程序代码实现的陷阱

Helianthus

2006/01/05 09:43

如果把编写代码看做是在河水的上游倾倒垃圾，而将测试代码找出 BUG 看做是在河水的下游来治理污水的话，那么毫无疑问治理污染河流的首要环节应该是在上游尽最大可能的控制垃圾的倾倒数量，而不是在下游热火朝天的治理污水却全然不顾有人在上游肆意的倾倒垃圾。

前言

通常来说，好的软件一定有一个好的结构设计。但是反之结论却是不能成立的，因为一个好的软件结构设计并不一定能带来好的软件产品，这中间依赖很多其他的因素，例如代码性能，代码的健壮性以及代码的可维护性。正如 Frederic Brooks 博士在他那本著名的软件工程著作《神秘的人月》中所说的那样，如果将一段普通程序转变成产品中的代码的话，那是需要付出多达 9 倍的时间的。这中间的原因有很多方面，因为普通的一段程序通常仅仅是为了实现某一个功能或是达到对某种预测的验证，只要强调这段程序结果正确了就可以结束了；但是对于产品代码来说，仅仅实现了功能性的要求是远远不够，因为一个软件产品中往往存在大量的功能性模块，如果在某个功能模块执行之后造成了足以影响其他模块正常工作的负效应，那么我们把导致其他模块不能正常工作的这样的功能模块称之为脏模块，因为它在完成了自己的功能后污染了整体的软件运行环境，做了一些不该做的事情。可以说，软件工程师通过各种调试手段将软件中各种形形色色的 Bugs 找出来然后发 Patches 的过程，其实就是在根据需求逐渐的将一个不稳定的产品中各种 Dirty 的模块转变成 Clean 模块的过程。

然而，随着软件质量控制意识的加强以及软件工程的发展，我们逐渐意识到在软件开发的早期将各种 Bug 解决掉所需的时间成本是最低的，因为在产品后期再来解决 Bug 的话，随着软件模块数量增多，环境复杂度的增加，需要将问题定位到某一个具体功能模块所需的时间开销基本上是很大的（当然和调试人员的基本素质是有关的）。这可能就是很多软件质量控制的流程中都比较强调要在软件模块设计的过程中引入 Review 的过程，在代码编码实现后首先也必须由别人负责 Review 后再进入模块的单元测试，在单元测试结束后再进入产品的集成测试。这样的好处无疑是显然，在模块集成入产品前首先保证该模块是一个 Clean 的模块，这样在产品集成过程中就可以放心的将对问题的各种怀疑集中到各个相关的模块接口之上，而不是漫无目的的对所有模块内部都怀疑，这样做的一个严重的后果就是随着模块内部代码的分支增多，我们将无所适从，因为程序分支的指数增加将迫使我们打消这样的念头。

这篇文章的目的不是和大家探讨软件工程中对软件质量的控制的，本文的写作目的主要是试图从如何提高代码实现质量的角度上来探讨一下各种各样的代码陷阱问题，因为我们在开篇就说过，一个好的思想最终是需要一个好的代码质量来实现的，在代码的编码阶段就能够提前避免各种代码陷阱，无疑会加速模块的开发进度，而且也会大大降低产品的后续维护工作量的。当然，除了软件代码的实现陷阱外，我们还会讨论一些具体的软件调试技术以及工程中需要注意的各种细节问题，希望本文能够给那些刚刚进入或是期望进入软件中的兄弟姐妹们有个提醒。本文中所有的陷阱的例子都是从实际的项目中得来的，为了更好的说明问题，在某些例子中可能会根据实际的需要引入部分汇编程序，其目的是为了将问题说明得更加清楚。记得有一位计算机专家这么说过“计算机程序语言只是一种用来表达人们思想的工

具而已”，也就是说我们没有必要在各种语言的好坏上争论不休，当然也不能认为只有自己常用的语言就是最好的，我们需要在熟练掌握一两种语言的基础上再了解其他的语言即可。编程语言是发展很快的，我们试图紧跟每一种语言是不切实际，也没有太大的必要。常规的做法只需要把底层的汇编语言（了解计算机 CPU 的体系结构）和自己喜欢的一种高级语言就可以了，至于为自己喜欢的语言去和别人打口水仗就没有必要了。其实各种高级语言之间本身并没有那么大的差异性。就像在使用汇编语言写程序的时候，你所获得的速度优势是以你必须自己协调各个寄存器的使用，等到了 C 语言这个级别，分配寄存器，各个局部变量在堆栈中具体的地址在哪里，如果操作它们，使用哪一条指令来等效实现 C 语言的抽象语法所需要的指令等等之类非常繁杂的事情现在基本上都是由 C 语言的编译器全部解决了掉，这样做的好处就在于能够把人的精力从前面那些繁杂的，规律性很强的事务中解放出来，集中精力去思考高层的设计问题。再上升到 C＋＋语言的时候，又引入了更多新的机制，其目的和前面一样，是希望尽量把一些重复性很强的东西固定化，由编译器负责替你默默的把这些事情全部做完，这样的话就充分利用了程序的优势。在这一点上，就好像有一个内嵌在编译器中的 CheckList 一样，每次都会忠实的按照预定的执行次序顺序执行一些程序化的操作，这样就避免了人手工做这种事情可能会出现漏洞的可能性。比如说 C 语言中你可能要写很多的 Initialize 和 Finalize 的函数，这下好了，C++里面不再需要你象写 C 语言那样在声明和撤销某一个数据单元的时候再显式的调用 Initialize 函数和 Finalize 函数，这些烦琐的事情不再需要你去关心了。当然 C++的优势并不仅仅局限在这一点上了，但是从很多语言的发展路径上我们还是能够很清晰的看到这一点。就是因为 C 语言（C++中依然存在）中对内存使用常常会出现泄漏的问题（人们常常会在分配使用完内存之后常常忘记释放内存），JAVA 语言中就实现了一个所谓比较智能化的内存回收机制。但是当我们将各种语言美丽的标签统统去掉之后，我们就会发现其实所有的程序语言在本质上是一样的。语言也是发展的，每一种新的语言的产生都在试图解决老语言中存在的问题，但是随着时间的流逝，新的语言也会慢慢产生出很多其他新的问题，而这些新的问题又会催生出更新一代语言的产生。这是一个递归的过程，也从一个侧面反映出人们试图追求完美境界的美好愿望。但是不管语言怎么变化，对于那些有志于在程序设计方面有所追求的朋友们来说，了解底下处理器的硬件体系结构，清楚 CPU 是怎么运行的，掌握汇编语言是非常重要的，就像那句话说的那样“当我开始学会用心去看这个世界的时候，突然间我发现整个世界是如此的清楚”。在本文各个部分，我们会分别展示很多陷阱，应该说很多陷阱其实和语言是不相关的，也就是说这种陷阱的例子并不仅仅局限于我们在例子中展示的这种语言里面。其他的语言中也会存在同样的问题，可能不同的是在这些语言中具体的表现形式可能会有所不同而已，但是我们只需要仅仅把握住问题的实质和解决问题的方法就可以了。

下面我们就开始从产品中代码的各个阶段（代码的编写，代码的编译，代码的调试和代码的集成）来和大家交流一下我们可能会遇到的各种陷阱，同时也会把部分避免这些陷阱的技术和大家一起分享。如果把完成一个优秀的软件作品看做是必须通过一片到处是危险，到处是陷阱的原始森林的话，那么我们就能理解为什么仅仅准备好必要的技术装备是远远不够的，成功走出这些陷阱更需要的是丛林猎手的经验和生存技能。好的，现在让我们开始这段充满挑战的旅程，希望每一位朋友在经过这些必要的训练之后都能称为一名好的丛林猎手（Jungle Hunter）。

关于类型的声明

```
typedef    unsigned char     t_byte;
typedef    signed char       t_char;
typedef    unsigned long     t_dword;
```

第一章 编写代码前的准备知识

有些朋友也许会有疑惑，编写代码难道还需要先了解什么准备知识吗?任何一个看懂了编程语言语法书的都会写代码啊，有必要多此一举的搞什么准备知识的培训吗？对于有这样疑问的朋友们，我还是决定先问下面的几个问题，

1. 你写代码的时候是习惯使用 C 语言提供的基本数据类型如 long, char 呢还是倾向于使用自己定义出来的类型如 t_size 之类的?
2. 你写代码的时候常常用到会用 int 类型来声明的变量吗?
3. C 语言中的前缀关键字 register 用于声明将指定的变量尽量放在寄存器中以便加快运行速度，但是这样做对于其他的处理器也是一样的吗？比如 ARM 或是 MPC860 芯片？
4. C 语言里面有指针的数据类型,你可能已经知道指针类型的变量占用的空间是 4 个字节,但是你知道为什么是 4 个字节呢？我声明一个 byte * 的指针和一个 byte ** 的指针，它们之间究竟存在什么差异？ 这些差异性是在什么阶段产生的?
5. C 语言存在几个比较特殊的前缀描述符号,比如 const, static, volatile, interrupt, register,inline 等等,可是你知道它们是什么含意和具体使用在什么场合吗?
6. 你可能已经非常熟悉使用 include 语句了,也知道它属于预编译指令，除此之外，你还知道其他的预编译指令如 error, warning, pack, define 吗?你会使用某些特殊的预编译指令提供出来的非常强大的功能吗?
7. C 语言的语法书上告诉我们，在一个函数内部用 static 声明出来的变量不会随着函数执行的结束而消失，但是这种变量也仅仅能够供该函数内部使用，在该函数外部是不能使用的，但是情况真的是这样的吗?
8. 你会在程序中使用__FILE__, __LINE__吗?你知道这个变量是从哪里来的吗?你还知道其实还有几个其他的类似的变量吗?
9. 我们知道在调试程序的时候，了解问题发生的时候函数的调用顺序是非常关键的，但是你知道如何手工的恢复出问题发生时刻函数的调用栈顺序以及部分函数的执行的调用参数吗?
10. 我们可以使用 int, long 等基本数据类型来声明我们想要的变量，但是为什么我们不能用 void 来声明一个变量呢?然而我们用 void * 来声明一个变量却是可以的?
11. 为什么我们在使用一些商业工具如 Insure 或是 BoundsCheck 之类的代码检查工具来扫描代码的时候会出现精度损失(precision loss)的警告？这样的警告究竟是什么意思，为了消除这样的警告，我们直接加一个强制类型转换就可以了吗?
12. 你也许听说过 big_endian 和 little-endian，但是你知道它们的具体含意吗?什么样的情况下我们需要考虑到这样的情况呢?
13.

如果你觉得上面的这些问题你大部分能够非常清楚的给出答复的话，那么我必须恭喜你，说明你不仅仅对 C 语言的语法规则非常清楚，而且对底下具体的处理器的结构以及汇编语言也非常清楚，那么你可以直接越过这一章节进入下一章，因为这一章的知识主要是为了后续章节做基础的。但是如果有朋友对以上问题不是很清楚的话，那么我们觉

得还是有必要安下心来把这一章节读一读，也许你在读完之后会有所收获，当然这种收获并不仅仅局限于 C 语言,实际上有些内容和具体的编程语言是无关的。好了，下面让我们开始进入吧。

1.1 了解我们代码运行的平台－处理器

我们的代码被编译器编译成机器代码后最终将被某种具体的 CPU 芯片解析执行。为了更好地理解后续章节部分的内容，我们还是有必要了解一些基本的 CPU 知识的。

（此处可以从计算机结构体系中拷贝）

下面我们将着重介绍几个常用的 CPU 芯片。

1.1.1 x86(以 8086 为例)

