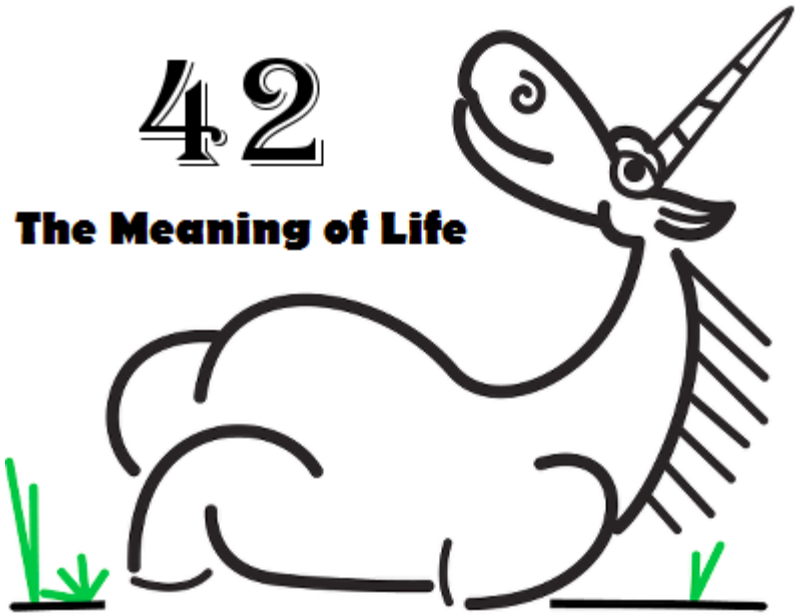
C++编程的 42 条建议

没错，你猜对了——答案是“42”。在这篇文章中，你就会看到关于 C++编程的 42 条建议，这些建议可以帮助程序员避免很多错误，节省时间和精力。本文的作者是 Andrey Karpov ——“程序验证系统”项目（Program Verification Systems，PVS）的技术总监,他们这个项目组主要是负责 PVS-Studio 静态代码分析器。检查过那么多开源项目的代码，我们看到过很多犯错的方式，所以我们有很多经验可以分享给读者。每一条建议我们都用一个例子来解释以证明问题的时效性。这些建议主要是面向 C/C++开发人员的，但是大多数是通用的，所以可能对其他语言的开发人员也有益。



前言

关于作者，我叫 Andrey Karpov。我的兴趣就是 C/C++以及改进代码分析的方法论。在 Visual C++的五年里，我获得了微软最具价值专家奖（Microsoft MVP)。我的这些文章和工作的首要目的就是让代码更安全。若这些建议能够让你避免一些常见的错误并写出更好的代码，我将不胜荣幸。那些为企业写代码规范的人也可从本文中得到一些有用的信息。

有个小背景。不久前，我创建了一个资源文件夹，在那里我分享了一些对 C++编程比较有用的技巧。但这个资源文件夹的订阅用户并没有我预料的那么多，所以我觉得没有在这里放链接的必要。这个文件夹应该会在网上放一段时间，但最终，会被删掉。然而，文件夹中的技巧依旧有价值。这也是为什么我要更新它们，增加了几条新的技巧，然后把它们整合在一个单独的文本里。阅读愉快。

1.别做编译器的活

来看一段选自 MySQL 项目代码块，这段代码包含有一个错误，在 PVS-Studio 代码分析器中表现为：V525 代码包含一系列相同的代码块。检查在 680，682,684,684,689,691,693,695 行的“0”，“1”，“2”，“3”，“4”，“1”，“6”项。

```
static int rr_cmp(uchar *a,uchar *b)
{
  if (a[0] != b[0])
    return (int) a[0] - (int) b[0];
  if (a[1] != b[1])
    return (int) a[1] - (int) b[1];
  if (a[2] != b[2])
    return (int) a[2] - (int) b[2];
  if (a[3] != b[3])
    return (int) a[3] - (int) b[3];
  if (a[4] != b[4])
    return (int) a[4] - (int) b[4];
  if (a[5] != b[5])
    return (int) a[1] - (int) b[5];     <<<<====
  if (a[6] != b[6])
    return (int) a[6] - (int) b[6];
  return (int) a[7] - (int) b[7];
}
```

解释

这是一个关于代码复制粘贴的典型错误。很显然，程序员复制代码“ if (a[1] != b[1])
return (int) a[1] - (int) b[1];”，然后开始改变下标，但是忘了将“1”改为“5”。结果导致在这个比较函数中会返回不正确的值。这个问题比较不容易注意到，而且很难被检测到，因为在我们使用 PVS-Studio 浏览 MySQL 之前所有的测死用力都没有检测出来。

正确代码

```
if (a[5] != b[5])
  return (int) a[5] - (int) b[5];
```

建议

尽管这段代码简洁、易读，但是开发人员还是比较容易忽视掉这个错误。在阅读类似这样的代码的时候你会比较难集中精力，因为你所看的都是相似的代码。

这些相似的语句块经常会导致一个结果，就是程序员会想要尽可能的优化代码。他手动“展开循环”。我认为在这里，循环展开并不是一个好法子。

首先，我会怀疑程序员能否真的从这里获得任何益处。现代的编译器都非常的智能，也非常的善于在可以提升代码能力的情况下自动展开循环。

其次，这段代码块的 bug 是因为他想要优化代码。如果你把它写成一段比较简单的循环的话，犯错的几率应该会小一点。

我会建议用下面的方式重写这个函数：

```
static int rr_cmp(uchar *a,uchar *b)
{
  for (size_t i = 0; i < 7; ++i)
  {
    if (a[i] != b[i])
      return a[i] - b[i];
```

```
  }
  return a[7] - b[7];
}
```

优点：

- 函数更易于阅读和理解
- 犯错的几率比较小

我敢肯定这个函数不会比原来那个版本运行慢。

所以，我的建议就是——写简单和易于理解的代码。具体来说，简单的代码就是正确的代码。不要尝试去做编译器的活——比方说，展开循环。如果展开会比较好的话，编译器会去做的。做这种手动优化的工作只有在一些比较重要的代码块那里才会有意义，而且这只有在 profiler 已经将这些代码块判断为有问题（慢）的时候才成立。

2.大于 0 并不意味着是 1

下面这个代码块选自 CoreCLR 项目。这个代码包含一个错误，在 PVS-Studio 分析器中诊断为：V698 表达式“memcmp(....) == -1”不正确。这个函数的返回值不是只有“-1”，可以是任意的负数值。可以考虑用“memcmp(....) < 0”来代替。

```
bool operator( )(const GUID& _Key1, const GUID& _Key2) const
  { return memcmp(&_Key1, &_Key2, sizeof(GUID)) == -1; }
```

解释

让我们来看一下 *memcmp()*函数的描述：

*int memcmp (const void * ptr1, const void * ptr2, size_t num);*

将 *ptr1* 指向的存储单元里的前 *num* 个字节和 *ptr2* 指向的存储单元里的前 *num* 个字节做比较。如果它们都相等就返回 *0*，如果不等，返回一个非 *0* 的数来表示大于。

返回值：

- *<0* —— 两个存储单元里的字节不相等，*ptr1* 里的值小于 *ptr2* 里的值（如果以无符号 *char* 值作为比较）
- *==0* ——两个存储单元里的内容相等
- *>0* ——两个存储单元里的字节不相等，*ptr1* 里的值大于 *ptr2* 里的值（如果以无符号 *char* 值作为比较）

注意到如果存储块里的东西不一样的话，函数的返回值会大于或小于 *0* 。大于或小于。这很重要。你不能将 *memcmp(), strcmp(), strncmp(),*等等这些函数的返回值与常数 *1* 或*-1* 进行比较。

有趣的是，这段将返回值和 *1/-1* 进行比较的错误代码，多年来也能返回程序员所期望的值。但只是运气而已，别无其他。函数的行为可能产生意想不到的变化。比如说，你可能换了一个编译器，或者开发人员会以一种新的方式优化 *memcmp()*函数，那你的代码就不起作用了。

正确代码

```
bool operator( )(const GUID& _Key1, const GUID& _Key2) const
  { return memcmp(&_Key1, &_Key2, sizeof(GUID)) < 0; }
```

建议

不要觉得函数现在起作用就好。如果文档说一个函数可以返回大于和小于 0 的值，那么它就真的会。它意味着这个函数可以返回-10 ，2 ，或 1024. 你经常见到它返回-1 ，0 ，1 并不意味着什么。

还有，函数能返回类似于 1024 的事实表明了 *memcmp()*的运行结果不能以 *char* 变量进行排序。这是一个比较多人会犯的一个错误，其后果很严重。这个错误就是 MySQL/MariaDB 5.1.61、5.2.11， 5.3.5 ，5.5.22 版本以前一些高危漏洞的根源。当用户连上 MySQL/MariaDB 的时候，代码会记录一个值（对 hash 和密码进行 SHA 安全哈希算法后的值），然后这个值将会用来和 memcmp() 函数的返回值进行比较。但是在一些平台下，返回值会超出 {-128…127}。结果就是，有 1/256 的几率在比较 hash 值与预期值的时候总返回 true，不管 hash 值是怎样的。因此，只要一条简单的命令行黑客就能无须密码黑进 MySQL 服务器。原因就是“sql/password.c”文件里的这部分代码：

```
typedef char my_bool;
...
my_bool check(...) {
  return memcmp(...);
}
```

关于这个问题更详细的描述可以看这里：Security vulnerability in MySQL/MariaDB.

3.一次复制，多次检查

下面这段代码块选自 Audacity 项目。PVS-Studio 检测到的错误是：V501 在“-”的左边和右边有相同的子表达式。

```
sampleCount VoiceKey::OnBackward (....) {
  ...
  int atrend = sgn(buffer[samplesleft - 2]-
                   buffer[samplesleft - 1]);
  int ztrend = sgn(buffer[samplesleft - WindowSizeInt-2]-
                   buffer[samplesleft - WindowSizeInt-2]);
  ...
}
```

解释

“buffer[samplesleft – WindowSizeInt-2]”这个表达式减去它本身。出现这个错误是因为在复制代码块(复制-粘贴)的时候，程序员忘了把 2 改为 1.

这真是一个很不起眼的错误，但它终究是个错误。这样的错误于程序员而言确实是一个残酷的事实，所以我们要在这里多讲几遍。我要向它们宣战。

正确代码

```
int ztrend = sgn(buffer[samplesleft - WindowSizeInt-2]-
```

```
                buffer[samplesleft - WindowSizeInt-1]);
```

建议

在复制代码的时候须谨慎小心。

让程序员拒绝使用复制粘贴是没意义的。因为它太方便，太有用了，我们怎能弃之不用。

所以，要谨慎小心，不要急——凡事欲则立。

记得复制代码可能会引起很多错误。大部分被诊断为 V501 错误的代码都是因复制粘贴引起的。

如果你要复制代码后编辑它，记得检查一下代码，不要偷懒。

我们稍后会详谈复制粘贴。我希望你能记住，这个问题比我们想象的要严重。

4.当心三元运算符“？：”，用括号把它括起来

下面这段代码块选自 Haiku 项目（BeOS 的继承者）。这段代码中的错误被 PVS-Studio 诊断为：V502,有可能“？：”不能返回你所期望的值。“？：”的优先级低于“-”。

```
bool IsVisible(bool ancestorsVisible) const
{
  int16 showLevel = BView::Private(view).ShowLevel();
  return (showLevel - (ancestorsVisible) ? 0 : 1) <= 0;
}
```

解释

让我们来看一下 C/C++ 运算符的优先级.三元运算符“？：”的优先级非常低，比/，+，<这些运算符的优先级要低，甚至低于减号运算。所以，刚刚那段代码的返回值可能不能如程序员预期的那样。

程序员认为那些操作会以如下顺序执行：

```
(showLevel - (ancestorsVisible ? 0 : 1) ) <= 0
```

但事实上，它的顺序是这样的：

```
((showLevel - ancestorsVisible) ? 0 : 1) <= 0
```

在这样简单的代码发生这样的错误，也说明了“？：”有多么难搞。在使用它的时候很容易出错，尤其是当这个三元运算符的判断条件比较复杂的时候，它可能就是代码中唯一的错误。而且不是你会出错这么简单，这样的表达式也比较不易读。

真的，要当心“？：”运算。我看到过蛮多在使用这个运算符的时候出现的错误。

正确代码

```
return showLevel - (ancestorsVisible ? 0 : 1) <= 0;
```

建议

在之前的文章中，我们已经讨论过关于三元运算符所引起的错误，但自从那以后我就变得更多疑了。上面给的那个例子说明了在使用三元运算符时是多么容易出错，就算在简短简单的表达式中也不例外，这也是为什么我要修改我之前的那些技巧。

我不建议完全不用“？：”。有时它也很有用很必要的。然而，请勿滥用。如果你已经决定要使用它，那我的建议就是：要加括号。

假设你有这样一个表达式：
A = B ? 10 : 20;

那我建议你这样写：
A = (B ? 10 : 20);

好吧，括号在这里有点多余……

但是， 当以后你或者你的同事要重构函数时，比如说在 10 或 20 那里加一个变量 x：
A = X + (B ? 10 : 20);

没有括号，你可能会忘记“？：”的优先级非常低，然后就会出错哦。

当然啦，你也可以把“x+”写在括号里，然后也会导致同样的错误。虽然看上去好像有保护到它，但并没有。

5.使用可利用的工具去分析你的代码

下面的代码来自 LibreOffice 项目。其错误在 PVS-Studio 中诊断为：V718 “Create Thread”不能被“DllMain”调用。

```
BOOL WINAPI DllMain( HINSTANCE hinstDLL,
                     DWORD fdwReason, LPVOID lpvReserved )
{
  ....
  CreateThread( NULL, 0, ParentMonitorThreadProc,
                (LPVOID)dwParentProcessId, 0, &dwThreadId );
  ....
}
```

解释
过去有段比较长的时间我都有在兼职网站找兼职做。有次，我没能完成我接的任务。任务本身就没表述正确，但是我那个时候并没有意识到。毕竟它乍一看挺简单清晰的。

在 DllMain 里的特定条件下，我要使用 Windows API 的函数来做一些操作，具体什么操作我忘了，但是应该不难。

我花了蛮多时间在上面，然而代码就是运行不了。更离奇的是，当我新做一个标准化应用的时候，它又运行得了，但在 DllMain 函数里面就是运行不了。好神奇，对不对？那个时候我没能找出问题的根源。

直到多年后的现在，我在 PVS-Studio 部门工作，我才突然意识到当年我代码运行不了的原因。在 DllMain 这个函数里，你所能做的动作是有限的。因为有些动态链接库还没有加载进来，那当然你就不可以调用它们啦。

现在，当程序员要在 DllMain 里删除一些危险操作的时候，我们就会有一个诊断程序来提醒程序员。也就是这个，来源于我当年的一个任务。

细节

更多关于 DllMain 的使用都可以在 MSDN 的这篇文章找到：Dynamic-Link Library Best Practices，在这里我放一些那篇文章的摘要：

在加载器锁被挂起的时候就会调用 DllMain。因此那些能调用 DllMain 的函数会有比较多的限制。结果就是，DllMain 被设计为只能使用小部分的微软的 Windows API 来执行很少的初始化任务。你不能在 DllMain 里直接或间接的调用任何函数来尝试申请加载器锁。不然，你的应用程序会有可能死锁或崩溃。DllMain 里的一个错误可能危及整个进程以及它的所有线程。

最理想的 DllMain 应该是一个空的桩代码（stub）。但是考虑到很多应用程序的复杂性，这样太受限了。所以使用 DllMain 的一条比较好的原则就是尽可能的延长初始化。晚点初始化能增强应用程序的鲁棒性，因为当加载器锁被挂起的时候，初始化无法完成。而且，晚点初始化能让你更安全地使用比较多的 Windows API。

一些初始化任务无法推迟。举个例子，如果布局文件（configuration file）出错了或者包含有垃圾，那么依赖于布局文件的动态链接库（DLL）就无法加载。对于这种类型的初始化，那些动态链接库应该尝试着去加载，如果加载失败就立马退出，而不是浪费其他资源去做其他事。

在 DllMain 里面你不应该做以下这些事：

- 调用 LoadLibrary 或者 LoadLibraryEx(不管是直接还是间接)。这样会引起死锁或崩溃。
- 调用 GetStringTypeA, GetStringTypeEx,或者 GetStringTypeW(不管是直接还是间接），这样会引起死锁或崩溃。
- 于其他进程同步。这样会引起死锁。
- 申请一个同步对象，而该产生该对象的代码正在等待申请加载器锁。这样会引起死锁。
- 在一定条件下用 CoInitializeEx 初始化 COM 进程。这个函数会调用 LoadLibraryEx.
- 调用注册函数。这些注册函数是由 Advapi32.dll 来实现的。如果 Advapi32.dll 没有在你的动态链接库之前初始化完成，那你的动态链接库可能就会访问非初始化的内存，然后引起进程崩溃。
- 调用 CreateProcess. 创建一个进程会加载其他动态链接库。
- 调用 ExitThread. 在动态链接库没加载的情况下退出一个进程需要再次申请加载器锁，然后就会引起死锁或者进程崩溃。
- 调用 CreateThread. 新建一个进程后就算你不把它与其他进程同步它可能也能运行，但是很危险。

- 创建一个命名 pipe 或者其他命名对象（仅针对 Windows 2000）。在 Windows 2000 中，命名对象由终端服务的动态链接库（Terminal Services DLL).如果这个动态链接库还没有初始化，调用它就会引起进程崩溃。
- 使用 C 语言的动态运行时库（CRT）里的内存管理函数。如果 CRT DLL 还没有初始化，调用这些函数就会引起进程崩溃。
- 调用 User32.dll 或 Gdi32.dll 里的函数。有些函数会加载其他 DLL，而这些 DLL 可能还没有初始化。
- 使用受监督代码

正确代码

上面那段引自 LibreOffice 项目的代码块有的时候能运行，有的时候不能运行。——全凭运气。

要解决类似这样的问题比较麻烦。你需要尽可能地重组你的代码以使 DllMain 函数更简单，更简短。

建议

很难给出建议。你不可能万事皆知，每个人都有可能遇到这种离奇的错误。所能给的建议无非是：对于你所负责的项目，要仔细阅读所有文档。但你也知道，我们无法预见所有的问题。当你把所有的时间都用来阅读文档，那就没有时间来编程了。还有，就算你已经看了 N 篇文章，你也无法确定你有没有漏掉那些告诉你如何应付其他问题的文章。

我希望我可以给你一些有用的小贴上，但很遗憾，我只能想到一个：使用静态分析器。不，这也不能保证你的代码没有任何 bug。如果多年前有一个代码分析器告诉我说，我不能再 DllMain 调用 foo 函数，我可能会节省很多时间和精力：无法完成那个任务真的让我蛮生气的，快要气炸了。

6.检查所有指针强制转换成 integer 类型的代码快

下面的代码选自 IPP Samples 项目。这里的错误被 PVS-Studio 诊断为：V205 强制转换指针类型为 32 位 integer 类型：（unsigned long）（img）

```
void write_output_image(...., const Ipp32f *img,
                        ...., const Ipp32s iStep) {
  ...
  img = (Ipp32f*)((unsigned long)(img) + iStep);
  ...
}
```

注意。有人会说因为一些原因，这段代码并不是最好的例子。我们并不关注为什么一个程序员会需要以这样奇怪的方式进入数据缓冲区。我们在意的仅仅是，指针被转换成了“unsigned long”类型。我选择这个例子只是因为它简洁。

解释

一个程序员想要把一个指针转换为特定数目的字节。这段代码在 win32 下能够正确运行，因为指针的大小正好等于 long 类型的大小。但是如果我们编译的是这段代码的 64 位版本，指针就会变成 64 位，将它强制转换成 long 类型会确实高位。

