


C 语言编程中的 5 个常见错误及对应解决方案

C语言与CPP编程 昨天

以下文章来源于Linux中国，作者邀你一起成为开源贡献者



Linux中国
[Linux中国](https://linux.cn/)开源社区官方公众号。专注于开源技术研究、开源思想...



导读：增强 C 语言程序的弹性和可靠性的五种方法。

本文字数：8391 · 阅读时长大约：**10**分钟

<https://linux.cn/article-13894-1.html>
作者：Jim Hall
译者：unigeorge

即使是最好的程序员也无法完全避免错误。这些错误可能会引入安全漏洞、导致程序崩溃或产生意外操作，具体影响要取决于程序的运行逻辑。

C 语言有时名声不太好，因为它不像近期的编程语言（比如 Rust）那样具有内存安全性。但是通过额外的代码，一些最常见和严重的 C 语言错误是可以避免的。下文讲解了可能影响应用程序的五个错误以及避免它们的方法：



1、未初始化的变量

程序启动时，系统会为其分配一块内存以供存储数据。这意味着程序启动时，变量将获得内存中的一个随机值。

有些编程环境会在程序启动时特意将内存“清零”，因此每个变量都得以有初始的零值。程序中的变量都以零值作为初始值，听上去是很不错的。但是在 C 编程规范中，系统并不会初始化变量。

看一下这个使用了若干变量和两个数组的示例程序：

```
1.  #include <stdio.h>
2.  #include <stdlib.h>
3.
4.  int
5.  main()
6.  {
7.      int i, j, k;
8.      int numbers[5];
9.      int *array;
10.
11.     puts("These variables are not initialized:");
12.
13.     printf("  i = %d\n", i);
14.     printf("  j = %d\n", j);
15.     printf("  k = %d\n", k);
16.
17.     puts("This array is not initialized:");
18.
19.     for (i = 0; i < 5; i++) {
20.         printf("  numbers[%d] = %d\n", i, numbers[i]);
21.     }
22.
23.     puts("malloc an array ...");
24.     array = malloc(sizeof(int) * 5);
25.
26.     if (array) {
27.         puts("This malloc'ed array is not initialized:");
28.
29.         for (i = 0; i < 5; i++) {
30.             printf("  array[%d] = %d\n", i, array[i]);
31.         }
32.
33.         free(array);
34.     }
35.
36.     /* done */
37.
38.     puts("Ok");
39.     return 0;
40. }
```

这个程序不会初始化变量，所以变量以系统内存中的随机值作为初始值。在我的 Linux 系统上编译和运行这个程序，会看到一些变量恰巧有“零”值，但其他变量并没有：

```
1. These variables are not initialized:
2.   i = 0
3.   j = 0
4.   k = 32766
5. This array is not initialized:
6.   numbers[0] = 0
7.   numbers[1] = 0
8.   numbers[2] = 4199024
9.   numbers[3] = 0
10.  numbers[4] = 0
11. malloc an array ...
12. This malloc'ed array is not initialized:
13.   array[0] = 0
14.   array[1] = 0
15.   array[2] = 0
16.   array[3] = 0
17.   array[4] = 0
18. Ok
```

很幸运，`i` 和 `j` 变量是从零值开始的，但 `k` 的起始值为 32766。在 `numbers` 数组中，大多数元素也恰好从零值开始，只有第三个元素的初始值为 4199024。

在不同的系统上编译相同的程序，可以进一步显示未初始化变量的危险性。不要误以为“全世界都在运行 Linux”，你的程序很可能某天在其他平台上运行。例如，下面是在 FreeDOS 上运行相同程序的结果：

```
1. These variables are not initialized:
2.   i = 0
3.   j = 1074
4.   k = 3120
5. This array is not initialized:
6.   numbers[0] = 3106
7.   numbers[1] = 1224
8.   numbers[2] = 784
9.   numbers[3] = 2926
10.  numbers[4] = 1224
11. malloc an array ...
12. This malloc'ed array is not initialized:
13.   array[0] = 3136
14.   array[1] = 3136
15.   array[2] = 14499
16.   array[3] = -5886
17.   array[4] = 219
18. Ok
```

