

写 C 时遇到的一个小问题

2025-08-23 · Journals · Programming, C

■ 数组越界不报错？有点意思

起因

```
#include <stdio.h>

int main(void)
{
    int sum = 0, i = 0;
    char input[5];

    while (1) {
        sum = 0;
        scanf("%s", input);
        for (i = 0; input[i] != '\0'; i++)
            sum = sum*10 + input[i] - '0';
        printf("input=%d\n", sum);
    }
    return 0;
}
```

在浏览 [Linux C 编程一站式学习 第十章 gdb 3. 观察点](#) 时遇到了上述代码。这段代码目的很简单：把从输入设备输入的整数字符串转换为整数并输出。原文采用了以下调试步骤发现了问题：

```
$ ./main
123
input=123
67
input=67
12345
input=123407
```

原文的解释是：在内存中，局部变量 `i` 紧跟在 `input[4]` 后，所以 `input[5]` 指的就是局部变量 `i`。而从键盘输入了 `12345`，分别给 `input` 的各个元素赋值，便成了：

```
input[0] = '1';
input[1] = '2';
input[2] = '3';
input[3] = '4';
input[4] = '5';
i = '\0';
```

这里的 `i` 被赋值为 `'\0'`，因为键盘读取了一行字符串，而字符串以 `'\0'` 结尾。C 语言的 `scanf()` 函数不会读取空白字符，所以末尾不包含 `\n`。

注意到：

```
for (i = 0; input[i] != '\0'; i++)
```

`for` 循环的控制条件是 `input[i] != '\0'`，而这个数组并不包含 `'\0'`，因此出现了访问越界的情况。

原文使用了 GDB 调试，部分调试信息如下：

```
(gdb) n
11      for (i = 0; input[i] != '\0'; i++)
(gdb) p sum
$3 = 12345
(gdb) n
12      sum = sum*10 + input[i] - '\0';
(gdb) x/7b input
0xbfb8f0a7: 0x31 0x32 0x33 0x34 0x35 0x05 0x00
```

i 后面一个地址位置的值是 0x00，因此最后一次循环执行了 $12345 \times 10 + 0x05 - '\0'$ ，得到了 123407。

然而，事情真的有这么简单吗？

发现问题

我在使用 CLion 调试上述代码，正常得很！

```
12345
input=12345
114514
input=114514
1919810
input=1919810
^C
```

怎么回事呢？

看了一眼原文的上一页，发现原作者是这么编译运行的：

```
$ gcc main.c -g -o main
$ ./main
```

行吧，我也这么办。

？

怪事，一切正常啊。难道这个 bug 这么没有鲁棒性？

这时我有点摸不着头脑，看一眼我的 GCC 和 Linux 版本：

```
$ uname -a
Linux sakimidare-arch 6.16.2-arch1-1 #1 SMP PREEMPT_DYNAMIC Wed, 20 Aug 2025 21:43:45 +0000
x86_64 GNU/Linux
```

```
$ gcc --version
gcc (GCC) 15.2.1 20250813
Copyright © 2025 Free Software Foundation, Inc.
本程序是自由软件；请参看源代码的版权声明。本软件没有任何担保；
包括没有适销性和某一专用目的下的适用性担保。
```

好嘛，再试试其他编译器呢？总不会这个 Bug 到现代机器上不复存在了吧……

```
$ clang main.c -o main
$ ./main
12345
input=123407
666666
input=666617
123456789
```

```
input=123407
114514
input=114467
```

问题出现了！有时出错的代码比正确而不可靠的代码更有意义。

经常听到有人调侃道：代码跑起来就不要去管它了。这似乎是生产环境下的无奈之举。毕竟人是要吃饭的，谁也不可能抱着一段正常运行的代码研究一辈子。如果把时间全用来研究一段代码，程序带来的效率收益与投入的时间成本相比可能并不划算。

但与生产不一样的是，我们是在学习。在学习一门语言，理解一门语言时如果不深入了解每一行代码的意义，用“程序跑起来就不用管它”来麻痹自己，那么程序里必然有我们发现不了的隐患，生产中有所谓“我这边能跑啊，你那边环境没配对吧”的借口，学习中有知其然而不知其所以然的糟糕态度。

这些看似微不足道的怠惰悄悄构成了思维上的不完备，让我们不习惯于全面地研究问题。长年累月下去，只会追悔莫及。

为什么会这样？

好了好了扯远啦，我们来看看为什么这两种编译器编译出来的程序有不同的行为。

我们知道，从一个 `.c` 源代码文件到可执行程序共分为四步：

1. 预处理 (Preprocessing)
2. 编译 (Compilation)
3. 汇编 (Assembling)
4. 链接 (Linking)