注意。Linux 使用不同的数据模型。在 *64* 位的 *Linux* 项目中，*"long"*类型也是 *64* 位。虽然如此，但用*"long"*来存储指针委实不是一个好主意。首先，这样的代码比较常用于 *Windows* 应用程序中，但是在 *Windows* 下面，它是出错的。其次，有专门用来存储指针类型的数据类型，比如说，*intptr_t.* 用这些类型可以让程序看上去更干净一些。

在上面的例子中，我们可以看到一个发生于 *64* 位程序中的典型错误。更直白一点说，在 *64* 位编程的道路上，程序员还会遇到很多其他的错误。但是把指针转换为 *32* 位 *integer* 变量是一个最常见也最隐秘的问题。

这个错误可以用下面的图来表示：

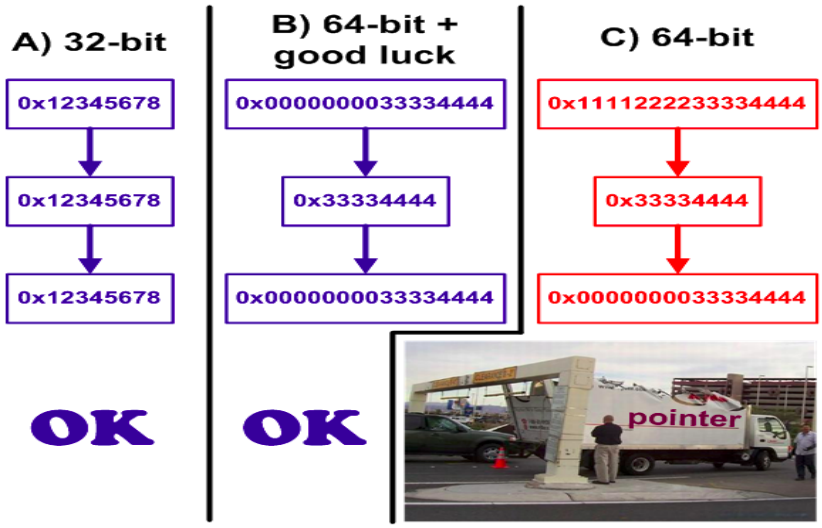


图 *1 A）32* 位程序 *B）*指向低位地址的 *64* 位指针。*C）*被破坏的 *64* 位指针。

讲一下它的隐秘性，这个错误有时候真的很难注意到。程序大多数都运行正确。在使用这个函数时，丢失指针里的重要位数这种事可能只会出现在少有的几个小时里。因为分配到的内存是来自低位地址的，也就是为什么所有的对象和数组都保存在前 *4G* 内存里。一切都很顺利。

当程序持续运行，内存变得四分五裂的，这样即使这个程序并没有用到那么多内存，新建的对象也可能会被创建在前 *4GB* 以外。然后，问题开始产生了。要重现这样的错误非常难。

正确代码

你可以使用 *size_t, INT_PTR, DWORD_PTR, intrptr_t,* 等等这些类型来存储指针。

```
img = (Ipp32f*)((uintptr_t)(img) + iStep);
```

事实上，我们可以不强制转换指针的。并没有见到在哪里有提到过说格式化不同于标准化的，也就是为什么使用*__declspec(align(#))*这样函数也没产生作用。所以，除非要转换成数字类型的指针能被 *Ipp32f* 整除，不然我们会触发一些没有定义过的行为。（详情看：*EXP36-C*）

所以，我们可以这样写：

*img += iStep / sizeof(*img);*

建议
使用特定的类型来存储指针——把 int 和 long 忘掉吧。最常用来存储指针的类型是：*intptr_t* 和 *uintptr_t*。在 *Visual C++* 里面，也可以用这些类型：*INT_PTR, UINT_PTR, LONG_PTR, ULONG_PTR, DWORD_PTR.* 从它们的名字你就可以看出来，用它们来存储指针很安全。

size_t 和 *ptrdiff_t* 也可以用来存储指针，但是我不建议用，因为它们原本的目的是用来存储大小和下标的。

你不能用 *uintptr_t* 类里的成员函数来存储指针。成员函数跟标准函数还是有些许不同的。除去指针本身，成员函数会隐藏掉指向其他类对象的 *this* 指针的值。然而，这也不重要——在 *32* 位程序中，你无法把这样一个指针赋值给 *unsigned int* 类型。这些指针一般会以一些特殊的方式处理的，这就是为什么 *64* 位程序里没那么多错误。至少，我没有看到这样的错误。

如果你打算把你的程序编译成 *64* 位，第一件事就是，你需要检查并修改所有把指针强制转换为 *32* 位 *integer* 类型的代码块。记住——可能程序里还是有其他错误，但是你应该从指针开始检查。

对于那些在开发或者打算开发 *64* 位应用程序的，我建议你们看一下这个材料：*Lessons on development of 64-bit C/C++ applications.*

7 别在循环里调用 alloca() 函数

在 Pixie 项目中发现了这个 bug。该错误被 PVS-Studio 诊断为：V505 在循环里使用了“alloca”函数。这样很快就会导致栈溢出。

```
inline  void  triangulatePolygon(....) {
  ...
  for (i=1;i<nloops;i++) {
    ...
    do {
      ...
      do {
        ...
        CTriVertex *snVertex =
          (CTriVertex *) alloca(2*sizeof(CTriVertex));
        ...
      } while(dVertex != loops[0]);
      ...
    } while(sVertex != loops[i]);
    ...
```

```
  }
  ...
}
```

解释

*alloca(size_t)*函数使用栈来分配内存。由 *alloca()* 分配到的内存要在函数执行完的时候才被回收。

分给程序的栈内存通常不多。当你在 *Visual C++*里新建一个项目的时候，你可以看到默认就分配 *1MB* 的栈内存，这也就是为什么如果 *alloca()*函数是在循环里的话，它很快就把可用的栈内存耗光了。

在上面那个例子中，一次就有三个循环，所以会造成栈溢出。

在循环里使用类似 *A2W* 的宏也不安全，因为它们中都有调用 *alloca()*的函数。

就像我们前面所说的，*Windows* 程序默认使用 *1MB* 的栈内存。我们可以修改这个值，在项目设置里找到并*“Stack Reserve Size”* 和*“Stack Commit Size”*的参数值。详情：*“/STACK (Stack Allocations)”*。 然而我们应该明白，扩大栈内存并不能解决这个问题——你只是推迟了项目栈溢出的时间。

建议

不要在循环里调用 alloca() 函数。如果你有一个循环，然后你又需要开辟一块临时缓冲区，用下面的 3 个法子中任意一个来解决：

1. 提前申请内存，然后使用一个缓冲区来进行所有操作。如果你每次需要的缓冲区大小都不一样，申请最大块的内存。如果不行（你不知道具体它会需要多少内存），使用法 2.
2. 把循环单独放在一个函数里。这样，缓冲区在每一次迭代的时候都会申请、释放。如果这样也不行，就只剩下法 3 了。
3. 用 malloc()或者其他操作来代替 alloca()，或者使用类似 *std::vector* 函数。但是这样分配内存会比较耗时间。如果使用 *malloc/new* 你就要考虑到它的释放。另一方面，这样你在用比较大的数给客户演示程序的时候不会发生栈溢出。

*8.*记住析构函数里的异常很危险

这个问题是在 *LibreOffice* 项目里发现的。在 *PVS-Studio* 中诊断为：*V509* 应该在 *try……catch* 里放‘*dynamic_cast<T&>*’操作，因为可能会产生异常。在析构函数里出现异常是非法的。、

```
virtual ~LazyFieldmarkDeleter()
{
  dynamic_cast<Fieldmark&>
    (*m_pFieldmark.get()).ReleaseDoc(m_pDoc);
}
```

解释
析构函数可以在异常发生后在清理栈内存的过程中调用，也可以在对象生命周期结束时被调用。如果在异常已经发生时调用析构函数，而在清理栈内存时又发生异常，异常之上又有异常，C++库只能调用 *terminate()*来终止程序。所以从中我们可以知道，不要在析构函数中抛出异常。析构函数中的异常要在同一个析构函数中处理。

上面引用的代码不只是危险那么简单。动态强制转换操作失败的话会生成一个名为“*std::bad_cast*”的异常。

类似的，任何可以抛出异常的构造函数也是危险的。比如说，在析够函数中用 *new* 操作来分配内存就不安全，因为如果分配不成功会抛出“*std::bad_alloc*”异常。

正确代码
要解决 *dynamic_cast* 可以不用引用类型而是用指针类型。这样，就算转换类型失败也不会抛出异常。

```
virtual ~LazyFieldmarkDeleter()
{
  auto p = dynamic_cast<Fieldmark*>m_pFieldmark.get();
  if (p)
    p->ReleaseDoc(m_pDoc);
}
```

建议

让析够函数尽可能的简单。析够函数不是用来分配内存和文件读取的。

当然，有时候我们无法做到让析够函数变得简单，但是我们要尽力达到这一点。此外，复杂的析够函数也说明类的设计不合理，以及这个方案不是最简便的方案。
你在析够函数里写的代码越多，它越难把问题解决好。因为这样，更难指出那个代码块可以或者不可以抛出异常。

如果有一定几率会发生异常的话，我们可以把它都最后写在 *catch(...)*里：

```
virtual ~LazyFieldmarkDeleter()
{
  try
  {
    dynamic_cast<Fieldmark&>
      (*m_pFieldmark.get()).ReleaseDoc(m_pDoc);
  }
  catch (...)
  {
    assert(false);
  }
}
```

没错，这样使用可能会掩盖住一些发生在析构函数里的错误，但这也有助于程序更稳定的运行。

我并不坚持说一定不要在构造函数里抛出异常——视情况而定吧。有时在析构函数里抛出异常会更有用。我也看到过一下专门的类中是可以抛出异常的，但这些情况很少，而且那些类就是被设计为可以在析够函数里抛出异常的。如果是比较常见的类如："own string","dot", "brush" "triangle", "document"等等，就不应该在析构函数里抛出异常。

只要记住，在一个异常之上又抛出异常会让程序终止。所以，一切都取决于你想不想让这种事发生在你的程序身上。

9.用'\0'来结束空字符

片段选自 Notepad++项目。其错误被 PVS-Studio 诊断为：V528 'char' 指针要跟'\0' 比较值很奇怪。 可能是想表达：*headerM != '\0'。

```
TCHAR headerM[headerSize] = TEXT("");
...
size_t Printer::doPrint(bool justDoIt)
{
  ...
  if (headerM != '\0')
  ...
}
```

解释

感谢这段代码的作者，使用'\0'来作为结束空字符，这让我们很容易定位和修正这个错误。作者大大做得好，但还不够好。

假设代码是这样写的：

```
if (headerM != 0)
```

数组地址和 `0` 比较，这其实没什么意义，因为比较结果总是 `true`。所以这是什么—一个错误或者多余的检查？很难说，尤其是如果这个代码是别人的代码或者说这个代码写了很久。

但既然程序员在这段代码中用了`'\0'`，我们就假设程序员是想要检查某一个字符的值。然后我们又知道，比较 *`headerM`* 指针和 *`NULL`* 没意义。所以我们认为作者是想知道那个字符串是否为空或者在写检测的时候没出错。要修改这段代码，我们需要增加一个指针间接引用操作。

正确代码

```
TCHAR headerM[headerSize] = TEXT("");
...
size_t Printer::doPrint(bool justDoIt)
{
  ...
  if (*headerM != _T('\0'))
  ...
```

```
}
```

建议
0 可以表示 NULL，false,空字符'\0'，或者仅仅是 0。所以请不要偷懒—避免在任一个案例中使用 0 作为简写的符号。

用下面的符号：

- 0—整数 0;
- nullptr—C++中的空指针;
- NULL—C 中的空指针
- '\0', L'\0', _T('\0'）—空结束
- 0.0, 0.0f—浮点类型的 0
- false，FALSE—’false‘值

用这种方法会让你的代码更干净，也让你和其他程序员在重新检查代码时更容易定位 bug。

10. 避免使用过多的小块#ifdef 代码

代码选自 CoreCLR 项目。其错误被 PVS-Studio 诊断为：V522 可能会发生空指针‘hp’的间接引用.

```
heap_segment* gc_heap::get_segment_for_loh (size_t size
#ifdef MULTIPLE_HEAPS
                                            , gc_heap* hp
#endif //MULTIPLE_HEAPS
                                            )
{
#ifndef MULTIPLE_HEAPS
    gc_heap* hp = 0;
#endif //MULTIPLE_HEAPS
    heap_segment* res = hp->get_segment (size, TRUE);
    if (res != 0)
    {
#ifdef MULTIPLE_HEAPS
        heap_segment_heap (res) = hp;
#endif //MULTIPLE_HEAPS
  ....
}
```

解释

我总觉得*#ifdef/#endif* 是有害的——一个无可避免的害处。不幸的是，它们是必须的，我们不得不使用它们。所以我不主张让你停止使用*#ifdef*，这么做没意义。但我想让你注意别“滥用”它。

我猜，你们中很多人都有见到过那种代码，满屏的*#ifdef*. 要看那种每十行就来一个*#ifdef*，或者更频繁的代码，真的很痛苦。这种代码是依赖于系统的，而且你不能不用*#ifdef*。尽管它的存在没有让你更快乐一些。

你看，要阅读上面那段代码是多么难！有些程序员的基本工作就是阅读代码。是的，你没看错。我们花了非常非常多的时间来检查和学习已经写好的代码，比花在写新代码的时间上要多得多。这也就是为什么阅读难读的代码会降低我们的效率，而且也比较容易出错。

回过头来看我们的代码，错误是在空指针间接引用的时候发现的，在没有声明 *MULTIPLE_HEAPS* 宏的时候发生的。为让你更容易理解，我们来展开那个宏：

```
heap_segment* gc_heap::get_segment_for_loh (size_t size)
{
  gc_heap* hp = 0;
  heap_segment* res = hp->get_segment (size, TRUE);
  ....
```

程序员声明了 *hp* 变量，初始化为 *null*，然后立刻就间接引用它了。如果此时 *MULTIPLE_HEAPS* 还没有定义，自然会出错。

正确代码

虽然我的同时已经在“*25 Suspicious Code Fragments in CoreCLR*”这篇文章中报告了这个错误，但是这个问题依然存在于 *CoreCLR*（*12.04.2016*）中。所以我也不知道解决这个错误的最好的法子是什么。

我的想法是，既然*(hp == nullptr)*, 那变量‘*res*’就应该也初始化为其他值——但是我不知道具体是什么值。所以这次我们什么也不用做。

建议

从你的代码中消除小块*#ifdef/#endif*——它们的存在真的让代码阅读与理解变得很困难。过多的 *ifdef* 更容易出现错误。

并没有适用于任何情况的建议——都得视具体情况而定。无论如何，记住，*ifdef* 是麻烦之源。所以你应该努力使你的代码尽可能的干净整洁。

Tip N1: 试着拒绝*#ifdef*

#ifdef 有时可以用常量和 *if* 代替。比较一下下面两段代码：一个宏的变体：

```
#define DO 1

#ifdef DO
static void foo1()
{
  zzz();
}
#endif //DO

void F()
{
#ifdef DO
  foo1();
#endif // DO
  foo2();
```

```
}
```

这段代码不好读，你不会愿意读的。我打赌，你直接略过了，有没有？ 现在来看下面的代码：

```
const bool DO = true;

static void foo1()
{
  if (!DO)
    return;
  zzz();
}

void F()
{
  foo1();
  foo2();
}
```

现在更好读一点了。有人会说这样的话代码效率就没那么好了，因为它要调用函数，然后还要在函数里检查它有没有定义过。我不同意这样的说法。首先，当代的编译器都非常智能，你非常有可能会在 *release* 版本看到这样的代码：没有任何外部检查和函数调用。其次，这一点点损失就没必要过分纠结了，好吗。代码的干净整洁更重要。

Tip N2 ：让你的*#ifdef* 代码块更大一点

如果我要写那个 *get_segment_for_loh()*函数，我不会在其中使用很多*#ifdef*，我会用两个版本的函数来代替。对，字可能会比较多一点，但是函数很容易阅读和编辑了。

再者，有人会说，这是复制代码，既然每个*#ifdef* 都有很多长度比较长的函数。每个函数都有两个版本容易让人在修改其中一个版本时忘记修改另一个版本。

嘿，等一下。为什么你的函数长度那么长？把所有的逻辑放到额外的备用函数——这样你两个版本的函数都会变得更段一些。我敢肯定这样能让你更容易找出它们之间的不同。

我知道这个 *Tip* 并不能包治百病，但也要考虑这么做。

Tip N3. 考虑使用模板（*templates*）——有时它们也能帮到你。

*Tip N4.*在使用*#ifdef* 之前花点时间来考虑：你可以不用它吗？或者，你可以少用它并且只在一个地方使用它吗？

11.不要把所有的操作运算都压缩到同一行

下面的代码选自 Godot Engine 项目。该错误被 PVS-Studio 诊断为：V567 未定义行为（undefined behavior）。在序列点（sequence point）中使用两次会修改变量“t”的值。

```
static real_t out(real_t t, real_t b, real_t c, real_t d)
{
  return c * ((t = t / d - 1) * t * t + 1) + b;
}
```

解释

有的时候，你会遇到那种代码，作者通过复杂的结构把所有的运算都压缩在一小段代码中。这样做并不会帮到编译器半分，但是在让代码更难懂，让其他程序员（甚至作者自己）更难理解上面确实起了不少作用。而且这种代码出错的概率也比较高。

就是这种代码，程序员想要只用短短几行来放大量代码，通常会出现有关未定义行为的错误。他们经常在一个序列点内读写同一个变量。为更好理解这个问题，我们需要讨论一下“无定义行为”和“序列点”。

未定义行为是一些编程语言的一个属性，表现为一个问题的结果取决于编译器的执行或者优化的选择。很多出现未定义行为的例子（包括我们现在讲的例子）大都和“序列点”相关。

序列点的定义是一个程序执行中的点，这个点的特殊性在于，在这个点之前语句产生的所有副作用都将生效，而后面语句的副作用还没有发生。在 C/C++中的序列点出现在下述位置：

- "&&", "||", ","。如果不过分使用，这些操作遵循从左至右的运算顺序
- 三元运算符“？：”
- 完整表达式结束处（通常以“;”为标记）；
- 函数调用时的函数入口点，但是是在参数初始花后
- 函数返回时，在返回值已经复制到调用上下文

注意。在新的 C++标准中已经摒弃了“序列点”这个概念，但是我们还是用上面的解释，因为这能让不熟悉这门语言的你们更容易更快理解一点。这个解释比新的解释更简单些，也更方便我们理解为什么不能把所有的操作都压缩在一起。

在一开始我们给出的例子中，并没有上面的所提到的序列点。但是，“=”，还有括号，不能这样处理。因此，我们我们不知道在计算这个式子的时候 t 的值是多少。

换句话说，这个表达式有一个序列点，所以编译器不知道以什么样的顺序来使用 t。比如说，“t*t”可以在“t=t/d-1”之前或之后进行。

正确代码

```
static real_t out(real_t t, real_t b, real_t c, real_t d)
{
  t = t / d - 1;
  return c * (t * t * t + 1) + b;
}
```

建议

很明显，把所有的操作都放在一行代码中并不明智。除了难读，还容易放错。

把表达式分成两部分我们能同时解决两个问题——让代码更易读，和避免了由于增加序列点带来的未定义行为。

上面讨论的代码并不是出错的唯一例子。这里是另外一个：

```
*(mem+addr++) =
   (opcode >= BENCHOPCODES) ? 0x00 : ((addr >> 4)+1) << 4;
```

