Crash。

**方案：**从不暂停。甚至你暂停了，也要重新正确启动。

让我们举一个典型程序案例，一个客户端必须连接到一个服务器：

```
ConnectionType *the_connect =  
    new ConnectionType(CONNECTION_PARAMETERS);
```

问题是当服务器没有运行的时候，会发生什么？一个典型的非嵌入式应用的例子中，你紧急地得到一条错误消息。基本上，你告诉用户“你遇到一个问题，修复它，然后再次运行我。”

```
// Typical non-embedded code  
the_connection->connect_to_server();  
if (the_connection.bad()) {  
    std::cerr <<  
        "FATAL ERROR: Could not connect to server " <<  
    std::endl;  
    abort();  
}
```

**注意：**在这个例子中，ConnectionType 类设置一个错误标志而不是抛出一个异常。编程异常是一种更简洁的做事方式。但是它们用一些额外的语句混淆了这个例子，所以编写明智的标志很清晰。但是你应该在实际编程中使用异常。现在，在嵌入式编程中，用户不能暂停和启动程序。所以，如果一个重要的程序退出，对于他来说机器时无用的。甚至存在一个术语，动词“抻”像“当他尝试使用无线功能时，他把机器抻成了砖头。”

由于我们不想我们的系统像这样退出，我们需要修改我们的编程方法，如果可能，永远不要停止。这是一个嵌入式黑客如何处理连接服务器：

```
// Good embedded code
```

```
while (1) {
    the_connection->connect_to_server();

    if (! (the_connection.bad()))
        break;
    log_warning(
        "Unable connect. Will retry in %d seconds" ,
        CONNECT_DELAY);
    sleep(CONNECT_DELAY);
}
```

所以，如果服务器没准备好，会发生什么？在这个例子中，我们记录了一个错误（来告诉某人我们有一个问题），然后等待几秒（无需责怪服务器），重试。所以，即使服务器需要一段时间上线，我们最终会连接上它。如果服务器从不上线，然后我们从未上线，但是那不是我们的问题。取决于谁在管理服务器，找出它上线的方式，当它上线时我们准备好了。

所以，基本的规则是程序会持续尝试直到连接成功。它从不会停止，因为几乎不可能再次启动它。（在糟糕的程序系统中，重新死机的唯一方式是重启机器。）

但是，如果你的程序遇到了一个问题，就是重试不是一个选择，会发生什么。例如：

```
try {
    char *buffer_we_must_have = new char[BUFFER_SIZE];
    // ....
}
catch (std::bad_alloc e) {
    // ?? What do we do now?
}
```

直到我们有足够的内存，否则没有方式让我们可以持续重试 **new** 操作。毕竟，我们确实希望当我们的程序持续运行时，通过用户外出，购买一个内存芯片，然后安装它，来解决这个问题。

不，这个例子中我们必须退出。程序不再有效。但是，在嵌入式程序中，任何时候我们应该保持事情运行。所以，围绕退出问题的一个方案是确保程序是否确实退出，我们再次正确启动它。

所以，我们建立了一个小的执行程序，监视我们的主程序：

```
while (1) {
    system( "main_program" );
    log_warning( "main_program stopped. Will restart." );
}
```

```
sleep(SETTLE_TIME);  
}
```

作为黑客，我们知道绕过“不可能”限制总是有一些方法。在嵌入式系统中，我们不能停止，如果我们确实停止，我们不会停止太长时间。无论发生什么，系统必须运行。

## 方案 120：检测饥饿

**问题：**在一个嵌入式系统中，一个高优先级的进程卡住，使低优先级的进程饥饿是有可能的。

**方案：**使用一个看门狗和一个低优先级进程的联合，保持系统运行。

一个看门狗是一个设计用于帮助保持系统运行的设备。在你设置之后，它坐在那里，等待你每个一段时间 ping 它。如果，由于一些原因，你没有在一个指定的时间段联系它，它会重置你的系统。

通过使用一个看门狗和一个低优先级的任务来给你的看门狗挠痒痒，你可以确保你的系统有一个问题然后锁起来，然后它会重置，接着运行。

这是一个典型的程序，旨在为看门狗定时器提供服务：

```
// If we don't do anything for 5 minutes, reboot  
const int WATCHDOG_TIME = 60 * 5;  
// Tickle the watchdog every 1/2 minute  
const int SLEEP_TIME = 30;  
int main()  
{  
    init_watchdog();  
    set_my_priority(LOWEST_POSSIBLE_PRIORITY);  
  
    while (true) {  
        tickle_watchdog();  
        sleep(SLEEP_TIME);  
    }  
}
```

思路是看门狗期待每五分钟有人给它挠痒痒。如果没有，它会生气，然后重启系统。

我们每 30 秒给它挠痒痒。（这很低于 5 分钟）所以我们应该安全的。

如果由于某些原因系统确实非常忙，如此忙以至于在五分钟内它不能给我们可怜的挠痒痒进程任何 CPU，然后程序不会给看门狗挠痒痒，看门狗会失望，我

们就会重启。

有许多基于硬件的看门狗可用，以及仅软件内核驱动。这些都被记录在内核源目录`/usr/src/linux/Documentation/watchdog`

应该注意的是，这种方案是最后的解决方法。它应该仅用于让系统永久挂起或重置系统的选择。希望你已经在系统中内置其他的保护措施，防止你不得不让看门狗完成它的工作。但是对抗糟糕的事情它是一个好的故障安全。

## 第 12 章：Vim 编辑方案

Vim 是一个基于旧的 UNIX Vi 程序的强大的编辑器。然而，它有超过 Vi 命令集的异常扩展的特性。

因为 Vim 是程序员制造给程序员的。由于它包含很多设计用于使程序员的工作更容易的命令。这章讨论的主题包括：

- 语法着色
- 使用 Vim 的内置 make 系统
- 自动缩进
- 源代码搜索
- 查看复杂函数逻辑
- 日志文件查看

即使你在 Microsoft Windows 上编程，你也可以使用 Vim。尽管编程环境，像 Visual C++，提供给你一个编辑器，Vim 在编辑方面比内部编辑器优越很多。

**注意：**这章假设你已经开启 Vim 扩展特性。在 UNIX 和 Linux 系统中，通过建立 ~/.vimrc 文件完成。在 Microsoft Windows 中这是默认完成的。

### 什么是 Vim

Vim 是一个使编辑文本更快的高质量文本编辑器。它为这样的人设计，像程序员，不得不编辑大量文本。

Vim 的一个问题是它的陡峭的学习路线。注意的游标移动键是 **h**, **j**, **k** 和 **l**。如果你观察键盘，你会发现这些键在你右手的“home”类型位置。换句话说，对于大多数人它们是最快和最容易的键。

但它们完全不是助记符。（例如，右边是 **l**（小写的 L。））学习 Vim 花费时间，但是一旦你学会它们，你可以比其它任何编辑器编辑得更快。

## 方案 121：打开语法着色

**问题：**使用简单的等宽字体进行文本编辑，没有颜色。

**方案：**打开 Vim 的语法着色

为了在 Vim 打开语法着色，简单执行命令：

```
:syntax on
```

现在，例如关键字，字符串，注释和其他的语法元素会有不同的颜色。（如果你使用一个黑白的终端，它们会有不同的属性，例如粗体，下划线，闪烁等等。）

实际上，你不用常常执行手动执行这个命令。相反，将它放到你的 \$HOME/.vimrc 文件，然后每次 Vim 启动的时候它会自动执行。

## 方案 122：使用 Vim 的内部 make 系统

**问题：**你不得不关闭 Vim 去构建一个程序，然后回到 Vim 去修复问题。

**方案：**使用 Vim 的:make 命令执行构建。

Vim 的:make 命令通过运行 make 程序构建程序。（在 Windows 上使用的是 nmake。）为了启动构建进程执行命令：

```
:make [arguments]
```