0. 预处理

预处理是指

1. 处理 `#include` 展开头文件；
2. 处理 `#define` 替换宏；
3. 处理条件编译指令 `#if` `#ifdef` `#ifndef`；
4. 删除注释。

用 `gcc` 和 `clang` 分别预处理这个 `main.c`，看看生成的预处理文件有什么不一样吧！

```
$ gcc -E main.c -o gcc.i
$ clang -E main.c -o clang.i
```

发现 `main()` 函数部分代码一样，都是：

```
int main(void)
{
    int sum = 0, i = 0;
    char input[5];

    while (1) {
        sum = 0;
        scanf("%s", input);
        for (i = 0; input[i] != '\0'; i++)
            sum = sum*10 + input[i] - '0';
        printf("input=%d\n", sum);
    }
}
```

```
    return 0;
}
```

这也符合我们的认知，因为预处理只是进行了替换操作，不涉及修改函数的逻辑。

在 `main()` 上方有八百多行代码，两个编译器处理后的文件不一样。不过我们先不去管它，因为这个问题出现的主要原因是 `input` 数组和 `i` 的位置相邻。根据我们的直觉，问题不在头文件。

先把这两个文件放在一边，我们继续。

1. 编译

此处的汇编语言是 x86-64 GNU 汇编语言，Windows 无法直接运行。

编译是指把预处理后的 C 代码翻译成汇编代码。这一步包括语法检查、语义分析、优化等，于是我们有理由怀疑编译器在这一步做了不一样的操作，导致汇编逻辑不一样。

出发吧！

```
$ gcc -S gcc.i -o gcc.s
$ clang -S clang.i -o clang.s
```

把这两个文件都贴出来：

```
.file "main.c"
.text
.section .rodata
.LC0:
.string "%s"
.LC1:
.string "input=%d\n"
.text
.globl main
.type main, @function
main:
.LFB0:
.cfi_startproc
pushq %rbp
.cfi_def_cfa_offset 16
.cfi_offset 6, -16
movq %rsp, %rbp
.cfi_def_cfa_register 6
subq $32, %rsp
movq %fs:40, %rax
movq %rax, -8(%rbp)
xorl %eax, %eax
movl $0, -24(%rbp)
movl $0, -20(%rbp)
.L4:
movl $0, -24(%rbp)
leaq -13(%rbp), %rax
leaq .LC0(%rip), %rdx
movq %rax, %rsi
movq %rdx, %rdi
movl $0, %eax
call __isoc23_scanf@PLT
movl $0, -20(%rbp)
```

```

    jmp .L2
.L3:
    movl    -24(%rbp), %edx
    movl    %edx, %eax
    sall    $2, %eax
    addl    %edx, %eax
    addl    %eax, %eax
    movl    %eax, %edx
    movl    -20(%rbp), %eax
    cltq
    movzbl  -13(%rbp,%rax), %eax
    movsbl  %al, %eax
    addl    %edx, %eax
    subl    $48, %eax
    movl    %eax, -24(%rbp)
    addl    $1, -20(%rbp)
.L2:
    movl    -20(%rbp), %eax
    cltq
    movzbl  -13(%rbp,%rax), %eax
    testb   %al, %al
    jne     .L3
    movl    -24(%rbp), %eax
    leaq    .LC1(%rip), %rdx
    movl    %eax, %esi
    movq    %rdx, %rdi
    movl    $0, %eax
    call    printf@PLT
    jmp     .L4
    .cfi_endproc
.LFE0:
    .size   main, .-main
    .ident  "GCC: (GNU) 15.2.1 20250813"
    .section .note.GNU-stack,"",@progbits

    .file   "main.c"
    .text
    .globl  main                                # -- Begin function main
    .p2align 4
    .type   main,@function

main:                                           # @main
    .cfi_startproc
# %bb.0:
    pushq   %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset %rbp, -16
    movq    %rsp, %rbp
    .cfi_def_cfa_register %rbp
    subq    $32, %rsp
    movl    $0, -4(%rbp)
    movl    $0, -8(%rbp)
    movl    $0, -12(%rbp)
.LBB0_1:                                       # =>This Loop Header: Depth=1