永远都要记得初始化程序的变量。如果你想让变量将以零值作为初始值，请额外添加代码将零分配给该变量。预先编好这些额外的代码，这会有助于减少日后让人头疼的调试过程。



2、数组越界

C 语言中，数组索引从零开始。这意味着对于长度为 10 的数组，索引是从 0 到 9；长度为 1000 的数组，索引则是从 0 到 999。

程序员有时会忘记这一点，他们从索引 1 开始引用数组，产生了“大小差一”^(off by one) 错误。在长度为 5 的数组中，程序员在索引“5”处使用的值，实际上并不是数组的第 5 个元素。相反，它是内存中的一些其他值，根本与此数组无关。

这是一个数组越界的示例程序。该程序使用了一个只含有 5 个元素的数组，但却引用了该范围之外的数组元素：

```
1.  #include <stdio.h>
2.  #include <stdlib.h>
3.
4.  int
5.  main()
6.  {
7.      int i;
8.      int numbers[5];
9.      int *array;
10.
11.     /* test 1 */
12.
13.     puts("This array has five elements (0 to 4)");
14.
15.     /* initialize the array */
16.     for (i = 0; i < 5; i++) {
17.         numbers[i] = i;
18.     }
19.
20.     /* oops, this goes beyond the array bounds: */
21.     for (i = 0; i < 10; i++) {
22.         printf(" numbers[%d] = %d\n", i, numbers[i]);
23.     }
24.
25.     /* test 2 */
26.
27.     puts("malloc an array ...");
28.
29.     array = malloc(sizeof(int) * 5);
30.
31.     if (array) {
32.         puts("This malloc'ed array also has five elements (0 to 4)");
33.
34.         /* initialize the array */
35.         for (i = 0; i < 5; i++) {
36.             array[i] = i;
37.         }
38.
39.         /* oops, this goes beyond the array bounds: */
40.         for (i = 0; i < 10; i++) {
41.             printf(" array[%d] = %d\n", i, array[i]);
42.         }
43.
44.         free(array);
45.     }
46.
47.     /* done */
48.
49.     puts("Ok");
50.     return 0;
51. }
```

可以看到，程序初始化了数组的所有值（从索引 0 到 4），然后从索引 0 开始读取，结尾是索引 9 而不是索引 4。前五个值是正确的，再后面的值会让你不知所措：

```
1. This array has five elements (0 to 4)
2.   numbers[0] = 0
3.   numbers[1] = 1
4.   numbers[2] = 2
5.   numbers[3] = 3
6.   numbers[4] = 4
7.   numbers[5] = 0
8.   numbers[6] = 4198512
9.   numbers[7] = 0
10.  numbers[8] = 1326609712
11.  numbers[9] = 32764
12. malloc an array ...
13. This malloc'ed array also has five elements (0 to 4)
14.   array[0] = 0
15.   array[1] = 1
16.   array[2] = 2
17.   array[3] = 3
18.   array[4] = 4
19.   array[5] = 0
20.   array[6] = 133441
21.   array[7] = 0
22.   array[8] = 0
23.   array[9] = 0
24. Ok
```

引用数组时，始终要记得追踪数组大小。将数组大小存储在变量中；不要对数组大小进行（hard-code）硬编码。否则，如果后期该标识符指向另一个不同大小的数组，却忘记更改硬编码的数组长度时，程序就可能会发生数组越界。