跟前面的例子一样，这段复杂难懂的代码也犯了类似的错误。程序员打算把 addr 的自增操作放在一个会导致未定义行为的表达式中，因为编译器不知道在表达式右边，addr 会取何值——它原来的值还是自增后的值。

对于这个问题最后的解决方案跟前面那个一样——不要没缘由地做那么复杂的操作。把操作分成几部分，而不是把它们都放在同一行里：

```
*(mem+addr) = (opcode >= BENCHOPCODES) ? 0x00 : ((addr >> 4)+1) <<
4;
addr++
```

从此处我们可以得到一个简单而有用的结论——不要尝试着把一系列的操作用短短几行来完成。把代码分成几部分会更好，这样不但易于理解也减少了犯错的几率。

下次当你打算写一个复杂的结构的时候，停下来一会，然后想想，你这样做会付出什么代价，而你，是否已经准备好付出这个代价。

12.在复制粘贴的时候，要特别留心最后几行

这个 bug 是在 Source SDK 中发现的。该错误被 PVS-Studio 诊断为：V525 代码包含了相同的语句块。检查'SetX', 'SetY', 'SetZ', 'SetZ'这几项。

```
inline void SetX( float val );
inline void SetY( float val );
inline void SetZ( float val );
inline void SetW( float val );

inline void Init( float ix=0, float iy=0,
                  float iz=0, float iw = 0 )
{
  SetX( ix );
  SetY( iy );
  SetZ( iz );
  SetZ( iw );
}
```

解释

我敢肯定，这段代码是复制粘贴的。复制第一行，然后修改特定的字母。最后，复制粘贴没能让程序员如愿：他的注意力被减弱，而且他最后忘了把‘Z’改为‘W’。

在这个例子中，我们不关注作者犯错了这个事实，我们关心的是，这种错误是由于一系列单调的动作引起的。

我非常建议阅读“最后一行效应”这篇文章。由于公共利益，这篇文章的科学版本已经出版。

简单来说，但使用复制-粘贴术来复制代码的时候，你在最后几行出错的可能性非常高。这不是我的猜测，有统计数据的。

正确代码

```
{
  SetX( ix );
  SetY( iy );
  SetZ( iz );
  SetW( iw );
}
```

建议

我希望你以阅读了上面我所提到的文章。所以，再一次，我们要解决接下来的问题。当要写差不多的代码块，程序员会复制粘贴，然后做些轻微的修改。这么做的时候，他们常常会忘了修改特定的单词或字母，而由于注意力被削弱，这种事一般会出现在最后几行。

为减少出错的次数，这里有些小贴士：

- 把你那些看起来相似的代码放在一个“表格”里：这样错误应该会更突出。我们会在下一节讨论“表格”的布局。可能在这个例子里表格布局不会有多大用处，但在编程中它依旧是个很有用的东西。
- 在复制粘贴的时候要谨慎小心，注意力集中。集中精神，再次检查你的代码——尤其是最后几行。
- 现在你已经知道了最后几行效应，把它记在心上，顺便告诉你的同事。知道错误是怎样产生的，将有助于你避开它们。
- 把“最后一行效应”这篇文章分享给你的同事。

13.设计表格风格的格式

下面的代码来自 ReactOS 项目（兼容于 Windows 的开源操作系统）。该错误被 PVS-Studio 诊断为：V560 部分条件表达式的值总为 true：10035L

```
void adns__querysend_tcp(adns_query qu, struct timeval now) {
  ...
  if (!(errno == EAGAIN || EWOULDBLOCK ||
        errno == EINTR || errno == ENOSPC ||
        errno == ENOBUFS || errno == ENOMEM)) {
  ...
}
```

解释

上面给的代码很少，你应该很容易定位其错误。但是在现实世界的代码，bug 都很难注意到。当阅读这样的代码的时候，你会无意识的跳过有类似比较的代码块，直接到下一片段。

这种错误的出现与判断条件的无格式有关。你不会愿意在其上花费太多注意力，因为需要精力，而且我们会想，既然判断条件都差不多，应该不会出错吧，一切应该都好好的吧。

一种能解决这种错误就是，把代码表格化。

如果你很懒，不想去找出上面那段代码的错误。我可以告诉你，有一个判断条件少了“errno==”，这样就会导致条件永远为真，因为 EWOULDBLOCK 不等于 0.

正确代码

```
if (!(errno == EAGAIN || errno == EWOULDBLOCK ||
      errno == EINTR || errno == ENOSPC ||
      errno == ENOBUFS || errno == ENOMEM)) {
```

建议

首先，让我们来看一段最简单的表格化代码。其实我也不喜欢这样。

```
if (!(errno == EAGAIN  || EWOULDBLOCK     ||
      errno == EINTR   || errno == ENOSPC ||
      errno == ENOBUFS || errno == ENOMEM)) {
```

现在看起来好多了，但是还不够好。

我不喜欢这种布局有两种原因。首先，错误还是不够明显; 其次，为了对齐，要插入好多空格。

这就是为什么我们需要对这种格式化风格进行两点改进。第一，一行不要超过一个比较：这样容易发现错误。比如：

```
a == 1 &&
b == 2 &&
c      &&
d == 3 &&
```

第二个改进就是用更合理的方式写&&, ||这些运算符。比如，写在左边而非右边。

看一下用空格来对齐有多么诡异：

```
x == a          &&
y == bbbbb      &&
z == ccccccccccc &&
```

把运算符写在左边的话，编程会更快更简单的：

```
   x == a
&& y == bbbbb
&& z == ccccccccccc
```

让我们把这两种改进运用到刚开始的代码块里：

```
if (!(   errno == EAGAIN
      || EWOULDBLOCK
      || errno == EINTR
      || errno == ENOSPC
      || errno == ENOBUFS
      || errno == ENOMEM)) {
```

是，这样更长了——但是错误也更明显了。

我同意，这看起来很诡异，然而我还是很推荐用这种法子。我用这个法子已经半年了，觉得不错。所以我自信这个建议能帮到你。

我并不认为代码变得更长是个问题。我曾经用这样写过：

```
const bool error =    errno == EAGAIN
                   || errno == EWOULDBLOCK
                   || errno == EINTR
                   || errno == ENOSPC
                   || errno == ENOBUFS
                   || errno == ENOMEM;
if (!error) {
```

代码太长太密集了，很失望？我同意，所以我们把它写成一个函数吧！

```
static bool IsInterestingError(int errno)
{
  return    errno == EAGAIN
         || errno == EWOULDBLOCK
         || errno == EINTR
         || errno == ENOSPC
         || errno == ENOBUFS
         || errno == ENOMEM;
}
....
if (!IsInterestingError(errno)) {
```

你可能会觉得我有点夸张，太过完美主义。但我向你保证，在复杂的表达式中，这种错误很常见。如果不是它们如此常见，我才不会提出来呢。它们随处可见，并且难以发现。

下面是另外一个例子，来自 WinDjView：

```
inline bool IsValidChar(int c)
{
  return c == 0x9 || 0xA || c == 0xD ||
         c >= 0x20 && c <= 0xD7FF ||
         c >= 0xE000 && c <= 0xFFFD ||
         c >= 0x10000 && c <= 0x10FFFF;
}
```

这个函数虽然只有几行，但是仍然有个错误。函数返回值永远为 true。从长远来看，这种错误跟格式混乱，以及程序员坚持使用这种代码多年，不愿去认真阅读它有很大关系。

让我们重构这段代码，将其表格化。我会增加一些括号：

```
inline bool IsValidChar(int c)
{
  return
       c == 0x9
    || 0xA
    || c == 0xD
    || (c >= 0x20    && c <= 0xD7FF)
    || (c >= 0xE000  && c <= 0xFFFD)
    || (c >= 0x10000 && c <= 0x10FFFF);
}
```

你无须用我说的这种方法来设计你的代码。这一条建议的目的是让你注意因书写混乱引起的错误。通过将代码表格化，你可以避免很多细微的拼写错误，这就够了。所以，我希望这条建议有帮到你。

注意

坦白说，我不得不提醒你，表格化有时也有害。来看这个例子：

```
inline
void elxLuminocity(const PixelRGBi& iPixel,
                   LuminanceCell< PixelRGBi >& oCell)
{
  oCell._luminance = 2220*iPixel._red +
                     7067*iPixel._blue +
                     0713*iPixel._green;
  oCell._pixel = iPixel;
}
```

代码来自 eLynx SDK。程序员想要对齐代码，所以他在 713 前面加了个 0. 不幸的是，他忘记了，0 在首位表示该数字是八进制。

字符串数组

我希望，我有把代码格式化这个建议表达清楚。但是我还想在给几个例子。让我们再来看几个例子。在这里，我要说的是，表格化不仅仅可以用于判断条件，也可以用在其他结构中。

下面的代码块来自 Asterisk 项目。其错误被 PVS-Studio 诊断为：V653 一个含有两部分的可疑字符串被用来数组初始化。可能是少了逗号。检查："KW_INCLUDES" "KW_JUMP".

```
static char *token_equivs1[] =
{
  ....
  "KW_IF",
  "KW_IGNOREPAT",
  "KW_INCLUDES"
  "KW_JUMP",
  "KW_MACRO",
  "KW_PATTERN",
  ....
};
```

这里有个拼写错误——少了一个逗号。结果就是两个完全不同含义的字符串被合成一个了。这样，我们其实是用它来初始化了：

```
....
  "KW_INCLUDESKW_JUMP",
  ....
```

若是程序员用表格化，应能避免此错。那样，如果少了逗号，很容易就能定位到。

```
static char *token_equivs1[] =
{
  ....
  "KW_IF"         ,
  "KW_IGNOREPAT" ,
  "KW_INCLUDES"  ,
  "KW_JUMP"       ,
  "KW_MACRO"      ,
  "KW_PATTERN"    ,
  ....
};
```

注意，就像上次一样，如果我们把分割符（在这个例子中是逗号）放在右边，我们要用到很多空格，这样很不方便。如果有一行/词组很长要加空格是非常不方便的：所以我们需要重新设计整个表格。

这就是我为什么要再次建议用接下来的法子来表格化：

```
static char *token_equivs1[] =
{
  ....
  , "KW_IF"
  , "KW_IGNOREPAT"
  , "KW_INCLUDES"
  , "KW_JUMP"
  , "KW_MACRO"
  , "KW_PATTERN"
  ....
};
```

现在就很容易定位到缺失逗号的地方，而且无需使用过多的空格——这样的代码漂亮又易懂。可能这样的格式化并不常见，但你很快就会 适应的——试一下吧。

最后，我的座右铭就是：一般说来，漂亮的代码往往就是正确的代码。

14. 好的编译器和编程风格还不够

我们已经将了好的编程风格，但这次，我们来看个反面例子。对于写出好代码来说，好的编程风格还不够：仍然会有各种错误，好的编程风格并非包治百病。

下面的代码片段来自 PostgreSQL. 其错误被 PVS-Studio 诊断为：V575 'memcmp'函数要处理‘0’元素。检查第三个参数。

Cppcheck 分析器也检测到了同样的错误。它给出一个警告：memcmp() 第三个参数无效。这里需要一个非布尔类型的值。

```
Datum pg_stat_get_activity(PG_FUNCTION_ARGS)
{
```

```
  ....
  if (memcmp(&(beentry->st_clientaddr), &zero_clientaddr,
             sizeof(zero_clientaddr) == 0))
  ....
}
```

解释

反括号放错了地方。这仅仅是一个拼写错误，但不幸的是，它完全改变了代码的意思。

sizeof(zero_clientaddr) == 0 这个表达式永远为‘false’，因为任何对象的大小都大于 0。false 又等于 0,这样就导致 *memcmp()*要比较 *0* 字节。如此，函数就会认为数组相等然后返回 *0*.这就意味着上面的代码可以化简为 *if (false)*。

正确代码

```
if (memcmp(&(beentry->st_clientaddr), &zero_clientaddr,
           sizeof(zero_clientaddr)) == 0)
```

建议

这就是一个我无法给出任何安全编程技巧来避免拼写错误的例子。我能想到的唯一一个就是"尤达条件"，把常数写在比较运算符的左边：

```
if (0 == memcmp(&(beentry->st_clientaddr), &zero_clientaddr,
            sizeof(zero_clientaddr)))
```

但我不建议这种写法。我不喜欢也不用它出于两个原因：

第一，这样判断条件的可读性变差。我不知道要怎样表达，但是这种风格以尤达命名并非没有理由。

第二，这样对于解决把括号放在错误的地方并没有什么帮助。你出错的方式有千万种。下面的例子就是尤达条件无法阻止括号放错的：

```
if (0 == LoadStringW(hDllInstance, IDS_UNKNOWN_ERROR,
    UnknownError,
    sizeof(UnknownError) / sizeof(UnknownError[0] -
    20)))
```

上面的代码片段来自 ReactOS 项目。它的错误很难发现，所以让我帮你指出来：
sizeof(UnknownError[0] - 20)

所以尤达条件在这里没用。

我们可以创造一些人为的风格来确保每一个反括号都很其正括号匹配。但这样会使代码很笨重、难看，而且没有人会愿意那样写。

所以，再说一遍，我无法推荐任何一种编程风格来避免反括号放在错误的地方。

而这也就是编译器派上用场的地方，编译器应该会提醒我们这个奇怪的结构的，对不对？额，它应该，但它没有。我运行 Visual Studio 2015,详述了/Wall switch...然后没有收到任何提示。但我们不能因此责备编译器，事实上，它有很多工作要做。

从这里我们得出的最重要的结论就是：好的编程风格和编译器（我确实喜欢 VS2015 编译器）并不总能解决问题。有有时会听到这样的言论：“你只需要把编译器的提醒设到最高级，然后使用好的编程风格，那么一切都会好的。”不，并非如此。我不是说有些程序员不善于编程，只是，每个程序员都会出错。每一个，没有例外。在编译器和好的变成风格下面还是会有很多拼写错误。

所以好的编程风格+编译器提醒这个组合很重要，但是还不够。这也是为什么我们需要使用多种法子来搜寻 bug。并没有什么杀手锏，高质量的代码只能通过使用多种搜索 bug 的技术结合来得到。

我们这里讨论的错误可以用下面的法子来找到：

- 检查代码
- 单元测试
- 手动测试
- 静态代码分析
- 等等

我想你已经猜到了，我个人从对静态代码分析方法最感兴趣。顺便说一句，静态代码分析是解决这种特殊问题最适当的方法。因为它能在最早检测到错误，比如说，就在代码写完后。

事实上，这种错误用类似 Cppcheck 或 PVS-Studio 的工具很快就能找到。

结论 有些人没有足够的技巧来避免错误。每个人都会出错——无可避免。即使是大师级的人物，也会时不时犯些拼写错误。那么既然它无可避免，责备程序员或者编译器，编程风格都没有意义。这样并没有任何帮助。所以，我们要使用多种软件的组合来提高技术。

15. 若可以，开始在你的代码中使用 enum class

所有关于这个错误的代码都很长。我已经选了最短的那一个了，但还是比较长，非常抱歉。

这个 bug 是在 Source SDK 库中发现的。该错误被 PVS-Studio 诊断为：V556 比较不同枚举类型的值：Reason == PUNTED_BY_CANNON.

```
enum PhysGunPickup_t
{
  PICKED_UP_BY_CANNON,
  PUNTED_BY_CANNON,
  PICKED_UP_BY_PLAYER,
};

enum PhysGunDrop_t
{
  DROPPED_BY_PLAYER,
  THROWN_BY_PLAYER,
  DROPPED_BY_CANNON,
  LAUNCHED_BY_CANNON,
};

void CBreakableProp::OnPhysGunDrop(...., PhysGunDrop_t Reason)
{
  ....
  if( Reason == PUNTED_BY_CANNON )
  {
    PlayPuntSound();
  }
  ....
}
```

解释

Reason 是枚举类型 *PhysGunDrop_t* 的一个变量。这个变量现在是跟一个属于另一个枚举类型的常量 *PUNTED_BY_CANNON* 在做比较，这样的比较很明显是逻辑错误。

这种 *bug* 非常常见。我在类似 *Clang, TortoiseGit, and Linux Kernel* 的项目里都有遇到过。

这种错误那么常见的原因就是，枚举在标准 C++里并非拼写安全的。你很容易对哪个应该跟哪个比较感到困惑。

正确代码

我不知道这段代码的正确版本应该是怎么样的。我猜，在 PUNTED_BY_CANNON 那个位置的应该是 DROPPED_BY_CANNON 或者 LAUNCHED_BY_CANNON。就当是 LAUNCHED_BY_CANNON 吧。

```
if( Reason == LAUNCHED_BY_CANNON )
{
  PlayPuntSound();
}
```

建议

如果你用 C++编程，你很幸运。我建议你开始使用 enum class，而且编译器不会让你比较两个不同枚举类型的值。你不会再拿英镑和便士做比较了。

我对有些 C++的创新不太信任。比如说，auto 关键字。我坚信过度使用它会有害。这是我对它的看法：程序员看代码比写代码的时间要多，所以我们要确保，编程文档要易读。在 C 中，变量要在函数的前面声明，所以当你写到中间或者后边的时候，总是不容易指出一些变量到底指什么。这也是为什么有那么多命名方式。比如说，前置命名，pfAlice 可能代表“浮点类型指针(pointer to float)”。

在 C++中，你可以在任何你需要的时候声明变量，这种方式也被认为是一种好的风格。使用前缀或者后缀命名法都不再流行。此时，auto 关键词出现了，导致了程序员开始使用大量类似“auto Alice = Foo();”的诡异结构。

很抱歉，离题了。我就是想告诉你，有些新特性既好又坏。但 enum class 不是这样的:我坚信 enum class 只好不坏。

在使用 enum class 的时候，你要明确指出哪个常量是属于哪个枚举类的。这有助于代码出现新的错误。然后，代码会长这样：

```
enum class PhysGunDrop_t
{
  DROPPED_BY_PLAYER,
  THROWN_BY_PLAYER,
  DROPPED_BY_CANNON,
  LAUNCHED_BY_CANNON,
};

void CBreakableProp::OnPhysGunDrop(...., PhysGunDrop_t Reason)
{
  ....
  if( Reason == PhysGunDrop_t::LAUNCHED_BY_CANNON )
  {
    PlayPuntSound();
  }
  ....
}
```

没错，修改旧的代码会有一定的困难。但我强烈建议你从今天起，在新的代码中开始使用 enum class。你的项目会从中受益的。

我不觉得我要在这里介绍 enum class 的必要。下面的几条链接可以帮助你学到 C++11 这个完美的新特性。

4. 维基百科。C++11.Strongly typed enumerations
5. Cppreference。Enumeration declaration.
6. StackOverflow. Why is enum class preferred over plain enum?

16 .“看我能做什么！”——编程中无法接受的

这一部分会和“不要把所有的操作运算都压缩到同一行”有些许相似，但这一次我想要把焦点放在另一个地方。有时候我会觉得程序员好像在跟某个人较劲，想要写最短的代码。

我不是要讲复杂的模板。我们要讨论的是一个不同的课题，因为很难讲清楚模板在哪里有害，又在哪里有益。现在我要提及一个更简单的情况，和 C 与 C++程序员有关。他们会倾向于写很复杂的结构，想着，“我这么做是因为我能做到”。

下面的代码来自 KDE4 项目。其错误被 PVS-Studio 诊断为：V593 检查类似'A = B == C'的表达式。这种表达式是按'A = (B == C)'的顺序运算的。

```
void LDAPProtocol::del( const KUrl &_url, bool )
{
  ....
  if ( (id = mOp.del( usrc.dn() ) == -1) ) {
    LDAPErr();
    return;
  }
  ret = mOp.waitForResult( id, -1 );
  ....
}
```

解释

看完这段代码，我总有这样一个问题：这么做有什么意义呢？你想要节省行数？你想展示说，你可以把几个动作放在同一个表达式里？

结果我们得到一个 典型的错误模板——用类似 if (A = Foo() == Error)这种表达式。

比较运算的优先级大于赋值运算的优先级。这也是为什么"mOp.del(usrc.dn()) == -1"先执行，然后就只有 *true* (1) 或 *false* (0) 这两值可以赋给变量 id。

如果 mOp.del() 返回‘-1’，那函数就会终止。否则，函数会继续执行，但是变量 id 会被赋予一个错误值。所以它会总等于 0.