如果产生了错误，它们被捕获，编辑位置在第一个错误出现的地方。

查看一个典型的:make 会话。（典型的:make 会话生成更多的错误和更少的愚蠢错误。）图 6 显示了结果。从这你可以看出你在两个文件中有错误，main.c 和 sub.c。

```
:!make | & tee /tmp/vim215953.err
gcc -g -Wall -o prog main.c sub.c
main.c: In function `main':
main.c:6: too many arguments to function `do_sub'
main.c: At top level:
main.c:10: parse error before `]'
sub.c: In function `sub':
sub.c:3: `j' undeclared (first use in this function)
sub.c:3: (Each undeclared identifier is reported only once
sub.c:3: for each function it appears in.)
sub.c:4: parse error before `]'
sub.c:4: warning: control reaches end of non-void function
make: *** [prog] Error 1
2 returned
"main.c" 11L, 111C
```



```
(3 of 12): too many arguments to function `do_sub'
Press RETURN or enter command to continue
```

图 6: **make** 输出

当你按 Enter (Vim 调用返回)，你看到图 7 中显示的结果。

```
int main()
{
    int i=3
    do_sub("foo");
    ++i;
    return (0);
}
~
(3 of 12): too many arguments to function do_sub
```

图 7: 第一个错误

编译器移动到第一个错误的地方。这是 `main.c` 的第 6 行。你不需要指定文件或行数，Vim 自动知道 位置。下面的命令到达下一个错误出现的位置（参考图 8）：

```
:cnext
int main()
{
    int i=3
    do_sub("foo");
    ++i;
    return (0);
}
~
(5 of 12): parse error before `}'
```

图 8: **:cnext**

命令 **:cprevious** 或 **:cNext** 到前一个错误。类似，命令 **:clast** 到最后一个错误，**:crewind** 到第一个错误。**:cnfile** 到下一个文件的第一个错误消息（参考图 9）。

```
int sub(int i)
{
    return (i * j);
}
~
~
~
```

```
~  
~  
~  
(7 of 12): `j' undeclared (first use in this function)
```

图 9: `:cnfile` 命令

如果你忘记了当前的错误是什么，你可以使用下面的命令显示它：

```
:cc
```

如果你已经运行了 `make`，产生了你自己的错误文件，你可以通过使用 `:cfile error-file` 命令告诉 Vim。`error-file` 是 `make` 命令或编译器的输出文件的名字。如果 `error-file` 没有被指定，文件通过使用的 `'errorfile'` 选项指定。

其他可能对你有用的命令是 `:copen` 命令。它打开一个包含一系列错误的新的窗口，通过移动它和按 `<enter>`，让你选择你想测试的错误。

最后 `:help` 命令用于得到展示在这里的所有命令的文档。

## 方案 123：自动缩进代码

**问题：**缩进 C++ 程序是一件苦差事。

**方案：**让 Vim 自动完成它。

Vim 编辑机器包含一个非常智能的缩进系统。为了访问它，你所有需要做的就是使用下面的命令打开它：

```
:set cindent
```

默认缩进大小是 8 个空格。如果你想一些不同，像例如缩进大小是 4，你需要执行这个命令：

```
:set shiftwidth=4
```

再一次，这些通常放在 `$HOME/.vimrc` 文件，这样每次 Vim 启动的时候它们会执行。

现在，你键入代码，当你按下回车，程序会自动缩进。图 10 阐释 `'cindent'` 如何工作。

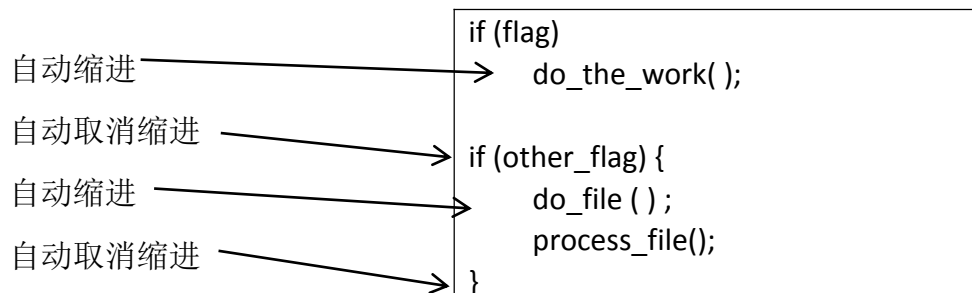


图 10: cindent

**注意：**对于所有类型的文件，你可能不想都打开‘cindent’。它真的会把一个文本文件弄乱。下面的命令，当把它们放在一个.vimrc 文件中，只对 C 和 C++ 文件打开‘cindent’。

```
:filetype on
:autocmd FileType c,cpp :set cindent
```

## 方案 124: 缩进错在的代码块

**问题：**提供给你一个已经编辑多次的遗留程序，缩进是无效的。

**方案：**使用 Vim 内部缩进功能，让它为你工作。

为了缩进文本块，使用 Vim 执行下面的命令：

1. 即将光标放在被缩进的第一行。
2. 执行 **V**（大写的“v”）命令启动“视觉”模式。
3. 使用任何 Vim 的移动命令，移动到被缩进的最后一行。缩进的代码块为高亮。（参考图 11.）

4. 执行 Vim 命令=来缩进高亮的文本。

从这开始  
————→

按 V 移动到这儿  
↓

```
// Code intentionally indented wrong
int do_it()
{
    if (today)
        do_today();
        do_now();
    do_required();
do_finish();
}
int main()
{
    do_it();
    return(0);
}
```

按=结果：

```
// Code intentionally indented wrong
int do_it()
{
    if (today)
        do_today();
        do_now();
}
```

```
    do_required();
    do_finish();
}

int main()
{
    do_it();
    return(0);
}
```

图 11: 缩进代码

## 方案 125: 使用标签导航代码

**问题:** 你在处理一块你不熟悉的代码, 希望一步步的遍历代码。你到达一个调用 `do_the_funny_bird`, 你想看看这个函数做了什么, 但是你不知道它在哪里定义。在这样的情况下, 你怎样高效的导航代码。

**方案:** 使用 `ctags` 命令产生一个本地文件 (叫做 `tags`) Vim 可以用于定位函数定义。

在你开始编辑之前, 你需要生成 `tags` 文件。这使用命令完成:

```
$ ctags *.cpp *.h
```

现在, 当你在 Vim 中, 你想找到一个函数定义, 你可以通过使用下面的命令跳到函数定义:

```
:tag do_the_funny_bird
```

这个命令会寻找函数, 即使它是另一个文件。**CTRL-]**命令将光标跳到单词的标签。这使得搜索繁琐的 C++ 代码变得容易。

例如, 假设你在函数 `write_block` 中。你可以看到它调用了 `write_line`。但是 `write_line` 做了什么? 通过将光标放在调用 `write_line` 上面, 键入 **CTRL-]**, 你会跳转到这个函数的定义 (参考图 12)。Write\_line 函数调用 `write_char`。你需要查明它做了什么。所以你定位光标到 `write_char`, 然后按 **CTRL-]**。现在你位于 `write_char` 的定义 (参考图 13)。

```
void write_block(char line_set[])
{
    int i;
    for (i = 0; i < N_LINES; ++i)
        write_line(line_set[i]);
}
```



**CTRL-]**到了 `write_line` 的定义（如果需要会转换文件）。命令 **:tag write\_line** 做同样的事情。

```
void write_line(char line[])    CT
{                               of w
    int i;                     if n
    for (i = 0; line[0] != '\0')
        write_char(line[i]);
}
~
"write_line.c" 6L,
```

图 12：使用 **CTRL-]**进行标签跳转

```
void write_char(char ch)
{
    write_raw(ch);
}
~
"write_char.c" 4L, 48C
```

光标在这时，使用 **CTRL-]**，  
让我们来到这。

图 13：跳转到 `write_char` 标签。

**:tags** 命令显示一系列你已经遍历的标签（参考图 14）。

```
~
:tags
# TO tag FROM line in file/text
1 1 write_block 1 write_block.c
2 1 write_line 5 write_block.c
3 1 write_char 5 write_line.c
>
Press RETURN or enter command to continue
```