可以说，x86 芯片是我们接触得最多的芯片类型。相对于其他的 CPU 芯片来说，x86 芯片属于硬件比较简单的一种。常规的几个算术寄存器 AX,BX,CX,DX,SI,DI,两个堆栈寄存器 SP,BP, 再加上几个段寄存器如 CS,DS,SS,ES,及状态寄存器 FLAGS 和指令地址寄存器 IP。和其他的芯片相比，x86 芯片因为寄存器比较少，所以需要频繁地执行内存到寄存器间的数据搬移的动作。但是也正是因为 x86 寄存器比较少，反过来也使得 x86 芯片完成函数调用等方面，处理起来非常简洁。A 函数调用 B 函数时，A 函数只需要将函数的入口参数压入堆栈，然后调用 CALL B 指令就可以了。CALL 指令在执行时会自动地将 A 函数中调用 B 函数的指令地址压入堆栈，这样当 B 函数返回的时候就可以从堆栈中直接获取返回地址（具体堆栈内容结构请参考后续部分）。请参考下图中指令序列和堆栈的对应关系的说明。

x86 芯片还有一个比较重要的特征，就是可以任意地一次性的往某个指定地址读写多个字节而不需要顾及该地址是否和读写字节数目边界对齐（当然可能是内部的微指令通过多次边界对齐的字节寻址出来的）其实是内部的。怎么理解呢？例如对于 0x0003 这样一个地址，x86 可以写入一个字节的内容 0x55（0x03=0x55），可以写入 2 个字节的内容 0x55AA(0x03=0xAA, 0x04=0x55) ， 也 可 以 写 入 4 个 字 节 的 内 容 0x11223344 （ 0x3=0x44, 0x4=0x33,0x5=0x22,0x6=0x11）。这样的操作对于某些处理器来说就不一定可以获得支持，因为有些芯片强制要求 2 个字节内容的读写动作其读写地址必须满足是 2 个字节对齐的(也就是说只能是 0x00, 0x02,0x04,0x06 之类的偶地址)，强制要求 4 个字节内容的读写动作其读写地址必须是满足 4 个字节对齐的(也就是说只能是 0x00, 0x04, 0x08 之类的地址)，否则的话将会出现总线存取的错误，ARM 就是这样的芯片。

1.1.2 MOTOROLA MPC860

1.1.3 ARM(ARM7)

和 x86 芯片比较起来，ARM 处理器的第一个印象就是运算寄存器数目众多。ARM7 可以提供 R0 到 R15 一共 16 个常规寄存器(其中 R13=SP，R14=LR，R15=IP)以及一个状态寄存器 CPSR。由于有众多的寄存器可用，所以 ARM 处理器可以在代码执行过程中尽最大可能地减少数据在内存和寄存器之间的交互。此外，由于寄存器数目的优势，这也直接导致 ARM 上函数调用处理和 x86 有所不同。根据 ARM 编译器的预定，函数的前 4 个参数分别放入 R0 至 R3 这 4 个寄存器，第 4 个以后的参数将通过堆栈进行传递。由于大多数函数的参数都不足 4 个，这就使得 ARM 处理器可以不必象 x86 那样将参数压入堆栈，在函数执行过程中从堆栈中获取参数执行，然后在函数执行后退栈，而是可以直接在调用的时候将参数推入寄存器，然后在函数执行时直接从寄存器中获取参数，由于省略了多次存取堆栈的操作（也就是存取内存），所以同样的程序在 ARM 上将可

以获得比 x86 更高的执行效率。ARM 和 x86 另一个不同点是，ARM 引入一个寄存器 LR 来专门保存函数的调用地址。比如对于函数 A 调用 B，然后 B 函数再调用 C 函数的这样一个函数调用序列，请参考下图中指令序列和堆栈的对应关系的说明。

需要注意的是 ARM 对内存地址的存取操作是要求严格的字节边界对齐的。

1.2 关于程序设计语言

1.3 编译器的知识

1.4 处理器的预备知识

1.5 函数调用引出栈的知识，寄存器分配

在了解完必要的处理器的基本知识之后，让我们再来看看程序设计语言中的一些重要概念在具体处理器上的表现形式。

1.5.1 变量

变量是程序语言中的一个非常重要的部分。可以想像一下，如果 CPU 仅仅执行机器指令却不会引发任何数据内容的变化，那么这样的 CPU 操作对我们来说是没有意义的。我们看到的任何程序，从 CPU 的角度看过去，无非就是执行机器指令，然后完成对某些特定内存地址数据单元内容的更改。当然，这样的解释是从 CPU 的角度看过去的，换成程序语言思考的角度，我们怎么在高级语言的层次上来实现对 CPU 某个特定内存地址的数据单元操作呢？这就是通过变量来实现的。从抽象的逻辑角度来理解的话，我们可以这样认为，C 语言中的变量其实本质上都指向了某一个内存地址上，我们可以通过操作变量来操作这个内存地址中的内容。我们读写这个内存地址中的内容就被转换成我们对这个变量的读写（通过编译器完成了中间的抽象变化过程），因为我们可以往某个指定内存地址上写 1 个/2 个/或者 4 个字节，因此对应的变量的基本类型也分成了字节(byte=8bit), 双字节(word=16bit), 字(dword=32bit), 如果变量没有具体的数据类型的话，那么处理器是无法完成操作的，因为处理器根本就不知道如何

1.5.2 函数，堆栈及函数调用分析

函数在程序设计语言中占有相当重要的角色，可以说正是函数将人们从烦琐的具体程序实现细节中解脱出来，将复杂的软件分割成多个子模块，然后在各自独立完成这些子模块的编码调试之后再将各个之模块象堆积木的方式集成起来。想像一下，如果 C 语言中没有系统提供的那么多库函数的话，那就意味着你自己必须一切从零开始，自己写类似于 memcpy，fopen 之类的库函数实现，在当前软件复杂度日益增加的今天，一切期望全部靠自己从头开始的软件开发模式已经很少见了。所以我们必须依靠库函数。

函数是一段程序代码的集合，依赖外部的参数输入来实现某种特殊的功能。比如 memset 函数就是专门用来将某个指定地址开始的内存空间全部填充成某个固定字节的库函数。由于函数

1.5.3 堆栈

1.5.4 指针

第二章　　代码的编写

有人也许会问，为什么需要专门来说说代码的编写呢？无论是谁，只要懂得具体语言的语法规则不就可以了吗？ 难道写了这么多程序了还需要再重新回过头来重新接受如何编码的培训？其实代码的编写在开发软件产品的过程中相当重要，写得好的代码不仅能够有效的预防各种潜在的错误，同时也能够能够有助于开发者在代码的维护阶段更快的最终各种错误。就像我们前面就谈到的，Bug 应该是在源头阶段就尽量被避免。这也是本文写作的目的所在，如果能够在读完本文后，在编写代码的时候就能自动的想起本文中的种种陷阱，那么你就能快速的通过这片陷阱地。

1. 培养编写 Defense 代码的习惯来主动的抵御不可知的代码错误

   什么叫 Defense 代码呢？我们习惯把那些能够在发生错误的时候主动告警或是被动告警的那些代码称之为 Defense 代码。能够在错误的情况下主动的发出提示告警信息是很重要的，因为如果在错误的时候没有提示信息的话，那么这样的代码一旦进入集成阶段就

会给我们带来很多难以想像的问题，等到那时再试图回头来解决编码阶段的问题已经为时晚矣。

1.1 多用 ASSERT（断言）

1.1.1 利用断言来完成函数的必要的参数检查

很多写过产品代码的朋友们都或多或少的读过 CODING Rule(编码的规范)，里面几乎无一例外的都提到一点就是使用断言。断言其实就是一种非常典型的 Defense 代码，强调了软件代码的能够正确执行的先决条件，如果条件不满足，就会抛出告警信息。

```
1    #include “stdarg.h”
2    void sys_trace(const t_char * fmt, ...)
3    {
4    #define MAX_MEM_TRACE_INFOLEN 256
5        va_list     ap;
6        t_unit      len;
7        t_char      pszConsoleBuffer[MAX_MEM_TRACE_INFOLEN + 1];
8
9        ASSERT(NULL != fmt);
10
11       va_start(ap, fmt);
12       len = vsprintf(pszConsoleBuffer, fmt, ap);
13       va_end(ap);
14
15       if (len >= MAX_MEM_TRACE_INFOLEN)/*Limit the Range*/
16       {    /* output information already crash the stack… */
17           len = MAX_MEM_TRACE_INFOLEN;
18           ASSERT(FALSE);
19       }
20       pszConsoleBuffer[len] = ‘\0’;
21       printf("%s", pszConsoleBuffer);
22   }
```

23 C1-1 一个使用 ASSERT 的例子

正如我们在例子 C1-1 中看到的，函数 sys_trace 的目的是用于输出指定长度的字符串信息，由于使用了和 printf 类似的可变参数，所以我们在调用这个函数的时候可以象调用 printf 那样来输出我们需要的信息，这种采用统一信息输出通道的方式可以避免我们在调试的阶段随意的嵌入 printf 的语句，从而破坏了功能代码的美观。

现在我们来注意一下程序中的第 9 行和第 18 行处的两个 ASSERT 语句。其中第 9 行处的 ASSERT 属于参数检查性质的断言，用于声明 sys_trace 的调用者绝对不会使用空的格式声明（类似于 printf 常用的“This code has %d lines.\r\n”之类的信息）。事实上从应用的角度上看，sys_trace 的参数 fmt 为 NULL 的可能性是很低的，因此我们才在第 9 行放置了这样的一个 ASSERT 语句，以便能够在某一个粗心的朋友在写代码的时候真的那样做的时候提出告警信息。但是第 18 行处的 ASSERT 却有所不同，因为库函数 vsprintf 自身的安全缺陷问题（关于这一点，我们会在后续的部分给出分析），可能会导致缓冲区溢出，所以我们在第 22 行放置了这样的一个断言，以便在信息输出真的出现缓冲区溢出的时候及时的告警。试

想一下，函数 sys_trace 在用户需要输出的信息字节数目小于 256 的时候都是能够正常工作的，但是某一天一个新来的朋友在不知情的情况突然使用这个函数输出 300 多个字节的信息的时候，就会触发 18 行处的断言，给出一个断言警告（其实这个时候 sys_trace 的函数调用栈可能已经被破坏了）来提示我们有一个异常的调用出现。

需要着重说明一点的是，ASSERT 功能基本上主要用于调试阶段，正式的产品代码中往往被关闭了，这一点可以从 ASSERT 自己的代码实现中看得非常清楚。原因很简单，一旦所有的错误在调试阶段被清除掉后，在产品中保留 ASSERT 的代码检查不仅没有意义，而且会因为 ASSERT 的语句执行占用过多的 CPU 从而导致程序运行很慢。

```
1    #ifdef _DEBUG
2    #define ASSERT(x) {\
3        if (!(x))\
4        {\
5            printf("\r\nASSERT Fail at %d %s with %s.\r\n", __LINE__, __FILE__, #x);\
6            while(1);\
7        }\
8    }
9    #else
10       #define   ASSERT(x)   0
11   #endif/* #ifdef _DEBUG */
```

12 C1-2 ASSERT 的实现

1.1.2 利用断言来实现对不可预知错误的检查

为了加深朋友们对 Defense 代码的理解，作者准备再举一个例子来说明 ASSERT 在关键时刻所能发挥的特殊作用。

在开发某个视频产品过程中，由于各个软件模块都是由相关的开发人员来负责测试开发维护的。在 V0.5 版本集成联调成功后不久，突然有一天发现在使用了很多新的 patch 代码的 V0.53 版本上出现了很严重的问题，机器在很远端设备互相连通后不久就死机了。后来对板子做过分析后发现了一个非常震惊的情况，内存中的很多的程序代码全部被改写成乱七八糟的指令了！在随后对代码的隔离分析后马上就发现了原因：这个问题是因为一个软件模块 A 在调用另外一个模块 B 提供的解码函数造成的。

解码函数在 V0.5 的时候是这样的，

```
t_long   stream_decode(void *dest_buf, long dest_bufsize, void * src_buf, long src_bufsize);
```

函数的目的也非常简单，就是对用户提供的原始数据缓冲区进行解码并将解码之后的数据根据目的缓冲区实际大小的限制对应的写入到目的缓冲区中。函数的返回值用来告知调用者实际写入到目的缓冲区中的数据字节信息的大小。这样的一个函数实际上可以称为一个标准的模范函数声明。

然后让我们再来看一下模块 A 中对这个函数的调用代码是怎么样的？如 C1-3 所示

```
1    t_long   decoded_len;
2
3    decoded_len ＝ stream_decode（dest_buf, MAX_SIZE, src_buf, MAX_SIZE）;
4    memcpy(output_buf[recv_cnt], dest_buy, decoded_len);
5    recv_cnt += decoded_len;
```

6 C1-3 一段没有 ASSERT 保护但是功能正确的代码

在 C1-3 中我们可以看到，其实模块 A 的代码也是满足编码规范的，可以说也没有太多的瑕疵可找。那么为什么软件从 V0.5 升级到 V0.53 就出现了问题呢？

问题的根本原因就出现在 B 模块的解码函数上。原来 B 模块的开发者也许是出于好意，在 stream_decode 函数的内部实现上私自增加了一个新的小功能－－在原始码流出错或是解码函数内部状态有错的情况下会利用 long 数据类型可以携带负数的特征，在解码出错的情况下会返回一个负数值用来告知调用者出现了何种错误！也就是说如果数据正常的情况下，decoded_len 得到的是一个正数，而如果原始码流出错的情况下 decoded_len 将是一个负数！很不幸的是，decoded_len 如果是负数的话，由于 memcpy（dest,src, count）函数的原型中将 count 解释成无符号类型的，这就是说如果 decoded_len 是－1 的话，会被后面的 memcpy 函数解释成需要拷贝 4294967295 个字节到目的地址，由于地址在 0xFFFFFFFF 处会回归 0,从而导致整个内存空间被刷掉！

