**内存调试技巧**

**C 语言最大难点揭秘**

本文将带您了解一些良好的和内存相关的编码实践，以将内存错误保持在控制范围内。内存错误是 C 和 C++ 编程的祸根：它们很普遍，认识其严重性已有二十多年，但始终没有彻底解决，它们可能严重影响应用程序，并且很少有开发团队对其制定明确的管理计划。但好消息是，它们并不怎么神秘。

0https://dw1.s81c.com/developerworks/i/v17/dw-cmts-arrow.png [评论](https://www.ibm.com/developerworks/cn/aix/library/au-memorytechniques.html#icomments)

[Cameron Laird](https://www.ibm.com/developerworks/cn/aix/library/au-memorytechniques.html#authorN1001B) ([claird@phaseit.net](mailto:claird@phaseit.net?subject=%E5%86%85%E5%AD%98%E8%B0%83%E8%AF%95%E6%8A%80%E5%B7%A7)), 副总裁, Phaseit Inc.

2007 年 6 月 21 日

* [expand](https://www.ibm.com/developerworks/cn/aix/library/au-memorytechniques.html#toggle)**内容**



在 IBM Bluemix 云平台上开发并部署您的下一个应用。

[**开始您的试用**](https://developer.ibm.com/sso/bmregistration?lang=zh_CN&ca=dwchina-_-bluemix-_-aix-au-memorytechniques-_-sidebar)

**引言**

C 和 C++ 程序中的内存错误非常有害：它们很常见，并且可能导致严重的后果。来自计算机应急响应小组（请参见[参考资料](https://www.ibm.com/developerworks/cn/aix/library/au-memorytechniques.html#resources)）和供应商的许多最严重的安全公告都是由简单的内存错误造成的。自从 70 年代末期以来，C 程序员就一直讨论此类错误，但其影响在 2007 年仍然很大。更糟的是，如果按我的思路考虑，当今的许多 C 和 C++ 程序员可能都会认为内存错误是不可控制而又神秘的顽症，它们只能纠正，无法预防。

但事实并非如此。本文将让您在短时间内理解与良好内存相关的编码的所有本质：

* [正确的内存管理的重要性](https://www.ibm.com/developerworks/cn/aix/library/au-memorytechniques.html#management)
* [内存错误的类别](https://www.ibm.com/developerworks/cn/aix/library/au-memorytechniques.html#categories)
* [内存编程的策略](https://www.ibm.com/developerworks/cn/aix/library/au-memorytechniques.html#strategies)
* [结束语](https://www.ibm.com/developerworks/cn/aix/library/au-memorytechniques.html#conclusion)

**正确的内存管理的重要性**

存在内存错误的 C 和 C++ 程序会导致各种问题。如果它们泄漏内存，则运行速度会逐渐变慢，并最终停止运行；如果覆盖内存，则会变得非常脆弱，很容易受到恶意用户的攻击。从 1988 年著名的*莫里斯蠕虫* 攻击到有关 Flash Player 和其他关键的零售级程序的最新安全警报都与缓冲区溢出有关：“大多数计算机安全漏洞都是缓冲区溢出”，Rodney Bates 在 2004 年写道。

在可以使用 C 或 C++ 的地方，也广泛支持使用其他许多通用语言（如 Java™、Ruby、Haskell、C#、Perl、Smalltalk 等），每种语言都有众多的爱好者和各自的优点。但是，从计算角度来看，每种编程语言优于 C 或 C++ 的主要优点都与便于内存管理密切相关。与内存相关的编程是如此重要，而在实践中正确应用又是如此困难，以致于它支配着面向对象编程语言、功能性编程语言、高级编程语言、声明性编程语言和另外一些编程语言的所有其他变量或理论。

与少数其他类型的常见错误一样，内存错误还是一种隐性危害：它们很难再现，症状通常不能在相应的源代码中找到。例如，无论何时何地发生内存泄漏，都可能表现为应用程序完全无法接受，同时内存泄漏不是显而易见。

因此，出于所有这些原因，需要特别关注 C 和 C++ 编程的内存问题。让我们看一看如何解决这些问题，先不谈是哪种语言。

**内存错误的类别**

首先，不要失去信心。有很多办法可以对付内存问题。我们先列出所有可能存在的实际问题：

* 内存泄漏
* 错误分配，包括大量增加 free() 释放的内存和未初始化的引用
* 悬空指针
* 数组边界违规

这是所有类型。即使迁移到 C++ 面向对象的语言，这些类型也不会有明显变化；无论数据是简单类型还是 C 语言的 struct 或 C++ 的类，C 和 C++ 中内存管理和引用的模型在原理上都是相同的。以下内容绝大部分是“纯 C”语言，对于扩展到 C++ 主要留作练习使用。