图 14： **:tags** 命令

现在回来。**CTRL-T** 命令到达前面的标签。这个命令计数参数，指定跳回多少个标签。

所以，你已经向前了，现在回退。让我们再次向前。下面的命令到达列表中的标签。

你可以使用一个计数的前缀，向前跳转多少个标签。例如：

```
:3tag
```

图 15 阐明了各种类型的标签导航。

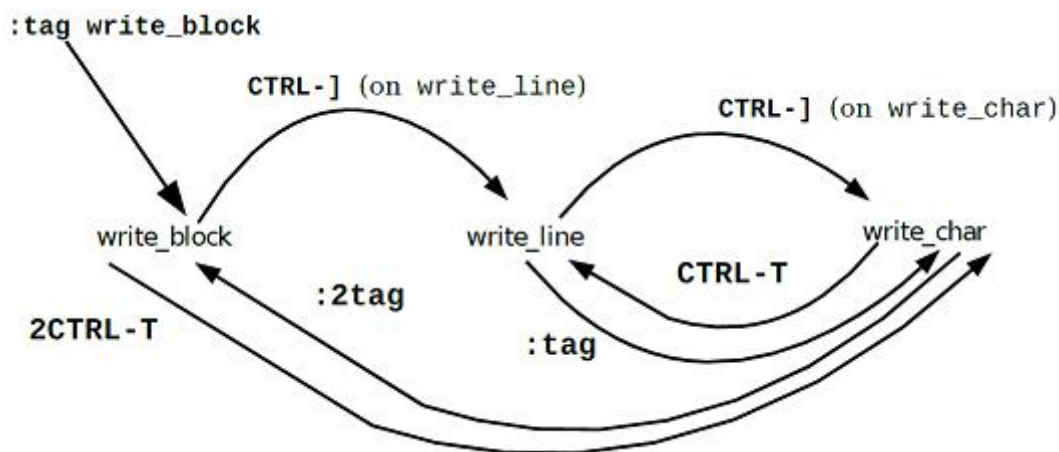


图 15: 标签导航

## 方案 126: 你需要定位你只知道部分名字的过程

**问题:** 你“有点”知道你想要查找的过程名字吗？对于 Microsoft Windows 程序员, 这是一个普遍的问题, 因为 Windows API 中的过程命名约定非常不一致。UNIX 程序员好不到哪去。只有命名转换的不一致才是一贯的; 唯一的问题是 UNIX 喜欢讲字母留在系统调用名称之外。(例如, `creat`)。

**方案:** 使用 `:tag` 命令的正则表达式搜索过程。

如果过程名以/开始, `:tag` 命令假设名字是一个正则表达式。如果你想查找一个过程, 名字为“一些写一些,”例如, 你可以使用下面的命令:

```
:tag /write
```

这查找所有名字中带有单词 `write` 的过程, 定位游标在第一个。如果你想查找所有以 `read` 开始的过程, 你需要使用下面的命令:

```
:tag /^read
```

如果你不确定过程是否是 `DoFile`, `do_file`, 或 `Do_File`, 你可以使用这个命令:

```
:tag /DoFile\|do_file\|Do_File
```

或者

```
:tag /[Dd]o_[Ff]ile
```

这些命令可以返回多个匹配。你可以使用下面的命令得到一系列标签:

```
:tselect name
```

图 16 显示了一个典型 `:tselect` 命令的结果。

```
~
# pri kind tag file
> 1 F C f write_char write_char.c
```

```
void write_char(char ch)
2 F f write_block write_block.c
void write_block(char line_set[])
3 F f write_line write_line.c
void write_line(char line[])
4 F f write_raw write_raw.c
void write_raw(char ch)
Enter nr of choice (<CR> to abort):
```

图 16: **:tselect** 命令

第一列是标签的数量。第二列是优先列。这包含 3 个字母的联合。

F 完全匹配（如果缺少，则忽略大小写匹配）

S 静态标签（如果缺失，一个全局标签）

F 当前文件中的标签

**:tselect** 命令的最后一行给你一个改进，使你可以输入你想要的标签数目。

否则你只能按 Enter（Vim 术语中的<CR>）让事情孤立。

**g]**在游标标识下做一个**:tselect**。**:tjump** 命令只对像**:tselect** 命令有效，除非选择结果只有一个条目，它自动被选择。**gCTRL-]**命令在游标单词下做一个**:tjump**。

一些其他相关的命令与这个标签选择集有关，包括：

**:count tnext** 到达下一个标签

**:count tprevious** 到达前一个标签

**:count tNext** 到达下一个标签

**:count trewind** 到达第一个标签

**:count tlast** 达到上一个标签

图 17 展示如何在一个**:tag** 或**:tselect** 命令的匹配标签之间使用这些命令。

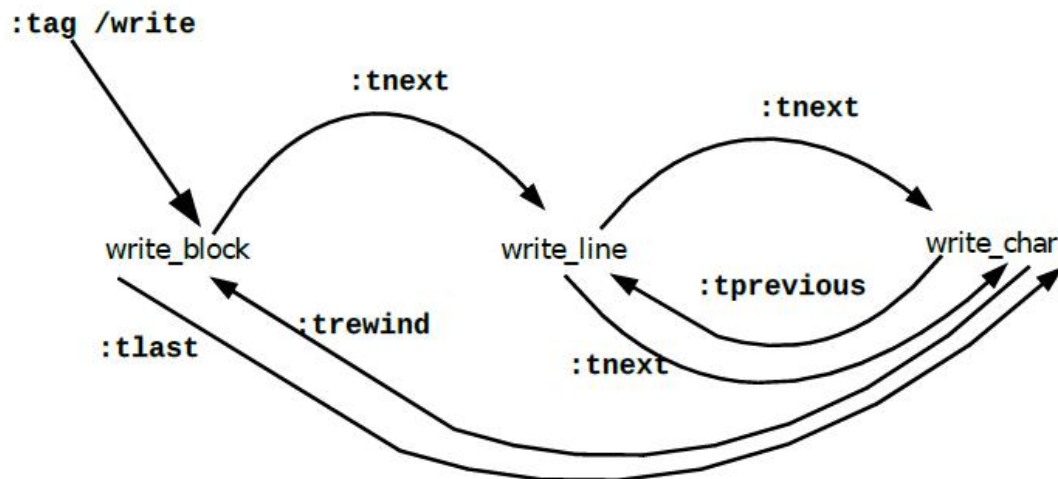


图 17: 标签导航

### 方案 127: 使用:vimgrep 搜索变量或函数

**问题:** 你想发现变量被使用的每个地方。

**方案:** 使用 Vim 的内部:vimgrep 命令为你查询。

:vimgrep 命令表现得像:make。它运行外部程序 grep，捕获输出。为了查找变量 ground\_point 的所有匹配项，例如，你使用这个命令：

```
:vimgrep /\<ground_point\>/ **/*.cpp
```

\<...\>告诉 Vim 只查找完整的单词。Ground\_point 是你正在查找的字符串。

最后，一列搜索的文件，所有目录（\*\*）和所有的 C++文件（\*.cpp）。图 18 展示结果。

```

    ++i;
    return (0);
}
}
main.cpp:5: int i=3;
main.cpp:7: ++i;
sub.cpp:1:int sub(int i)
sub.cpp:3: return (i * j)
(1 of 4): : int i=3;
Press RETURN or enter command to continue

```

图 18: :vimgrep 输出

**注意:** 关于 C++语法，grep 程序什么也不知道。所以它会查找 ground\_point，即使它出现在字符串或注释的内部。

你可以使用:cnex, :cprevious 和:cc 命令对一系列匹配进行翻页。:crewind 达



到第一个匹配，**:clast** 到达上一个。最后，下面的命令到达下一个文件的第一个匹配：

```
:cnfile
```

## 方案 128：查看大型函数的逻辑

**问题：**你得到一个像这样的遗留代码：

```
if (flag) {
    start_of_long_code();
    // 1,834 more lines of code
    end_of_long_code();
} else {
    return (error_code);
}
```