原因找到之后，我们也许会埋怨 B 模块 decoder 函数的开发者违反流程，私自修改了接口意义（尽管是出于好的目的），从而导致了这个严重问题。但是如果我们把问题反过来想的话，我们为什么不能在要求 A 模块的开发者在 C1-3 中的第 3 行和第 4 行之间主动插入下一条断言来主动的抵御因为各种不可知的原因导致的异常呢?请参考 C 1-4 的实现

```
1    t_long   decoded_len;
2
3    decoded_len ＝ stream_decode（dest_buf, MAX_SIZE, src_buf, MAX_SIZE）;
4    ASSERT((0 <= decoded_len) && (MAX_SIZE >= decoded_len));
5    memcpy(output_buf[recv_cnt], dest_buy, decoded_len);
6    recv_cnt += decoded_len;
7    C1-4 一个能主动预告错误的例子
```

1.1.3 利用断言来实现对环境假设的检查

我们知道，相同的代码如果运行在不同的平台上出现了问题，往往说明代码中包含有和特定平台紧紧关联的，不可移植性的东西。这是因为我们在一开始写代码的时候往往脑海里是有一个基本的平台环境的，如具体使用的编译器或是具体运行的 CPU 芯片类型。所以这就要求我们尽量的在写代码的时候将自己对环境的某些约束条件的要求显式的表现出来，这可以通过断言来实现,当代码移植到新的平台上出现环境条件不满足的时候，ASSERT 就可以在第一时间里面及时告知我们这样的变化的的确确发生了。

```
1    typedef enum
2    {
3       FILEX_FLG_NORMAL     = 0x00,
4       FILEX_FLG_DIRECTORY = 0x01,
5       FILEX_FLG_LINK           = 0x02,
6       FILEX_FLG_BUTT
7    }E_FILE_ATTRIB;
8    void demo_func(byte * buf, dword buf_size)
9    {
10      E_FILE_ATTRIB file_attrib;
11      ……
12      ASSERT(sizeof(dword) == sizeof(E_FILE_ATTRIB));//检查枚举是不是 4 个字节?
13      memcpy(buf, &file_attrib, 4);
14      ……
```

```
15    }
16    C1-5 一个对编译环境变量做断言检查的例子
```

如上面的例子，我们假定在函数 demo_func 内部枚举(enum)类型的 file_attrib 变量总是要用于填充输入缓冲区 buf 的前 4 个字节的。但是我们知道不同的编译器在处理枚举类型的时候是不尽相同的，有些编译器将枚举固定解释成 4 个字节,有些则根据枚举类型定义的范围大小给出具体的字节数目,比如 E_FILE_ATTRIB 范围在 255 之内，因此只用一个字节就可以了，这样就出现了解释的不一致。而第 12 行的断言无疑会及时的将这样的不一致性检查出来并告知给我们，避免了因为没有意识到编译器对枚举类型解释的变化而无端的耗费大量的精力去定位问题的原因。

1.2 了解模块的设计需求，以代码的健壮性，稳定性，可维护性为基本出发点

为什么需要单独写一个小节来叙述这个问题呢？作者在维护一些老的代码的时候有一个很深的感触，很多老的模块代码因为开发的时间比较早，在早期的功能需求中测试通过后就用于产品中直到有一天突然有新的需求出现。在针对新需求而开发出来的新代码提交使用后，却发现在某些特殊的情况下会出现部分数据随机丢失，同时在索引表中会异常出现多个相同记录的情况。后来经过仔细的调试才发现了原因所在：问题出现在对一个库函数 memcpy 的调用上面。功能代码在每次接收到新的数据后都会在原有的数据索引表上面建立起新的数据索引表，然后在新的索引表建立之后就会用新表覆盖掉旧表，这个覆盖的动作就是由 memcpy 函数调用来完成。但是后来的代码维护者并没有注意到在某些特殊的情况下面，新的索引表和旧索引表会出现交叉（共享部分数据），这样就造成刚好引发了 memcpy 函数的一个缺陷（具体的缺陷请参考后续部分 1.3 中的说明），从而导致了数据异常丢失的问题。如果有哪位朋友对于 memcpy 的缺陷不是很清楚的话，请通过 C1-5 的例子来体会，在 C1-5 中，在 memcpy 函数执行之后，string 里面的内容将变成 HHHHHH????（此处的？表示未知的内容信息）,而不是我们所期待的 HHello。

```
1     void demo_func(void)
2     {
3           t_char string[10] = “Hello”;
4           memcpy(string ＋ 1, string, 6);//嗯，此处我们还是选用更安全的 memmove 函数？
5     }
6     C1-5
```

很显然，这个问题的确是因为代码后续的维护者可能在不知道此处存在代码缺陷的情况下造成了这个问题。但是反过来说，我们可以想像一下，如果代码的原始开发者在刚开始写这段代码的时候就注意到后续代码维护可能会出现的情况，出于稳定和慎重的考虑之后选择使用比 memcpy 更为安全的库函数 memmove，那么这个错误可以说就不会遗留下来，尽管 memove 是以牺牲了部分代码执行效率为代价，但是 memove 函数在牺牲效率的同时减少了代码在后续维护阶段中可能出错的机会，在某种意义上来讲，事实上是提高了工作的效率。要直到在产品后期维护阶段想从数据随机丢失这个现象中一步一步的追踪到这条语句的代码行可不是一件容易的事情，除了时间的开销之外，还需要调试者对模块有相当的熟悉度才有可能快速定位。比较起来，你是愿意在写代码阶段就尽量的避免这样的陷阱还是愿意在产品发布之后，在某天夜里被电话叫醒去解决紧急 Bug 呢？

注：因为目前大多数的 memcpy 函数为了出于效率的考虑，在实现上通常是尽量将数据按照 4 个字节做为一个拷贝单元对齐之后再拷贝，再加上各个系统中对 memcpy 的内部实现存在

差异，所以用户如果试图使用 C1-5 的代码来获得 string=”HHello”可能是不可靠的。反之，使用 memove 则能够保证一定在函数执行之后获得的 string 为“HHello”
LINUX 上执行 C1-5 后，string＝“HHeelo”；
WINDOWS 2000 上执行 C1-5 后，string＝“HHello”；
UNIX 上执行 C1-5 后，string＝“HHHHHH”；//典型的逐字节拷贝

1.3 提防库函数的陷阱

通过上面的几个具体的陷阱例子，可能朋友们会问，既然你说库函数有问题，那么我需要用到系统的库函数的时候怎么办？事实上本文作者的意思并不是说不让大家使用库函数，应该说绝大多数库函数都是稳定可靠的，至于我们在前面部分所提及的那些有缺陷的库函数，我们鼓励在有可替换功能库函数存在的情况下使用更为安全的库函数，而不是使用这些存在缺陷的库函数，即使我们真的无法避免使用这些存在缺陷的库函数，我们也应该尽量的在了解了它们的负效应后再谨慎使用它们。

1.3.1 避免使用容易主动引起缓冲区溢出的库函数

也许朋友们在浏览网页或是和朋友交流的过程中或多或少的都听说过缓冲区溢出这个名词，也知道一些黑客就是利用精心设计的缓冲区溢出的机会将伪装成数据的代码指令写入栈空间，替换掉原有的指令返回地址，然后获取非法的权限的故事。在这里我们不想详细的讨论黑客们是如何利用缓冲区溢出的机会获取非法权限的具体步骤（已经有很多好的文章详细的讨论了这个问题的具体实现机制），相反，我们将着重讨论我们如何在编写代码的时候就意识到某些代码或是库函数是存在缺陷和后门的，从而在编写代码阶段就将这些后门尽可能的堵上，尽量不给他们任何机会。

是否加上函数调用的栈结构分析？？

很多存在缓冲区溢出缺陷的函数或多或少都是因为他们自身没有对输出信息大小进行有效的控制。一个非常知名的函数就是 gets（void * dest_buf）；从这个函数的原型声明我们就可以看出，这个函数入口参数只有用于存储用户输入信息的缓冲区的头地址，而没有对这个缓冲区最大能够容纳多少用户输入的信息字节给出任何约束。这就是说，在调用这个函数的时候，如果某一个用户不小心输入的数据长度远远大于目的缓冲区中的最大长度，就会出现缓冲区溢出的情况，如果这个缓冲区刚好是在函数内部的局部栈上声明出来的话，那么就会出现栈溢出，部分黑客就是通过这机会获取指令的运行权的；如果这个缓冲区是声明在全局堆空间上的，那么会造成该缓冲区周围的全局变量内容被异常修改，同样弄脏了这个软件的运行环境。具体代码请参考 C1-6 的例子。

```
1    void process_user_input(void)
2    {
3        t_char    cmd_buf[256];
4
5        gets(cmd_buf);//嗯，是不是换成更安全的  fgets(cmd_buf, 256, stdin);????
6        ……..
7    }
```

8 C1-6 一个典型的可能存在缓冲区溢出的例子

那么朋友们可能会问了，既然 gets 函数是不安全的，那么我需要获取用户的输入信息的时候怎么办呢？用可替换的安全函数。在 C 语言的文件系统的库函数中存在一个类似功能的安全函数 fgets(buf, buf_maxsize, stdin);通过使用这个函数，我们能够严格的控制可能存在的非法数据输入造成对缓冲区的溢出，从而避免由于缓冲区溢出带来的各种不可知的错误

因素（比如因为其他数据被覆盖导致其他的模块运行不正常）。

类似不安全的库函数主要有：gets，sprintf, vsprintf，scanf 等等。可以分别改用 fgets，snprintf，vsnprintf 等等。

1.3.2 避免使用容易被动引发使用缺陷的库函数

我们目前使用的库函数当中有不少这样的函数，这些函数自身在调用使用上实际上可能存在缺陷的，如果用户在使用之前对这样的库函数的负效应了解不多的话，很容易就会陷入这些函数的陷阱中，而且更为可怕的是这些陷阱只有在某些特殊的情况下才会被触发，也就是说这些函数在大多数情况下还是可以工作的，这就好比我们坐在一座火山口的喷发口处，而我们却不知道什么时候火山会爆发出来…

1.3.2.1 小心内存泄漏

隐形的内存泄漏就像我们身上一个永远不会愈合的创口一样，会慢慢地把我们系统中的资源慢慢地吃完掉，最后系统因为资源耗尽而死机或是重启。所以我们在使用这些函数的时候就必须非常的小心，否则的话，不论是经验及其丰富的老程序员还是刚刚进入的新人都容易被这些陷阱缠住。

在这一类的函数中，比较知名的一个是 void * realloc(void * orig_ptr, t_size buf_size);

通常的说来用户调用 realloc 函数的目的是希望动态的改变指定的内存块的大小，将原有的内存指针做为 orig_ptr 参数，需要的新的内存大小传入，函数将会把新分配成功的内存指针返回给调用者。如 C1-7 所示，用户在第 3 行处分配了一个 100 字节的内存块供自己动态使用。但是在随后的过程中却可能发现前一次分配的内存不够用了，可能需要加大内存，因此通常会调用 realloc 函数来实现这一点。但是如果是象 C1-7 那样做的话，就在不经意间造成了一个内存泄漏。

```
1    void demo_func(void)
2    {
3        void * mem_ptr;
4        mem_ptr   =   malloc(100);
5        …….
6        mem_ptr   =   realloc(mem_ptr, 65536);
7        …….
8    }
```

9 C1-7 一个典型的容易引发潜在内存泄漏的例子

问题就出在第 6 行对 realloc 的调用上。我们假定第 3 行分配内存成功了，得到的内存指针是 0x0034F678(当然也可以是其他的地址),当执行到第 6 行的时候，事情就可能出现了变化。realloc 这个函数调用能否成功完全取决了当前系统的内存占用状况，如果当前内存比较宽裕，那么一切都正常；但是如果内存非常紧张的时候，realloc 将无法完成指定的内存分配要求，从而导致其返回值为 NULL,这就直接导致原有的内存指针 0x0034F678 被刷成了新的值 NULL!也就是说原来的那个 100 个字节左右的内存块指针 mem_ptr 由于被 NULL 覆盖而完全丢失掉了,这样就造成了一个内存泄漏。很显然，这样的内存泄漏是非常隐蔽的，如果不是内存非常紧张的话是不会被暴露出来的。正确的做法应该参考 C1-8 的做法。

```
1    void demo_func_2(void)
2    {
3        void * mem_ptr，* tmp_ptr;
4        mem_ptr   =   malloc(100);
```

```
5         …….
6         tmp_ptr    =    realloc(mem_ptr, 65536);
7         if (tmp_ptr)
8         {
9             mem_ptr = tmp_ptr;
10        }
11    }
```

