

浙江大学

探索实验 1：对计算机编程语言的探索与实验分析

课程名称：汇编与接口

实验名称：汇编视角下 C 语言内存越界和缓冲溢出的分析和应用

姓 名：高铭健

学 院：计算机学院

系：图灵 2101

专 业：计算机科学与技术

学 号：3210102322

指导教师：蔡铭

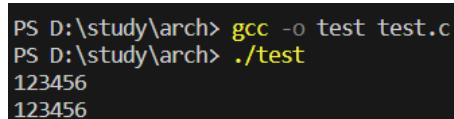
2023 年 11 月 8 日

1. 背景说明

在编写C语言程序时，遇到最多也最我头疼的错误就是段错误，也就是内存的越界访问。在最初学习C语言的时候，我发现了一件当时感到很神奇的事，对于C语言 `gets()` 函数的使用（比如下面的 `test.c` 程序）：

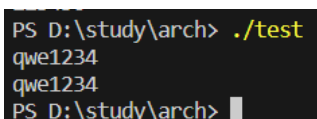
```
1 //test.c
2 #include<stdio.h>
3 void test(){
4     char input[1];
5     gets(input);
6     puts(input);
7 }
8 int main(){
9     test();
10 }
```

当我输入的字符数比较小的时候，这一段是可以运行的且好像没有任何问题，比如输入6、7个字符（虽然可能已经越界了），如图1.1和图1.2所示：



```
PS D:\study\arch> gcc -o test test.c
PS D:\study\arch> ./test
123456
123456
```

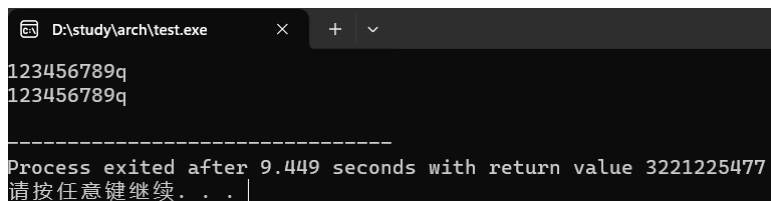
图1.1



```
PS D:\study\arch> ./test
qwe1234
qwe1234
PS D:\study\arch>
```

图1.2

但是当我输入字符比较多时，程序就会出错，比如10个字符，这里用 `devc++` 可以更明显地看出问题，运行时先是卡住一会，再返回3221225477（一般表示内存访问错误），如图1.3所示



```
D:\study\arch\test.exe
123456789q
123456789q
-----
Process exited after 9.449 seconds with return value 3221225477
请按任意键继续. . .
```

图1.3

当时在网上搜索后我了解到C语言的 `gets()` 等很多库函数是没有边界检查的，在新的C语言标准中已经不建议使用。但是 `gets()` 函数为什么在越界不多的时候还能正常运行，再越界更多的时候就会出错？这样的错误会有什么结果？能利用这种错误做什么？我都不清楚。这次的探索实验我就想用汇编视角来看 `gets()` 函数，并通过这个函数对内存的操作进一步了解一下C语言在内存访问和缓冲区中做的一些事情。

2. 探索过程

2.1 调试尝试

- 首先我想通过单步调试直接进入 `gets()` 函数内部，在具体的每一步观察堆栈的变化，但是当我使用什么IDE进行单步调试时，无法直接进入函数内部，在 `step into` 后总是会直接跳过 `gets()`，然后提示出现堆栈的异常，如图2.1.1所示



图2.1.1

- 随后我发现 `gets()` 函数在编译器内使用关键字 `inline` 和 `__attribute__((__always_inline__))` 进行了内联优化，因此无法跟踪函数调用栈，在内联函数中添加断点好像也不起作用，如图2.1.2所示：

```
79 extern inline __attribute__((__always_inline__)) char *
80 gets (char *_str)
81 {
82     if (__ssp_bos (_str) != (size_t) -1)
83         return __gets_chk (_str, __ssp_bos (_str));
84     return __gets_alias (_str);
85 }
```

图2.1.2

2.2 源码分析

- 然后我想通过调用函数的源码来分析，于是找到C语言的库文件，在命令行中输入 `git clone git://sourceware.org/git/glibc.git` 将其下载下来，如图2.2.1所示：

```
fengnian@MICROSO-VU1H310:/mnt/d/study/arch$ git clone git://sourceware.org/git/glibc.git
Cloning into 'glibc'...
remote: Enumerating objects: 705700, done.
remote: Counting objects: 100% (27370/27370), done.
remote: Compressing objects: 100% (9235/9235), done.
remote: Total 705700 (delta 20364), reused 20650 (delta 17771), pack-reused 678330
Receiving objects: 100% (705700/705700), 231.65 MiB | 3.10 MiB/s, done.
Resolving deltas: 100% (601089/601089), done.
Updating files: 100% (19801/19801), done.
```