**内存泄漏**

在分配资源时会发生内存泄漏，但是它从不回收。下面是一个可能出错的模型（请参见[清单 1](https://www.ibm.com/developerworks/cn/aix/library/au-memorytechniques.html#list1)）：

**清单 1. 简单的潜在堆内存丢失和缓冲区覆盖**

void f1(char \*explanation)

{

char p1;

p1 = malloc(100);

(void) sprintf(p1,

"The f1 error occurred because of '%s'.",

explanation);

local\_log(p1);

}

您看到问题了吗？除非 local\_log() 对 free() 释放的内存具有不寻常的响应能力，否则每次对 f1 的调用都会泄漏 100 字节。在记忆棒增量分发数兆字节内存时，一次泄漏是微不足道的，但是连续操作数小时后，即使如此小的泄漏也会削弱应用程序。

在实际的 C 和 C++ 编程中，这不足以影响您对 malloc() 或 new 的使用，本部分开头的句子提到了“资源”不是仅指“内存”，因为还有类似以下内容的示例（请参见[清单 2](https://www.ibm.com/developerworks/cn/aix/library/au-memorytechniques.html#list2)）。FILE 句柄可能与内存块不同，但是必须对它们给予同等关注：

**清单 2. 来自资源错误管理的潜在堆内存丢失**

int getkey(char \*filename)

{

FILE \*fp;

int key;

fp = fopen(filename, "r");

fscanf(fp, "%d", &key);

return key;

}

fopen 的语义需要补充性的 fclose。在没有 fclose() 的情况下，C 标准不能指定发生的情况时，很可能是内存泄漏。其他资源（如信号量、网络句柄、数据库连接等）同样值得考虑。

**内存错误分配**

错误分配的管理不是很困难。下面是一个示例（请参见[清单 3](https://www.ibm.com/developerworks/cn/aix/library/au-memorytechniques.html#list3)）：

**清单 3. 未初始化的指针**

void f2(int datum)

{

int \*p2;

/\* Uh-oh! No one has initialized p2. \*/

\*p2 = datum;

...

}

关于此类错误的好消息是，它们一般具有显著结果。在 AIX® 下，对未初始化指针的分配通常会立即导致 *segmentation fault* 错误。它的好处是任何此类错误都会被快速地检测到；与花费数月时间才能确定且难以再现的错误相比，检测此类错误的代价要小得多。

在此错误类型中存在多个变种。free() 释放的内存比 malloc() 更频繁（请参见[清单 4](https://www.ibm.com/developerworks/cn/aix/library/au-memorytechniques.html#list4)）：

**清单 4. 两个错误的内存释放**

/\* Allocate once, free twice. \*/

void f3()

{

char \*p;

p = malloc(10);

...

free(p);

...

free(p);

}

/\* Allocate zero times, free once. \*/

void f4()

{

char \*p;

/\* Note that p remains uninitialized here. \*/

free(p);

}

这些错误通常也不太严重。尽管 C 标准在这些情形中没有定义具体行为，但典型的实现将忽略错误，或者快速而明确地对它们进行标记；总之，这些都是安全情形。

**悬空指针**

悬空指针比较棘手。当程序员在内存资源释放后使用资源时会发生悬空指针（请参见[清单 5](https://www.ibm.com/developerworks/cn/aix/library/au-memorytechniques.html#list5)）：

**清单 5. 悬空指针**

void f8()

{

struct x \*xp;

xp = (struct x \*) malloc(sizeof (struct x));

xp.q = 13;

...

free(xp);

...

/\* Problem! There's no guarantee that

the memory block to which xp points

hasn't been overwritten. \*/

return xp.q;

}

传统的“调试”难以隔离悬空指针。由于下面两个明显原因，它们很难再现：

* 即使影响提前释放内存范围的代码已本地化，内存的使用仍然可能取决于应用程序甚至（在极端情况下）不同进程中的其他执行位置。
* 悬空指针可能发生在以微妙方式使用内存的代码中。结果是，即使内存在释放后立即被覆盖，并且新指向的值不同于预期值，也很难识别出新值是错误值。