你如何计算这样一个函数的“逻辑”？

**方案：**使用 Vim 的折叠特性。

以将光标放在有困惑的代码的第一行开始。

```
if (flag) {
    start_of_long_code();
    // 1,834 more lines of code
    end_of_long_code();
} else {
    return (error_code);
}
```

按 **V** 启动可视模式。这行会被高亮。

```
if (flag) {
    start_of_long_code();
    // 1,834 more lines of code
    end_of_long_code();
} else {
    return (error_code);
}
```

到达长代码的末尾。你可以有几种方式：

1. 使用通常的光标移动命令例如 **j**（向下）或 **/**（搜索）移动光标。
2. 向上到花括号并按 **%**（找到匹配的括号），然后向上移动一行。
3. 按下操作符挂起命令 **ib** 来选择 **{}** 中的文本。

当你完成移动光标到行，**{}** 内部的会被高亮。

```
if (flag) {
```

```
start_of_long_code();
// 1,834 more lines of code
end_of_long_code();
} else {
    return (error_code);
}
```

现在，输入命令 `zf` 来“折叠”高亮文本。代码的 1836 行现在被一行替换，告诉你这里有一个折叠的代码被替换。

```
if (flag) {
  +--- 1836 lines: start_of_long_code(); -----+
} else {
    return (error_code);
}
```

编辑器现在让你了解函数的逻辑是怎么样的。

如果你想再次查看文本，将游标放在折叠的行，然后键入 `zo`。

这里有很多，所以你可能像要浏览帮助文本。为了在命令记录中获得帮助执行下面这些命令：

帮助命令	话题
<code>:help V</code>	可视模式（高亮）
<code>:help %</code>	匹配{}（或其他的东西）
<code>:help v_iB</code>	选择括号内的（{}中的文本）
<code>zf</code>	建立折叠
<code>zo</code>	移除（打开）折叠

方案 129：使用 Vim 查看日志文件

**问题：**你得到一个日志文件，但是它有 18373 行。你如何减少它的大小。

**方案：**使用你的文本编辑器。对于文本操作和浏览它是一个理想的工具。

你可以使用 Vim 编辑器查看一些东西。我们假设你已经熟悉一些简单的命令，比如搜索和剪切。

但是那有一些其它我们可以使用的命令。例如，假设我们只对包含字符串“Frame Header”的行感兴趣。我们可以使用下面的命令告诉 Vim，删除除了包含“Frame Header”的行：

1. 达到代码中的第一行（**1G**）。
2. 开启一个操作，让文本经过一个筛选命令。要过滤的文本部分，从当前光标位置运行到文件末尾（**!G**）<感叹号><字符 G>

3. 输入过滤命令。这个例子中，我们使用 `grep` 搜索行，所以我们的过滤命令是：**`grep 'Frame Header'<enter>`**。

这个命令作为一行字符串是：

```
1G!Ggrep 'Frame Header'<enter>
```

结果是一个只包含我们感兴趣的信息的一个日志文件，如果我们使幸运的，只需要适量的信息来发现错误。

在 Vim 有很多方式操作文本。如果在这里列出来就太多了。

扩展日志和一个优秀的文本编辑器的结合，使得定位和查看我们想要的信息变得容易。这种非标准地使用文本编辑器，是一个优秀黑客可以思考“开箱即用”，并充分利用对于他来说可用工具的一种方式。

## 第 13 章：聪明但是无用

不是所有的方案都有实际的用处。一些黑客发明做事的非常聪明的方式，只是显示了一些非常聪明的代码，所以他们可以拥有吹牛的权利。这并不是这是完全无用的努力。弄懂如何在不使用临时变量值的情况下交换两个变量的值，需要你了解计算机和编程中一些较为模糊的方面。尝试和学习这些技能是优秀黑客的真正特性之一。

### 方案 130：翻转变量 1 和 2

**问题：**一个变量 `flag` 可以有值 1 或 2。如果它是 1，则变为 2，如果它是 2，则变成 1。

现在最直接的方式是这样做：

```
if (flag == 1) {
    flag = 2;
} else {
    flag = 1;
}
```

偏执的黑客会使用一个 **switch**，并检查超出范围的值：

```
// This is the best and safest solution if we don't
// care about speed
switch (flag) {
    case 1:
        flag = 2;
        break;
    case 2:
        flag = 1;
        break;
    default:
        assert( "flag is not 1 or 2" == 0);
        abort(); // Make sure we stop
}
```

如果我们不介意速度，并且希望兼顾简洁和偏执，先前的位代码是优先选择的方案。

但是这个例子中，我们的目标是表明我们使如何聪明，然后产生一个非常聪明的方案。

**方案：**使用一个简单的减法语句，让变量在两个值之间切换。

现在，你可以花费几秒钟验证这条语句的正确性。

然而这个代码功能，是不完整的。如果你必须要在你的程序中包含像这样聪明的代码，请添加注释告诉追随你的人你做了什么。

```
// This clever piece of code flips the flag variable
// between the values of 1 and 2.
flag = 3 - flag;
```

### 方案 131：不使用临时变量交换两个数

**问题：**你如何不使用临时变量的情况下交换两个数。

**方案：**使用下面的代码：

```
void swap(int& a, int& b) {
    a ^= b;
    b ^= a;
    a ^= b;
}
```

这是经典的 CS201 答案。但是作为黑客，我们永远不会满足于孤立的东西。因为我是一个黑客，我想看看在真实的世界中是如何操作的。

然后我们发现另外一种不使用一个临时变量来交换变量的方法：

```
static void inline swap(int& a, int& b) {
    int tmp = a;
    a = b;
    b = a;
}
```

现在，你可能思考，我怎么能说这没有使用一个临时变量，当我在函数的第二行声明了一个临时变量。秘密是我声明这个函数是 **inline**，然后开启了优化。这让编译器免于计算交换两个变量的最好方式。

让我们看看优化器为我们做了什么。第一个测试代码看起来这样：

```
Int main()
{
    int a = 1;
    int b = 2;

    print(a,b);
    swap(a,b);
    print(a,b);
    return(0);
}
```

```
}
```

使用编译器运行这个代码，查看产生的汇编代码，我们看到编译器重写了我们的主函数：

```
print(a,b);  
print(b,a);
```

swap 调用被完全消除。所以，不仅编译器删除了临时变量，它也删除了所有的代码。你不会得到比这更多的优化。

如果我们改变我们的测试，强制变量的实际交换，我们得到下面的代码：

```
mov reg1, a  
mov reg2, b  
mov a, reg2  
mov b, reg1
```

现在，作为临时变量寄存器没有计数（至少我做了计数）所以再一次，我们有一个没有临时变量交换变量的例子。

这个问题让我们表明一个真正黑客的几个属性。第一个是愿意跳出框框进行思考。任何人都可以在网络上查看标准代码。但是只有一个优秀的黑客会问问题“嗨，如果……会发生什么”，然后进行一系列的测试回答那个问题。如果黑客是真的优秀，你会发现你自己的答案是惊讶的，比“标准”答案好得多。

### 方案 132：不用临时变量，在一个字符串中翻转单词

**问题：**你如何在不使用一个临时变量的情况下，在一个字符串中翻转单词。

**解决方案：**翻转字符串，然后遍历字符串，翻转单词。

例如，以这个字符串开始：

```
Now is the time
```

翻转整个字符串：

```
Emit eht si woN
```

然后翻转独立的单词：

```
time eht si woN  
time the is woN  
time the is Now
```

证据表明从一个字符串的结尾移动一个单词到另一个是一个不同的操作。翻转一个字符串是简单的。所以，而不是移动单词到它们合适的位置，我们翻转它们，然后翻转单词，让它们以正确的方式出现。