12 C1-8 对 realloc 函数正确的调用方法

严格说来，象 C1-7 这样造成隐性内存泄漏的例子完全属于编码不当导致的，和被调用的函数 realloc 本身并没有太大的关系。但是有些库函数却会在被调用后隐性地产生了内存分配的行为，对于这类函数我们必须非常地小心，因为在软件中，内存问题从来都不会是小问题。让我们来看一看一个这样的函数 strdup，请参考 C1-9 的例子。

```
1     void dir_find_obj_name(t_char * obj_path)
2     {
3         t_char * tmp_path;
4         t_char * sub_dir, * dir_split;
5         ….
6         tmp_path = strdup(obj_path);
7         sub_dir = tmp_path;
8         ….
9         While (sub_dir)
10        {
11            dir_split = strchr(sub_dir, DIR_SPLITOR);
12            if (dir_split)    *dir_split = '\0';// we get one new sub_dir item!!!
13            dir_check_obj_exist(sub_dir);
14            …….
15            sub_dir = ++dir_split;//to next sub_item
16         }
17         ……
18         free(tmp_path);//嗯，这个语句是必要的!!!
19    }
```

20 C1-9 一个由于 strdup 造成内存泄漏的例子

正如我们在 C1-9 中所看到的一样，函数的主要目的是期望根据输入的文件路径信息来查找该文件是否存在。我们假定输入的 obj_path 是一个带有绝对路径的文件路径信息（比如说/tmp/download/log/20060101.log），由于我们需要从根目录开始，逐个目录的检查路径信息的完整性以及访问权限等等信息，因此我们需要将整个路径分段的分割成下面的几个对象名字, tmp，download, log, 20060101.log,然后调用 dir_check_obj_exist 依次检查这些子目录的有效性。为了逐段的获取各个子目录项，我们需要将每一个'/'的地方替换成字符串结束符'\0'，可是这样一来我们却会破坏了原始的字符串信息。因此解决的方法无非就是需要使用一个临时的内存（malloc 一片内存，局部变量，或是复制该字符串）来复制一下原始的字符串信息。在这里，我们为了更好的说明这个容易造成内存泄漏的问题，特意选用了 char * strdup(char * orig_str).这个函数很简单,就是分配一块临时内存，并将原始字符串的信息原封不动的拷贝到新的临时内存块，最后将该临时内存块的地址返回。从描述我们就可以看出,

如果我们不在 C1-8 的第 18 行处执行一个对该临时内存块的释放动作，那么毫无疑问，一个隐性的内存问题就这样产生出来了。

1.3.3 避免使用函数功能实现的盲区

所谓函数功能实现的盲区就是指可能会引发不可预知问题的函数调用条件。大多数的库函数通常是为了实现某一个功能而设计的，为了实现这些功能，往往在调用这些函数的时候我们需要给出一定的执行条件，这个所谓的执行条件就是我们所说的函数输入参数了。参数不同，函数执行出来的结果自然就会不同。但是，有些函数参数的组合可能会引发库函数进入到一个盲区，也就是函数执行出来的结果完全是不可预知的。请参考 C1-10 的例子。

```
1    void process_request(dword size)
2    {
3        t_byte    *    prefix;
4        prefix = (byte *)malloc(size);
5        if (pre_fix)
6        {
7            *prefix = 'B';
8            ……..
9            free(pre_fix);
10       }
11   }
12   C1-10 一个可能引发函数盲区的例子
```

从表面上看，上面的代码似乎没有任何问题的。实现必要的内存分配后，再检查一下分配出来的内存是否是有效的之后，然后对这块内存进行必要的操作之后再释放掉它。没有内存泄漏，一切都是合理合法的操作。但是结果真的是这样的吗？

让我们重新来仔细的看看这个函数代码的实现。我们知道，库函数 malloc 主要是用来向系统分配指定字节大小的内存块的。在常规情况下，如果分配成功的话，该函数就会将指定长度的内存块的首地址返回给函数的调用者，如果分配内存失败时，就会返回 NULL 以告知用户当前内存不够。从这样的函数使用说明看起来，代码似乎还是没有问题啊。但是，如果我们给出的 size 长度是 0 的话，请问 malloc 函数将会把什么返回给我们呢？事实上，C1-10 代码的问题就在这里，输入参数 SIZE 为 0 对于 malloc 函数来说是一个盲区. 在这一点上，各个系统平台上的库函数 malloc 返回的结果是不尽相同的，完全依赖于底层的实现。有些平台下面的 malloc 函数在 size 为 0 的时候，返回值可能是非 NULL 的，返回值用来指示当前可用内存地址块的最低地址。但是另一些平台下的版本却直接返回 NULL 了事.这样一来，问题就出来了，C1-10 例子中的第 4 行处的 malloc 调用在 size 参数为 0 的时候可能依然能够得到的非 NULL 的返回值，但是这个值并不是表示一块有效的内存地址块，紧接着在顺利的通过了第 5 行的代码检查后，将会执行第 7 行处的代码，而这条赋值语句的执行无疑将会导致指定的内存地址处的信息被异常改写，引发不可预知的错误。

另外一个存在函数实现盲区的例子是 void * memcpy(void *, const void *, size_t). 记得我们曾经在 1.2 中谈到过这一点。函数 memcpy 在大多数情况下都是能够正常工作的，但是，当原始内存块和目的地址块出现重叠(overlap)的时候，memcpy 函数执行的结果将是不可预料的，也就是说会出现一个盲区。当然，这个盲区的产生和平台是有很大的关系的，但是归根到底来说，主要还是因为这样的操作的结果是没有明确规定的，所以导致了各个平台下实现各不相同。下面我们就来仔细的剖析一下 C1-11 在各个平台下执行的结果为什么会有这么大的差异。

```
1    void demo_func_1(void)
2    {
3        t_char string[20] = “Hello”;
4        memcpy(string + 1, string, 6);
5    }
```

6 C1-11 存在库函数盲区的例子

在 UNIX 下，由于库函数 memcpy 的实现在内部是依照顺序逐字节拷贝的，因此我们就能从下面分布拷贝的步骤中看出来为什么最后的结果是”HHHHHHH???”.

UNIX 下 memcpy 内部实现的伪代码为

```
1    void * memcpy(t_byte * dest, t_byte * src, t_dword len)
2    {
3        while (len--) *dest++ = *src++;
4        …..
5    }
```

	执行前 string 的内容	执行后 string 的内容	操作解释
S1	Hello	HHllo	H0→e1(0 位置 H 覆盖了 1 位置的字符 e)
S2	HHllo	HHHlo	H1→H2
S3	HHHlo	HHHHo	H2→H3
S4	HHHHo	HHHHH	H3→H4
S5	HHHHH	HHHHHH	H4→H5 //结束符被覆盖!!!
S6	HHHHHH????	HHHHHHH????	H5→H6

在 LINUX 2.4.21-4 下面，情况又有所不同了，结果变成了”HHeelo”,还是让我们通过分步骤来看清楚结果是怎么出来的? LINUX 2.4.21-4 下面的 memcpy 的伪代码如下图所示（当然下面的这段 C 语言的伪代码是从汇编代码转换得到的）

```
1    void * memcpy(t_byte * dest, t_byte * src, t_dword len)
2    {
3        if ((len & 1)) *((U8*)dest)++ = *((U8*)src)++;//两个字节对齐!!!
4        len >>= 1;
5        if ((len & 1)) *((U16*)dest)++ = *((U16*)src)++;//四字节对齐!!!
6        len >>= 1;
7        for (i = 0; i < len; i++) *((U32*)dest)++ = *((U32*)src)++;
8    }
```

	执行前 string 的内容	执行后 string 的内容	操作解释
S1	Hello	HHelo	He→el//拷贝了两个字节!!!
S2	HHelo	HHeelo	elo?→lo??//拷贝了四个字节!!

WINDOWS-2000 下由于伪代码比较复杂，我们不再列出具体的执行步骤和每一次执行后具体的结果，但是我们还是可以从汇编代码中清楚的看到微软的 memcpy 函数的实现已经不再象我们在前面看到的 UNIX 的伪代码那样,只是简单的逐字节拷贝.事实上微软的 memcpy 库函数已经能够象 memmove 那样在监测到原始地址块和目标地址块出现重叠的时候自动的做相应的处理,也就是说微软的 memcpy 函数和 memmove 函数在调用效果是一样的了。

也许有些朋友可能会问,库函数 memcpy 怎么在各个平台下会有这么大的差异呢?不是说库函数的可移植性应该由平台来保证吗?如果说跨平台后库函数会出现这么大的差异性,那谁还能相信在这个平台上运行的非常好的代码在移植到别的平台下没有一点问题? 这个问

题问得相当的好。事实上，如果你发现移植到一个新的平台下后你的代码出现了问题，这恰恰说明你原来的代码中存在了一些不可移植的东西，正是这些东西导致了老代码在新平台下出现的问题(平台自身升级导致原有功能不支持引发的问题暂时不在我们考虑范围之内).而这些所谓的不可移植的东西就包含了我们前面说的函数的盲区。如果我们在刚刚开始写代码的时候就忽视了函数盲区的存在，就会在将来的某个时刻受到程序狠狠的惩罚。既然我们已经知道了函数 memcpy 存在着实现的盲区，那么我们就会在以后编码的阶段时刻注意到这个盲区，尽量的去使用更为安全，通用性更高的 memmove 函数。要知道，和 memcpy 函数不同的是，无论原始地址块是否会和目的地址块发生重叠与否，memmove 函数在每一个平台下得到的结果都是相同的!

1.3.4 避免使用容易传递错误的库函数

这句话是什么意思呢？和前面我们所提到的 memcpy 等函数不同，我们在这一部分谈到的函数大部分在功能实现上是没有问题的，也就是说不会存在功能实现上的盲区。但是，这一类的函数本身也没有比较好的保护机制，也就是说，一旦外部输入有错误，那么这一类的库函数就会将错误传递下去，而这是我们所不希望看到的，请看下面的这个例子。

```
1    void process_name(char * file_name)
2    {
3        t_char    tmp_name[MAX_FILE_NAME_LEN + 1];
4        ASSERT(NULL != file_name);
5        ……
6        strcpy(tmp_name, file_name);
7        ……
8    }
```

9 C1-11 一个容易传递错误的库函数例子

从表面上一眼看上去，C1-11 的例子似乎无懈可击。断言也有了，调用的函数也是标准库函数，看上去都是没有问题的。可是，说 C1-11 代码都没有问题的这种论断只能是在没有异常的情况下才成立的。为什么这么说呢？不妨让我们考虑一下这样的情况，在某个模块调用函数 process_name 之前，输入的参数指针 file_name 所指向的字符串因为某些其他程序的错误，导致结束符'\0'的丢失或者是输入的字符串长度远远大于 MAX_FILE_NAME_LEN，然后调用函数 process_name，会发生什么呢？聪明的朋友们可能已经看出来了，因为原始字符串的结束符丢失或长度过长，将会导致第 6 行处的 strcpy 函数在执行的时候会向目标地址写入大大超出 tmp_name 数组最大长度数目的字符数，而且因为 tmp_name 数组是从堆栈中分配的，这就导致一个隐性的堆栈溢出。

也许有些朋友会问，既然可能会导致这么严重的问题，那么我们为什么不能象第 3 行那样，加入一个 ASSERT(MAX_FILE_NAME_LEN >= strlen(file_name))的语句呢？对的，这样的想法毫无疑问是一个非常好的建议，这就恰恰说明朋友们脑海里面已经有了写防御代码的良好意识了。但是我们知道，ASSERT 语句一般只是在 DEBUG 版本上才工作的，一旦进入了 RELEASE 版本，这样的保护就将不复存在。而我们在这里期望的是一种更高的编码要求：即使函数的调用者给我们的是垃圾数据，我们也应该尽量做到不因为输入的是垃圾信息就紧跟着产生新的垃圾效应给下一层函数，让因为错误的输入数据引发的负效应链在我们的代码中中止。那么我们应该怎么写代码才能保证做到这一点呢？请参考 C1-12 代码的实现。

```
1    void process_name(t_char * file_name)
2    {
3        t_char     tmp_name[MAX_FILE_NAME_LEN + 1];
```

```
4
5       ASSERT(NULL != file_name);
6       ASSERT(MAX_FILE_NAME_LEN >= strlen(file_name));
7       memcpy(tmp_name, file_name, MAX_FILE_NAME_LEN);
8       tmp_name[MAX_FILE_NAME_LEN] = '\0';
9       ……
10  }
11
12  #define COPY_FIX_LEN_STR(dest, src, len) \
13  {\
14      memcpy((dest), (src), (len));\
15      *((char*)(dest) + (len)) = '\0' ;\
16  }
17  C1-12  一个能够主动抵御外在错误的例子
```