悬空指针不断威胁着 C 或 C++ 程序的运行状态。

**数组边界违规**

数组边界违规十分危险，它是内存错误管理的最后一个主要类别。回头看一下[清单 1](https://www.ibm.com/developerworks/cn/aix/library/au-memorytechniques.html#list1)；如果 explanation 的长度超过 80，则会发生什么情况？回答：难以预料，但是它可能与良好情形相差甚远。特别是，C 复制一个字符串，该字符串不适于为它分配的 100 个字符。在任何常规实现中，“超过的”字符会覆盖内存中的其他数据。内存中数据分配的布局非常复杂并且难以再现，所以任何症状都不可能追溯到源代码级别的具体错误。这些错误通常会导致数百万美元的损失。

**内存编程的策略**

勤奋和自律可以让这些错误造成的影响降至最低限度。下面我们介绍一下您可以采用的几个特定步骤；我在各种组织中处理它们的经验是，至少可以按一定的数量级持续减少内存错误。

**编码风格**

编码风格是最重要的，我还从没有看到过其他任何作者对此加以强调。影响资源（特别是内存）的函数和方法需要显式地解释本身。下面是有关标头、注释或名称的一些示例（请参见[清单 6](https://www.ibm.com/developerworks/cn/aix/library/au-memorytechniques.html#list6)）。

**清单 6. 识别资源的源代码示例**

/\*\*\*\*\*\*\*\*

\* ...

\*

\* Note that any function invoking protected\_file\_read()

\* assumes responsibility eventually to fclose() its

\* return value, UNLESS that value is NULL.

\*

\*\*\*\*\*\*\*\*/

FILE \*protected\_file\_read(char \*filename)

{

FILE \*fp;

fp = fopen(filename, "r");

if (fp) {

...

} else {

...

}

return fp;

}

/\*\*\*\*\*\*\*

\* ...

\*

\* Note that the return value of get\_message points to a

\* fixed memory location. Do NOT free() it; remember to

\* make a copy if it must be retained ...

\*

\*\*\*\*\*\*\*\*/

char \*get\_message()

{

static char this\_buffer[400];

...

(void) sprintf(this\_buffer, ...);

return this\_buffer;

}

/\*\*\*\*\*\*\*\*

\* ...

\* While this function uses heap memory, and so

\* temporarily might expand the over-all memory

\* footprint, it properly cleans up after itself.

\*

\*\*\*\*\*\*\*\*/

int f6(char \*item1)

{

my\_class c1;

int result;

...

c1 = new my\_class(item1);

...

result = c1.x;

delete c1;

return result;

}

/\*\*\*\*\*\*\*\*

\* ...

\* Note that f8() is documented to return a value

\* which needs to be returned to heap; as f7 thinly

\* wraps f8, any code which invokes f7() must be

\* careful to free() the return value.

\*

\*\*\*\*\*\*\*\*/

int \*f7()

{

int \*p;

p = f8(...);

...

return p;

}

使这些格式元素成为您日常工作的一部分。可以使用各种方法解决内存问题：

* 专用库
* 语言
* 软件工具
* 硬件检查器

在这整个领域中，我始终认为最有用并且投资回报率最大的是考虑改进源代码的风格。它不需要昂贵的代价或严格的形式；可以始终取消与内存无关的段的注释，但影响内存的定义当然需要显式注释。添加几个简单的单词可使内存结果更清楚，并且内存编程会得到改进。

我没有做受控实验来验证此风格的效果。如果您的经历与我一样，您将发现没有说明资源影响的策略简直无法忍受。这样做很简单，但带来的好处太多了。

**检测**

检测是编码标准的补充。二者各有裨益，但结合使用效果特别好。机灵的 C 或 C++ 专业人员甚至可以浏览不熟悉的源代码，并以极低的成本检测内存问题。通过少量的实践和适当的文本搜索，您能够快速验证平衡的 \*alloc() 和 free() 或者 new 和 delete 的源主体。人工查看此类内容通常会出现像[清单 7](https://www.ibm.com/developerworks/cn/aix/library/au-memorytechniques.html#list7) 中一样的问题。

**清单 7. 棘手的内存泄漏**

static char \*important\_pointer = NULL;

void f9()

{

if (!important\_pointer)

important\_pointer = malloc(IMPORTANT\_SIZE);

...

if (condition)

/\* Ooops! We just lost the reference

important\_pointer already held. \*/

important\_pointer = malloc(DIFFERENT\_SIZE);

...