正确代码

我想要强调，增加额外的括号并非解决此问题的方法。是的，错误可以消除。但此法不通。

原代码有额外的括号——仔细看。很难看出他们想要做什么，可能是程序员想要避免编译器给出警告，也可能是，他觉得运算的优先级不对，想要解决这个问题，但他失败了。不管怎样，加括号是没用的。

这里有一个更深层次的问题。如果有可能不把代码弄得更复杂，就别弄了。这样写会好点：

```
id = mOp.del(usrc.dn());
if ( id == -1 ) {
```

建议

请不要偷懒，多加一行：毕竟，复合表达式很难读。先赋值，然后才到比较。你这样做，后面帮你维护代码的人也更容易些，而且，这能减少出错的几率。

所以我的结论就是——不要总想着表现。

这个建议听起来有点微不足道，但是我希望它能帮到你。干净有条理的代码比“看我多酷”风格的代码要好。

17.用专门的函数来清楚私有数据

下面的代码来自 **Apache HTTP Server** 项目。该错误被 **PVS-Studio** 诊断为：**V597** 编译器会删掉'memset'函数调用，这个函数是用来清除‘**x**’缓冲区的。应该用 RtlSecureZeroMemory()来擦除私有数据。

```
static void MD4Transform(
  apr_uint32_t state[4], const unsigned char block[64])
{
  apr_uint32_t a = state[0], b = state[1],
               c = state[2], d = state[3],
               x[APR_MD4_DIGESTSIZE];
  ....
  /* Zeroize sensitive information. */
  memset(x, 0, sizeof(x));
}
```

解释

这段代码是调用 *memset()*函数来擦出私有数据。但这不是最好的法子，因为数据并没有被擦除。更确切的说，它们是否被擦除取决于编译器，编译器的设置以及时机。

试着从编译器的角度看这段代码。它尽他最大的努力来让你的代码尽快运行，所以它会进行一些优化。其中一个优化就是删除一些函数调用，这些函数调用不会影响程序的功能，所以从 *C/C++*的角度看有点多余。以上面的代码来说，就是 *memset()*函数。没错，这个函数会修改‘*x*’缓冲区，但是这个缓冲区后面都没有再用到，所以，这意味着，调用 *memset()*函数可以——而且，应该——删掉。

重要！我现在要告诉你的不是一个关于编译器行为的理论化模型——而是现实的。在这样的例子中，编译器真的会删掉 *memset()*函数。你自己可以做几个实验来检查。关于这个问题的更多细节和例子，请看下面的几篇文章：

- *Security, security! But do you test it?*
- *Safe Clearing of Private Data*.

- *V597*.编译器会删掉*'memset'*函数调用，这个函数是用来清除‘***x***’缓冲区的。应该用*RtlSecureZeroMemory()*来擦除私有数据。
- *Zero and forget -- caveats of zeroing memory in C*（也要看这篇文章的 *discussion*）
- *MSC06-C. Beware of compiler optimizations.*

让这个错误，删掉调用 *memset()*，变得难搞的是，它很难追踪。在调试的时候，你很可能用的是非优化代码，函数调用还在。你只有在研究汇编器信息的时候才会发现这个问题，汇编器信息是在形成优化应用程序版本时形成的。

有些程序员坚信，需要在编译器中解决问题，而且没有理由把 *memset()*这样重要的函数去掉。但这个例子不一样。这个函数并没有比其他函数更重要或者更不重要，所以编译器有绝对的理由在它调用处优化它。毕竟，这样的代码很多余。

正确代码

```
memset_s(x, sizeof(x), 0, sizeof(x));
```
or
```
RtlSecureZeroMemory(x, sizeof(x));
```

建议

你应该使用专门的内存清理函数，那种编译器就算出于优化目的也不允许删除的函数。

比如说，*Visual Studio* 就提供了 *RtlSecureZeroMemory* 函数，还有，从 *C11* 开始，你可以用 *memset_s* 函数。如果必要，你甚至可以自己创建一个安全的函数——网上有很多例子。下面这些是其中的一部分：

版本 *1*：

```
errno_t memset_s(void *v, rsize_t smax, int c, rsize_t n) {
  if (v == NULL) return EINVAL;
  if (smax > RSIZE_MAX) return EINVAL;
  if (n > smax) return EINVAL;
  volatile unsigned char *p = v;
  while (smax-- && n--) {
    *p++ = c;
  }
  return 0;
}
```

版本 *2*：

```
void secure_zero(void *s, size_t n)
{
    volatile char *p = s;
    while (n--) *p++ = 0;
}
```

有些程序员更厉害，用函数实现给数组赋伪随机值。这些函数通过运行不同次数来确保不受时间测量的攻击。你也可以在网上找到这些函数的实现。

*18.*你关于一种语言的知识并不总适用于其他语言

下面的代码片段来自 **Putty** 项目。无效的代码被 **PVS-Studio** 诊断为：**V814** 降低了性能。在计算循环条件的时候，**strlen** 函数被调用了多次。

```
static void tell_str(FILE * stream, char *str)
{
  unsigned int i;
  for (i = 0; i < strlen(str); ++i)
    tell_char(stream, str[i]);
}
```

解释

这里并没有明确的错误，但是这样的代码在我们处理长字符串的时候效率非常低，因为在每一次循环迭代的时候都要调用 *strlen()*函数。所以这里非要说有错的话，就是效率低。

一般的，主要是在那些之前用 *Pascal* （或 *Delphi*）的程序员所写的代码中发现这种错误。在 *Pascal* 中，判断循环的终止条件只计算一次，所以这个代码很合适而且常见。

让我们看一个用 *Pascal* 写的例子。单词‘*called*’只会打印一次，因为只调用 *pstrlen()* 一次。

```
program test;
var
  i   : integer;
  str : string;

function pstrlen(str : string): integer;
begin
  writeln('called');
  pstrlen := Length(str);
end;

begin
  str := 'a pascal string';
  for i:= 1 to pstrlen(str) do
    writeln(str[i]);
end.
```

高效代码

```
static void tell_str(FILE * stream, char *str)
{
  size_t i;
  const size_t len = strlen(str);
  for (i = 0; i < len; ++i)
    tell_char(stream, str[i]);
}
```

建议

不要忘记， 在 *C/C++*中，循环种植条件会在每一次迭代中重新计算一次。所以调用低效的函数做判断条件委实不是个好主意，尤其是，你可以在进入循环前只计算一次。

在一些例子中，编译器可能会优化含有 *strlen*（）的代码，比如说，如果一个指针总是依次指向相同的字符串。但我们不能总依赖它。

*19.*如何正确的从一个构造函数中调用另一个

这个问题是在 ***LibreOffice*** 项目中发现的。该错误被 ***PVS-Studio*** 诊断为：***V603*** 对象已经创建但还没有使用。如果你想要调用构造函数，应该用“***this->Guess::Guess(....)”***。

Guess::Guess()

```
{
  language_str = DEFAULT_LANGUAGE;
  country_str = DEFAULT_COUNTRY;
  encoding_str = DEFAULT_ENCODING;
}

Guess::Guess(const char * guess_str)
{
  Guess();
  ....
}
```

解释

好的程序员会讨厌写重复的代码。这很好。但是到构造函数这一块，很多人在想要使代码更简短有条理时会自食其果。

你知道，构造函数和普通函数不一样。如果我们写*"A::A(int x) { A(); }"*，会创建一个临时的无名的 *A* 类对象，而不是调用一个无参的构造函数。

这也是上面的代码所发生的：创建一个临时的无名的 *Guess()*对象，然后即刻被摧毁，因为没有初始化函数成员。

正确代码

有三种方法可以避免在构造函数中写重复代码。让我们来看看具体是什么。

第一种方法，设计一个独立的初始化函数，然后在两个构造函数里调用它。我会给你看几个例子——这样就会更明显了。

这是一种很好，可靠，干净又安全的技术。然而，有些程序元希望他们的代码更短一些。所以我不得不提到另外两种方法。

这两种方法有点危险，而且需要你对它们如何运作，你将面临怎样的后果有足够的了解。

第二种方法：

```
Guess::Guess(const char * guess_str)
{
  new (this) Guess();
  ....
}
```

第三种方法：

```
Guess::Guess(const char * guess_str)
{
  this->Guess();
  ....
}
```

第二种和第三种方法比较危险是因为基类被初始化了两次。这样的代码会引起一些不明显的bug，有害而无益。来看一个例子，在何处调用构造函数是对的，在何处不对。

现在是对的：

```
class SomeClass

{
  int x, y;
public:
  SomeClass() { new (this) SomeClass(0,0); }
  SomeClass(int xx, int yy) : x(xx), y(yy) {}
};
```

这个代码是安全的而且运作正常，因为代码只包含了简单的数据类型，而且没有继承其他类。调用两次构造函数不会有任何危险。

然后，现在看另一个例子，在这里调用构造函数会引起错误。

```
class Base

{
public:
 char *ptr;
 std::vector vect;
 Base() { ptr = new char[1000]; }
 ~Base() { delete [] ptr; }
};

class Derived : Base
{
  Derived(Foo foo) { }
  Derived(Bar bar) {
     new (this) Derived(bar.foo);
  }
  Derived(Bar bar, int) {
     this->Derived(bar.foo);
  }
}
```

所以我们用表达式"new (this) Derived(bar.foo);" 或者 "this->Derived(bar.foo)"来调用构造函数。

基对象已经创建，也已经初始化。再次调用构造函数会引起二次初始化。那么指向新分配的内存空间的指针就会被写到 ptr 里，最终导致内存溢出。至于对 *std::vector* 类对象的二次初始化，其后果更难以预测。唯一清楚的就是：这样的代码是不被允许的。

最后你想那么麻烦吗？如果你还不能正确使用 *C++11* 的新特性，那就是用方法 *1*（创建一个初始化函数）吧。在很少见的情况下才会直接调用构造函数。

建议

最后，我们知道了一个 *C++*的新特性可以在构造函数上帮到我们。

C++11 允许在构造函数中调用同类的另一个构造函数（也就是委托「*delegation*」）。这样就一个构造函数只要用几行代码就可以使用另一个构造函数的行为了。

比如：

```
Guess::Guess(const char * guess_str) : Guess()

{
  ....
}
```

想要理解更多关于委托构造函数的，看下面的链接：

1. 维基百科 C++11Object construction improvement.
2. *C++11 FAQ Delegating constructors.*
3. MSDN Uniform Initialization and Delegating Constructors.

20.单是文件终止符（EOF）的检查还不够

下面的代码片段来自 **SETI@home** 项目。该错误被 **PVS-Studio** 诊断为：**V633** 可能会无限循环。要跳出循环‘**cin.eof()**’这个条件还不够。考虑加‘**cin.fail()**’这一函数调用到条件表达式中。

```
template <typename T>
std::istream &operator >>(std::istream &i, sqlblob<T> &b)
{
  ....
  while (!i.eof())
  {
    i >> tmp;
    buf+=(tmp+' ');
  }
  ....
```

```
}
```

解释

从一个流对象中读数据的操作并非那么微不足道。程序员常会调用 **eof()**方法来检查是否已经到了终点。然而，但是检查这个还不够，因为它不是充分条件，而且你无法确定是否出现了读取错误或者流出错，这两个都会引起特定的问题。

注意。本文说提到的信息包括输入流和输出流。为避免重复，在这里，我们只讨论其中一种。

上面的代码存在着一个错误：如果有任何数据读取错误，就会导致无限循环，因为 **eof()**总返回 **false**。最重要的是，循环会一直处理错误的数据，因为不知道什么样的值会写到 **tmp** 变量里。

为避免这样的问题，我们需要额外的方法来检查流的状态：***bad(), fail()***.

正确代码

我们可以利用流对象可以强制转换为布尔类型的特性。true 意味着读值操作完成。更多关于这段代码如何运作的细节可以在 StackOverflow 上找到。

```
template <typename T>
std::istream &operator >>(std::istream &i, sqlblob<T> &b)
{
  ....
  while (i >> tmp)
  {
    buf+=(tmp+' ');
  }
  ....
}
```

建议

当用流来读取数据的时候，不要只使用 eof()，还要检查其他故障。

使用 *bad()* 和 *fail()*函数来检查流的状态。第一个函数是用来流的完整性，第二个函数是用来检查数据读取错误的。

但是，用 *bool()*操作会更便捷一些，就像在正确代码那里显示的。

21.检查文件终止符（EOF）有正确到达终点

让我们继续讨论和文件有关的话题，再来看 EOF。但这次我们要讨论的是完全不同的 bug 类型。它通常只出现在本地化软件版本里。

下面的代码片段是选自 **Computational Network Toolkit**。错误被 **PVS-Studio** 诊断为：**V739 EOF** 不应该和 **char** 类型比较。‘**c’**应该是 **int** 类型。

```
string fgetstring(FILE* f)
{
  string res;
  for (;;)
  {
    char c = (char) fgetc(f);
    if (c == EOF)
      RuntimeError("error reading .... 0: %s", strerror(errno));
    if (c == 0)
      break;
    res.push_back(c);
  }
  return res;
}
```

解释

让我们来看 EOF 是怎样声明的：

```
#define EOF (-1)
```

如你所见，EOF 就是一个值为‘-1’的 int 类型。*Fgetc()*返回一个 *int* 类型的值。也就是说，它可以返回 *0* 到 *255* 的一个数，或者是*-1*（*EOF*）。

使用扩展字符集（*Extended ASCII Codes*）的用户在程序中某一个字母处理不当的时候可能会出错。

比如说，在 *Windows1251* 编码表中，俄文的最后一个字母是 *0xFF* ，而在程序中就会被编译为文件终止符。

正确代码

```
for (;;)
{
  int c = fgetc(f);
  if (c == EOF)
    RuntimeError("error reading .... 0: %s", strerror(errno));
  if (c == 0)
    break;
  res.push_back(static_cast<char>(c));
}
```

建议

在这里并没有什么特殊的建议，但既然我们谈到了 *EOF*，我想展示一些人没有注意到的比较有趣的错误。

只要记住，如果一个函数的返回值是 *int* 类型，就不要急于把它转化为 *char* 类型。停下来检查看有没有错。顺便提一句，我们在第二个技巧“大于 *0* 并不意味着是 *1*”那里讨论 *memcmp*（）函数的时候用到了相似的代码。（关于 *MySQL* 漏洞那一块）。

22.不要用#pragma warning(default:X)

下面的代码来自 ***TortoiseGIT*** 项目。该错误被 ***PVS-Studio*** 诊断为：***V665*** 也许，在此处，‘***#pragma warning(default: X)***’用得不对。应该用’***#pragma warning(push/pop)***’

```
#pragma warning(disable:4996)
LONG result = regKey.QueryValue(buf, _T(""), &buf_size);
#pragma warning(default:4996)
```

解释

程序员经常认为，早期让警告实现的*"pragma warning(disable: X)"*指令在使用*"pragma warning(default : X)"*后就能继续工作。但并非如此。*'pragma warning(default : X)'*这条指令是把‘*X*’警告设置为 *DEFAULT* 状态，这两个不是同一件事。

假设一个文件在编译时用了*-wall*，那么就会出现 *C4061* 警告。如果你增加了*"#pragma warning(default : 4061)"* 指令，那么这条警告就不会出现，因为它已经被默认关掉了。

正确代码

```
#pragma warning(push)
#pragma warning(disable:4996)
LONG result = regKey.QueryValue(buf, _T(""), &buf_size);
#pragma warning(pop)
```

建议

返回警告的之前状态的正确方法是使用*"#pragma warning(push[,n])"* 和*"#pragma warning(pop)"*这两条指令。可以看 *Visual C++11* 关于这些指令描述的文档：*: Pragma Directives. Warnings*.

库开发者应该要特别注意 *V665* 警告。忽视定制警告会在库使用者这边引起巨大的灾难。

关于这个话题的，值得一看的好文章：*So, You Want to Suppress This Warning in Visual C++*

23.自动计算字符串长度

下面的代码来自 ***OpenSSL*** 库。错误被 ***PVS-Studio*** 诊断为：***V666*** 检查‘***strncmp’***的第三个参数。它可能和字符串的长度不一致，字符串长度取决于第二个参数。

```
if (!strncmp(vstart, "ASCII", 5))
  arg->format = ASN1_GEN_FORMAT_ASCII;
else if (!strncmp(vstart, "UTF8", 4))
  arg->format = ASN1_GEN_FORMAT_UTF8;
else if (!strncmp(vstart, "HEX", 3))
  arg->format = ASN1_GEN_FORMAT_HEX;
else if (!strncmp(vstart, "BITLIST", 3))
  arg->format = ASN1_GEN_FORMAT_BITLIST;
else
  ....
```

解释

很难停止使用魔数。而且没有理由不使用类似 *0, 1, -1, 10* 这些常数。更难的是，给这些常数命名，而且，它们会使阅读代码变得更复杂。

然而，减少使用魔数的个数是有用的。比如说，避免使用用来定义字符串长度的魔数就很有用。

让我们来看一下上面给出的代码。代码看上去很像是复制粘贴的。程序员复制了下面这一行：

```
else if (!strncmp(vstart, "HEX", 3))
```

之后，用*"BITLIST"*代替*"HEX"* ，但是程序员忘了把 *3* 改为 *7*. 结果就是，字符串不是和*"BITLIST"*做比较，而是仅仅比较了*"BIT"*。这个错误似乎不太严重，但终究是个错误。

用复制粘贴来写代码真的很糟糕。更严重的是，字符串长度由魔数来定义。一直以来，我们遇到这样的错误，就是字符串长度和确定的数字不一致的错误，都是因为拼写错误或者程序员的疏忽。所以，这是一个典型的错误，我们需要做点什么来避免它。让我们来看看应该如何避免。

正确代码

乍一看，只要把 *strncmp()*换成 *strcmp()*就好了。那样魔数就消失了。

```
else if (!strcmp(vstart, "HEX"))
```

不好——我们改变了代码运作的逻辑。*strncmp()*的作用是检查一个字符串是否是以‘*HEX*’开始的，而 *strcmp()*是检查两个字符串是否相等的。它们做的是不同的检查。

要解决这个问题最简单的方法就是改变常数：

```
else if (!strncmp(vstart, "BITLIST", 7))
  arg->format = ASN1_GEN_FORMAT_BITLIST;
```

这个代码是正确的，但是魔数 *7* 还在那里，这就有点不好了。这也是为什么我要建议另一种方法。

建议

如果我们能够正确估计字符串的长度，这样的错误就能够避免。最简单的方法就是用 *strlen()*函数。

```
else if (!strncmp(vstart, "BITLIST", strlen("BITLIST")))
```

在下面例子中，如果你忘记更改字符串，你能更快地发现它们的不匹配。

```
else if (!strncmp(vstart, "BITLIST", strlen("HEX")))
```

但是这个建议有两个缺点：

- 无法保证编译器会不会优化 *strlen()*调用：用一个常数来代替它。
- 你要逐字复制字符串。看上去不好看，而且也会出错。

第一个问题可以在编译阶段用专门计算字符串长度的结构来解决。比如说，你可以用这样的宏：

```
#define StrLiteralLen(arg) ((sizeof(arg) / sizeof(arg[0])) - 1)
....
else if (!strncmp(vstart, "BITLIST", StrLiteralLen("BITLIST")))
```

但这个宏有点危险。在重构的时候会出现下面的代码：

```
const char *StringA = "BITLIST";
if (!strncmp(vstart, StringA, StrLiteralLen(StringA)))
```

在这个例子中，*StrLiteralLen* 宏会返回一些没意义的东西。取决于指针的大小（*4* 或 *8* 字节），我们会得到 *3* 或 *7* 。但是在 *C++*中我们可以避免这种尴尬的局面，通过使用更复杂的技巧：

```
template <typename T, size_t N>
char (&ArraySizeHelper(T (&array)[N]))[N];
#define StrLiteralLen(str) (sizeof(ArraySizeHelper(str)) - 1)
```

现在，如果 *StrLiteralLen* 宏的参数是一个简单的指针，我们可能无法编译上面的代码。

让我们来看第二个问难题（复制字符串）。我不知道要对 *C* 程序员说什么。你可以为它写一个特别的宏，但是我个人不喜欢这样。我不热衷于宏。所以我知道要建议什么。

在 C++中，一切都很好。而且，我们可以用更聪明的法子来解决第一个问题。模板函数会帮我们很大的忙。你可以用不同的方法写，但是一般是长这样的：

```
template<typename T, size_t N>
int mystrncmp(const T *a, const T (&b)[N])
{
  return _tcsnccmp(a, b, N - 1);
}
```

现在，字符串只用一次。在编译阶段就能计算字符串的长度。你不会遇到简单指针也不会错误的计算字符串的长度。完美!