代码在下面：

```
#include <iostream>
#include <ctype.h>
// Given two character pointers, reverse
// the string between them
static void rev_chars(
    char* ptr1,
    char* ptr2
)
{
    char tmp;
    while (ptr1 < ptr2) {
        tmp = *ptr1;
        *ptr1 = *ptr2;
        *ptr2 = tmp;
        ptr1++;
        ptr2--;
    }
}
// Reverse the words in a string
// (Without a temporary)
void reverse(char* const str)
{
    // Reverse the entire string
    rev_chars(str, str + strlen(str)-1);

    // Pointer to the first character of a word
    // to reverse
    char* first_ptr = str;

    // Keep looping until we run out of string
    while (*first_ptr != '\0') {
        // Move up to the first letter of the word
        while (! isalpha(*first_ptr)) {
            if (*first_ptr == '\0')
                return;
            first_ptr++;
        }
        char* last_ptr; // The last letter of the word + 1

        // Find last letter
        for (last_ptr = first_ptr+1;
            isalpha(*last_ptr); ++last_ptr)
            continue;
```

```
    // Reverse word
    rev_chars(first_ptr, last_ptr-1);
    // Move to next
    first_ptr = last_ptr +1;
}
}
```

优秀的黑客会超越眼前的问题，然后问“为什么你要做这样的一件事情？”可能是某些命令解析器正在使用这些单词，而这些解析器需要反向使用它们。如果那是真的，那就没有理由翻转单词，在命令解析器中分离它们。只是在逆向的顺序中命令，调整命令解析器分离字符串为单词。

伟大的黑客不仅在他们之前能够解决问题，而且超越问题，提供影响更大范围的解决方案。

### 政府-反黑客

一些黑客为政府工作，当他们碰到编写非常糟糕的函数，从一个 JOVAL 版本转换一簇代码到另一种。

可以算出，他们可以重写代码，代码变得整洁和更有效。为了确保他们的新设计对于已经存在的代码是有效的，他们搜索每一处函数调用的地方。

他们什么也没找到。什么也没有。函数从没被调用。

所以他们找到老板，“这个函数从未被使用，我们可以丢弃它。”

老板告诉他们，他知道函数从未被使用。至少三个发行版本中没有被调用。但是因为这是一个政府程序，做文书工作的成本远远超过支付某人更新费用的成本。

### 方案 133：使用单个指针实现一个双向链表

**问题：**对于每个链接，你如何值使用单个指针实现一个双向链表？

（这个问题最初是问，那是当内存花费\$\$\$而不是 c，就说得通。）



**方案：**异或（XOR）前向和后向链接，在指针中加载值：

```
node.link = next_ptr ^ prev_ptr;
```

当前向遍历链表时，你可以使用指向前一个的指针重构指向下一个节点的指针。

```
cur_ptr = first_ptr;
prev_ptr = NULL;

while (cur_node != NULL) {
    // Do something with the current node
    next_ptr = cur_ptr->link ^ prev_ptr;
    prev_ptr = cur_ptr;
    cur_ptr = next_ptr;
}
```

一个相似的系统可以被用于你想遍历链表。

这个代码中省略了一些强制转换和大量细节。而且，我让读者计算插入和移除节点。

然而这些天，内存是廉价的，程序员是昂贵的，将双向链表实现为双向链表更具成本效益，因此这只是一个学术练习。

### 方案 134：不使用一个锁，访问共享内存

**问题：**如何在不使用测试和设置或上锁指令的情况下，同步共享内存的两个进程。

**方案：**有几个算法可以做这件事情。其中之一列在下面。

这个算法使用下面的代码进入临界区并访问共享资源。

1. 设置一个 **flag** 标识这个进程现在正在尝试进入临界区。
2. 设置一个 **turn** 变量来标识这个进程想进入。**turn** 变量会保存最近一个尝试进入的进程的 ID。
3. 如果其它进程想进入临界区，我们是最近一个访问 **turn** 变量。（此例中使用了轮转消耗等待的 CPU。）
4. 做临界区的事务。
5. 设置 **flag** 标识我们不在临界区中。

```
// Index for flag and turn variables
enum process {PROCESS_1 = 0, PROCESS_2 = 1};
```

```
// If true, I'm trying to go critical
volatile bool flag[2] = {false, false};

// Who's turn is it now
volatile int turn = PROCESS_1;

void process_1()
{
    flag[PROCESS_1] = true;
    turn = PROCESS_1;
    while (flag[PROCESS_2] && (turn == PROCESS_1))
        continue; // do nothing

    do_critical();
    flag[PROCESS_1] = false;
}

void process_2()
{
    flag[PROCESS_2] = true;
    turn = PROCESS_2;
    while (flag[PROCESS_1] && (turn == PROCESS_2))
        continue; // do nothing

    do_critical();
    flag[PROCESS_2] = false;
}
```

作为黑客，我们趋向于思考超出问题的边界。在这个例子中，我们谨慎地使用 **volatile** 关键字来确保内存实际上被改变，优化器不会修改我们的代码。

但是硬件也可以对我们耍技巧。让我们假设这些进程运行在不同的 CPU 上。如果进程有缓存，**flag** 和 **turn** 的改变只会保存在一个本地缓存，不会被拷贝到共享内存。如果是这样的情况，我们需要在这个系统中添加一些缓存刷新指令。

还有其他的一些问题要问，例如为什么我们没有某种硬件锁定机制，或至少一个测试和设置指令。有很多没有答案的问题需要思考。

但是，当它涉及在多线程编程中计算所有可能发生的有趣的事情。你不得不意识到任何进程可以运行在任何时间，任何共享变量可以在任何时间改变，当它涉及这类编程时，是真的会有影响。

## 方案 135：回答面向对象的挑战

**问题：**有些人热衷于他们的编程方法，并拒绝承认任何可能更好的系统。

很多人认为面向对象编程时唯一拿来使用的方式，如果每个人都使用面向对象设计，所有世界的编程问题会被解决。作为一个长期的黑客，我经历过很多潮流，包括专家编程，人工智能，极限编程，结构化设计等等，还有其它的。所有的这些承诺结束世界上的编程问题。它们中都未曾做到。

一个面向对象的提议者层发出一个挑战。“不存在一个程序，通过使用面向对象无法做的更好，”他说。这个挑战，是寻找这样一个程序。

**方案：** `true` 命令。

为了公平，我挑选了一个真正做有用工作的程序作为一个例子。它实际上是 Linux 和 UNIX 标准命令集的一部分。命令是 `true`。

它的工作是返回 `true` 的状态到操作系统以用于脚本编写。下面是一个使用这个命令的 shell 脚本的小例子：

```
if [ true ]; then
    echo "It is true"
fi
```

现在，这是我编写的 C++，没有使用面向对象设计：

```
int main()
{
    return (0);
}
```

我想寻找面向对象的大师如何通过用户对象使这个程序更好。

现在，有些人可能会说有点延伸了规则。我是，但是那是黑客应该做的。我们测试我们生活的世界的限制，有时我们结束制造新的限制。

我们也不要将我们自己锁在一种技术中，因为有人说它是好的。对于我们使用它，它不得不实际上是好的。作为黑客，我们唯一热衷的是良好的代码，并将使用任何东西来创建它。

关于这个问题有一些其他的事情需要注意。首先这个代码的 C 版本与 C++ 版本完全相同。

在 UNIX 中，这最初作为包含 0 行的 shell 脚本实现。

在我们的 Solaris 机器上，它是一个 shell 脚本，其中只有 5 个版权声明，警

告诉我内容是未发布的繁多代码，版本号：1.6.没有命令。

版本号最有趣，因为脚本没有代码。从 1.0 版本到 1.5 版本他们得到了什么错误，使其在 1.6 版本中被修复？Linux 附带的命令版本包含选项。一个打印命令的版本号，另外的打印帮助文本，告诉你唯一的选项是打印版本号。顺便提一下，我系统上的版本号是 5.2.1。再一次，有人不得不思考以前的版本出了什么问题？

所以，true 命令提供两个命令。一个是它表明像“我可以使用对象使它更好”这样如此绝对的语句不是总是对的。它也表明管理者，版权律师和其他力量如何为一个程序添加不必要的复杂性，这个程序时我所知道的最接近无程序的程序。