在 C1-12 的例子中，我们能够非常清楚的看到，不论是在 DEBUG 版本还是 RELEASE 版本下，不管输入的字符串是否是正常的，C1-12 的函数 process_name 都能够正常的处理而不会带来其他的负效应的问题。C1-12 就是一个我们向朋友们展示了如何写出更为强壮的函数来抵御外在不可预知的风险的代码例子。当然，如果朋友们觉得象 C1-12 那么写比较烦琐的话，就不妨考虑使用 C1-12 的第 21 行到第 25 行的宏定义方式的 COPY_FIX_LEN_STR 或是直接改成 inline 函数也是可以的。

类似的函数主要是 strcpy, strcat 之类的,也许正是因为这种容易引发潜在性的问题的缘故，所以在不少软件公司的内部编程规范中都将此类的函数列入了禁用之列。

2. 小心边界条件引发的陷阱

事实上，翻开关于 BUG 的档案记录，我们非常吃惊的发现有无数的程序员在这个方面都出现过纰漏，当然也包括作者本人。严重的直接导致系统重启，死机甚至进入代码的死循环，程度轻一些的也会造成其他模块工作环境被破坏，以至于直接影响了整个系统功能的正常执行。下面我们就来逐个的探讨一下这一方面的陷阱。

2.1 条件判断的陷阱

通常说来，条件判断的陷阱主要还是表现在部分边界的判断上。比如说改用'>='的时候却使用了大于，这样就会导致本来应该执行的代码却没有被执行，本来不该执行的代码却被执行了，造成了不应该的环境污染,造成对不该操作的内存空间的数据的隐性破坏，从而导致其他的问题。请参考 C1-13 一个对循环缓冲区的写函数的例子。

```
1   t_boolean   loopbuf_write(s_loopbuf * loopbuf, t_byte info)
2   {
3       ……
4       *loopbuf->write++ = info;
5       if (loopbuf->write > loopbuf->base + loopbuf->size)
6       {
7           loopbuf->write = loopbuf->base;
8       }
9       …….
10  }
```

```
11  typedef struct
12  {
13      t_dword     size;
14      t_byte      * base;
15      t_byte      * write;
16      t_byte      * read;
17  }s_loopbuf;
```

18 C1-13 一个存在边界问题的缓冲区操作的例子

从 C1-13 中我们可以很明显的看出来第 5 行到第 8 行的代码是试图做一个 rewind 的动作，因为是循环缓冲区，所以我们必须在写到缓冲区的尾部的时候将写指针重新回退到缓冲区的头部，这也是循环缓冲区的特性所决定了的。但是问题就出现在第 5 行上，我们知道缓冲区的总长度如果是 100 个自己的话，那么我们只能操作从偏移量为 0 的地址到偏移量为 99 共 100 个字节的空间，所以 write 指针所能到达的最大有效偏移地址是 size – 1！这就是说在 C1-13 的例子里面会造成一个对非法地址的写一个字节的操作，在大多数情况下会导致一个异常死机。正确的代码应该是将第 7 行处的条件判断语句修改为

if (loopbuf->write >= loopbuf->base + loopbuf->size)。

类似的问题还出现在对数组元素的操作上面。在 C 语言中声明一个数组 char info[100] 仅仅表明你有存取从索引 0 到索引 99 这 100 个字节单元空间的权利。我们是没有权利存取索引为 100 的字节单元(info[100])的权利的，读写这样单元的操作是应该被严格禁止的。也许有些朋友会反驳说，我有的时候就是在函数内部声明了一个数组变量，然后写了你所说的合法数组单元之外的地址也没有出现象你说的死机问题啊。但是事实并不像表面上看上去的那么简单，严重一点的就是你可能已经破坏了这个数组周边的声明的变量内容，轻度一些的也许没有覆盖任何其他变量的内容，那不是没有问题，只不过是编译器在无意中挽救了你（编译器为了操作数据快速的缘故，在默认的情况下会自动的将部分保留字节空间填充在你声明的各个变量之间（具体技术细节请参考第二章），而刚好你就是往这些填充字节空间中写入的数据而已），但是如果换一个编译器或是修改了编译器的编译环境后你觉得你还能依然这么幸运吗？

还有一种是比较特殊的一种条件判断陷阱。我们知道通常来说在做条件判断的时候，一般都要求条件判断符两边的数据的类型是对等的

Float value

if (value < 0.1)

2.2 忽略了对象本身的隐性约束条件

什么叫对象本身的隐性约束条件呢？就像我们在 2.1 对 C 语言数组讨论的那样，尽管代码上你声明的数组空间大小是 100，但是你能操作的单元范围只能是 0 到 99（SIZE – 1），这个隐性的约束条件让很多初学 C 语言的朋友们都犯过错误。所以我们常常会看到象类似于下面 C1-14 这样的例子，

```
1   void demo_func(void)
2   {
3       t_long i;
4       for (i = 0; i <= size; i++)
5       {
6           array[i] = i + 1;
7       }
```

```
8    }
```

9 C1-14 C 语言数组操作越界的问题

除了数组之外，还有很多类似存在隐性约束条件的对象，其中一个最典型的就是 C 语言中的字符串了。例如我们可以通过代码 char prefix[10] = “Hello”来声明字符串”Hello”. 和 PASCAL 语言有所不同的是（PASCAL 语言在字符串首地址前面会保留一段区域用于存储该字符串的真实长度和以及用于存储该字符串的内存块的总长度），C 语言中是通过在字符串的尾部放置一个特殊的字符 0 表示字符串的信息到此为止的。也就是说，对于类似于 prefix 的声明，在 PASCAL 语言中和 C 语言中的大致内部结构信息如下(little-endian 的模式，当然仅仅是个示意图，具体的结构可能会有所变化),

PASCAL(prefix 将指向字节偏移为+0 的地址)

字节偏移	-4	-3	-2	-1	+0	+1	+2	+3	+4
字节信息 1	0x0A	0x00	0x05	0x00	0x48	0x65	0x6C	0x6C	0x6F
字节信息 2	10		5		H	e	l	l	0
具体意义	数组总长度		字符串总长		字符串的内部内容				

C 语言(prefix 将指向字节偏移为+0 的地址)

字节偏移	-4	-3	-2	-1	+0	+1	+2	+3	+4	+5	+6
字节信息 1	N/A	N/A	N/A	N/A	0x48	0x65	0x6C	0x6C	0x6F	0x00	??
字节信息 2	N/A		N/A		H	e	l	l	o	0	??
具体意义	N/A		N/A		字符串的内部内容						

从上面这两张表中我们可以明显的看出来在字符串管理方面 PASCAL 要比 C 语言要安全而且高效一些。说高效是因为 PASCAL 语言在做字符串操作之前就已经知道了字符串的真实大小和实际最大的存储空间，因此操作的时候是不用象 C 语言那样首先去获取串长(请参考 strlen, strcat 等等)。而说安全是因为 PASCAL 语言可以提前知道存储空间的最大长度从而避免出现拷贝出现溢出的情况。反过来看 C 语言，在声明完 char prefix[10] = “Hello” 之后，数组 prefix 的空间长度信息其实就已经丢失了（这其实也这是我们为什么说 strcpy 之类的函数是不安全的根本原因所在），因此后面对字符串操作的安全性几乎完全掌握在程序员的手中，而无法象 PASCAL 语言那样可以通过内部的安全机制来自动防御错误，这其实相应地提高了对 C 程序员在操作字符串代码时的要求。除此之外，我们也许就可以理解为什么 C 语言中字符串的结尾符号’\0’尽管并不记入字符串的内容但却需要计入字符串的存储空间的原因了，因为没有它将无法知道字符串的内容究竟在什么地方才算结束。很多 C 语言的程序员都会忽略这个非常重要的字符的存在以至于写出会导致存取内存非法错误的代码。

```
1    void demo_func(void)
2    {
3        t_char    machine_name[MAX_FILE_NAME_LEN];
4        get_machine_name(machine_name, MAX_FILE_NAME_LEN);//????
5    }
6    void get_machine_name(t_char * dest_buf, t_dword buf_size)
7    {
8        static t_char local_name[ ] = “Helianthus_001”;
9        strcpy(dest_buf, local_name);
10   }
```

```
11  void demo_func2(t_char * file_name)
12  {
13      t_char * tmp_name;
14      tmp_name = (char *)malloc(strlen(file_name));//????
15      ……..
16  }
```

17 C1-15 忘记给'\0'留下空间了！

正如 C1-15 中的例子，第 4 行和第 14 行都忘记给这个结束符预留空间了！这就直接导致当库函数在操作完自动写这个结束符号时因为没有它的空间而将这个结束符写到紧随其后的那个字节空间内，而这个空间往往是别人使用的空间！从而引发堆栈被破坏或是数据被异常改写的错误。

2.3 数据类型引发的边界条件陷阱

相对于前面两种情况引发的边界条件陷阱，这种由于数据类型自身内在特性导致的边界条件约束更为隐蔽，非常容易带来很隐蔽的陷阱。

第一种陷阱是因为数据类型自身的所能表达的数据域范围所引发的。我们在前面的章节已经知道，一个字节的所能表达的数据梯度只能达到 256 级，这是因为一个字节只有 8 个比特(bit)的缘故。同样，一个字(word)所能表达的梯度可以达到 65536 级，这是因为一个字有 16 个比特的缘故。因此，这就包含了一个隐性的约束条件，比如，你不能拿梯度只有 256 级别变量去和梯度超过 256 的做比较，否则就会引发问题。

```
1   typedef enum
2   {
3       SYSX_MSG_0000 = 0,
4       SYSX_MSG_0001,
5       SYSX_MSG_0002,
6       SYSX_MSG_0300,
7       SYSX_MSG_MAX
8   }SYS_MSG_ID;
9   void demo_func(void)
10  {
11      t_byte    i = 0;
12      for (i = 0; i < SYSX_MSG_MAX; i++)
13      {     /* Check whether this is valid message??? */
14            …….
15      }
16  }
17  void demo_func_2(void)
18  {
19      t_byte    i = 10;
20      while (i-- >= 0)
21      {
22          ……..
23      }
24  }
```

25　C1-16 因为数据类型表达范围局限性导致的问题

当消息比较少（少于 256 个）的时候，因为枚举值 SYSX_MSG_MAX 也小于 256 的时候，函数 demo_func 还是可以工作的。但是如果突然有一天，需要大量增加消息 ID 的时候，你按照要求往枚举 SYS_MSG_ID 里面添加了很多新的消息 ID 定义之后也许就会惊奇的发现，函数 demo_func 突然出现死循环了！为什么？原因就在 C1-16 的第 12 行上！因为你增加了很多新的消息 ID，导致 SYSX_MSG_MAX 的值也会远远大于 256。但是因为第 12 行的循环变量却使用了一个只能表达出 256 级梯度的变量，这就直接造成当变量 i 等于 255 的时再做加 1 动作的时候，0xFF + 1 = 0x100，但是因为只能存储一个字节,所以 i 的值不是 256，而是重新退回到 0 了。所以程序就会陷入在第 12 行的死循环上。当然,如果我们使用合适的循环变量类型或是加入对 SYSX_MSG_MAX 范围检查的断言也是可以避免这个问题的。类似的，demo_func_2 函数也存在这样的问题。