```

```

                                #      Child Loop BB0_2 Depth 2
movl  $0, -8(%rbp)
leaq  -17(%rbp), %rsi
leaq  .L.str(%rip), %rdi
movb  $0, %al
callq __isoc99_scanf@PLT
movl  $0, -12(%rbp)
.LBB0_2:                                #      Parent Loop BB0_1 Depth=1
                                #  =>  This Inner Loop Header: Depth=2
    movslq  -12(%rbp), %rax
    movsbl  -17(%rbp,%rax), %eax
    cmpl  $0, %eax
    je  .LBB0_5
# %bb.3:                                #      in Loop: Header=BB0_2 Depth=2
    imull  $10, -8(%rbp), %eax
    movslq  -12(%rbp), %rcx
    movsbl  -17(%rbp,%rcx), %ecx
    addl  %ecx, %eax
    subl  $48, %eax
    movl  %eax, -8(%rbp)
# %bb.4:                                #      in Loop: Header=BB0_2 Depth=2
    movl  -12(%rbp), %eax
    addl  $1, %eax
    movl  %eax, -12(%rbp)
    jmp  .LBB0_2
.LBB0_5:                                #      in Loop: Header=BB0_1 Depth=1
    movl  -8(%rbp), %esi
    leaq  .L.str.1(%rip), %rdi
    movb  $0, %al
    callq printf@PLT
    jmp  .LBB0_1
.Lfunc_end0:
    .size main, .Lfunc_end0-main
    .cfi_endproc
                                # -- End function
    .type .L.str,@object                                # @.str
    .section .rodata.str1.1,"aMS",@progbits,1
.L.str:
    .asciz  "%s"
    .size .L.str, 3

    .type .L.str.1,@object                                # @.str.1
.L.str.1:
    .asciz  "input=%d\n"
    .size .L.str.1, 10

    .ident  "clang version 20.1.8"
    .section ".note.GNU-stack","",@progbits
    .addrsig
    .addrsig_sym __isoc99_scanf
    .addrsig_sym printf

```

来看看 gcc.s :

```

.LFB0:
.cfi_startproc
pushq %rbp
.cfi_def_cfa_offset 16
.cfi_offset 6, -16
movq %rsp, %rbp
.cfi_def_cfa_register 6
subq $32, %rsp          # 预留 32 字节栈帧给局部变量
movq %fs:40, %rax
movq %rax, -8(%rbp)
xorl %eax, %eax
movl $0, -24(%rbp)      # sum = 0;
movl $0, -20(%rbp)      # i = 0;
.L4:
movl $0, -24(%rbp)      # sum = 0;
leaq -13(%rbp), %rax     # input[0] 的位置在-13(%rbp)
...

```

再看看 `clang.s` 是如何处理的:

```

main:                                # @main
.cfi_startproc
# %bb.0:
pushq %rbp
.cfi_def_cfa_offset 16
.cfi_offset %rbp, -16
movq %rsp, %rbp
.cfi_def_cfa_register %rbp
subq $32, %rsp          # 预留 32 字节栈帧给局部变量
movl $0, -4(%rbp)        # 返回值临时保留位, 本程序未使用
movl $0, -8(%rbp)        # sum = 0;
movl $0, -12(%rbp)       # i = 0;
.LBB0_1:                  # =>This Loop Header: Depth=1
                          #      Child Loop BB0_2 Depth 2
movl $0, -8(%rbp)        # sum = 0;
leaq -17(%rbp), %rsi     # input[0] 的位置在-17(%rbp)
...

```

好啦, 这下就清楚了!

我们来画一下栈:

GCC 栈

位置	变量
-9	input[4]
-10	input[3]
-11	input[2]
-12	input[1]
-13	input[0]
...	...
-20	i
-24	sum

Clang 栈

位置	变量
-8	sum
-12	i
-13	input[4]
-14	input[3]
-15	input[2]
-16	input[1]
-17	input[0]

因此，我们得出了结论：

GCC 为 `i` 和 `input[0]` 之间留足了栈帧，并且 `input[4]` 之后也没有变量可以影响循环，因此没出问题。而 Clang 让 `input[4]` 和 `i` 紧靠在一起，增加了数组越界的风险。

2. 汇编

3. 链接

哎呀这两个标题和本文没关系，加上只是为了目录更好看（

验证猜想

我们用 GCC 看看 `input[-7]`？按道理就是 `i` 了吧！修改程序为

```
#include <stdio.h>
```

```
int main(void)
{
    int sum = 0, i = 0;
    char input[5];

    while (1) {
        sum = 0;
        scanf("%s", input);
        for (i = 0; input[i] != '\0'; i++)
            sum = sum*10 + input[i] - '0';
        printf("input=%d\n", sum);
        printf("%d\n", input[-7]);
    }
    return 0;
}
```

运行程序：

```
$ gcc test.c -o test
$ ./test
12345
input=12345
5
```

大功告成！果然，`input[-7]` 就是 `i`！

如何规避风险?

| -00 了吗? -fsanitize=address 了吗? 快加上!