}

如果 condition 为真，简单使用自动运行时工具不能检测发生的内存泄漏。仔细进行源分析可以从此类条件推理出证实正确的结论。我重复一下我写的关于风格的内容：尽管大量发布的内存问题描述都强调工具和语言，对于我来说，最大的收获来自“软的”以开发人员为中心的流程变更。您在风格和检测上所做的任何改进都可以帮助您理解由自动化工具产生的诊断。

**静态的自动语法分析**

当然，并不是只有人类才能读取源代码。您还应使*静态语法分析* 成为开发流程的一部分。静态语法分析是 lint、*严格编译* 和几种商业产品执行的内容：扫描编译器接受的源文本和目标项，但这可能是错误的症状。

希望让您的代码无 *lint*。尽管 lint 已过时，并有一定的局限性，但是，没有使用它（或其较高级的后代）的许多程序员犯了很大的错误。通常情况下，您能够编写忽略 lint 的优秀的专业质量代码，但努力这样做的结果通常会发生重大错误。其中一些错误影响内存的正确性。与让客户首先发现内存错误的代价相比，即使对这种类别的产品支付最昂贵的许可费也失去了意义。清除源代码。现在，即使 lint 标记的编码可能向您提供所需的功能，但很可能存在更简单的方法，该方法可满足 lint，并且比较强键又可移植。

**内存库**

补救方法的最后两个类别与前三个明显不同。前者是*轻量级* 的；一个人可以容易地理解并实现它们。另一方面，内存库和工具通常具有较高的许可费用，对部分开发人员来说，它们需要进一步完善和调整。有效地使用库和工具的程序员是理解轻量级的*静态* 方法的人员。可用的库和工具给人的印象很深：其作为组的质量很高。但是，即使最优秀的编程人员也可能会被忽略内存管理基本原则的非常任性的编程人员搅乱。据我观察，普通的编程人员在尝试利用内存库和工具进行隔离工作时也只能感到灰心。

由于这些原因，我们催促 C 和 C++ 程序员为解决内存问题先了解一下自己的源。在这完成之后，才去考虑库。

使用几个库能够编写常规的 C 或 C++ 代码，并保证改进内存管理。Jonathan Bartlett 在 developerWorks 的 2004 评论专栏中介绍了主要的候选项，可以在下面的[参考资料](https://www.ibm.com/developerworks/cn/aix/library/au-memorytechniques.html#resources)部分获得。库可以解决多种不同的内存问题，以致于直接对它们进行比较是非常困难的；这方面的常见主题包括*垃圾收集*、*智能指针* 和 *智能容器*。大体上说，库可以自动进行较多的内存管理，这样程序员可以犯更少的错误。

我对内存库有各种感受。他们在努力工作，但我看到他们在项目中获得的成功比预期要小，尤其在 C 方面。我尚未对这些令人失望的结果进行仔细分析。例如，业绩应该与相应的*手动* 内存管理一样好，但是这是一个灰色区域——尤其在垃圾收集库处理速度缓慢的情况下。通过这方面的实践得出的最明确的结论是，与 C 关注的代码组相比，C++ 似乎可以较好地接受智能指针。

**内存工具**

开发真正基于 C 的应用程序的开发团队需要运行时内存工具作为其开发策略的一部分。已介绍的技术很有价值，而且不可或缺。在您亲自尝试使用内存工具之前，其质量和功能您可能还不了解。

本文主要讨论了基于软件的内存工具。还有硬件内存调试器；在非常特殊的情况下（主要是在使用不支持其他工具的专用主机时）才考虑它们。

市场上的软件内存工具包括专有工具（如 IBM Rational® Purify 和 Electric Fence）和其他开放源代码工具。其中有许多可以很好地与 AIX 和其他操作系统一起使用。

所有内存工具的功能基本相同：构建可执行文件的特定版本（很像在编译时通过使用 -g 标记生成的调试版本）、练习相关应用程序和研究由工具自动生成的报告。请考虑如[清单 8](https://www.ibm.com/developerworks/cn/aix/library/au-memorytechniques.html#list8) 所示的程序。

**清单 8. 示例错误**

int main()

{

char p[5];

strcpy(p, "Hello, world.");

puts(p);

}

此程序可以在许多环境中“运行”，它编译、执行并将“Hello, world.\n”打印到屏幕。使用内存工具运行相同应用程序会在第四行产生一个数组边界违规的报告。在了解软件错误（将十四个字符复制到了只能容纳五个字符的空间中）方面，这种方法比在客户处查找错误症状的花费小得多。这是内存工具的功劳。

**结束语**

作为一名成熟的 C 或 C++ 程序员，您认识到内存问题值得特别关注。通过制订一些计划和实践，可以找到控制内存错误的方法。学习内存使用的正确模式，快速发现可能发生的错误，使本文介绍的技术成为您日常工作的一部分。您可以在开始时就消除应用程序中的症状，否则可能要花费数天或数周时间来调试。