第二种类型和第一种却有所不同。在第一章里面关于 C 语言中各种数据类型的讨论之后我们知道，当我们用调试器去看某一个变量值的时候，调试器所显示出来的值的大小取决于两个方面，一个是当前内存中的该变量存储空间里面具体的数据，另外一个就是这个变量的数据类型，这就好比我们是通过各种变色镜看一个物体，尽管物体还是那个物体，可是物体的颜色看起来却是不相同的了。举一个例子来讲，我们假定内存 0x2000 地址上的那个字节内容为 0xAA， 即 (0x2000)= 0xAA;如果我们用一个 unsigned char* 的指针指向 0x2000 然后再看该指针指向的内容，你就会发现因为这个字节本身的内容为 0xAA，但是因为这个最终的对象是一个无符号的，因此这个你看到的值将是 0xAA(Hex)/170(decimal)/ ª(char). 但是当你用 signed char * 类型的指针也指向 0x2000 这个地址的再显示该指针指向的内容的时候，尽管里面的内容没有任何变化，依然是 0xAA,但是由于数据类型从无符号转变成有符号之后，那么你看到的值将变成 0xAA(Hex)/-86(decimal)/ ª(char). 十进制模式下值从 170 突变成-86 的根本原因在于 0xAA 的 bit7=1 在有符号的情况下这个 bit7=1 需要被解释成一个负号的标志,而不能象无符号类型那样被当作有效的数据位计入最终的显示值中，因此在有符号的情况下（注意中间有个求反加 1 的过程），
0xAA(Hex) = 10101010(binary) = -(01010101 + 1) = - 0x56 = -86(decimal)。正因为存在这样的情况，因此在某些特殊的数据类型在做数学运算的时候就很容易会进入一些很隐蔽的陷阱。

```
1    void demo_func(void)
2    {
3        t_char    value = 127;
4        t_int     total = 0;
5        value++;
6        total += value;
7    }
```

8　C1-17　一个因为数据类型边界溢出导致负号反转的例子。

如果我问朋友们,在 C1-17 中，第 6 行代码执行之后的 total 的数据值是多少呢？如果没有仔细分析的话，粗心的朋友们可能会说出 128 这个答案来。但是真的是这样的吧？让我们来仔细的分析一下。由于我们在上面分析的原因，相信大多数朋友们应该不会对这个消息感到很吃惊：第 5 行执行后，value 的值应该是-128 而不是+128!在了解了这样的事实之后，我相信大多数朋友应该可以得出正确的答案了：total 的值应该是-128。原因比较简单，
127 + 1 = 0x7F + 1 = 0x80 = -(0x7F + 1) = -128。

```
1    void demo_func(dword check_range)
2    {
3        t_int i;
4        for (i = 0; i < check_range; i++)
5        {
6            ……
7        }
8    }
```

9 C1-18 另一个可能因为符号反转导致死循环的例子

在 C1-18 中同样存在着一个潜在的死循环问题，请朋友们想一想为什么？死循环是在什么情况下发生的？你现在明白了为什么很多编程规范中都反复强调为什么在有符号和无符号的变量之间做条件判断的时候要非常小心了吗？

2.4 边界截断

其实在了解了 2.3 中的数据类型约束之后，我们就能马上明白这种陷阱了。

```
1    void demo_func(t_word size_limit)
2    {
3        t_byte      size;
4        t_word      i;
5        size = size_limit;
6        for ( i = 0; i < size; i++)
7        {
8            …….
9        }
10   }
```

11 C1-19 一个出现数据截断的例子

由于数据类型表达范围的约束，因此如果出现 C1-19 第 4 行的语句将会引发一个数据截断的动作。比如说输入的 size_limit 是 0x034A, 但是在第 4 行处会出现一个数据截断。导致 size 变量得到的真正值是 0x4A, 而不是想像中的 0x034A。更为严重的例子可能会导致 for 循环语句无法执行。比如说当 size_limt 的值为 0xXX00(X 表示任意值)，都会导致 size 为 0，从而导致循环体根本就不会执行了。一般说来，象诸如此类的问题一般可以通过一些商用的软件工具(如 Insure 或是 BoundsCheck 之类的)来检测出来，提示的警告信息一般都是数据类型转换精度损失（conversion precision loss）。

下面的 C1-20 是另外一个例子，你能说出来为什么 demo_func_1 函数可能会有问题，为什么要尽量地修改成 demo_func_2 函数的写法呢？（当然不是唯一的写法）

```
1    void demo_func_1 (t_word elem_num,    t_word each_elem_size)
2    {
3        t_dword    total_size;
4        void    * elem_ptr;
5        total_size = elem_num * each_elem_size;
6        elem_ptr = malloc(total_size);
7        …….
8    }
9    void demo_func_2(t_word elem_num, t_word each_elem_size)
```

```
10  {
11      t_dword    total_size;
12      void       * elem_ptr;
13      total_size      = each_elem_size;
14      total_size    *= elem_num;
15      elem_ptr = malloc(total_size);
16      ….
17  }
```

18 C1-20 另外一个典型的因为数据边界截断的例子

3. 关注多出口函数的陷阱

相信很多写过多年程序的程序员都或多或少的看到或者是听到过编程规范有要求一个函数尽量的少出现多个出口，尽量保持一个出口比较好。为什么呢？其实这款教条其实是在诸多因为多出口引发的陷阱造成的各种各样的问题之后总结出的血的教训。为什么这么说呢？下面让我们来看一看这样的例子

3.1 资源泄漏

资源泄漏的问题属于代码编写引发出的一个相当古老的问题，和软件自身的结构设计几乎是没有关系的。即使是在今天，依然会有很多这样的问题逃过我们的眼睛，进入最终的产品中。尽管资源泄漏的问题在编码阶段试图完全避免几乎是不可能的，但是依然还是有不少方法被提出来用于抵御这样的问题，比如尽量保持一个函数的出口。在函数的每一个出口处我们都应当尽量的问一问自己，函数中分配的局部资源是不是都已经全部被释放了，信号量，内存，或者其他的资源对象？

```
1   t_boolean demo_func_1(void)
2   {
3       t_byte * mem1, * mem2;
4       mem1 = (t_byte *)malloc(100);
5       if (NULL == mem1) return FALSE;
6       mem2 = (t_byte*)malloc(100);
7       if (NULL == mem2) return FALSE;
8       ……..
9       return TRUE;
10  }
11  t_boolean demo_func_2(void)
12  {
13      t_byte * mem1, * mem2;
14      mem1 = malloc(100);
15      if (NULL == mem1) goto common_error;
16      mem2 = malloc(200);
17      if (NULL == mem2) goto common_error;
18      ……..
19      return TRUE;
20  common_error:
21      if (mem1) free(mem1);
22      if (mem2) free(mem2);
```

```
23        return FALSE;
24    }
```

25 C1-21 一个典型的因为函数的多个出口导致的内存泄漏

聪明的朋友可能已经看出来了，在 C1-21 中的第 7 行存在一个隐形的内存泄漏问题：如果此时直接返回，就会导致前面 mem1 内存块的泄漏。在大多数的运行环境中，第 6 行 mem2 的内存分配要求是可以成功的，这就意味着一个隐形的内存泄漏问题被带入了最终的产品代码。一个比较好的替代方案如 C1-21 中的 demo_func_2 函数所示，里面使用的 goto 语句的目的主要是希望保持整个函数在逻辑结构上的简洁性。当然，尽量的减少函数的出口不仅仅能有效的减少代码的资源泄漏问题，同时还能够有效的抵御代码在维护阶段可能带来的问题，请看下面的例子。

```
1    t_boolean demo_func_3(void)
2    {
3        byte * mem1, * mem2;
4        mem1 = malloc(100);
5        if (NULL == mem1) return FALSE;
6        mem2 = malloc(200);
7        if (NULL == mem2)
8        {
9            free(mem1);
10           return FALSE;
11       }
12       …….
13       return TRUE;
14   }
```

15 C1-22 一个因为函数多出口可能导致维护代码阶段引发问题的例子

有些朋友可能会说，我不是很赞同 C1-21 例子中使用 demo_func_2 解决 demo_func_1 函数隐形内存泄漏的方法。我只要象 C1-22 中的 demo_func_3 那样注意到 mem1 的内存泄漏，并且将这个内存泄漏解决掉不就可以了吗？为什么非要遵循所谓的那个函数尽量只有一个出口的规则呢？确实，我们承认，在 C1-22 中的函数 demo_func_3 中并没有采用我们所谓的尽量保持一个函数出口的原则也照样解决了 C1-21 中 demo_func_1 中的内存泄漏问题。但是仅仅是在目前。为什么这么说呢？我们知道，软件代码的生命周期其实是很长的，即使产品发布了，生命周期依然没有结束，还有后续的代码维护阶段以及新特性开发的阶段。因此，我们在一开始编写代码的时候就必须意识到这一点，尽量在编写代码的时候就尽量去考虑到如何在后续的阶段避免因为后来者因为经验或是对模块特性了解不足引发的人为因素所造成的各种软件维护问题。尽管按照避免资源泄漏问题来看，C1-22 的 demo_func_3 和 C1-21 中的 demo_func_2 都达到了目的，但是很显然，demo_func_3 的代码在维护阶段依然可能引发象 demo_func_1 那样的隐形的内存泄漏问题。比如说，对于 demo_func_3 函数，新的代码维护者可能需要动态的分配 300 个字节的内存块 mem3 来实现一个新的特性需求。对于 demo_func_2 来说，因为内存分配的代码比较统一规范，他很可能只要在上面增加一个内存分配语句，在公共的出错口处增加一个内存释放就可以了。但是对 demo_func_3 来说，问题就不那么简单了，他需要增加一个 mem3 的内存分配语句，同时还必须保证如果出现了内存分配失败的情况，必须将 mem3 之前分配出的所有内存全部释放！如果还有所谓的 mem4，mem5 的话，代码的维护量又会增加多少呢？可见，尽量保持一个函数出口的编码风格，对于主动的抵御潜在引发的代码问题是具有相当重要的意义的。之所以这样说的根本原因就在

于人们难以避免在问题复杂度增加的情况下不犯错误。在当前软件复杂度日益增加的今天，如果不能在编码初期阶段就保持一个良好的习惯，请问，当面对下面 C1-23 例子中函数时候，如果不能遵循尽量保持函数一个出口的原则，你觉得自己依然能够幸运地躲过形形色色资源泄漏问题的陷阱吗?在以后的维护阶段呢？例子中除了内存泄漏外还会发生什么情况呢？

```
1     void demo_func_1(void)
2     {
3           t_byte    * mem1, * mem2;
4           t_word    idle_slot;
5           spin_lock(&api_lock);
6           mem1 = malloc(100);
7           if (mem1) return FALSE;
8           idle_slot = table_getslot( );
9           if (INVALID_SLOT == idle_slot) return FALSE; /* no idle slot!!!! */
10          if (ERROR == process_slot(idle_slot)) return FALSE;/* we meet failures!!! */
11          …… ;/* we continue process with all resource available */
12          table_putslot(idle_slot);/* return slot back to resource pool */
13          free(mem1);
14          spin_unlock(&api_lock);
15          return TRUE;
16    }
17    C1-23 一个因多出口可能引发多种复杂资源泄漏的例子
```

4. 小心程序语法规则引发的陷阱

每一种程序设计语言都有自己独特的语法规则，C 语言也不例外。因此，在开始编写代码之前，我们最好来看一看 C 语言中哪些语法规则可能会偷偷地埋下陷阱。

4.1 小心 C 语言中地代码行结束符';'

在 C 语言中，符号';'是用来标识一个代码行的结束，因此我们在写代码的时候就必须非常小心，以免造成不应有的错误，C1-24 就是这样一个新手非常容易犯的错误例子。

```
1     void demo_func_1(void)
2     {
3         t_byte size = 0, value = 0;
4         while (value++ < 100);
5             size++;
6     }
7     C1-24 一个';'引起的 while 逻辑错误
```

4.2 切莫将'='误用成关系运算符号'=='

在 BASIC 之类的程序设计语言中，用在条件判断语句中的符号'='是表示关系运算的。但是在 C 语言中，这个符号是用来表示赋值操作的，不少 C 语言的新手往往会陷入在这个陷阱中。

```
1     void demo_fun_1(t_byte size)
2     {
3           t_byte value = 0;
4           if (size = 100)
```

```
5          {
6                value++;
7          }
8     }
```

9 C1-25 一个误将赋值操作符号'='当作关系运算符'=='的例子

如 C1-25 中所示，第 4 行处的代码就属于典型的这一类的错误。C 语言中做条件判断的关系运算符应该是' == '，很多 C 语言的新手就因为没有搞清楚这一点而犯错。事实上，由于第 4 行的语句是'size=100'，这就是说首先会将 100 赋值给 size 变量，随后因为语句'size=100'的值是 100，因此 if 语句的条件将为真，从而执行 value++的动作。在这个案例中，最危险的操作是将关系运算符误用成赋值符，从而导致 size 变量的值被异常刷新！为了在编写代码阶段有意识地来抵御这种疏忽，因此很多编程规范中都会强调在写条件判断语句的时候，常量必须放在符号的左边。这样的话，C1-25 中的第 4 行语句就变为'100=size'，很显然，语句'100=size'会引发编译器的编译错误，因为我们是不可能将一个变量赋值给一个常数的。规范正是试图通过这样的一种有意暴露错误的方式来主动避免这种陷阱的。