一位群友如是说。好吧好吧, 我们加上这两个参数再编译一次试试:

```
$ gcc -00 -fsanitize=address test.c -o test
```

```
$ ./test
```

```
1234567
```

```
=====
```

```
==16748==ERROR: AddressSanitizer: stack-buffer-overflow on address 0x7b4c18f00025 at pc  
0x7f4c1ba6e51d bp 0x7ffdce2744c0 sp 0x7ffdce273c48
```

```
WRITE of size 8 at 0x7b4c18f00025 thread T0
```

```
#0 0x7f4c1ba6e51c in scanf_common /usr/src/debug/gcc/gcc/libsanitizer/sanitizer_common/  
sanitizer_common_interceptors_format.inc:342
```

```
#1 0x7f4c1ba8edee in __isoc23_vscanf /usr/src/debug/gcc/gcc/libsanitizer/  
sanitizer_common/sanitizer_common_interceptors.inc:1554
```

```
#2 0x7f4c1ba8f5f5 in __isoc23_scanf /usr/src/debug/gcc/gcc/libsanitizer/  
sanitizer_common/sanitizer_common_interceptors.inc:1584
```

```
#3 0x564e475b9254 in main (/home/sakimidare/CLionProjects/c_study/test+0x1254) (BuildId:  
35fabe8824d13d3c1ca4e2836107a3b16992a4a9)
```

```
#4 0x7f4c1b627674 (/usr/lib/libc.so.6+0x27674) (BuildId:  
4fe011c94a88e8aeb6f2201b9eb369f42b4a1e9e)
```

```
#5 0x7f4c1b627728 in __libc_start_main (/usr/lib/libc.so.6+0x27728) (BuildId:  
4fe011c94a88e8aeb6f2201b9eb369f42b4a1e9e)
```

```
#6 0x564e475b90d4 in _start (/home/sakimidare/CLionProjects/c_study/test+0x10d4)  
(BuildId: 35fabe8824d13d3c1ca4e2836107a3b16992a4a9)
```

```
Address 0x7b4c18f00025 is located in stack of thread T0 at offset 37 in frame
```

```
#0 0x564e475b91b8 in main (/home/sakimidare/CLionProjects/c_study/test+0x11b8) (BuildId:  
35fabe8824d13d3c1ca4e2836107a3b16992a4a9)
```

```
This frame has 1 object(s):
```

```
[32, 37) 'input' (line 6) ← Memory access at offset 37 overflows this variable
```

```
HINT: this may be a false positive if your program uses some custom stack unwind mechanism,  
swapcontext or vfork
```

```
(longjmp and C++ exceptions *are* supported)
```

```
SUMMARY: AddressSanitizer: stack-buffer-overflow (/home/sakimidare/CLionProjects/c_study/  
test+0x1254) (BuildId: 35fabe8824d13d3c1ca4e2836107a3b16992a4a9) in main
```

```
Shadow bytes around the buggy address:
```

```
0x7b4c18effd80: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

```
0x7b4c18effe00: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

```
0x7b4c18effe80: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

```
0x7b4c18efff00: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

```
0x7b4c18efff80: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

```
=>0x7b4c18f00000: f1 f1 f1 f1[05]f3 f3 00 00 00 00 00 00 00 00 00 00
```

```
0x7b4c18f00080: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

```
0x7b4c18f00100: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

```
0x7b4c18f00180: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

```
0x7b4c18f00200: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

```
0x7b4c18f00280: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

```
Shadow byte legend (one shadow byte represents 8 application bytes):
```

```
Addressable: 00
```

```
Partially addressable: 01 02 03 04 05 06 07
```

```
Heap left redzone:      fa
Freed heap region:     fd
Stack left redzone:    f1
Stack mid redzone:     f2
Stack right redzone:   f3
Stack after return:    f5
Stack use after scope: f8
Global redzone:        f9
Global init order:     f6
Poisoned by user:      f7
Container overflow:    fc
Array cookie:          ac
Intra object redzone:  bb
ASan internal:         fe
Left alloca redzone:   ca
Right alloca redzone:  cb
==16748==ABORTING
```

看得出来，加上参数确实有助于规避数组越界风险。

不过，最有效的方法还是事先考虑好所有情况，防范任何可能出现的 Bug! —(酒吧点炒饭.txt)

写在最后

这是我第一次写这种类型的文章，算是对自己独立解决问题能力的一次检验吧！

也不知道会不会有人看这篇文章，当作日记得了。如果有人看到这里，感谢大家阅读！