总结：在处理字符串的时候避免使用魔数。使用宏或模板函数。这样函数不仅会变得更安全，而且会更漂亮更简短。

作为例子，你可以看 *strcpy_s ()*的声明：

```
errno_t strcpy_s(
   char *strDestination,
   size_t numberOfElements,
   const char *strSource
);
template <size_t size>
errno_t strcpy_s(
   char (&strDestination)[size],
   const char *strSource
); // C++ only
```

第一个是针对于 C 语言的，或者在不是提前知道缓冲区大小的情况的。如果我们要处理缓冲区，创建一个栈，这样我们就可以在 C++中用第二个了。

24.Override 和 final 标识符应该成为你的新朋友

下面的代码片段选自 MFC 库。该错误被 PVS-Studio 诊断为：V301 意外的函数重载行为。观察函数'WinHelpW'的第一个参数，这个函数在派生类'CFrameWndEx'和基类'CWnd'中。

```
class CWnd : public CCmdTarget {
  ....
  virtual void WinHelp(DWORD_PTR dwData,
                       UINT nCmd = HELP_CONTEXT);
  ....
};
class CFrameWnd : public CWnd {
  .…
};
class CFrameWndEx : public CFrameWnd {
  ....
```

```
  virtual void WinHelp(DWORD dwData,
                       UINT nCmd = HELP_CONTEXT);
  ....
};
```

解释

当你重写一个虚函数的时候，很容易在函数签名的时候出错，也很容易变成定义一个新的函数，这个函数跟基类里的虚函数没有任何关联。在这个例子中，有这几类错误：

- 在被重写的函数中，参数类型不同。
- 在被重写的函数中，参数个数不同。在参数比较多的时候，这种情况会更严重。
- 重写的函数在 const 修饰符这里不同。
- 基类函数不是虚函数。这种情况是以为在派生类中的函数会重载基类中的函数，但事实上，并没有，派生类隐藏了它。

相同的错误也会在修改已存在的代码中的参数类型或参数个数的时候发生。当程序员改变要改变虚函数签名的时候，他可能改了几乎所有有继承关系的类里的，但忘了在一些派生类中改，于是，就出错了。

在将代码移植到 64 位平台的时候，这种错误出现得更平凡。因为这个时候要把 DWORD 改成 DWORD_PTR , LONG , LONG_PTR 等等。细节. 这就是我们这个例子中发生的错误。

即使有这种错误的代码能够在 32 位系统正常运行，在 64 位机子上也会出错。因为在 32 位上，DWORD 和 DWORD_PTR 是 *unsigned long* 的同义词，但是在 64 位平台上，DWORD_PTR 是 *unsigned __int64* 的同义词。

正确代码

```
class CFrameWndEx : public CFrameWnd {
  ....
  virtual void WinHelp(DWORD_PTR dwData,
                       UINT nCmd = HELP_CONTEXT) override;
  ....
};
```

建议

现在有方法来避免上面描述的错误了。在 *C++11* 里添加了两个标识符：

- *Override*——表明该函数是重写基类中的虚函数。
- *Final*——表明该函数在派生类中无需重写。

我们对 override 这个标识符比较感兴趣。这个标识符可以指示编译器去检查某个虚函数是不是重写基类里的函数，如果不是，就报错。

如果在写 CFrameWndEx 类里的 WinHelp 函数的时候我们有用 override 指明，这个应用的 64 位版本在编译的时候会出错。所以，这个错误应该在早期清除掉。

在重写虚函数的时候记得用 override（或 final）标识符。更多关于 override 和 final 的知识请看下面的文章：

- Cppreference.com. override specifier (since C++11)
- Cppreference.com. final specifier (since C++11)
- 维基百科. Explicit overrides and final.
- Stackoverflow.com. 'override' in c++11.

25.不要再拿‘this’和 nullptr 比较了。

下面的代码来自 CoreCLR 项目。这段危险的代码被 PVS-Studio 诊断为：V704 应避免'this == nullptr'这个表达式——这个表达式在新的编译器上总是 false，因为‘this’指针永远不会为 NULL。

```
bool FieldSeqNode::IsFirstElemFieldSeq()
{
  if (this == nullptr)
    return false;
  return m_fieldHnd == FieldSeqStore::FirstElemPseudoField;
}
```

解释

人们以前常拿 this 指针来和 0/ *NULL* / *nullptr* 来做比较。这种情况只有在 *C++*开发阶段的早期是正常的。我们是在‘考古学’研究的时候发现这样的代码的。我建议阅读一篇关于 *checking Cfront* 的文章了解更多关于它们的细节。而且，在以前，*this* 指针的值是可以改变的，但过去太久了，这件事已经被遗忘。

让我们再回过头来看 *this* 和 *nullptr* 之间的比较。

现在它是非法的。根据现代 *C++*标准，*this 永远*不会等于 nullptr。

根据 C++标准，调用 IsFirstElemFieldSeq()方法来访问空指针 this 会导致未定义行为。

看上去它是想表示，如果 this==0,则在运行这个函数的时候就不能进入该区域。但事实上，这段代码有两种完全不同的实现方式。根据 C++标准，this 指针用于不会为空，所以编译器会通过简化它来优化函数调用：

```
bool FieldSeqNode::IsFirstElemFieldSeq()
{
  return m_fieldHnd == FieldSeqStore::FirstElemPseudoField;
}
```

对了，这里还有一个陷阱。假设这里有一个多重继承：

```
class X: public Y, public FieldSeqNode { .... };
....
X * nullX = NULL;
X->IsFirstElemFieldSeq();
```

假设 Y 类的大小是 8 字节。然后源指针 NULL（0x00000000）在这样的情况下是对的，然后它又指向了下一个对象 FieldSeqNode 的开始处，然后你要偏移 sizeof（Y）字节。所以指向 IsFirstElemFieldSeq()函数的 this 指针应该是 0x00000008。最后，检查"this == 0"也没什么意义了。

正确代码

很难给出正确代码的例子。只是把判断条件从这个函数中移除是不够的。你要重构代码，用一种你永远不会调用到使用了空指针的函数。

建议

所以，现在已经宣布"if (this == nullptr)"是不合法的了。然而你还是可以在很多应用程序和库（比如说，MFC 库）中发现这条代码。这就是为什么 Visual C++还在孜孜不倦地比较 this 和 0。我猜，编译器开发者不会那么疯狂把正确工作了那么多年的代码删掉。

但这条规则已经正式通过了。所以从现在开始，让我们避免将 this 和 null 比较。还有，如果你有时间，检查一下那些非法的比较，已经重写那些代码。

很多编译器会有如下的反映。首先，它们会给出一个关于比较的警告。可能在我还没有研究这个问题的时候，它们已经给了。然后，在某一时刻，它们要支持新标准，那你的代码就不能正常运行了。所以我强烈建议你现在就开始遵循这条规则，后面也会很有用的。

P.S. 在重构的时候，你需要空对象模式。

另外两个关于这个话题的链接：

7. Still Comparing "this" Pointer to Null?
8. Diagnostic V704.

26.狡猾的 VARIANT_BOOL

下面的代码选自 NAME 项目。代码中的错误被 PVS-Studio 诊断为：V721 VARIANT_BOOL

类使用不正确。true(VARIANT_TRUE)等于-1。检查第一个参数。

```
virtual HRESULT __stdcall
  put_HandleKeyboard (VARIANT_BOOL pVal) = 0;
....
pController->put_HandleKeyboard(true);
```

解释

有一段很机智的话是这么写的：

自那些年我们学习了 Basic 后，我们就不得不为我们自己所犯下的错误而买单了。

这个提示就是关于邪恶的。VARIANT_BOOL 是 VB 里的一种类型。我们现在遇到的一些编程问题都跟这个类型有关。因为，它的 true 等于-1。

让我们来看一下这个类型的声明和表示 true/false 的常数。

typedef short VARIANT_BOOL;

#define VARIANT_TRUE ((VARIANT_BOOL)-1)

#define VARIANT_FALSE ((VARIANT_BOOL)0)

看起来也没什么麻烦的。false 是 0 ，true 是非 0.所以-1 是一个很合适的常熟。但是在使用 true 和 false 来代替 VARIANT_TRUE 的时候很容易出错。

正确代码

```
pController->put_HandleKeyboard(VARIANT_TRUE);
```

建议

如果你看到一个不知道的类型，别慌，去看一下文档。即使那个类型的名字里有 BOOL，但也不意味这你可以把 1 赋值给这个类型的变量。

在使用 HRESULT 这个类型的时候，有些程序员也会犯同样的错误。他们想要拿它来和 FALSE 和 TRUE 比较，但是忘记了：

```
#define S_OK     ((HRESULT)0L)
#define S_FALSE  ((HRESULT)1L)
```

所以我要提醒你要特别注意那些你没见过的类型，在编程的时候也别太着急。

27.狡诈的 BSTR 字符串

让我们来讨论另一个令人讨厌的数据类型——BSTR（基础字符串「Basic string」或者二进制字符串「binary string」）。

下面的代码选自 VirtualBox 项目。这段代码中的错误被 PVS-Studio 分析器诊断为：V745 一个'wchar_t *'类型的字符串被错误的转换为‘BSTR’字符串。可以考虑使用函数 'SysAllocString' 。

```
....
HRESULT EventClassID(BSTR bstrEventClassID);
....
hr = pIEventSubscription->put_EventClassID(
                    L"{d5978630-5b9f-11d1-8dd2-00aa004abd5e}");
```

解释

BSTR 是这样声明的：

```
typedef wchar_t OLECHAR;
typedef OLECHAR * BSTR;
```

乍一看，好像"wchar_t *" 和 *BSTR* 是同一个东西。但并非如此，而且程序员们也因此产生了很多困惑和错误。

让我们来讨论这个 BSTR 类型以对这个例子有一个更深的理解。

这个是 MSDN site 中 BSTR 的有关信息。读 MSDN 的文档确实不好玩，但不好玩也得做啊，是不？

BSTR (Basic string 或 *binary string)* 是 *COM*，*Automation* 和 *Interop* 函数所使用的一种字符串数据类型。在所有可以进入脚本的接口中都可以使用 *BSTR* 数据类型。*BSTR* 的描述：

- 长度前缀。*BSTR* 的前 *4* 个字节表示字符串长度。它的值不包含终止符 *null*。
- 数据字符串。一个 *Unicode* 的字符串。可能会多次嵌入 *null* 字符。
- 终止符：两个 *null* 字符。

一个 *BSTR* 是一个指针。这个指针指向数据字符串的第一个字符，而不是长度前缀。*BSTR* 用 *COM* 内存分配函数类分配内存，所以它们可以从函数中返回而无需考虑内存分配。

下面的代码是不对的：

```
BSTR MyBstr = L"I am a happy BSTR";
```

这行代码可以正确构建（编译和链接），但是不会正确运行，因为这个字符串没有长度前缀。如果你用调试程序去检查这个变量的内存位置，你也不会在数据字符串前面找到一个 *4byte* 的长度前缀。所以，应该这样写：

```
BSTR MyBstr = SysAllocString(L"I am a happy BSTR");
```

现在定位这个变量的调试程序就能够找到一个包含值为 *34* 的长度前缀。是 *34* 而非 *17* 是因为前面有个‘*L*’修饰符，*L* 把单字符（single-character）转换成了宽字符（*single-character*）。调试程序也能显示在数据字符串后面的 *2byte* 的终止符 *null*。

COM 函数的参数应该是 *BSTR*，如果你用简单的 *Unicode* 字符串，*COM* 函数会出错的。

我想这些已经足够你了解，为什么我们要区分 *BSTR* 和简单的*"wchar_t *"*字符串类型。

额外的链接：

- *MSDN.BSTR*
- *StackOverfow. Static code analysis for detecting passing a wchar_t* to BSTR.*
- *StackOverfow. BSTR to std::string (std::wstring) and vice versa.*
- *Robert Pittenger. Guide to BSTR and CString Conversions.*
- *Eric Lippert. Eric's Complete Guide To BSTR Semantics.*

正确代码

```
hr = pIEventSubscription->put_EventClassID(
       SysAllocString(L"{d5978630-5b9f-11d1-8dd2-00aa004abd5e}"));
```

建议

建议跟前面的一样。当你看到不认识的数据类型的时候，别慌，去看看文档。这很重要，所以这个建议再次出现也无须大惊小怪的。

*28.*能用简单函数完成的就别用宏

下面的代码选自 *ReactOS* 项目。代码中包含的错误被 *PVS-Studio* 诊断为：*V640* 代码的运算逻辑跟它的格式不一致。一直在执行第二句。可能是少了花括号。

```
#define stat64_to_stat(buf64, buf)   \
    buf->st_dev   = (buf64)->st_dev;   \
    buf->st_ino   = (buf64)->st_ino;   \
    buf->st_mode  = (buf64)->st_mode;  \
    buf->st_nlink = (buf64)->st_nlink; \
    buf->st_uid   = (buf64)->st_uid;   \
```

```
    buf->st_gid   = (buf64)->st_gid;   \
    buf->st_rdev  = (buf64)->st_rdev;  \
    buf->st_size  = (_off_t)(buf64)->st_size;  \
    buf->st_atime = (time_t)(buf64)->st_atime; \
    buf->st_mtime = (time_t)(buf64)->st_mtime; \
    buf->st_ctime = (time_t)(buf64)->st_ctime; \

int CDECL _tstat(const _TCHAR* path, struct _stat * buf)
{
  int ret;
  struct __stat64 buf64;

  ret = _tstat64(path, &buf64);
  if (!ret)
    stat64_to_stat(&buf64, buf);
  return ret;
}
```

解释

代码有点长。幸运的是，它也比较简单，所以应该不难理解。

大意是这样的，如果你成功地通过*_tstat64()*函数得到了文件的信息，那就把数据放到 *of _stat* 结构体中。我们用宏 *stat64_to_stat* 来保存数据。

这个宏不能正确运行。它执行的操作没有用花括号*{}*包起来。结果就是，条件运算符体只是宏里的第一个字符串。如果你展开那个宏，你会看到：

```
if (!ret)
  buf->st_dev   = (&buf64)->st_dev;
buf->st_ino   = (&buf64)->st_ino;
buf->st_mode  = (&buf64)->st_mode;
```

因此，不管有没有正确收到信息，结构体里的成员都会被复制。

这肯定是个错误，但在实际中，它不是致命的。我们比较幸运，这里只是白白复制未初始化的内存单元而已。但我有遇到过更严重的错误，也跟宏编写不当有关。

正确代码

最简单的改变就是给宏加个花括号。加 *do { .... } while (0)*会更好。在宏和函数后面，你可以放个分号';'。

```
#define stat64_to_stat(buf64, buf)   \
  do { \
    buf->st_dev   = (buf64)->st_dev;   \
    buf->st_ino   = (buf64)->st_ino;   \
    buf->st_mode  = (buf64)->st_mode;  \
    buf->st_nlink = (buf64)->st_nlink; \
    buf->st_uid   = (buf64)->st_uid;   \
    buf->st_gid   = (buf64)->st_gid;   \
    buf->st_rdev  = (buf64)->st_rdev;  \
    buf->st_size  = (_off_t)(buf64)->st_size;  \
    buf->st_atime = (time_t)(buf64)->st_atime; \
    buf->st_mtime = (time_t)(buf64)->st_mtime; \
    buf->st_ctime = (time_t)(buf64)->st_ctime; \
```

```
  } while (0)
```

建议

我不能说我喜欢宏。我知道代码中不能没有宏，尤其是 *C*。然而，如果可能我还是尽量避免使用它们。同时，我也呼吁你不要滥用宏。我不喜欢宏有三个原因：

- 代码会比较难调试。
- 容易出错。
- 代码变得更难理解，尤其是在一些宏中又调用其他的宏。

还有很多关于宏的错误。我给的这个例子已经能显示，我们不一定非要用宏不可。我实在无法理解为什么有的程序员不用简单的函数而要用宏。放弃用宏的优点：

9. 代码更简单。你不需要花额外的时间来写宏，而且还要用‘\’来对齐。
10. 代码更可读（例子里的错误不可能再在代码中出现）。

考虑到这些缺点，我只能想到优化。是，会有函数调用，但它不会那么危险。

无论如何，让我们假设这件事对我们来说很重要，然后考虑如何优化。首先，你可以用 *inline*。其次，把函数声明为 *static* 也不错。我觉得这对编译器编译这个函数，并且不用分离函数体已经够了。

事实上你一点也不用担心，因为现在的编译器都很智能。即使你的函数没有 *inline/static*，但是如果它觉得应该加，编译器也会自己加进来的。不要被这种细节困扰。最好还是写一个简单易懂的函数，这样更有利。

在我看来，上面的代码应该这样写：

```
static void stat64_to_stat(const struct __stat64 *buf64,
                           struct _stat *buf)
{
  buf->st_dev   = buf64->st_dev;
  buf->st_ino   = buf64->st_ino;
  buf->st_mode  = buf64->st_mode;
  buf->st_nlink = buf64->st_nlink;
  buf->st_uid   = buf64->st_uid;
  buf->st_gid   = buf64->st_gid;
  buf->st_rdev  = buf64->st_rdev;
  buf->st_size  = (_off_t)buf64->st_size;
  buf->st_atime = (time_t)buf64->st_atime;
  buf->st_mtime = (time_t)buf64->st_mtime;
  buf->st_ctime = (time_t)buf64->st_ctime;
}
```

说实话，在这里我们还可以做更多的改进。例如在 *C++*中，最好不用指针而用引用。用一个没有预先检查的指针看上去不会有多好看。但这又是另一个故事了，我不会在讲宏的部分里讨论它的。

29.在迭代器中使用前自增（++i）而不是用后自增（i++）

下面的代码选自 **Unreal Engine 4** 项目。这段代码被 **PVS-Studio** 诊断为不够高效，因为：**V803** 性能下降。在‘**itr**’是迭代器的例子中，使用前自增比用后自增高效。把 **iterator**++换成 ++**iterator**。

```
void FSlateNotificationManager::GetWindows(....) const
{
  for( auto Iter(NotificationLists.CreateConstIterator());
       Iter; Iter++ )
  {
    TSharedPtr<SNotificationList> NotificationList = *Iter;
    ....
  }
}
```

解释

如果你没有看到这一节的标题，我想你应该很难注意到这段代码的问题。乍一看，这段代码没什么问题，但是还不够完美。是的，我在讨论后自增——“Iter++”。相对于厚自增迭代器，你更应该用前自增的形式，比如用‘++Iter’代替‘ Iter++’。为什么我们要这么做呢，这么做的现实意义在哪？这里面是有玄机的。

正确代码

```
for( auto Iter(NotificationLists.CreateConstIterator());
     Iter; ++Iter)
```

建议

前自增和后自增的区别大伙儿都知道。我希望大家也同样知道这两个结构内部的区别（这个区别表现了我们运算的规则）。如果你曾经重载过这个运算，你肯定有注意到。如果没有——我会做一个简单的解释。（了解的人可以直接跳过这一段，直接跳到代码后面的。）

前自增运算会改变对象的状态，然后给自己返回改变过的值。不需要临时对象。前自增运算大概是这样的：

```
MyOwnClass& operator++()
{
  ++meOwnField;
  return (*this);
}
```

后自增同样也改变对象的状态，但是它返回的是对象之前的状态。它能做到返回之前状态是用到了临时对象。后自增重载代码应该是这样的：

```
MyOwnClass operator++(int)

{
  MyOWnCLass tmp = *this;
  ++(*this);
  return tmp;
}
```

通过上面的代码，你可以看到有一个加法运算用到了临时变量。在实际应用中，这有多重要呢?