4.3 运算符优先级陷阱

和前面两种语法陷阱对比起来，这一类的陷阱显得非常隐蔽，所以很多程序员都会在不经意间陷入到这样的陷阱中去了。

```
1     void do_sum(t_byte input)
2     {
3          t_word    sum = 0;
4          ……
5          sum =    input + sum >> 1;/*  嗯，写成  sum = input + (sum >> 1);?? */
6          …….
7     }
```

8 C1-26 运算符优先级引发的陷阱

C1-26 就是一个非常典型的案例。在第 5 行处，程序员原本的目的也许是希望将 sum 的值右移一位以后再和输入的 input 值相加。但是因为在 C 语言中，符号'+'的运算优先级要高于符号'>>'，这就导致第 5 行的代码被编译器解释成 '首先将 input 和 sum 做加法运算，然后将运算后的结果再右移 1 位'。很显然，代码真正的动作和程序员设想的是不同的，解决这个问题的办法就是使用括号'('和')'来提升指定目标的运算优先级别。由于 C 语言中的运算符是相当多的，因此，为了防止歧义同时提高代码的可读性，很多编程规范会强制要求加括号来显式地表明运算的先后次序。类似于 ' mid=(high << 8) | low ' 以及 ' if ((a & b) && (c &&d))'的语句都是这种好例子。

4.4 数据运算的转换规则

```
10 / 3;
```

5. 关注对象未初使化的问题

E5 文件系统　打开文件描述符号由于没有初使化，造成 DRM 文件关闭的时候有时会将某些使用中的文件描述符异常清空并释放，导致死机。

M5 & E5 中收到短信容易 RESET。原因是因为 PRES 里面某个结构体中包含了两个消息空间的指针，在最后判断为非空的时候会错误释放内存。

6. 只使用属于自己的合法空间

6．1 变量的声明

```
1    void demo_func_1(void)
2    {
3        t_char    str[16];
4        ……..
5        return    str;
6    }
```

另外一个例子，就是有些模块在企图释放栈空间上面的内存空间。

```
1    void demo_func(void)
{
    string = malloc(100);
    ….
    {
        char tmp_string[20];
        …..
        string = tmp_string;
    }
    if (string) free(string);
}
```

第三个例子是使用了悬挂指针。

比如 M5 中短信 inputer 中的全局变量在构造函数中被初使化，但是在析构函数中却没有被清除，导致使用过期对象，从而导致死机。

还有一种类似的现象就是内存被释放之后没有及时将内存指针清除，从而导致悬挂的指针，容易引发内存的二次释放，导致死机。

6．2 公共空间

```
1    void demo_func_1(t_char * input)
2    {
3        static    t_char m_local_str[MAX_BUF_SIZE + 1];
4        strcpy(m_local_str, input);
5    }
```

7. 尽量不要替编译器写代码

看到这里，有些性急的朋友们有可能会问了，我们什么时候替编译器写代码了？其实我们在这里想说的是，我们在编写代码的过程中常常是无意识的写出了很多无效的代码，这些代码其实让编译器来写可能会更好一些，除了编译器编译代码实现过程中可能因为存在差异性从而可能会导致各种潜在的问题之外，编译器在编译代码的过程中了解很多内部的细节信息，所以有些事情让编译器去做可能更合适。一个非常典型的例子就是编程规范中常说的多用 sizeof 语句。

7.1 让编译器去计算元素的大小

有的时候，有些朋友也会写出类似于下面 demo_func_1 这样的代码出来，虽然从逻辑功能的角度上来看，也许这样的代码并没有任何错误，m_cmd_cnt 变量只是用来标识 m_cmd_string 数组里面有多少个字符串而已，因为 demo_func_1 里面目前只有 3 个字符串，所以我们把 m_cmd_cnt 的值置为 3，如果以后需要增加或是减少 m_cmd_string 里面的元素个数，我们只需要将 m_cmd_cnt 的值做相应的改变就可以了。一切都很完美，不是吗?

```
1   t_char * demo_func_1(t_word index)
2   {
3       static char         * m_cmd_string[ ] = { “ls”, “cd”, “rm”, NULL};
4       static t_dword      m_cmd_cnt = 3;
5       index = (index >= m_cmd_cnt)? m_cmd_cnt: index;
6       return m_cmd_string[index];
7   }
8
9   t_char * demo_func_2(t_word index)
10  {
11      static char *         m_cmd_string[ ] = {“ls”, “cd”, “rm”, “cp”, “ln”, NULL};
12      static t_dword      m_cmd_cnt = sizeof(m_cmd_string) / sizeof(m_cmd_string[0]);
13      index = (index >= m_cmd_cnt)? m_cmd_cnt: index;
14      return m_cmd_string[index];
15  }
16  C71-2
```

从纯粹的代码功能实现上来讲，确实如此，demo_func_1 没有功能缺陷，但是 demo_func_1 的代码最大的问题在可维护性上！正如前面说的，每次增加/减少 m_cmd_string 数组元素个数的时候，我们必须对 m_cmd_cnt 的大小做出相应的改动以使之匹配。人都会犯错误的，很多时候会在维护代码的过程中引发新的问题。比如象 demo_func_2 那样，我们在 m_cmd_string 中添加了 2 个新的字符串”cp”, “ln”, 但是忘记将更新 m_cmd_cnt 从 3 更新到 5, 那么就会直接导致任何试图使用“cp”,”ln”命令的失败！因为函数会返回 NULL! 反之，让我们来仔细看看 demo_func_2 中的代码实现。由于我们使用了 sizeof 的语句，所以如果 m_cmd_string 发生任何变化的话，那么 m_cmd_cnt 变量的值会在编译阶段被自动的更新掉，在这个过程中并不需要朋友们手工的来改变 m_cmd_cnt,因为人们都是会犯错误的，为什么不能让编译器在代码发生变化的时候主动的去适应这种变化呢?

7.2 让编译器计算数据对象的地址

```
1   typedef struct taginfo
2   {
3       t_byte      elem_1;
4       t_dword   elem_2;
5   }t_info;
6   void demo_func_1(t_info * info)
7   {
8       t_dword     offset;
9       offset = (t_byte*)(&(info->elem2)) – (t_byte*)info;// 嗯，偏移值可不是 1 个字节!
```

```
10        ……
11    }
12
13    #define get_offset(struct_type, elem_name)    (&((struct_type*)0)->(elem_name)))
14    void demo_func_2(t_info * info)
15    {
16        t_dword    offset;
17        offset = get_offset(t_info, elem2);
18        ……..
19    }
```

20 C71-2 让编译器主动计算地址

在上面的例子中，我们试图获取

7. 关于 C 语言中的行内汇编

```
__asm
{
    MOV BX, AX;
}
```

第三章　了解代码的编译

3.1　请不要对编译器做假设

3.1.1　请不要对对象空间布局做出假设

```
1     void demo_func_1(void)
2     {
3         int i;/*  高地址  */
4         int j;/*  低地址  */
5         memset(&j, 0, 2 * sizeof(int));/*  嗯，想这样把 I & j  都置为 0???? */
6         …….
7     }
```

8 C3-1 一个依赖于编译器行为的例子

C3-1 中的例子当然是比较少见的，我们现在用这样一段古怪的代码的目的主要是试图用来说明我们应当尽量的避免对编译器的行为做出各种假设。编译器能够保证每一个数据对

象都存在一个唯一的地址空间，但是各个对象之间的内存分布位置则是不可预知的。这也就是说，我们在源代码中声明对象的时候可能是有先后关系的，但是编译器并不一定会按照我们在源码中声明对象的先后关系为对象分配地址空间。C3-1 的第 5 行代码就是这样的一个例子。这行代码试图通过 memset 函数调用将 j 和 i 地址空间里面的信息全部清 0 来达到将 i 和 j 两个变量赋值为 0 的目的。但是，这样的 memset 函数调用能否成功执行其实是有一个提前的，这个前提就是在堆栈空间上，j 变量的堆栈地址是在低处，而 i 变量的地址是在高处。我们知道堆栈空间的分配会导致堆栈指针 SP 减少（向低地址偏移），而堆栈空间回收会导致堆栈指针 SP 增加（向高地址偏移）。因此，如果编译器是按照变量的声明次序来分配堆栈空间的，那么 I 变量将被放在堆栈的高地址，而 j 变量将被存放在随后的低地址处，（这种情况下的堆栈布局请参考 P3-1）那么 C3-1 的函数也许可以幸运的逃过一劫，一切平安无事。但是如果编译器不是按照源码中变量声明的次序来分配对象的地址空间，那么 memset 语句将可能会破坏堆栈中的函数返回地址或是其他的对象内容信息，从而导致死机或是其他不稳定的问题。尽管大多数编译器的确是按照对象声明的顺序来实现的，但是这种行为并没有得到统一的支持，这也就意味着如果你是以这种假设为前提来编写你的代码的话，你的代码很可能会在你意想不到的时刻给你造问题。

3.1.2 小心编译器背后的动作—Padding(填充)

编译器除了可能以我们未曾了解的方式来产生对象空间布局之外，还有可能在必要的情况下偷偷地添加部分我们看不见的数据对象成员，这就是非常常见的现象—padding。Padding 常常是因为编译器受到机器硬件的约束或是完成效率优化的阶段被引入的。

其实 padding 并仅仅局限于对数据结构上。事实上，对于那些在函数内部局部堆栈上声明出来的局部变量同样存在这样的问题。从第一章的预备知识我们了解到，如果被存取的数据在内存地址上是边界对齐的，那么数据的存取效率将会大大提高，否则存取效率会很低或是导致硬件总线错误。正是出于类似的原因，因此编译器在编译源码的时候可能就会在发现数据成员没有边界对齐的时候自动地插入部分填充的字节空间来保证对数据对象是边界对齐的。C3-2 就是这样的一个典型例子。

```
1   typedef struct taginfo_a
2   {
3       t_byte      elem_1;
4       t_dword   elem_2;
5   }t_infoa; //一个健壮性较差的结构体声明例子
6   #define t_infoa_size   (sizeof(t_byte) + sizeof(t_dword))
7
8   typedef struct taginfo_b
9   {
10      t_byte      elem_1;
11      t_byte      reserve[3];
12      t_dword   elem_2;
13  }t_infob; //一个健壮性很好的结构体声明例子
14  void demo_func(void)
15  {
16      t_infoa * info;
17      info = (t_infoa *)malloc(t_infoa_size) ;//少算了 padding 的 3 个字节空间??
18      if (info)
```

```
19          {
20              info->elem_2 = 0;//此处将引发 3 个字节的写溢出动作!
21          }
22          ……
23      }
```

24 C3-2 由于编译器 padding 动作引发问题的例子

在 C3-2 中，我们看到对同样数据对象元素的两种不同的声明方式，t_infoa 和 t_infob。但是为什么我们说 t_infoa 是一种比较差的方式呢? 在 t_infoa 的声明方式中，第一个元素 elem_1 占用了一个字节的空间，elem_2 将占用四个字节的地址空间，出于存取效率的优化考虑，编译器将 elem_1 后面的 3 个字节的空间全部空置掉，然后将 elem_2 放置在距离 elem_1 偏移为 4 个字节的连续 4 个字节地址空间上。这就是说，t_infoa 结构体真正占用的空间大小为 elem_1 + padding + elem_2 = 1 + 3 + 4 = 8 个字节，而不是象我们在源代码声明里面所占用的空间大小，elem_1 + elem_2 = 1 + 4 = 5 个字节! C3-2 中第 17 行处由于分配内存的时候少分配了编译器自动 padding 出来的 3 个字节的空间，从而导致了第 20 行处操作 elem_2 的语句将会内存溢出 3 个字节! 这也就是很多有经验的老程序员更倾向于使用 sizeof(t_infoa) 来替换第 17 行处 malloc 中的 t_infoa_size，因为编译器比你更清楚源码后面的一切。padding 之前的内存布局请参考 P3-1，padding 之后的内存布局请参考 P3-2。

| elem_1 | X | X | X | elem_2

我们可以看到，P3-2 的内存分布和 C3-2 中的 t_infob 的分布其实是一致的。为了放置编译器在编译代码时候做出 padding 的动作从而改变这个结构体空间的大小布局分布，所以很多数据结构在声明时刻就会主动的考虑到 padding 的影响，从而将 padding 的影响降低到最低，这就是为什么我们说 t_infob 的声明方式要优于 t_infoa 的根本原因。