图2.2.1

- 但随后在库文件中的 `stdio` 库里我始终没有找到 `gets()` 函数的定义，在网上搜索后了解到

可能没有原代码，该函数的实体应该在某个.dll或者.obj文件中。如果该函数是在.dll文件中，链接的时候，链接程序会根据函数声明在相应的.lib文件（可能就是标准库里面）里面找到该函数所在的dll；如果是在.obj文件中，那么链接程序会在该.obj文件中找到已经编译好函数实体直接链入你的程序。¹

如图2.2.2所示：

```
fengnian@MICROSO-VU1H310:/mnt/d/study/arch$ cd glibc/stdio-common
fengnian@MICROSO-VU1H310:/mnt/d/study/arch/glibc/stdio-common$ find get.c
find: 'get.c': No such file or directory
```

图2.2.2

- 最后我找到了网上的 `gets()` 函数的一个实现，那么调用这个函数的代码如下：

```
1 //test.c
2 #include<stdio.h>
3 char *gets(char *input){
4     char *ret = input;
5     int cnt;
6     while ((cnt = getchar()) != '\n' && cnt != EOF){
7         *ret++ = cnt;
8     }
```

```

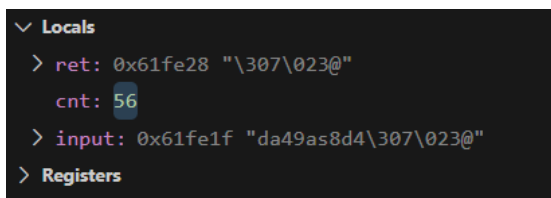
9      *ret++ = '\0';
10     return ret;
11 }
12 void test(){
13     char input[1];
14     gets(input);
15     puts(input);
16 }
17 int main(){
18     test();
19 }

```

这个函数从 `stdin` 中依次读取字符并写到 `input` 的位置直到回车，然后像C语言所做的在最后加上 `'\0'` 后返回

- 在之后调试该程序的时候发现，与调用库函数的现象基本一致

按正常逻辑来说，`gets()` 函数应该只能写到 `*input` 的一个地址，但实际上并没有限制它，使得它可以继续往后面的地址写入，如下图所示，显然已经超过了 `gets()` 应该访问的地方，并且在更后面的地址中写入了一些奇怪的东西 `"\307"\023@"` 等，如图2.2.3所示：



```

▼ Locals
> ret: 0x61fe28 "\307\023@"
  cnt: 56
> input: 0x61fe1f "da49as8d4\307\023@"
> Registers

```

图2.2.3

到了这一步，虽然我还是并不知道这个程序具体发生了什么，因此需要更通过编译器编译后的汇编代码来进一步分析

2.3 反汇编代码分析

- 在命令中输入 `gcc -S test.c -o test.s`，让GCC读取 `test.c` 文件并将其编译为 `test.s` 汇编代码文件，该文件包含与输入源代码等效的汇编指令，如图2.3.1所示：



```

PS D:\study\arch> gcc -S test.c -o test.s
PS D:\study\arch>

```

图2.3.1

- 打开生成的 `test.s` 汇编文件，我们先定位到 `test` 函数看看在调用 `gets()` 函数前做了什么，代码如下：

```

1  0000000004015bb <test>:
2      4015bb: 55                      push    rbp
3      4015bc: 48 89 e5                mov     rbp, rsp
4      4015bf: 48 83 ec 30             sub     rsp, 0x30
5      4015c3: 48 8d 45 ff             lea     rax, [rbp-0x1]
6      4015c7: 48 89 c1                mov     rcx, rax
7      4015ca: e8 81 ff ff ff         call    401550 <gets>
8      4015cf: 48 8d 45 ff             lea     rax, [rbp-0x1]

```

9	4015d3:	48 89 c1	mov	rcx, rax
10	4015d6:	e8 05 15 00 00	call	402ae0 <puts>
11	4015db:	90	nop	
12	4015dc:	48 83 c4 30	add	rsp, 0x30
13	4015e0:	5d	pop	rbp
14	4015e1:	c3	ret	

可以看到GCC首先为 `test()` 函数设置了48个字节的栈帧（通过 `subq $48, %rsp` 指令），将栈顶指针 `%rsp` 赋值给 `%rbp`，在将这个值减一当做 `gets()` 和 `puts()` 的参数地址，也就是说，`test()` 函数栈上为 `gets()` 保留的空间也只有1个字节