现在的编译器都很智能，能自己做优化，而且如果临时变量没有用的话，它们就不会去创建它。这就是为什么发布版本很难找出'it++' 和 '++it'的区别。

但是在调试的时候，又不一样了。在这个例子中，这两种表现的区别还是很重要的。

比如说，在这篇文章中，就用了几个例子测试代码在调试版本中用前自增和后自增所用的时间。我们可以看到用后自增时间差不多要长四倍。

有些人会说，“然后呢？他们在发布版本中是一样的！”，在他们这么说也对也不对。一般来说，在做单元测试的时候，我们会花比较多的时间在调试版本上去调试代码。虽然我们要花很长的时间在软件的调试版本上，但并不意味着我们愿意浪费时间呀。

总的来说，我觉得我们已经解决这个问题了。—我们应该用前自增代替后自增。是的，你应该。这样在调试的时候你花的时间就比较少了啊。而且，如果迭代器非常‘大量’，效果就更可观了。

参考文献（建议阅读）：

- Is it reasonable to use the prefix increment operator ++it instead of postfix operator it++ for iterators?
- Pre vs. post increment operator - benchmark

30.Visual C++和 wprintf（）函数

下面的代码选自 **Energy Checker SDK.**代码中包含的错误被 **PVS-Studio** 诊断为：**V576** 不正确的格式。建议检查**'wprintf'**函数的第二个参数。需要指向 **wchar_t** 字符串的指针标志。

```
int main(void) {
  ...
```

```
  char *p = NULL;
  ...
  wprintf(
    _T("Using power link directory: %s\n"),
    p
  );
  ...
}
```

解释

注意：第一个错误是用 *_T* 来标识一个一个宽字符的字符串格式。应该用前缀 *L* 才对。然而，这个错误不是太严重，也不是我们的兴趣所在。如果我们不用宽字符格式，代码只是不能编译而已，*_T* 也不会扩展的。

如果你想用 *wprintf()*来打印一个 *char**的字符串，在格式化字符串里你应该用*"%S"* 。

很多 *Linux* 程序员不知道陷阱在哪。事情是这样的，微软在 *wsprintf* 这类函数的实现上有些不同。如果我们在 *Visual C++*中用 *wsprintf* 函数，我们就应该用*“%s”*来打印宽字符串，同时，在打印 *char**字符串是用*“%S”*。所以，这个例子有点诡异。跨平台的应用程序经常会落入这个陷阱里。

正确代码

我这里给出的代码只是一个修正这个问题的方案，不是最完美的。但我仍然希望展示我们修改此类错误的首要观点。

```
char *p = NULL;
...
#ifdef defined(_WIN32)
wprintf(L"Using power link directory: %S\n"), p);
#else
wprintf(L"Using power link directory: %s\n"), p);
#endif
```

建议

在这里，我没有什么特别的建议。我只是告诉你一些在使用类似 *wprintf()*的函数时要注意的。

从 *Visual Studio2015* 开始，就有针对“建议写可移植性代码”的方案。为兼容 *ISO C (C99),*你应该指出预处理器，宏*_CRT_STDIO_ISO_WIDE_SPECIFIERS*。

这个例子中，下面的代码是正确的：

```
const wchar_t *p = L"abcdef";
const char *x = "xyz";
wprintf(L"%S %s", p, x);
```

分析器知道*_CRT_STDIO_ISO_WIDE_SPECIFIERS*，而且在做分析的时候也把它考虑进去了。

顺便说一句，如果你打开了兼容 *ISO C* 模式（宏*_CRT_STDIO_ISO_WIDE_SPECIFIERS* 已经声明），你可以用之间的版本，也就是用*"%Ts"*来标识格式。

总的来说，关于宽字符的标识非常错综复杂，已经不是短短的一篇文章能讲完的。为更深刻了解这个话题，我建议你阅读一些关于这个话题的文章：

- *Bug 1121290 - distinguish specifier s and ls in the printf family of functions*
- MBCS to Unicode conversion in swprintf
- Visual Studio swprintf is making all my %s formatters want wchar_t * instead of char *

31.在 C 和 C++中，数组不是按值传递的

下面的代码来自游戏'**Wolf**'.代码中包含的错误被 PVS-Studio 诊断为：V511 在'sizeof (src)'这个表达式中，sizeof()返回的是指针的大小，而不是数组的大小。

```
ID_INLINE mat3_t::mat3_t( float src[ 3 ][ 3 ] ) {
  memcpy( mat, src, sizeof( src ) );
}
```

解释

有时候，程序员会忘记在 C/C++中，你不可以把一个数组值传递给一个函数。因为传数组给一个函数，数组类型自动转换为指针类型。方括号里里的数字没什么意思，它们只是用来告诉程序员，多大的数组被传进去了。事实上，你可以传任意大小的数组。比如，下面的代码也能编译成功：

```
void F(int p[10]) { }
void G()
{
  int p[3];
  F(p);
}
```

类似的，*sizeof(src)*不是计算数组的大小，而是指针的大小。然后结果就是，*memcpy()*只是复制了一部分数组而已。也就是，*4* 或者 *8* 字节，这都取决于指针的大小（不算奇怪的结构）。

正确代码

最简单的修正是这样的：

```
ID_INLINE mat3_t::mat3_t( float src[ 3 ][ 3 ] ) {
  memcpy(mat, src, sizeof(float) * 3 * 3);
}
```

建议

有多种方法可以让你的代码更安全。

知道数组大小。你可以在函数中用数组的引用做参数。但并不是所有人都知道可以这么做。甚至更少的人知道怎么写。所以我希望下面的例子能够有用，有趣：

```
ID_INLINE mat3_t::mat3_t( float (&src)[3][3] )
{
  memcpy( mat, src, sizeof( src ) );
}
```

现在，就可以在传递数组的时候只用右边的部分。而且最重要的是，*sizeof()*计算的是数组的大小了，不再是一个指针的。

解决这个问题的另一个方法是用 *std::array* 类。

不知道数组大小。有些编程书的作者建议使用 *std::vector* 类和其他相似的类。但是，这实际应用做，并不总是那么方便。

有时，你也会想要用指针来解决这个问题。在这个例子中，你可以传两个参数给那个函数：一个指针，元素个数。然而，一般来说，这样做都不太好，因为它会导致很多 *bug*。

在这样的例子中，*"C++ Core Guidelines"*这篇文章里的观点蛮有用的。我建议读*"Do not pass an array as a single pointer"*。总的来说，在你有空的时候读*"C++ Core Guidelines"*真的不失为一件有益的事。它里面包含了很多有用的观点。

32.危险的 printf

下面的代码选自 TortoiseSVN 项目。代码中包含的错误被 PVS-Studio 诊断为：V618 以一种危险的方式调用 printf 函数，因为传递进去的那一行应该包含格式化说明。安全使用 printf 的例子：: printf("%s", str);

```
BOOL CPOFile::ParseFile(....)
{
  ....
  printf(File.getloc().name().c_str());
  ....
}
```

解释

当你打算打印或者，比如说，写一个字符串到文件中，很多程序员会这样写：

```
printf(str);
fprintf(file, str);
```

一个优秀的程序员应该时刻记得，这样组织代码是非常不安全的。事情是这样的，如果格式化说明不知怎样进入一个字符串里，将会导致无法预测的后果。

让我们回过头来看最初的例子。如果文件名是"file%s%i%s.txt"，那么这个程序会崩溃，或者输出一些不知所云的东西。但这不还是问题的全部。事实上，这样的函数调用是一个真正的漏洞。在它的帮助下，我们能发动攻击。用一些刻意的字符串，就可以打印内存里的私有数据。

更多关于这个漏洞的信息可以查看这篇文章。花点时间去通读一遍，我保证，会很有趣的。你不仅能看到理论基础，还可以看到实际的例子。

正确代码

```
printf("%s", File.getloc().name().c_str());
```

建议

像 printf()这样的函数会引起很多有关安全的问题。最好一点也不要使用它们，你可以用其他的来代替啊。比如说，你会发现 *boost::format* 或者 *std::stringstream* 也很有用。

一般来说，草率地使用 *printf(), sprintf(), fprintf(),*等等函数不仅会导致运行不当，而且会引发潜在的漏洞，然后别人就可以利用这个漏洞来攻击你。

33.永远不要间接引用空指针

bug 是在 *GIT* 的源代码中发现的。代码中包含的错误被 *PVS-Studio* 诊断为：*V595* 在还没有验证‘*tree*‘指针是否为空之前就使用它。检查 *134,136* 行。

```
void mark_tree_uninteresting(struct tree *tree)
{
  struct object *obj = &tree->object;
  if (!tree)
    return;
  ....
}
```

解释

无疑，这是一个糟糕的做法，因为它间接引用了空指针，而这样间接引用的结果就是未定义行为。我们都同意这背后的理论基础。

但是，当具体运用的时候，程序员们就开始争论不休了。总有人声称，这段代码能够正确运行。他们甚至以项上人头做担保——对他们来说，它也总是能运行。所以，我要给出更多的理由来证明我的观点。这就是为什么这篇文章是改变他们观点的又一尝试。

我故意选了这么一个能够引发更多讨论的例子。当 *tree* 指针被调用，类成员不仅是在使用，也在计算该成员的地址。那么，如果*(tree == nullptr)*，就永远不会用到成员的地址，而且函数已经退出了。很多人都认为这段代码是正确的。

但并不是。你不应该这么写代码。未定义行为不一定造成程序崩溃，比如赋值给空地址，或者诸如此类的行为。只要你调用了一个等于 *null* 的指针，未定义行为可以是任何操作。这个时候再讨论这段代码会如何运行已经没有意义了，因为此时它可以做它任何想做的操作。

未定义行为的一个标志是，编译器会把*"if (!tree) return;"*删掉——编译器看到指针已经被调用了，而指针不是空的，那么这一行检查就会被编译器移除。这只是众多版本中的一个，而这个版本会引起程序崩溃。

我建议阅读这一篇文章：http://www.viva64.com/en/b/0306/，里面给出了更多细节。

正确代码

```
void mark_tree_uninteresting(struct tree *tree)
{
  if (!tree)
    return;
  struct object *obj = &tree->object;
  ....
}
```

建议

要注意未定义行为，即使一切看上去都没什么问题。没必要冒险。即使我已经写了，但还是很难表现出它的价值。尝试着去避免未定义行为，即使一切看上来都没问题。

有人会想，他清楚的知道，未定义行为是怎样运作的。而且，他可能会想，这意味着，他可以做一些其他人不能做的事，还能保证代码不出错。但并非如此。下一章节将会说明未定义行为真的非常危险。

34.未定义行为比你想象的要贴近我们的生活

这次很难给出实际应用的例子。但是，我经常有看到会导致下面要所描述问题的可疑代码。这个错误是在处理大数组的时候出现的，而我不知道哪个项目会用到那么大的数组。我们没有真的收集到 64 位的错误，所以今天的例子是刻意的。

让我们来看一段刻意出错的代码：

```
size_t Count = 1024*1024*1024; // 1 Gb
if (is64bit)
  Count *= 5; // 5 Gb
```

```
char *array = (char *)malloc(Count);
memset(array, 0, Count);

int index = 0;
for (size_t i = 0; i != Count; i++)
  array[index++] = char(i) | 1;

if (array[Count - 1] == 0)
  printf("The last array element contains 0.\n");

free(array);
```

解释

如果你构建的是这个项目的 32 位版本，代码是可以正确运行的。但是如果我们要编译 64 位版本，情况就会变得很复杂。

这个 64 位项目开始的时候申请 5GB 的缓冲区，然后初始化为 0.接着，用循环修改为非 0 值：用“|1”来保证非 0.

现在来猜一下，如果在 x64 位版本下用 Visual Studio 2015 编译的时候，这段代码会如何运行？你有答案吗？如果有，那我们继续。

如果你运行的是这个项目的调试版本，它会因为下标溢出而崩溃。在一定程度上，下标会溢出，而且其值会变成?2147483648 (INT_MIN).

听上去很有逻辑，对不对？事情并不是这样的。这是一个未定义行为，任何事情都有可能发生。

为获得更多内容，我推荐下面的链接：

- Integer overflow
- Understanding Integer Overflow in C/C++
- Is signed integer overflow still undefined behavior in C++?

好玩的是——当我或者其他人说这段代码会引发未定义行为的时候，就会有人抱怨。我不知道为啥，但看起来好像是，他们觉得自己对 C++有绝对的了解，而且知道编译器是怎样运作的。

但事实上他们都没有注意到它。如果他们知道，他们不会这么说的（大众的观点）：

这在理论上没什么意义。好吧，是，‘int’溢出会导致未定义行为。但是这没什么，不过老生常谈而已。在实际应用中，我们知道代码运行后能得到什么。1 加 INT_MAX 等于 INT_MIN 嘛。但是，可能在宇宙中的某一个角落存在着能让这种情况（整形一出也能得我们想要的结果）发生的结构，但就是我的 Visual C++/GCC 没能给出正确的结果而已。

现在，没有任何魔法。我会用一个简单的例子来证明未定义行为，而且没有用到什么奇怪的结构，只是一个 win64 的项目。

把上面的例子构建成发布版本并运行就已经足够了。代码会崩溃结束，而且“最后一个数组元素包含 0”的提示也不会出现。

未定义行为一般以如下的方式出现。所有的数组元素都会被赋值，尽管数组下标的类型 int 并不足以覆盖所有的元素。那些至今还在怀疑我的人，可以看一下下面的代码：

```
  int index = 0;
  for (size_t i = 0; i != Count; i++)
000000013F6D102D  xor         ecx,ecx
000000013F6D102F  nop
    array[index++] = char(i) | 1;
000000013F6D1030  movzx       edx,cl
000000013F6D1033  or          dl,1
000000013F6D1036  mov         byte ptr [rcx+rbx],dl
000000013F6D1039  inc         rcx
000000013F6D103C  cmp         rcx,rdi
000000013F6D103F  jne         main+30h (013F6D1030h)
```

这里就有一个未定义行为。而且没有用到什么特殊的编译器，用的就是 VS2015.

如果你用 unsigned 代替 int，就不会有未定义行为。而数组就只有一部分被填充，最后我们会收到一条信息——“最后一个数组元素包含 0”。

相似的代码用 unsigned 的情况：

```
unsigned index = 0;
000000013F07102D  xor         r9d,r9d
  for (size_t i = 0; i != Count; i++)
000000013F071030  mov         ecx,r9d
000000013F071033  nop         dword ptr [rax]
000000013F071037  nop         word ptr [rax+rax]
    array[index++] = char(i) | 1;
000000013F071040  movzx       r8d,cl
000000013F071044  mov         edx,r9d
000000013F071047  or          r8b,1
000000013F07104B  inc         r9d
000000013F07104E  inc         rcx
000000013F071051  mov         byte ptr [rdx+rbx],r8b
000000013F071055  cmp         rcx,rdi
000000013F071058  jne         main+40h (013F071040h)
```

正确代码

在程序中你要用对正确的类型，这样才能确保它顺利运行。如果你要处理大数组，忘了 int 和 unsigned 吧。正确的类型有 *ptrdiff_t*, *intptr_t*, *size_t*, *DWORD_PTR*, *std::vector::size_type* 等等，在这里用 size_t.

```
size_t index = 0;
for (size_t i = 0; i != Count; i++)
  array[index++] = char(i) | 1;
```

建议

如果根据 C/C++的语言机制会导致未定义行为的话，就别跟它们争了，也不要尝试去预测它们会在将来做出什么表现。只要不写危险的代码就好了啊。

还是有很多固执的程序员不愿意在转换负数、比较 this 和 null 或者有符号类型溢出时看任何表示怀疑的言论。

不要这样。就算现在代码运行得好好的也不意味着一切都是好的。未定义行为是不可预测的。可预测的程序行为是未定义行为的一个变体。

35.在枚举中加了新的枚举常量后别忘了修改 switch 运算

下面的代码来自 Appleseed 项目。代码中包含的错误被 PVS-Studio 诊断为：V719 switch 语句没有覆盖枚举“InputFormat”的所有值，少了 InputFormatEntity.

```
enum InputFormat
{
    InputFormatScalar,
    InputFormatSpectralReflectance,
    InputFormatSpectralIlluminance,
    InputFormatSpectralReflectanceWithAlpha,
    InputFormatSpectralIlluminanceWithAlpha,
    InputFormatEntity
};
```

```
switch (m_format)
{
  case InputFormatScalar:
    ....
  case InputFormatSpectralReflectance:
  case InputFormatSpectralIlluminance:
    ....
  case InputFormatSpectralReflectanceWithAlpha:
  case InputFormatSpectralIlluminanceWithAlpha:
    ....
}
```

解释

有的时候我们需要在已经存在的枚举里加入新的元素，当我们做这个操作的时候，我们需要特别的谨慎——因为我们要检查全部代码看看哪里有用到这个枚举，比如说在 switch 和 if 中。上面给出的代码就是这种情况。

在 InputFormat 中加了 InputFormatEntity——这里是我想象的，因为确实在 InputFormat 的后面有加入这个常量。很多时候，程序员在枚举后面加了新的常量，然后忘了检查代码以确保他们有正确处理这个常量，也没有修改 switch 操作。

最后的结果就是，在这个例子中，并没有处理到"m_format==InputFormatEntity"的情况。

正确代码

```
switch (m_format)
{
  case InputFormatScalar:
  ....
  case InputFormatSpectralReflectance:
  case InputFormatSpectralIlluminance:
  ....
  case InputFormatSpectralReflectanceWithAlpha:
  case InputFormatSpectralIlluminanceWithAlpha:
  ....
  case InputFormatEntity:
  ....
}
```

建议

让我们想想，怎样在代码重构的时候避免这种错误？最简单，但不那么有效的解决方法就是加一个‘default’，它可以输出一个信息，像这样：

```
switch (m_format)
{
  case InputFormatScalar:
  ....
  ....
  default:
    assert(false);
    throw "Not all variants are considered"
}
```

现在，如果变量 m_format 是 *InputFormatEntity*，我们就可以看到一个异常。这样的处理方法有两个不好的地方：

- 因为有可能在测试阶段这个错误并没有显现出来（如果在测试的时候，*m_format* 不等于 *InputFormatEntity*），那这个错误就会流入发布版本，以后才会显现——在客户运行时出现。如果要顾客来反映这种问题，真的很糟糕。
- 如果我们把 *default* 考虑为一个错误，那我们就不得不写一个 *case* 来解决枚举所有可能的值。这样就很不方便，尤其是当一个枚举中有很多常量的时候。有时，用 *default* 来处理不同 *case* 真的很方便。

我建议用以下的方法来解决这个问题，我不敢说这个方法很完美，但至少它有解决到问题。

当你定义一个枚举的时候，要确保你也加了一条特殊的注释。你也可以用关键词和枚举名。

例子：

```
enum InputFormat
{
```

```
  InputFormatScalar,
  ....
  InputFormatEntity
  //If you want to add a new constant, find all ENUM:InputFormat.
};

switch (m_format) //ENUM:InputFormat
{
  ....
}
```

在上面的代码中，当你要改变枚举 *InputFormat*，你就可以直接在项目的源代码中查找“*ENUM:InputFormat*”。

如果你是在一个开发者团队中，你可以告诉你的小伙伴们这个约定，然后把它加入到你们的编程标准和风格指引中。如果有人没能遵守这条原则，真遗憾。

*36.*如果你的 *PC* 发生了什么奇怪的事，检查一下内存

我想在看了枚举类型的错误后，你肯定很累了。所以这次，让我们休息一会，不看代码了。

一个典型的场景——你们的项目没有正确运行。但你也不知道发生了什么。在这种情况下，我建议你不要急于去责备某个人，而应该把焦点放到你们的代码上。在 99.9%的案例中，罪恶之源就是你们的开发团队中某个人引入了 bug。通常这个 bug 都非常的愚蠢且平淡无奇。所以快点花点时间去找找。

实际上，这个一直出现的 bug 也不能说明什么。你可能只是遭遇了海森堡 bug（Heisenbug）.