3、字符串溢出

字符串只是特定类型的数组。在 C 语言中，字符串是一个由 `char` 类型值组成的数组，其中一个零字符表示字符串的结尾。

因此，与数组一样，要注意避免超出字符串的范围。有时也称之为 **字符串溢出**。

使用 `gets` 函数读取数据是一种很容易发生字符串溢出的行为方式。`gets` 函数非常危险，因为它不知道在一个字符串中可以存储多少数据，只会机械地从用户那里读取数据。如果用户输入像 `foo` 这样的短字符串，不会发生意外；但是当用户输入的值超过字符串长度时，后果可能是灾难性的。

下面是一个使用 `gets` 函数读取城市名称的示例程序。在这个程序中，我还添加了一些未使用的变量，来展示字符串溢出对其他数据的影响：

```

1.  #include <stdio.h>
2.  #include <string.h>
3.
4.  int
5.  main()
6.  {
7.      char name[10];                /* Such as "Chicago" */
8.      int var1 = 1, var2 = 2;
9.
10.     /* show initial values */
11.
12.     printf("var1 = %d; var2 = %d\n", var1, var2);
13.
14.     /* this is bad .. please don't use gets */
15.
16.     puts("Where do you live?");
17.     gets(name);
18.
19.     /* show ending values */
20.
21.     printf("<%s> is length %d\n", name, strlen(name));
22.     printf("var1 = %d; var2 = %d\n", var1, var2);
23.
24.     /* done */
25.
26.     puts("Ok");
27.     return 0;
28. }

```

当你测试类似的短城市名称时，该程序运行良好，例如伊利诺伊州的 `Chicago` 或北卡罗来纳州的 `Raleigh`：

```

1.  var1 = 1; var2 = 2
2.  Where do you live?
3.  Raleigh
4.  <Raleigh> is length 7
5.  var1 = 1; var2 = 2
6.  Ok

```

威尔士的小镇 `Llanfairpwllgwyngyllgogerychwyrndrobwlllllantysiliogogogoch` 有着世界上最长的名字之一。这个字符串有 58 个字符，远远超出了 `name` 变量中保留的 10 个字符。结果，程序将值存储在内存的其他区域，覆盖了 `var1` 和 `var2` 的值：

```

1.  var1 = 1; var2 = 2
2.  Where do you live?
3.  Llanfairpwllgwyngyllgogerychwyrndrobwlllllantysiliogogogoch
4.  <Llanfairpwllgwyngyllgogerychwyrndrobwlllllantysiliogogogoch> is length 58
5.  var1 = 2036821625; var2 = 2003266668
6.  Ok

```



```
7. | Segmentation fault (core dumped)
```

在运行结束之前，程序会用长字符串覆盖内存的其他部分区域。注意，`var1` 和 `var2` 的值不再是起始的 1 和 2。

避免使用 `gets` 函数，改用更安全的方法来读取用户数据。例如，`getline` 函数会分配足够的内存来存储用户输入，因此不会因输入长值而发生意外的字符串溢出。



4、重复释放内存

“分配的内存要手动释放”是良好的 C 语言编程原则之一。程序可以使用 `malloc` 函数为数组和字符串分配内存，该函数会开辟一块内存，并返回一个指向内存中起始地址的指针。之后，程序可以使用 `free` 函数释放内存，该函数会使用指针将内存标记为未使用。

但是，你应该只使用一次 `free` 函数。第二次调用 `free` 会导致意外的后果，可能会毁掉你的程序。下面是一个针对此点的简短示例程序。程序分配了内存，然后立即释放了它。但为了模仿一个健忘但有条理的程序员，我在程序结束时又一次释放了内存，导致两次释放了相同的内存：

```
1.  #include <stdio.h>
2.  #include <stdlib.h>
3.
4.  int
5.  main()
6.  {
7.      int *array;
8.
9.      puts("malloc an array ...");
10.
11.     array = malloc(sizeof(int) * 5);
12.
13.     if (array) {
14.         puts("malloc succeeded");
15.
16.         puts("Free the array...");
17.         free(array);
18.     }
19.
20.     puts("Free the array...");
21.     free(array);
22.
23.     puts("Ok");
24. }
```