1	40156a:	48 8d 50 01	lea	rdx, [rax+0x1]
2	40156e:	48 89 55 f8	mov	QWORD PTR [rbp-0x8], rdx
3	401572:	8b 55 f4	mov	edx, DWORD PTR [rbp-0xc]
4	401575:	88 10	mov	BYTE PTR [rax], dl

我们在关注 `gets()` 函数中写字符的部分写入地址是 `rax+0x1`，也就是往上增长的（与栈增长相反），那么当写入更多字符的时候就应该超过了 `test()` 函数的栈，这时那么就应该会写到调用它的 `main()` 的函数栈上，产生溢出后的覆盖

2.4 GDB调试

之后我用GDB对生成的可执行文件进行调试，主要关注在函数调用前后栈顶指针 `rsp` 和栈基指针 `rbp` 的值

- 首先在 `main()` 函数处打上断点并运行到这里，如图2.4.1所示：

```

Reading symbols from test...done.
(gdb) b main
Breakpoint 1 at 0x4015ef: file test.c, line 21.
(gdb) run
Starting program: D:\study\arch\test.exe
[New Thread 20612.0x5028]
[New Thread 20612.0x56bc]

Thread 1 hit Breakpoint 1, main () at test.c:21
21      test();
(gdb)

```

图2.4.1

- 查看当前的寄存器，如图 2.4.2所示：

```

(gdb) info register
rax            0x1          1
rbx            0x8          8
rcx            0x1          1
rdx            0x1d3bd0 1915856
rsi            0x17         23
rdi            0x1d3c10 1915920
rbp            0x61fe20 0x61fe20
rsp            0x61fe00 0x61fe00
r8             0x1d2610 1910288
r9             0x1d0000 1900544
r10            0x13         19
r11            0x61fc48 6421576
r12            0x10         16
r13            0x8          8
r14            0x0          0
r15            0x0          0

```

图2.4.1

可以看到，此时 `rsp` 的值为 `0x61fe00`

- 进入 `test()` 函数之后，在查看此时 `rbp` 的值为 `0x61fdf0`，如图 2.4.2所示：

```
(gdb) n
17      gets(input);
(gdb) info register
rax             0x1      1
rbx             0x8      8
rcx             0x1      1
rdx             0x663bd0 6699984
rsi             0x17     23
rdi             0x663c10 6700048
rbp             0x61fdf0 0x61fdf0
rsp             0x61fdc0 0x61fdc0
r8              0x662610 6694416
r9              0x660000 6684672
r10             0x13     19
r11             0x61fc48 6421576
r12             0x10     16
r13             0x8      8
r14             0x0      0
r15             0x0      0
rip             0x4015c3 0x4015c3 <test+8>
```

图 2.4.2

二者相减可以得到是16个字节，也就是说这16个字节可能有一些并没有使用过（或者说写到这些地方可能没有问题）

- 为了验证我就查看了地址范围的数据，如图 2.4.3所示：

```
(gdb) x/x 0x61fdf3
0x61fdf3: 0x00000000
(gdb) x/x 0x61fdf4
0x61fdf4: 0x00000000
(gdb) x/x 0x61fdf9
0x61fdf9: 0x00004015
(gdb) x/x 0x61fdfa
0x61fdfa: 0x00000040
(gdb) █
```

图 2.4.3

我发现前几个字节地址全都为0，没有被使用，而后几个字节的地址被赋了一些值。也就是说，当我们越界写入这些没有被使用的地址对现在整个程序的运行其实是没有影响的，但是当输入的字符过多，就会覆盖一些有用的地址甚至达到 `main()` 函数的栈，如果不进行保护就会影响返回地址、局部变量等等参数。至于这些使用了的地址具体在做什么，我没有得到结论。

2.5 进一步验证和分析

- 在上一步结束之后，我又想试一下改变写入数组的大小，看看会有什么变化，于是我改变了 `input[size]` 数组的大小，发现在汇编中为 `test()` 函数分配的栈空间是离散增长的，而且在 `size > 16`之后，传入 `gets()` 的参数不再是 `%rbp-size`，而是 `%rbp-constant`，其中 `constant` 是一个也是离散增长的值。如下表所示