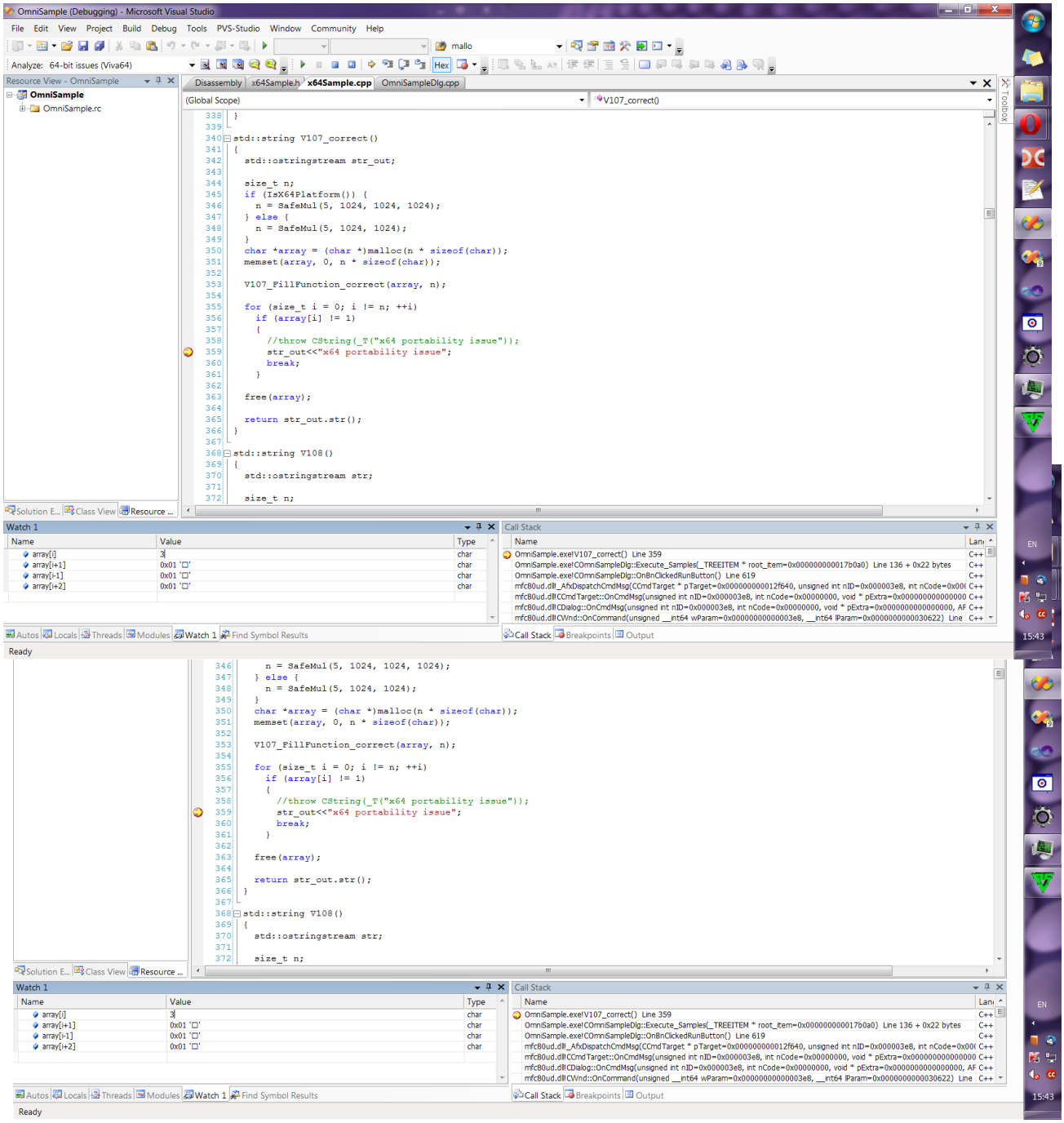
责备编译器也不是什么好主意。它当然也会出错啊，虽然真的非常少。比如说，你会发现就是仅仅错用了 sizeof()，这件事就会变得很棘手。我的博客里有篇博文就是关于它的：The compiler is to blame for everything。

但为了让记录更公正一点，我应该说，还是会有例外存在的。很少的情况下，bug 不对代码做什么。但我们还是应该注意到这种可能性的存在。这可以帮助我们保持理智。

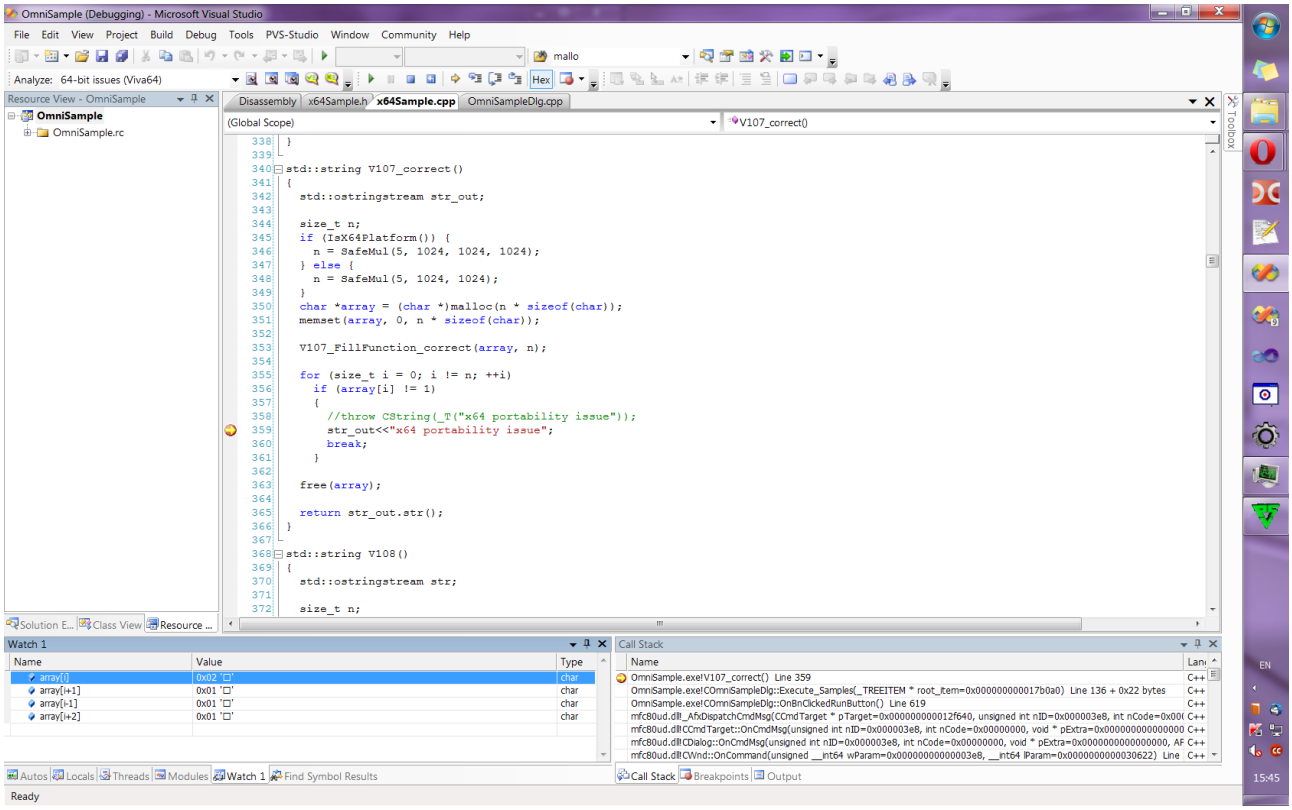
我会用一个我曾经遭遇过的例子来证明这种可能性。非常幸运，我当时有截图。

当时，我在做一个简单的测试项目，想要去演示 Viva64 分析器（PVS-Studio 的前身）的功能，但是它没能正确运行。

经过漫长而令人讨厌的调查之后，我发现是一个内存槽引起这所有的问题。更确切地说，是 1bit。你可以看看下面的照片，我当时正在调试，想往这个存储单元写入‘3’。



在改变内存之后，编译器开始读值并将其显示在窗口上，但它显示的是 2：看，地址是 0x02。尽管我已经把值设置为 3 了。低位总为 0.



一个内存测试项目证实了这个问题。非常奇怪，这台电脑一直都工作得好好的，没出什么问题。最后为了让项目正确运行，我换了内存条。

我真的很幸运。当时我处理的是一个简单的测试项目。虽然我还是花了很多时间去了解发生了什么。我花了两个多小时去检查汇编器目录，想要找到引起这种奇怪行为的原因是什么。是的，我当时有因此责备编译器。

我无法想象，如果是一个实际的项目，我要花多大的时间精力去解决这个问题。谢天谢地，我当时不用去调试其他的东西。

建议

要时刻检查你代码中的错误。别想着推卸责任。

但是，如果一个 bug 只在你的电脑上出现，而且都快一周了。那么，可能不是因为你的代码。

要持续寻找 bug。但是在回家前，可以运行一个一整夜的 RAM 测试。或许，这个简单的测试步骤能节省你的时间精力。

37.要小心在 do {...} while (...)里面的‘continue’操作

下面的代码来自 Haiku 项目（BeOS 的继承者）。代码中包含的错误被 PVS-Studio 诊断为：V696. 'continue'操作会终止'do { ... } while (FALSE)' 循环，因为判断条件一直是 false。

```
do {
  ....
  if (appType.InitCheck() == B_OK
    && appType.GetAppHint(&hintRef) == B_OK
    && appRef == hintRef)
  {
    appType.SetAppHint(NULL);
    // try again
    continue;
  }
  ....
} while (false);
```

解释

在 do-while 循环里的 continue 的运作机制可能跟某些程序员想象的不一样。当遇到 *continue* 的时候，会先要检查循环的终止条件。我想要更详细的解释这个问题。假设有个程序员写了这么一段代码：

```
for (int i = 0; i < n; i++)
{
  if (blabla(i))
    continue;
  foo();
}
```

或者是这样的：

```
while (i < n)
{
  if (blabla(i++))
    continue;
  foo();
}
```

很多程序员凭直觉理解，当遇到 *continue*，就要（重新）评估控制条件（$i<n$）,然后只有为真的时候才会进入下一个循环迭代。但，如果程序员的代码是这么写的：

```
do
{
  if (blabla(i++))
    continue;
  foo();
} while (i < n);
```

直觉就不起作用了哦。因为他们没有在 *continue* 上面看到判断条件，而且，似乎对于他们来说，*continue* 会马上触发下一个循环迭代。事情并非如此，*continue* 还是像它平常所做的那样——先重新评估控制条件。

除非踩了狗屎运，不然缺乏对 *continue* 的理解真的很难不出错。无论如何，如果循环条件 一直是 *false*，这个错误肯定会发生。因为在上面给出的代码中，程序员打算在之后的循环中执

行特定的操作。代码中的注释“*//try again* ”证明了他们有这个意图。当然不会有‘*again*’啦，因为判断条件一直是 *false*，而且一遇到 *continue*，循环就会终止哦。

换句话说，在这个 *do {...} while (false)*结构中，*continue* 就相当于 *break*。

正确代码

要写正确代码有很多选项。比如，创造一个无限循环，然后用 *continue* 继续，用 *break* 退出。

```
for (;;) {
  ....
  if (appType.InitCheck() == B_OK
    && appType.GetAppHint(&hintRef) == B_OK
    && appRef == hintRef)
  {
    appType.SetAppHint(NULL);
    // try again
    continue;
  }
  ....
  break;
};
```

建议

尝试着不要在 *do { ... } while (...)*用 *continue*。即使你真的知道这是如何运作的。因为一不小心你就会犯这种错误，而且/或者你的同时就不能正确理解代码啦，最后把它错改。我一直都这么说：一个优秀的程序员不是知道并使用其他语言技巧的那一个，而是能够写出干净易懂的，即使新手也能理解的代码的那一个。

38.从现在开始，用 nullptr 不要用 NULL

新的 C++标准引入很多有用的改变。有些我现在还没有用到，但是有些应该马上用起来，因为用它们有诸多益处。

其中一个现代化标准就是关键字 *nullptr*, 其旨在代替 NULL 宏。

让我提醒你一下，在 C++中，NULL 的定义是 0 ，没有其他的了。

当然，看起来它好像就是一些语法糖（syntactic sugar）。那么当我们写 nullptr 或者 NULL 的时候，其中的区别是什么呢？真的有区别！用 nullptr 可以帮我们避免大量的错误。我将会用一些例子来证明。

假设有两个重载函数：

```
void Foo(int x, int y, const char *name);
void Foo(int x, int y, int ResourceID);
```

一个程序员肯呢个会这么写函数调用：

```
Foo(1, 2, NULL);
```

然后那个程序员坚信自己这么做是在调用第一个函数。但因为 NULL 是 0 而非其他东西，然后 0 又是整形，所以其实调用的是第二个函数。

然而，如果程序员用的是 nullptr，就不会出现这样的问题，而且第一个函数也能正确调用。另一种比较常见的使用 NULL 的例子是这样的：

```
if (unknownError)
  throw NULL;
```

在我看来，传一个指针到异常里面真的很奇怪。尽管如此，还是有人这么做。这样看来，那些程序员应该是这么写代码的。无论如何，关于这样写是好是坏的讨论已经超纲了。

最重要的是，那个程序员打算在处理未知错误的时候抛出一个异常，然后‘发送’一个空指针到外部世界。

事实上，这不是一个指针，而是一个整形。结果就是，异常处理的结果不像程序员所期望的那样。

"throw nullptr;"这行代码能让我们避免不幸，但是并不意味着我相信这样的代码能被接受。

在一些例子中，如果你用 nullptr，不正确的代码就不会被编译。

假设 WinApi 函数返回一个 *HRESULT* 类型。*HRESULT* 类型没有能用来处理指针的东西。然而还是有可能会写出类似这种无意义的代码：

```
if (WinApiFoo(a, b, c) != NULL)
```

这行代码能够编译，因为 *NULL* 是 *0*,是一个整形，然后 *HRESULT* 是一个长整形。长整形和整形是可以比较值的。如果你有 *nullptr*，那下面的代码就不会编译。

```
if (WinApiFoo(a, b, c) != nullptr)
```

因为编译错误，程序员就能够注意到并修改这行代码。

我想你已经明白我的意思了。有很多这样的例子。但大部分是人为的例子。而有的时候这种例子不那么有说服力。所以，有真实的例子吗？有的。这是其中一个。唯一的问题——它不是那么好看或者简短。

这个代码是来自 *MTASA* 项目。

所以，有 *RtlFillMemory()*哈。这可以是一个真实的函数或者一个宏。没关系。类似于 *memset()*函数，但第二第三个参数互换了位置。我们先来看一下这个宏是如何声明的：

```
#define RtlFillMemory(Destination,Length,Fill) \
  memset((Destination),(Fill),(Length))
```

还有 *FillMemory()*，它跟 *RtlFillMemory()*没什么区别：

```
#define FillMemory RtlFillMemory
```

是，一切都很长很复杂。但至少这是一个真实案例中的错误代码。

这里还有个用到了 *FillMemory* 宏的代码。

```
LPCTSTR __stdcall GetFaultReason ( EXCEPTION_POINTERS * pExPtrs )
{
  ....
  PIMAGEHLP_SYMBOL pSym = (PIMAGEHLP_SYMBOL)&g_stSymbol ;
  FillMemory ( pSym , NULL , SYM_BUFF_SIZE ) ;
  ....
}
```

这段代码有很多 bug。我们可以清晰的看到这里至少 2 到 3 个参数很混乱。这就是为啥分析器给出了两个警告 V575：

- V575 'memset' 函数要处理”512“值。检查第二个参数。crashhandler.cpp 499
- V575 'memset' 函数要处理”0“元素。检查第三个参数。crashhandler.cpp 499

代码可以编译是因为 *NULL* 是 *0*. 结果就是 *0* 数组元素被填充。但实际上，问题不仅仅是这个。一般来说，*NULL* 出现在这里是不合适的。*memset()*函数按字节处理，所以让内存充满 *NULL* 值没什么意义。这有点荒谬。正确的代码应该是这样：

```
FillMemory(pSym, SYM_BUFF_SIZE, 0);
```

或者这样：

```
ZeroMemory(pSym, SYM_BUFF_SIZE);
```

但还不是重点，重点是，这段没什么意义的代码会编译成功。然而，如果一个程序员已经养成了用 *nullptr* 不是用 *NULL* 的习惯，那他应该就会这么写：

```
FillMemory(pSym, nullptr, SYM_BUFF_SIZE);
```

这样的话，编译器就会给出一个错误信息，然后程序员就会意识到他们可能哪里出错了，然后就会更加注意他们编程的方式。

注意。我知道，在这个例子中，我们不能把一切都归咎于 *NULL*。但是也因为 *NULL*，不正确的代码成功编译了，没有输出任何警告。

建议

开始使用 nullptr。从此刻开始。还有，在你的公司的编程标准中做必要的修改。

用 nullptr 会帮助你避免某些愚蠢的错误，然后就可以稍微加快开发进程。

39.为什么不正确的代码也能运行

这个 bug 是在 Miranda NG 的项目中发现的。代码中包含的错误被 PVS-Studio 分析器诊断为：V502 可能'?:' 操作的结果和预期的有差。'?:' 的优先级低于'|'。

```
#define MF_BYCOMMAND 0x00000000L
void CMenuBar::updateState(const HMENU hMenu) const
{
  ....
  ::CheckMenuItem(hMenu, ID_VIEW_SHOWAVATAR,
    MF_BYCOMMAND | dat->bShowAvatar ? MF_CHECKED : MF_UNCHECKED);
  ....
}
```