## 附录 A：黑客名言

一个无效程序的加速是不着边际的。--未知。

亵渎是所有程序员都知道的一种语言。--未知。

不要相信需求、管理或用户。就像他们不知道他们真的想要的那样编程。通常他们不对。--Steve Oualline。

日程安排的目的是告诉你你确实太迟了。--未知。

Oualline 的文档法则：90%时间的文档会丢失。剩余的 10%，9%的时间会成为错误的版本。那一份时间，你有正确的文档和文档的正确版本，它是用中文写的。

O'Shea 的法则：墨菲是最优的。

对于一个程序来说，满足要求是不可能的，除非要求已经真的被定义。--Steve Oualline。

跟上日程表示不可能的，除非你真的有一个。--Steve Oualline。

为什么从来没有足够的时间做它做对，但是总有足够的时间做完它。--未知。

你程序的前 90%会花费你分配的 90%时间来创建。剩下 10%也是如此。--Steve Oualline。

“生存还是毁灭，--这是个问题。”布尔代数上的莎士比亚，村庄，第三幕，场景 1。

“该死的指令，正在被教育，回来困扰着发明者，”维护程序的莎士比亚。麦克白戏剧 1，场景 6。

## Grace Hopper

第一个编译器（COBOL）的发明者，以寻找一个连接中继器计算机的死亡的飞蛾而闻名。（这件事情产生了一个传说（不正确），这是第一个计算机的错误。）

关于标准的奇妙之处在于它们有很多可供选择。

“语言中毁灭性最强的语句是：我们一直用这种方式完成它。”

如果它是一个好主意，开始做它。比起获得授权道歉更容易。

## Linus Torvalds

Linux 的作者。

任何程序都是有用的。

我通常是一个非常务实的人：就是有效，有效。

在很多情况下，对于一个商业公司来说，一个程序的用户接口是最重要的部分：程序是否正确，似乎是次要的。

Linux 相比其它项目有更少的规则，任何人都可以做他们想做的事情。

Microsoft 不是邪恶的，他们只是制造了的确糟糕的操作系统。

我从 Linux 得到的网络收获的形式是，拥有了解并信任我的网民，并且我可以反过来依赖他。

PowerPC 的内存管理可用于吓唬小孩子。

看，你不仅要成为一个优秀的程序员来创建像 Linux 这样的系统，你也必须是一个偷偷摸摸的混蛋。

如果你需要超过 3 层的缩进，无论如何你会搞砸，应该修复你的程序。

你知道你是聪明的，但也许你想知道从现在开始两周你做了什么。

交谈是廉价的。给我你的代码。

真的，我不是要摧毁 Microsoft。那只是一个完全无意的副作用。

事实上，ACPI 是由一群 LSD 的高级程序员设计的，并且是业内最糟糕的设计之一，这显然使得任何时候运行它都非常难看。

如果有足够的眼球，所有的漏洞都很浅显。

实际上，我会声明一个坏程序员和一个优秀的程序员的区别在于，他认为他的代码或数据结构谁更重要。坏的程序员担心代码。优秀的程序员担心数据结构和它们的关系。

## 附录 B：你知道如果……你是一个黑客

你知道如果……你是一个黑客。

你房间的电脑多于电视。

你至少通过 e-mail 地址知道七个人，但是已经忘记了他们的名字。

你观看了两个人在电脑桌上做爱的电影，试着想起背景中是哪种类型的计算机。

你至少记住一个午夜送到计算机实验室的披萨店的位置。

你靠着拉面生活超过一周。

你破解了一个程序，看看它为什么被勾选。

你遇到一个完全陌生的人，问他正在使用什么类型的笔记本电脑。

每个来你家找你的人都带着计算机问题。

你至少一次取下所有电脑的盖子。

在 Grace Hopper 以后，取名给你的女儿



Grace Oualline (3 岁)

一段时期你曾经使用术语 TCP/IP。

你的妻子曾经全裸的站在计算机旁，然后问你“你想在计算机上工作还是做爱？”然后你不得不思考答案。

你思考答案，然后决定再修复一个错误。

你在带到迪士尼乐园的笔记本电脑上编写程序。

在尝试破解东西，算出它是如何工作之前，玩不超过一个小时的电脑游戏。

你曾经计算出游戏是如何工作的，决定改进它。

将家用电器连接到计算机。

曾经说“现在我知道`&`和`&&`的区别”给完全理解你意思的人听。（`&`发音“and”。）



## 附录 C：黑客罪行

### 使用字母 O, I, l 作为变量名

倘若你没有注意到我们正在讨论大写字母“O”，大写字母“I”和小写字母（l）（小写的 L）。

事实上，我不得不在这本书中包含前面的句子，应该告诉你，很容易将这些变量看做其它的。例如，你可以告诉我字母 O 和数字 0 的不同吗。如果你可以，然后，你会知道我在上一句中把它们弄反了。

### 不分享你的工作

成为黑客最大的一部分是帮助其他人成为黑客，或者至少帮助他们更高效的使用他们的计算机。GPL 全部停止分享。只要你分享你的工作，你可以使用别人的工作。

### 没有注释

我曾经听说，下一个伟大的编程语言会更容易编写，以至于你不需要注释。不幸的是，这是我在 30 年前听说的，问题中的语言是 FORTRAN II。关于 COBOL 他们也这样说。他们是错误的。

英语仍然是与人们交流的最好方式之一。不使用它表达你的意思，会导致你创建的一个神秘的混乱不能共享给其他人。

..

### *IncOnsisTency*

当你一致性编程，你建立的模式允许其他人很容易地理解和扩充你的代码。不一致会产生混淆，然后扰乱工作流程。

（不一致性书写意味着你必须为复制编辑解释，为什么你的奇怪格式真的只是一个幽默的尝试。他们保持想念这个笑话，然后尝试修复它。）

## 复制代码（通过剪切和粘贴编程）

有一类程序员，当他们面对一个问题时，会简单地使用他的文本编辑器复制一些代码来做相似的，做一些小的改变，然后发布他的方案。他的“新”代码的95%是拷贝的，5%是原始的工作（如果是这么多）。这类程序员叫做骇客。（不管在名字上的相似性，“骇客”和“黑客”是完全相反的。）

一个黑客修改存在的功能，使得它对于大量观众来说变得更通用和更有用。

剪切和粘贴编程是创建不可能维护的大型程序的一种方式。黑客喜欢小的，好的手艺，优秀设计的代码，而不是快速构造的大型混乱。

## 附录 D：为黑客准备的开源工具

### ctags - 函数索引系统

原始的 UNIX ctags 确实有限制，所以 Exuberant Ctags 被创建。它有很多超过基本 ctags 功能的扩展特性。

Darren Hiebert 编写，可以在 <http://ctags.sourceforge.net/> 找到。

### dxygen

一个 C 或 C++ 程序（其它语言也是）中嵌入文档的系统。当用于一整个系统，对于生成代码文档来说这是一个非常有效的工具。

从 <http://www.doxygen.org> 获取。

### FlawFinder

用于分析代码的潜在安全问题。

从 <http://www.dwheeler.com/flawfinder/> 获取。

### gcc - GUN C 和 C++编译器套件

它带有很多高质量的扩展功能，使得编程更容易和更可读。它是黑客为黑客编写的，现在它出现了。

在 <http://www.gnu.org> 获取。

### lxr

Linux 交叉引用。实际上，这个会交叉引用任何程序，生成一组超链接文件，使得更容易导航大型程序。

这个程序可以在 <http://lxr.linux.no/> 获取。

### Perl（对于 perldoc 和相关工具）- 文档系统

Perl 包括在 POD（明文旧的文档）格式中操作文档的工具。这个格式是文档化 Perl 代码的标准方法，但是它对于 C 和 C++ 也有效。

你可以从 <http://www.perl.org> 获取 perl。

### **valgrind（内存检查工具）**

用于内存破坏检测的工具，也可以用于检查未初始化的内存。

你可以从 <http://www.valgrind.org> 下载程序。

### **Vim（Vi 升级版）**

一个文本编辑器。已经说过，它让工作变得很好，包含设计用于让编程更容易的很多特性。不像完整的开发环境（IDE）做很多工作，Vim 编辑文本，它就是这样。确实，它有接口用于编译，还有其它的工具，但是它的主要工作是文本编辑。