如果我们了解编译器在编译源码过程中的背后可能会发生的动作，那么我们不仅仅可以写出健壮性很好的结构体声明，我们同样可以写出非常紧凑的代码出来。

```
typedef struct taginfo
{
    t_byte      elem_1;
    t_dword     elem_2;
    t_byte      elem_3;
    t_word      elem_4;
}t_info; //可能引发编译器 padding,导致占用过多的字节空间!

typedef struct taginfo
{
    t_byte      elem_1;
    t_byte      elem_3;
    t_word      elem4;
    t_dword     elem2;
}t_info; //紧凑的数据声明方式
```

也许有些好奇的朋友可能就会问了，既然编译器

```
typedef struct taginfo
```

3. 了解 C 语言后面的汇编动作

3.1 static 变量和局部变量的差异性

在了解这个问题之前，让我们首先来了解一下变量在内存中的空间位置。

BSS, 未初使化的全局变量区域，一般是清除为 0，但是依赖于加载器/操作系统的行为。

DATA,放置被严格初使化的全局变量

C 语言的教科书上是这么说的，函数内部声明的 STATIC 变量是仅仅局限于函数内部使用的，在作用域之外是不能使用的。这样一个直接好处就是可以用来声明一些仅仅供自己使用的全局变量，以防止其他文件中异常修改同名的全局变量。但是在我们了解内部分布之后，我们才能知道，这种访问阻止仅仅局限于编译器在语法编译的阶段。

3.2 为什么这样的代码是低效的？

```
1    void demo_func_1(void)
2    {
3        char text1 = 0xAA;
           mov byte    ptr[SP + 24], 0xAA; /* char text1 = 0xAA */
4        long value[2][2] = {{0x34,0x2323}, {0x10, 0x45}};
           mov dword ptr[SP + 20], 0x45;/* long value[2][3] = ……. */
           mov dword ptr[SP + 16], 0x23;
           mov dword ptr[SP + 8], 0x3232;
           mov dword ptr[SP + 0], 0x34;
5        value[0][1] = 0;
           mov dword ptr[SP + 4], 0;
6          ……
7    }
```

由于数组的分配是在栈空间上分配出来的，所以每一次在函数执行的时候都必须重新执行一遍将数组中内容进行初使化的动作，这个动作是在编译阶段就产生出来了，因此相对而言效率是比较低的，但是如果数组在函数执行的过程中是不发生改变的话，那么我们应该尽量的将函数声明成全局的或者是局部静态的，这样的话，就根本不存在对函数数组内容进行初使化的代码语句，数组中的值是在编译阶段就被放入到全局数据域 DATA 中的__value 对象内了，然后仅仅产生一个引用指针地址就可以了,这样就节约了每一次进入函数时都必须对数组执行初使化动作的时间。

```
1    void demo_func_1(void)
2    {
3        char text1 = 0xAA;
           mov byte    ptr[SP + 0], 0xAA; /* char text1 = 0xAA */
4        static long value[2][2] = {{0x34,0x2323}, {0x10, 0x45}};
5        value[0][1] = 0;
           mov dword ptr[__value + 4], 0;
6          ……
7    }
```

全局 DATA 域

……..

__value dcd 0x34, 0x2323, 0x3232, 0x10, 0x23, 0x45;

…….

2. 注意编译器的差异性

第四章 代码的调试+++++++

2. DEBUG & RELEASE 的差别

3.局部变量的分配

4．堆栈的布局 以及 手工堆栈恢复（ＡＲＭ ＆ Ｘ８６）

5．了解 C 语言后面的动作

就像我们在开篇所说的那样，和底层的汇编语言相对比，C 语言是一种高级语言，这就意味着使用 C 语言可以使我们从烦琐的寄存器分配事务中解脱出来从而将更多的精力集中于软件设计。但是这绝对不等同于说我们从此可以不用再去了解汇编了，恰恰相反，对于那些有志于在技术专家路上发展的朋友们来说，了解每一条 C 语言后面机器到底干了些什么是非常重要的，因为只有了解了 CPU 在后面究竟做了些什么，我们才能够真正了解导致那些看起来似乎非常神奇的问题的原因究竟是什么，为什么结果会是这样的？

5.1 程序为什么会死机？

```
1    void demo_func_1(void)
2    {
3        t_char * str = “This is code world”;
4        sprintf(str, “%s”, “Hello”);
5        printf(“%s”, str);
6    }
7    void demo_func_2(void)
8    {
9        t_char str[ ] = “This is code world”;
10       sprintf(str, “%s”, “Hello”);
11       printf(“%s”, str);
12   }
```

13 C5-1 为什么一个会导致程序死机，一个不会呢？

作为一个比较有意思的案例，C5-1 可以用来充分说明很多东西。我们想问朋友们两个问题，第一个是 C5-1 中一共有 demo_func_1 和 demo_func_2 两个函数，哪一个在运行之后会出错，哪一个不会呢？第二个问题是为什么？可能有些聪明的朋友们知道了第一个问题的答案是 demo_func_1 会出错，但是 demo_func_2 函数不会， 但是对于第二个问题也许就不是能很容易的给出准确的答复出来了。

严格一点说起来，其实第一个问题的答案应该是”demo_func_1 非常容易引发错误，而 demo_func_2 函数不会出错”。为什么不是说“一定会出错”而是说“非常容易引发错误”呢？有些性急的朋友也许会说“我刚刚在 VC6 上跑过了，每次都会跳出来一个 Access Violation 的错误提示窗口来啊，你怎么能说是非常容易引发错误呢？照你的意思理解，demo_func_1 有的时候还不会出错了？那我怎么每次都会出错呢？“

现在让我们来仔细的分析一下

第五章 软件技术讨论

在这一个章节，我们将着重和大家一起来探讨一些 C 语言教科书中不曾涉及的一些特殊的软件技术。

5.1 堆栈溢出的检测

堆栈溢出的问题简直可以说是程序员的噩梦。为什么这么说呢？因为堆栈溢出问题和其他的程序问题比较起来，追查起来是很困难的，尤其是在那些没有硬件内存保护机制的的处理器芯片上(比如没有 MMU 的 8086 或是 ARM7)，这种问题显得异常困难。因为在这种芯片上，内存空间资源是供所有的程序使用的，也就是说如果某一个带有恶意的程序被执行，可能会因此把整个内存空间的数据全部写坏（当然也包括操作系统的核心代码和数据空间），从而使整个系统死机。后来人们意识到这个问题的严重性，在 CPU 硬件设计上引入了内存管理单元(MMU)，引入虚拟内存的机制，将各个进程之间的内存空间隔离开来，只允许每一个进程操作完全属于自己的内存空间，如果某个进程逻辑出现错误从而试图去操作其他进程的空间乃至于整个系统的核心空间，将会立即触发一个硬件异常，由于硬件异常是由操作系统来接管处理的，从而导致操作系统的介入，操作系统将中止这个出错进程的继续运行，从而保证整个系统的安全性不会因为某个进程的错误导致整个系统死机。这也就是为什么早期的 DOS 操作系统往往会因为某个程序运行有错会导致整个系统死机，而现在的 WINDOW2000 之类的操作系统却不会这样，我们可以在看见一个“Access Violation“之类的提示信息后，只需要点击鼠标将该出错程序中止即可，整个系统依然可以正常运行的原因所在。

但是无论 CPU 芯片是否能够提供硬件上的内存保护机制，做为程序员来说，首要的问题还是必须避免写出容易引发堆栈异常或是内存操作异常的程序代码。因为硬件仅仅是阻止了错误进程进一步的恶意行为，对于我们来说，更为重要的是学会如何在一开始写代码的时候就试图避免这样的逻辑错误，同时在问题发生的时候学会如何去追踪问题线索并找出问题原因所在。下面我们就来仔细分析一下堆栈溢出问题产生的原因。

5.1.1 堆栈溢出的基本分类及其症状表现

从前面章节的预备知识我们可以知道，局部变量是在堆栈空间上生成的。函数内部的局部变量空间的分配是在函数入口处实现的，当函数执行完毕后，所有的局部变量所占用的堆栈空间将会被回收。

5.1.2 全局堆栈溢出的检测

5.1.2 局部堆栈溢出的检测

5.2

参考文献

1 **Writing Solid Code Microsoft Techniques for developing Bug_free C Programs. Steve Maguire**

2 **Program Debugging Science**

后注：

本文属于一份总结性质的文章，其主要目的在于和大家一起分享各种代码实现过程中可能遇到的陷阱以及如何克服这些陷阱的方法。这么些年来一直在从事软件的开发工作，也遇到了很多良师益友并从他们身上汲取了很多很好的软件开发经验。同时也可以算是对我自己一段时期的一份总结吧。

Helianthus
2006.01.03

素材区域

语法自身的陷阱

1. 运算符号自身的优先级别(注意运算符的优先级，并用括号明确表达式的操作顺序，避免使用默认优先级）

Sum = input + sum >> 1;////????

2 讨论数据的存取原理，表明数据的数值是由其自身的类型来决定的；但是你不能声明一个 VOID 类型的数据变量但是你可以声明一个 VOID * 的？

3 尽量减少 return 的语句,使得函数的出口尽量只有一个(从维护的角度来讲)….

4 防止引用已经释放的内存空间

1. **引用已经被释放的空间，**

2. 引用堆栈的空间。

22. 关于可重入函数和自相关函数的概念＝＝＝代码的自加载技术

3 12 宏

[1]12-1：用宏定义表达式时，要使用完备的括号。

示例：如下定义的宏都存在一定的风险。

```
#define RECTANGLE_AREA( a, b ) a * b
#define RECTANGLE_AREA( a, b ) (a * b)
#define RECTANGLE_AREA( a, b ) (a) * (b)
```

正确的定义应为：

```
#define RECTANGLE_AREA( a, b ) ((a) * (b))
```

[1]12-2:将宏所定义的多条表达式放在大括号中。

示例：下面的语句只有宏的第一条表达式被执行。为了说明问题，for 语句的书写稍不符规范。

```
#define INTI_RECT_VALUE( a, b )\
    a = 0;\
    b = 0;

for (index = 0; index < RECT_TOTAL_NUM; index++)
    INTI_RECT_VALUE( rect.a, rect.b );
```

正确的用法应为：

```
#define INTI_RECT_VALUE( a, b )\
{\
    a = 0;\
    b = 0;\
}

for (index = 0; index < RECT_TOTAL_NUM; index++)
{
   INTI_RECT_VALUE( rect[index].a, rect[index].b );
```

```
}
```

¹12-3:使用宏时，不允许参数发生变化。

示例：如下用法可能导致错误。

```
#define SQUARE( a ) ((a) * (a))


int a = 5;
int b;
b = SQUARE( a++ ); // 结果：a = 7，即执行了两次增 1。
```

正确的用法是：

```
b = SQUARE( a );
a++; // 结果：a = 6，即只执行了一次增 1。
```

为用户提供良好的接口界面，使用户能较充分地了解系统内部运行状态及有关系统出错情况。

½6-1：防止将函数的参数作为工作变量。

说明：将函数的参数作为工作变量，有可能错误地改变参数内容，所以很危险。对必须改变的参数，最好先用局部变量代之，最后再将该局部变量的内容赋给该参数。

示例：下函数的实现不太好。(函数的局部化以及具体的阐述为什么这样做不好,从局部参数的分配来讨论，顺便讨论 BASIC 语言，是全局参数，因为是全局数据，所以比较难追踪值的变化，一旦出现值被修改，想满世界的去找出具体是哪一行代码修改掉的是非常困难的，因此软件技术的改进方法就是为这样的全局变量提供读写两个接口，这样就可以只需要控制读写函数这两个点就可以了)

```
void sum_data( unsigned int num, int *data, int *sum )
{
    unsigned int count;

    *sum = 0;
```

```
    for (count = 0; count < num; count++)
    {
        *sum  += data[count]; // sum 成了工作变量，不太好。
    }
}
```

[1]编写可重入函数时，若使用全局变量，则应通过关中断、信号量（即P、V操作）等手段对其加以保护。

说明：若对所使用的全局变量不加以保护，则此函数就不具有可重入性，即当多个进程调用此函数时，很有可能使有关全局变量变为不可知状态。

示例：假设 Exam 是 int 型全局变量，函数 Squre_Exam 返回 Exam 平方值。那么如下函数不具有可重入性。

```
unsigned int example( int para )
{
    unsigned int temp;

    Exam = para; // （**）
    temp = Square_Exam( );

    return temp;
}
```

不要在对实时性要求很高的函数类调用非常耗时的操作如 printf,malloc 之内的操作

1. 严禁使用未经初始化的变量(dangling)

2. 关于可重入函数和自相关函数的概念