运行这个程序会导致第二次使用 `free` 函数时出现戏剧性的失败：

```

1. malloc an array ...
2. malloc succeeded
3. Free the array...
4. Free the array...
5. free(): double free detected in tcache 2
6. Aborted (core dumped)

```

要记得避免在数组或字符串上多次调用 `free`。将 `malloc` 和 `free` 函数定位在同一个函数中，这是避免重复释放内存的一种方法。

例如，一个纸牌游戏程序可能会在主函数中为一副牌分配内存，然后在其他函数中使用这副牌来玩游戏。记得在主函数，而不是其他函数中释放内存。将 `malloc` 和 `free` 语句放在一起有助于避免多次释放内存。



5、使用无效的文件指针

文件是一种便捷的数据存储方式。例如，你可以将程序的配置数据存储在 `config.dat` 文件中。Bash shell 会从用户家目录中的 `.bash_profile` 读取初始化脚本。GNU Emacs 编辑器会寻找文件 `.emacs` 以从中确定起始值。而 Zoom 会议客户端使用 `zoomus.conf` 文件读取其程序配置。

所以，从文件中读取数据的能力几乎对所有程序都很重要。但是假如要读取的文件不存在，会发生什么呢？

在 C 语言中读取文件，首先要用 `fopen` 函数打开文件，该函数会返回指向文件的流指针。你可以结合其他函数，使用这个指针来读取数据，例如 `fgetc` 会逐个字符地读取文件。

如果要读取的文件不存在或程序没有读取权限，`fopen` 函数会返回 `NULL` 作为文件指针，这表示文件指针无效。但是这里有一个示例程序，它机械地直接去读取文件，不检查 `fopen` 是否返回了 `NULL`：

```

1. #include <stdio.h>
2.
3. int
4. main()
5. {
6.     FILE *pfile;
7.     int ch;
8.
9.     puts("Open the FILE.TXT file ...");
10.
11.     pfile = fopen("FILE.TXT", "r");
12.
13.     /* you should check if the file pointer is valid, but we skipped that */
14.

```



```
15. puts("Now display the contents of FILE.TXT ...");
16.
17. while ((ch = fgetc(pfile)) != EOF) {
18.     printf("<%c>", ch);
19. }
20.
21. fclose(pfile);
22.
23. /* done */
24.
25. puts("Ok");
26. return 0;
27. }
```

当你运行这个程序时，第一次调用 `fgetc` 会失败，程序会立即中止：

```
1. Open the FILE.TXT file ...
2. Now display the contents of FILE.TXT ...
3. Segmentation fault (core dumped)
```

始终检查文件指针以确保其有效。例如，在调用 `fopen` 打开一个文件后，用类似 `if (pfile != NULL)` 的语句检查指针，以确保指针是可以使用的。

人都会犯错，最优秀的程序员也会产生编程错误。但是，遵循上面这些准则，添加一些额外的代码来检查这五种类型的错误，就可以避免最严重的 C 语言编程错误。提前编写几行代码来捕获这些错误，可能会帮你节省数小时的调试时间。

via: <https://opensource.com/article/21/10/programming-bugs>

作者： [Jim Hall](#) 选题： [lujun9972](#) 译者： [unigeorge](#) 校对： [wxy](#)

本文由 [LCTT](#) 原创编译



C 语言与 C++ 编程

分享 C 语言 / C++，数据结构与算法，计算机基础，操作系统等
51 篇原创内容



公众号

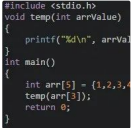
[阅读原文](#)

喜欢此内容的人还喜欢

基于C语言的最优HSM状态机架构实现
技术让梦想更伟大



整理的比较全面的C语言入门笔记！
STM32嵌入式开发



C 语言 PK 各大编程语言
C语言与C++编程