Vim 的学习曲线有些陡峭，但是一旦你学会如何使用它，你可以比使用任何其它编辑器更快地编辑文件。

Vim 可以从 <http://www.vim.org> 下载。

## 附录 E：安全设计模式

### 1. 消除副作用。将++和--放在它们自己的行

```
++i;      // Good
--i;      // Good

i++;      // Good but inefficient (See ###)

a = i++;  // Bad
a[i] = i++; // VERY bad
```

### 2. 不要将赋值语句放在其它语句中

```
// Bad and probably wrong
if (x = 5) ...

// Good
x = 5;
if (x != 0) ...
```

### 3. 定义常量

如果可以，使用 **const**：

```
const int WIDTH = 50;
```

如果你必须使用**#define**，将值放在()中：

```
#define AREA (50 * 10)
```

### 4. 使用内联函数而不是参数化的宏

```
inline int square(int i) { return (i*i); }
```

如果你必须使用参数化的宏，将括号放在每个参数两边：

```
#define SQUARE(i) ((i) * (i))
```

### 5. 使用{}消除含糊的代码

```
// Very bad
if (a) if (b) do_alpha() else do_beta();

// Good and indented too
if (a) {
```

```
    if (b) {
        do_alpha();
    } else {
        do_beta();
    }
}
```

有些人（有些 Perl 程序员）说你应该一直保护大括号。

## 6.让你做的一切变得明显，即使什么也不做

```
for (i = 0; buffer[i] != 'x'; ++i)
    continue;

switch (state) {
    case 1:
        do_stage_1();
        // Fall through
    case 2:
        do_stage_2();
        break;
    default:
        // Should never happen
        assert( "Impossible state" == 0);
        abort();
}
```

## 7.在一个 switch 中总是检查默认的案例

参考上面的。

## 8.优先级规则

- 1 乘法（\*）和除法（/）优先于加法（+）和减法（-）。
- 2 在其它任何东西两边放置小括号（）。

## 9.头文件

总是保护你自身的头文件。

```
/* File: report.c */
...
#include "report.h"
```

放置双重包含：

```
/* File report.h */
#ifndef __REPORT_H__
#define __REPORT_H__
// ....
#endif /* __REPORT_H__ */
```

## 10.检查用户输入

当检查用户输入时，总是检查什么是被允许的，永远不要检查排除的是什么。这种方式中，如果你犯了错误，你可能会意外地阻止一些好的数据，但是你不会让坏的数据通过。

## 11.数组访问

```
assert((index >= 0) &&
       (index < (sizeof(array) / sizeof(array[0]))));
value = array[index]; // or
array[index] = value;
```

## 12.复制一个字符串

```
strncpy(dest, source, sizeof(dest));
```

## 13.字符串连接

```
strncat(dest, source, sizeof(dest) - strlen(dest));
dest[sizeof(dest)-1] = '\0';
```

## 14.复制内存

```
memcpy(dest, source, sizeof(dest));
```

## 15.清零内存

```
memset(dest, '\0', sizeof(dest));
```

## 16.查找一个数组中的元素数量

```
n_elements = sizeof(array) / sizeof(array[0]);
```

## 17.使用 gets

不要使用 **gets**。没有方式让它安全。相反使用 **fgets**。

## 18.使用 fgets

```
fgets(string, sizeof(string), in_file);
```

## 19.当建立不透明的类型时，使它们可以被编译器检查

```
struct font_handle {  
    int handle;  
};  
  
struct window_handle {  
    int handle;  
}  
  
struct memory_handle {  
    int handle;  
}
```

## 20.在 delete/free 后清零指针，避免重用

```
delete ptr;  
ptr = NULL;  
  
free(b_ptr);  
b_ptr = NULL;
```

## 21. 总是检查自复制

```
class a_class {  
public:  
    a_class& operator = (a_class& other) {  
        if (&other == this) return (other);  
    }  
}
```

## 22. 使用 snprintf 建立字符串

```
snprintf(buffer, sizeof(buffer), "file.%d", sequence);
```



## Appendix F: Creative Commons License

### License

THE WORK (AS DEFINED BELOW) IS PROVIDED UNDER THE TERMS OF THIS CREATIVE COMMONS PUBLIC LICENSE ("CCPL" OR "LICENSE"). THE WORK IS PROTECTED BY COPYRIGHT AND/OR OTHER APPLICABLE LAW. ANY USE OF THE WORK OTHER THAN AS AUTHORIZED UNDER THIS LICENSE OR COPYRIGHT LAW IS PROHIBITED.

BY EXERCISING ANY RIGHTS TO THE WORK PROVIDED HERE, YOU ACCEPT AND AGREE TO BE BOUND BY THE TERMS OF THIS LICENSE. TO THE EXTENT THIS LICENSE MAY BE CONSIDERED TO BE A CONTRACT, THE LICENSOR GRANTS YOU THE RIGHTS CONTAINED HERE IN CONSIDERATION OF YOUR ACCEPTANCE OF SUCH TERMS AND CONDITIONS.

#### 1. Definitions

a. "**Collective Work**" means a work, such as a periodical issue, anthology or encyclopedia, in which the Work in its entirety in unmodified form, along with one or more other contributions, constituting separate and independent works in themselves, are assembled into a collective whole. A work that constitutes a Collective Work will not be considered a Derivative Work (as defined below) for the purposes of this License.

b. "**Derivative Work**" means a work based upon the Work or upon the Work and other pre-existing works, such as a translation, musical arrangement, dramatization, fictionalization, motion picture version, sound recording, art reproduction, abridgment, condensation, or any other form in which the Work may be recast, transformed, or adapted, except that a work that constitutes a Collective Work will not be considered a Derivative Work for the purpose of this License. For the avoidance of doubt, where the Work is a musical composition or sound recording, the synchronization of the Work in timed-relation with a

moving image ("synching") will be considered a Derivative Work for the purpose of this License.

c. "**Licensors**" means the individual, individuals, entity or entities that offers the Work under the terms of this License.

d. "**Original Author**" means the individual, individuals, entity or entities who created the Work.

e. "**Work**" means the copyrightable work of authorship offered under the terms of this License.

f. "**You**" means an individual or entity exercising rights under this License who has not previously violated the terms of this License with respect to the Work, or who has received express permission from the Licensor to exercise rights under this License despite a previous violation.

**2. Fair Use Rights.** Nothing in this license is intended to reduce, limit, or restrict any rights arising from fair use, first sale or other limitations on the exclusive rights of the copyright owner under copyright law or other applicable laws.

**3. License Grant.** Subject to the terms and conditions of this License, Licensor hereby grants You a worldwide, royalty-free, non-exclusive, perpetual (for the duration of the applicable copyright) license to exercise the rights in the Work as stated below:

a. to reproduce the Work, to incorporate the Work into one or more Collective Works, and to reproduce the Work as incorporated in the Collective Works;

b. to create and reproduce Derivative Works provided that any such Derivative Work, including any translation in any medium, takes reasonable steps to clearly label, demarcate or otherwise identify that changes were made to the original Work. For example, a translation could be marked "The original work was translated from English to Spanish," or a modification could indicate "The original work has been modified.";;

c.to distribute copies or phonorecords of, display publicly, perform publicly, and perform publicly by means of a digital audio transmission the Work including as incorporated in Collective Works;

d.to distribute copies or phonorecords of, display publicly, perform publicly, and perform publicly by means of a digital audio transmission Derivative Works.

e.For the avoidance of doubt, where the Work is a musical composition:

**i.Performance Royalties Under Blanket Licenses.** Licensor waives the exclusive right to collect, whether individually or, in the event that Licensor is a member of a performance rights society (e.g. ASCAP, BMI, SESAC), via that society, royalties for the public performance or public digital performance (e.g. webcast) of the Work.

**ii.Mechanical Rights and Statutory Royalties.** Licensor waives the exclusive right to collect, whether individually or via a music rights agency or designated agent (e.g. Harry Fox Agency), royalties for any phonorecord You create from the Work ("cover version") and distribute, subject to the compulsory license created by 17 USC Section 115 of the US Copyright Act (or the equivalent in other jurisdictions).