解释

我们有看到过很多例子，即使代码逻辑不对也能运行。这一次，我想提出一个不一样的、发人深思的话题来讨论。有时我们会看到完全不正确的代码很偶然的，即使困难重重，还是运行了。现在，对于有经验的程序员来说，这没什么好惊讶的（这是另一个故事了），但对那些 C/C++的初学者来说，这就有点令人感到困惑了。所以，今天我们来看看这样的例子。

在上面给出的代码中，我们需要调用带有一定标志的 *CheckMenuItem();* 而且，猛地一看我们会觉得，如果 *bShowAvatar* 是 *true*，那就做或运算 *MF_BYCOMMAND* | *MF_CHECKED*，相反的，如果是 *false*，就是 *MF_BYCOMMAND* | *MF_UNCHECKED*。很简单。

在上面给出的代码中，程序员选用了最意料之中的三元运算符来表达这个意思（这个表达式是 *if-then-else* 的简便版本）：

MF_BYCOMMAND | dat->bShowAvatar ? MF_CHECKED : MF_UNCHECKED

但问题是，‘|’的优先级高于‘*?:* ’的（参看 *C/C++*中的运算优先级）。这样就导致了一下子有两个错误。

第一个错误是，条件改变了。它不再是——像某人可能理解的——*"dat->bShowAvatar"*，而是*"MF_BYCOMMAND | dat->bShowAvatar"*。

第二个错误——只剩下一个标志可以选择——不是 *MF_CHECKED* 就是 *MF_UNCHECKED.* 标志 *MF_BYCOMMAND* 已经缺失了。

尽管有这些错误，代码还是正确运行了！原因——踩了狗屎运。那个程序员很幸运，因为标志 *MF_BYCOMMAND* 等于 *0x00000000L*。而因为 *MF_BYCOMMAND* 等于 *0*,所以最后没有影响到代码。可能一些有经验的程序员已经知道我想要表达的意思了，但我还是再做写解释吧，以便初学者理解。

首先，让我们看一眼加了括号的正确表达式。

MF_BYCOMMAND | (dat->bShowAvatar ? MF_CHECKED : MF_UNCHECKED)

然后用具体的值来代替宏：

0x00000000L | (dat->bShowAvatar ? 0x00000008L : 0x00000000L)

如果‘|’的其中一个操作数是 *0*,那么我们就可以简化上面的表达式，得到：

dat->bShowAvatar ? 0x00000008L : 0x00000000L

现在我们再来看原先那个不正确的表达式：

MF_BYCOMMAND | dat->bShowAvatar ? MF_CHECKED : MF_UNCHECKED

把具体的值代入：

0x00000000L | dat->bShowAvatar ? 0x00000008L : 0x00000000L

在子表达式*"0x00000000L | dat->bShowAvatar"* 中，其中一个操作数是 *0*。简化可得：

dat->bShowAvatar ? 0x00000008L : 0x00000000L

最后，我们得到了相同的表达式，这就是为什么有错误的代码还是能正确运行了。另一个编程奇迹发生了。

正确代码

有很多种方法来修正这段代码。其中一种就是加括号，另一种——加一个中间变量。还有比较老的 *if* 运算也是可以的：

```
if (dat->bShowAvatar)
  ::CheckMenuItem(hMenu, ID_VIEW_SHOWAVATAR,
                  MF_BYCOMMAND | MF_CHECKED);
else
  ::CheckMenuItem(hMenu, ID_VIEW_SHOWAVATAR,
                  MF_BYCOMMAND | MF_UNCHECKED);
```

我其实并不坚持让你用这个方法来修正代码。这样看上去是更易读了，但是有点长。所以，这更多的是偏好的问题。

建议

我的建议很简单——避免使用太复杂的表达式，尤其是三元运算符。还有，别忘了加括号。

就像在第四章已经说过的，‘*?: ’*这个运算真的很危险。有的时候一疏忽你就忘了它的优先级非常的低，然后你就会写一个不正确的表达式。人们会在他们想要阻塞一个字符串的时候用到它，所以，你别这么做。

*40.*开始使用静态代码分析

读了那么长一篇由静态代码分析器开发者所写的文章，却没有看到关于使用它的建议，是不是很奇怪。所以我们现在就讲这个。

下面的代码选自 *Haiku* 项目（*BeOS* 的继承者）。代码中包含的错误被 *PVS-Studio* 诊断为：*V501.'<'* 表达式左右两边的操作数是一样的：*lJack->m_jackType < lJack→m_jackType*

```
int compareTypeAndID(....)
{
  ....
  if (lJack && rJack)
  {
    if (lJack->m_jackType < lJack->m_jackType)
    {
      return -1;
    }
    ....
}
```

解释

这是一个很常见的拼写错误。在右边应该是 *rJack*，然后被错写成了 *lJack*。

这个拼写错误实际上是很简单的一种，但这种情况真的蛮复杂的。因为无论是编程风格，或者是其他的方法，在这里都无济于事。有的时候就是在拼写的时候犯了个错误，对此，你能做什么呢?

要特别强调说这不是某个人或者项目的问题。无疑，世人皆会出错，即使是在重大项目中的专业人员也会。这是关于我这个言论的证明。你可以看到最简单的拼写错误，诸如 *A == A,* 在 *Notepad++, WinMerge, Chromium, Qt, Clang, OpenCV, TortoiseSVN, LibreOffice, CoreCLR, Unreal Engine 4* 等等这些项目中出现。

所以这个问题是真实存在的，而且这不是学生的实验课作业。当有人跟我说，有经验的程序员是不会犯这样的错误的时候，我一般都会把这个链接甩给他们。

正确代码

```
if (lJack->m_jackType < rJack->m_jackType)
```

建议

首先，让我们先来讲几个不那么有用的小贴士。

- 在编程的时候要特别小心，不要让错误溜进你的代码。（话是挺漂亮的，然而并没有什么用）。
- 用好的编程风格。（没有哪一种编程风格能帮你避免在变量名上出错。）

那么，有用的建议呢?

- 代码审查
- 单元测试（测试驱动开发「*TDD*」）
- 静态代码分析

我应该说，每一种方法都有各自的强处和弱处。这就是为什么要写出最高效可靠的代码是把它们都用起来。

代码审查能帮我们找到大量的，不同的错误，而且最重要的是，它可以帮我们提高代码的可读性。但不幸的是，分享式阅读（*Shared reading*）代码有点昂贵、无聊而且不会保证正确性。很难一直保持警惕，然后在阅读这种代码的时候发现拼写错误：

```
qreal l = (orig->x1 - orig->x2)*(orig->x1 - orig->x2) +
          (orig->y1 - orig->y2)*(orig->y1 - orig->y1) *
          (orig->x3 - orig->x4)*(orig->x3 - orig->x4) +
          (orig->y3 - orig->y4)*(orig->y3 - orig->y4);
```

理论上，**单元测试**应该能帮到我们。但仅仅是理论上。在实际中，检查所有可能的运行路径是不现实的。而且，测试本身也会有一些错误 *:)*

静态代码分析仅仅是个程序，而非人工智能。分析器可能会跳过一些错误，有时还会误报，就是其实代码是正确的，但它弹出了错误信息。尽管有这些缺点，它还是一个很有用的工具的。在编写代码的早期，它可以检测到很多错误。

静态代码分析器也可以用做便宜版本的代码审查。让代码分析器而不是程序员来检查代码，也可以让它更全面的检查某一代码片段。

当然，我会推荐使用 *PVS-Studio* 静态代码分析器，这是我们开发的。额，这世上并非只有这一款。还有很多免费或付费的工具可以使用。比如说，你可以看看免费开放的 *Cppcheck* 分析器。维基百科上也给出了一系列静态代码分析器的单子：*List of tools for static code analysis*.

注意：

- 如果不正确使用静态代码分析器，可能会让你有点头疼。最典型的错误之一就是“把检查模式选项设为最大值，然后就陷入大量的提示信息之中。”我能给出的众多建议中的一个就是，为获得更多选项 ，去看看 *A*, *B* 会有用的。
- 静态分析器的使用应该有个界限，不是一直使用，或者当遇到问题的时候使用。参看 *C*, *D* 的解释。

真的，尝试着去使用静态代码分析器。你会喜欢它的。它是一个非常好的工具。

最后，我推荐阅读 *John Carmack* 的文章：静态代码分析.

*41.*避免在项目中引入新的库

假设你要在你的项目中实现 *X* 功能。软件开发理论家会说，你要用已经存在的库 *Y* 来实现你需要的功能。事实上，这是软件开发中一个典型的方法——重用你自己或者其他人之前已经创建好的库（第三方库）。然后大多数程序员也用这个方法。

但是，在一些文章或书籍中那些理论家忘了说，用某些第三方库近 *10* 年会有多悲剧。

我非常建议避免在项目中增加新的库。但是也别误解我。我不是让你丝毫都不用库，自己去实现所有的功能。这当然很没必要啊。但有时有些程序员心血来潮，想要往项目里加点‘*cool*’的特性，然后就引入了新的库。难的不是往项目里加新的库，而是从此以后整个项目要不得不带上它。

追溯几个大项目的演化，我看到了很多有第三方库引起的问题。我可能只会列举其中的一些，但这个单子已经能够引发我们的思考了：

11. 引入新的库会立即增加项目的大小。在我们这个有快速互联网和大型 *SSD* 驱动的时代，这当然不是什么大问题。但当从版本控制系统下载时间从 *1* 变成 *10* 的时候，这就令人有点不快了。
12. 即使你只用到了库功能的 *1%*，你还是得把它整个都导入到项目中。结果就是，如果这些库是用是在编译模块使用的（比如，*DLL*），分布大小会很快增长。如果你把库当作源代码使用，那编译时间会大大增长。
13. 跟项目的编译（*compilation*）连接的设备也会变得更复杂。有些库需要额外的组件。一个简单的例子：我们需要用 *Python* 来编译链接成目标文件（*building*）。结果就是，有时你需要很多额外的程序才能创建这个项目。而且有些地方会出错的几率也会上升。这很难解释，你需要自己去经历。在大项目中，有的地方就是一直运行不起来，你就得花很大的精力去解决它，让所有的一切都能正常编译运行。

14. 如果你有担心漏洞，你要时常更新第三方库。那些违反者（*violator*）应该对这个很有兴趣，研究代码库寻找漏洞。首先，很多库都是开源的，其次，如果发现了其中一个库的弱点，你可以写一个 *exploit* 去利用那些用到这个库的应用程序。
15. 那些库有可能突然就改变了许可证类型。第一，你要将这件事时刻放在心上，还要保持跟进。第二，如果真的发生了，你要做什么其实也不太清楚。比如说，有一次，广泛使用的 *softfloat* 库从个人协议移到了 *BSD*。
16. 在升级编译器的版本的时候你会遇到困难。肯定会有一些库没有跟新的编译器兼容，那你只能等了，或者你自己连接那个库。
17. 在移到不同的编译器上的时候，你也会遇到问题。比如说，你以前用的是 *Visual C++*，现在打算用 *Intel C++*。可定会有一些库出错。
18. 在移到不同平台的时候，也会遇到问题。有的时候甚至不是一个完全不同的版本。比如说，你打算把一个 *Win32* 应用程序移到 *Win64* 上。你就会遇到同样的问题。最可能的是，有些库没准备好，你要想要怎么处理他们。最令人不悦的就是，有些库根本就不开发了，也没有更新它了。
19. 或早或晚，如果你用了很多 *C* 库，它们的类不是放在命名空间（*namespace*）的，你就会遇到命名冲突（*name clash*）。这就会引起编译错误，或者隐藏错误。比如说，某个它用的枚举常量和你打算用的冲突了。
20. 如果你的项目已经用了很多库，再增加一个一个也不会造成多大危害。我们可以用破窗效应来比喻。但结果是，一直增长的项目会变成无法控制的混乱。
21. 引入新的库还有很多我没注意到的缺点。但无论如何，额外的库会增加项目支撑的复杂性。有些问题就会出现在它们最意想不到的地方。

我应该在一次强调，我不是说应该停止使用第三方库。如果我们要处理 *PNG* 格式的 图像，我们就应该用 *LibPNG* 库，而不是再重新写一个。

但即使我们要处理 *PNG*，我们也应该停下来想想。我们真的需要引入一个库吗？我们要用这张图片来做什么？如果任务只是要把一张图片存为*.*png* 文件，我们可以使用系统函数。比如说，如果你是在写一个 *Windows* 应用程序，你可以用 *WIC*。而且如果你已经用了 *MFC* 库，你就没有必要让代码变得更复杂了，因为 *MFC* 里面有 *CImage* 类。（参看 *StackOverflow* 里的讨论）。少用了一个库——真棒！

让我跟你讲一个我实际遇到的例子。在开发 *PVS-Studio* 分析器的过程中，我们需要在几个诊断中用一些简单的正则表达式。一般来说，我觉得静态分析器不是用正则表达式的好地方。

这是一个非常低效的方法。我甚至写了一篇关于这个话题的文章。但有时你就得用正则表达式来在一个字符串了找些东西。

是可以加一个库啦，但很明显，它们都很冗余。同时，我们还是得用到正则表达式，所以我们要想办法。

非常偶然的，我那个时候在读《代码之美》（*ISBN 9780596510046*）。这本书是关于简单漂亮的解决方法的。然后我就看到了一种很简单的实现正则表达式的方法。就只用简单的十几行。这就是我所寻找的！

我决定在 *PVS-Studio* 中用这个实现方法。你知道吗？这个实现方法对我们来说够用了，复杂的正则表达式对我们来说不是必需的。

结论：我们没有引入新的库而是花了半个小时来写需要的功能。我们克制住了再用一个库的欲望。而且事实证明，这是一个很正确的决定。时间证明了我们真的不用那个库。我说的不是几个月，我们用那个功能用了超过五年呢。

这个例子让我坚信，越简单的解决方案越好。通过避免引入新的库（若可以），你的项目会更简单的。

读者可能会对查找正则表达式的代码感兴趣。我会把它摘抄出来。看，多漂亮。这段代码在整合进 *PVS-Studio* 的时候做了些许的改变，但主要思想还在。那么，这就是书上的代码啦：

```
 // regular expression format
// c Matches any "c" letter
//.(dot) Matches any (singular) symbol
//^ Matches the beginning of the input string
//$ Matches the end of the input string
# Match the appearance of the preceding character zero or
// several times

int matchhere(char *regexp, char *text);
int matchstar(int c, char *regexp, char *text);

// match: search for regular expression anywhere in text
int match(char *regexp, char *text)
{
  if (regexp[0] == '^')
    return matchhere(regexp+1, text);
  do { /* must look even if string is empty */
   if (matchhere(regexp, text))
     return 1;
  } while (*text++ != '\0');
  return 0;
}

// matchhere: search for regexp at beginning of text
int matchhere(char *regexp, char *text)
{
   if (regexp[0] == '\0')
     return 1;
```

```
   if (regexp[1] == '*')
     return matchstar(regexp[0], regexp+2, text);

   if (regexp[0] == '$' && regexp[1] == '\0')
     return *text == '\0';
   if (*text!='\0' && (regexp[0]=='.' || regexp[0]==*text))
     return matchhere(regexp+1, text+1);
   return 0;
}

// matchstar: search for c*regexp at beginning of text
int matchstar(int c, char *regexp, char *text)
{
  do {   /* * a * matches zero or more instances */
            more instances */
    if (matchhere(regexp, text))
      return 1;
  } while (*text != '\0' && (*text++ == c || c == '.'));
  return 0;
}
```

是的，这个版本非常简单，但要不是它这几年来我们就得用更复杂的方法啦。这段代码的功能是有限的，但也不需要加其他更复杂的代码了，我也不觉得会有这样的代码。这是一个非常好的例子，说明一个简单的方案可以比复杂的方案做得更好。

建议

不要急于往项目中引入新的库，只有在非它不可的时候才引入。

这里有一些变通的方法：

- 看看你们系统的 *API* 或者已经用的库是否也有需要的功能。研究这个问题真的是一个好主意。
- 如果你只是用库里面很少的一部分功能，自己去实现会比较好。引入一个库来“以防万一”实在没有必要。几乎可以肯定的是，这个库再将来也不会用到太多。程序员有的时候就是想要一个并不需要的普遍性。
- 如果还有其他法子来解决你的问题，选能够满足你要求的并且最简单的那一个。就像我先前说的，摒弃那种想法“这个库好酷哦，我们引入它来以防万一吧。”
- 在引入新的库之前，坐下来想想。甚至可以休息一会儿，喝杯咖啡，和你的同事讨论一下。可能你就会意识到你可以用一种完全不同的方法来解决这个问题，而不需要使用第三方库。

P.S. 我这里讲的这一个观点可能并不能被每一个人所接受。比如说，实际上我会推荐用WinAPI,而不是一个通用的可移植库。可能会有人反对，因为这么做就把这个项目和一个操作系统‘绑’在一起了。这样要移植项目就会很困难。我不同意这种观点。很多时候，“稍后我

们会把它移植到其他操作系统上”这种观点只是存在于程序员的脑海里而已。对于管理者而言，这个任务并不是必须的。另一种观点——程序可能会因为其复杂性和一般性在它受到欢迎以及有必要移植之前就翘辫子了。还有，不要忘了上面列出的第八个问题。

42.不要用以“empty”命名的函数

代码选自 WinMerge 项目。代码中包含的错误被 PVS-Studio 诊断为：V530 要用到函数‘empty’的返回值。

```
void CDirView::GetItemFileNames(
  int sel, String& strLeft, String& strRight) const
{
  UINT_PTR diffpos = GetItemKey(sel);
  if (diffpos == (UINT_PTR)SPECIAL_ITEM_POS)
  {
    strLeft.empty();
    strRight.empty();
  }
  ....
}
```

解释

这个程序员是打算清空 *strLeft* 和 *strRight* 这两个字符串。它们都是 *String* 类型，*String* 类型跟 *std::wstring* 非常像。

出于这个这个目的，他调用了 *empty()*函数。但这样是不对的。*empty()*不会改变该对象，只是返回该字符串是否为空。

正确代码

要修正这个代码，你应该用 *clear()*或者 *erase()*取代 *empty()*。*WinMerge* 开发者更喜欢 *erase(),* 所以现在代码是这样的：

```
if (diffpos == (UINT_PTR)SPECIAL_ITEM_POS)
{
  strLeft.erase();
  strRight.erase();
}
```

建议

在这个例子中*"empty()"* 真的不合适。因为在不同的库中，这个函数代表着两种不同的操作。

在一些库中，*"empty()"* 函数是清除对象。在其他库中，它返回对象是否为空的信息。

我想说的是‘*empty*’这个词本身就很有歧义，每个人对它的理解都不同。有人会认为这是一个“动作”，也有人会认为这是一个“信息查询”。这就是原因。

只有一个解决方法。不要在类名中用‘*empty*’。

4. 用*"erase"* 或*"clear"*给函数命名。我会更愿意用*"erase"*，因为*"clear"* 有点表述不清。
5. 用其他的名字来给提供信息的函数命名。“*isEmpty*”比如说。

如果你认为这不是什么大问题，那么请看这里。这是影响力比较广的错误。当然，要改变类似 *std::string* 这样的类有点迟了，但最少我们可以不让错误继续流传下去。

结论

希望你喜欢这个小贴士集。当然，不可能把所有关于没写好代码的东西都写下来，而且也没意义。我的目的是想提醒程序员，让他们有一种忧患意识。可能，下一次当某个程序员遇到某些奇怪的事，他能想起我的这些小贴士，然后就不慌张了。有时，花几分钟去研究文档或者写简单/干净的代码就能够避免隐藏的错误，不让这个错误在之后的几年内造成你的同事或用户的痛苦。