**f.Webcasting Rights and Statutory Royalties.** For the avoidance of doubt, where the Work is a sound recording, Licensor waives the exclusive right to collect, whether individually or via a performance-rights society (e.g. SoundExchange), royalties for the public digital performance (e.g. webcast) of the Work, subject to the compulsory license created by 17 USC Section 114 of the US Copyright Act (or the equivalent in other jurisdictions).

The above rights may be exercised in all media and formats whether now known or hereafter devised. The above rights include the right to make such modifications as are technically necessary to exercise the rights in other media and formats. All rights not expressly granted by Licensor are hereby reserved.

**4. Restrictions.** The license granted in Section 3 above is expressly made subject to and limited by the following restrictions:

a.You may distribute, publicly display, publicly perform, or publicly digitally perform the Work only under the terms of this License, and You must include a copy of, or the Uniform Resource Identifier for, this License with every copy or phonorecord of the Work You distribute, publicly display, publicly perform, or publicly digitally perform. You may not offer or impose any terms on the Work that restrict the terms of this License or the ability of a recipient of the Work to exercise the rights granted to that recipient under the terms of the License. You may not sublicense the Work. You must keep intact all notices that refer to this License and to the disclaimer of warranties. When You distribute, publicly display, publicly perform, or publicly digitally perform the Work, You may not impose any technological measures on the Work that restrict the ability of a recipient of the Work from You to exercise the rights granted to that recipient under the terms of the License. This Section 4(a) applies to the Work as incorporated in a Collective Work, but this does not require the Collective Work apart from the Work itself to be made subject to the terms of this License. If You create a Collective Work, upon notice from any Licensor You must, to the extent practicable, remove from the Collective Work any credit as required by Section 4(b), as requested. If You create a Derivative Work, upon notice from any Licensor You must, to the extent practicable, remove from the Derivative Work any credit as required by Section 4(b), as requested.

b.If You distribute, publicly display, publicly perform, or publicly digitally perform the Work (as defined in Section 1 above) or any Derivative Works (as defined in Section 1 above) or Collective Works (as defined in Section 1 above), You must, unless a request has been made pursuant to Section 4(a), keep intact all copyright notices for the Work and provide, reasonable to the medium or means You are utilizing: (i) the name of the Original Author (or pseudonym, if applicable) if supplied, and/or (ii) if the Original Author and/or Licensor designate another party or parties (e.g. a sponsor institute, publishing entity,

journal) for attribution ("Attribution Parties") in Licensor's copyright notice, terms of service or by other reasonable means, the name of such party or parties; the title of the Work if supplied; to the extent reasonably practicable, the Uniform Resource Identifier, if any, that Licensor specifies to be associated with the Work, unless such URI does not refer to the copyright notice or licensing information for the Work; and, consistent with Section 3(b) in the case of a Derivative Work, a credit identifying the use of the Work in the Derivative Work (e.g., "French translation of the Work by Original Author," or "Screenplay based on original Work by Original Author"). The credit required by this Section 4(b) may be implemented in any reasonable manner; provided, however, that in the case of a Derivative Work or Collective Work, at a minimum such credit will appear, if a credit for all contributing authors of the Derivative Work or Collective Work appears, then as part of these credits and in a manner at least as prominent as the credits for the other contributing authors. For the avoidance of doubt, You may only use the credit required by this Section for the purpose of attribution in the manner set out above and, by exercising Your rights under this License, You may not implicitly or explicitly assert or imply any connection with, sponsorship or endorsement by the Original Author, Licensor and/or Attribution Parties, as appropriate, of You or Your use of the Work, without the separate, express prior written permission of the Original Author, Licensor and/or Attribution Parties.

## **5. Representations, Warranties and Disclaimer**

UNLESS OTHERWISE MUTUALLY AGREED TO BY THE PARTIES IN WRITING, LICENSOR OFFERS THE WORK AS-IS AND ONLY TO THE EXTENT OF ANY RIGHTS HELD IN THE LICENSED WORK BY THE LICENSOR. THE LICENSOR MAKES NO REPRESENTATIONS OR WARRANTIES OF ANY KIND CONCERNING THE WORK, EXPRESS, IMPLIED, STATUTORY OR OTHERWISE, INCLUDING, WITHOUT LIMITATION, WARRANTIES OF TITLE, MARKETABILITY, MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, NONINFRINGEMENT, OR THE ABSENCE OF LATENT OR OTHER

DEFECTS, ACCURACY, OR THE PRESENCE OF ABSENCE OF ERRORS, WHETHER OR NOT DISCOVERABLE. SOME JURISDICTIONS DO NOT ALLOW THE EXCLUSION OF IMPLIED WARRANTIES, SO SUCH EXCLUSION MAY NOT APPLY TO YOU.

**6. Limitation on Liability.** EXCEPT TO THE EXTENT REQUIRED BY APPLICABLE LAW, IN NO EVENT WILL LICENSOR BE LIABLE TO YOU ON ANY LEGAL THEORY FOR ANY SPECIAL, INCIDENTAL, CONSEQUENTIAL, PUNITIVE OR EXEMPLARY DAMAGES ARISING OUT OF THIS LICENSE OR THE USE OF THE WORK, EVEN IF LICENSOR HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

## **7. Termination**

a.This License and the rights granted hereunder will terminate automatically upon any breach by You of the terms of this License. Individuals or entities who have received Derivative Works (as defined in Section 1 above) or Collective Works (as defined in Section 1 above) from You under this License, however, will not have their licenses terminated provided such individuals or entities remain in full compliance with those licenses. Sections 1, 2, 5, 6, 7, and 8 will survive any termination of this License.

b.Subject to the above terms and conditions, the license granted here is perpetual (for the duration of the applicable copyright in the Work). Notwithstanding the above, Licensor reserves the right to release the Work under different license terms or to stop distributing the Work at any time; provided, however that any such election will not serve to withdraw this License (or any other license that has been, or is required to be, granted under the terms of this License), and this License will continue in full force and effect unless terminated as stated above.

## **8. Miscellaneous**

a.Each time You distribute or publicly digitally perform the Work (as defined in Section 1 above) or a Collective Work (as defined in Section 1 above), the Licensor offers to the recipient a license to the Work on the same terms and conditions as the license granted to You under this License.

b.Each time You distribute or publicly digitally perform a Derivative Work, Licensor offers to the recipient a license to the original Work on the same terms and conditions as the license granted to You under this License.

c.If any provision of this License is invalid or unenforceable under applicable law, it shall not affect the validity or enforceability of the remainder of the terms of this License, and without further action by the parties to this agreement, such provision shall be reformed to the minimum extent necessary to make such provision valid and enforceable.

d.No term or provision of this License shall be deemed waived and no breach consented to unless such waiver or consent shall be in writing and signed by the party to be charged with such waiver or consent.

e.This License constitutes the entire agreement between the parties with respect to the Work licensed here. There are no understandings, agreements or representations with respect to the Work not specified here. Licensor shall not be bound by any additional provisions that may appear in any communication from You. This License may not be modified without the mutual written agreement of the Licensor and You.

### **Creative Commons Notice**

Creative Commons is not a party to this License, and makes no warranty whatsoever in connection with the Work. Creative Commons will not be liable to You or any party on any legal theory for any damages whatsoever, including without limitation any general, special, incidental or consequential damages arising in connection to this license. Notwithstanding the foregoing two (2) sentences, if Creative Commons has expressly identified itself as the Licensor hereunder, it shall have all rights and obligations of Licensor.

Except for the limited purpose of indicating to the public that the Work is licensed under the CCPL, Creative Commons does not authorize the use by either party of the

trademark "Creative Commons" or any related trademark or logo of Creative Commons without the prior written consent of Creative Commons. Any permitted use will be in compliance with Creative Commons' then-current trademark usage guidelines, as may be published on its website or otherwise made available upon request from time to time. For the avoidance of doubt, this trademark restriction does not form part of the License.

Creative Commons may be contacted at <http://creativecommons.org/>.