size=1~16	<pre> 415 0000000000004015b0 <test>: 416 4015bb: 55 push rbp 417 4015bc: 48 89 e5 mov rbp,rbp 418 4015bf: 48 83 ec 30 sub rsp,0x30 419 4015c3: 48 8d 45 fb lea rax,[rbp-0x5] 420 4015c7: 48 89 c1 mov rcx,rax 421 4015ca: e8 81 ff ff ff call 401550 <gets> 422 4015cf: 48 8d 45 fb lea rax,[rbp-0x5] 423 4015d3: 48 89 c1 mov rcx,rax 424 4015d6: e8 05 15 00 00 call 402ae0 <puts> 425 4015db: 90 nop 426 4015dc: 48 83 c4 30 add rsp,0x30 427 4015e0: 5d pop rbp 428 4015e1: c3 ret </pre> <p>size=5/栈大小 =0x30/*input=%rbp-5</p>	<pre> 415 0000000000004015b0 <test>: 416 4015bb: 55 push rbp 417 4015bc: 48 89 e5 mov rbp,rbp 418 4015bf: 48 83 ec 30 sub rsp,0x30 419 4015c3: 48 8d 45 f6 lea rax,[rbp-0xa] 420 4015c7: 48 89 c1 mov rcx,rax 421 4015ca: e8 81 ff ff ff call 401550 <gets> 422 4015cf: 48 8d 45 f6 lea rax,[rbp-0xa] 423 4015d3: 48 89 c1 mov rcx,rax 424 4015d6: e8 05 15 00 00 call 402ae0 <puts> 425 4015db: 90 nop 426 4015dc: 48 83 c4 30 add rsp,0x30 427 4015e0: 5d pop rbp 428 4015e1: c3 ret </pre> <p>size=10/栈大小 =0x30/*input=%rbp-10</p>	<pre> 415 0000000000004015b0 <test>: 416 4015bb: 55 push rbp 417 4015bc: 48 89 e5 mov rbp,rbp 418 4015bf: 48 83 ec 30 sub rsp,0x30 419 4015c3: 48 8d 45 f0 lea rax,[rbp-0x10] 420 4015c7: 48 89 c1 mov rcx,rax 421 4015ca: e8 81 ff ff ff call 401550 <gets> 422 4015cf: 48 8d 45 f0 lea rax,[rbp-0x10] 423 4015d3: 48 89 c1 mov rcx,rax 424 4015d6: e8 05 15 00 00 call 402ae0 <puts> 425 4015db: 90 nop 426 4015dc: 48 83 c4 30 add rsp,0x30 427 4015e0: 5d pop rbp 428 4015e1: c3 ret </pre> <p>size=16/栈大小 =0x30/*input=%rbp-16</p>
size=17~33	<pre> 415 0000000000004015b0 <test>: 416 4015bb: 55 push rbp 417 4015bc: 48 89 e5 mov rbp,rbp 418 4015bf: 48 83 ec 40 sub rsp,0x40 419 4015c3: 48 8d 45 e0 lea rax,[rbp-0x20] 420 4015c7: 48 89 c1 mov rcx,rax 421 4015ca: e8 81 ff ff ff call 401550 <gets> 422 4015cf: 48 8d 45 e0 lea rax,[rbp-0x20] 423 4015d3: 48 89 c1 mov rcx,rax 424 4015d6: e8 05 15 00 00 call 402ae0 <puts> 425 4015db: 90 nop 426 4015dc: 48 83 c4 40 add rsp,0x40 427 4015e0: 5d pop rbp 428 4015e1: c3 ret </pre> <p>size=17/栈大小 =0x40/*input=%rbp-32</p>	<pre> 415 0000000000004015b0 <test>: 416 4015bb: 55 push rbp 417 4015bc: 48 89 e5 mov rbp,rbp 418 4015bf: 48 83 ec 40 sub rsp,0x40 419 4015c3: 48 8d 45 e0 lea rax,[rbp-0x20] 420 4015c7: 48 89 c1 mov rcx,rax 421 4015ca: e8 81 ff ff ff call 401550 <gets> 422 4015cf: 48 8d 45 e0 lea rax,[rbp-0x20] 423 4015d3: 48 89 c1 mov rcx,rax 424 4015d6: e8 05 15 00 00 call 402ae0 <puts> 425 4015db: 90 nop 426 4015dc: 48 83 c4 40 add rsp,0x40 427 4015e0: 5d pop rbp 428 4015e1: c3 ret </pre> <p>size=25/栈大小 =0x40/*input=%rbp-32</p>	<pre> 415 0000000000004015b0 <test>: 416 4015bb: 55 push rbp 417 4015bc: 48 89 e5 mov rbp,rbp 418 4015bf: 48 83 ec 50 sub rsp,0x50 419 4015c3: 48 8d 45 d0 lea rax,[rbp-0x30] 420 4015c7: 48 89 c1 mov rcx,rax 421 4015ca: e8 81 ff ff ff call 401550 <gets> 422 4015cf: 48 8d 45 d0 lea rax,[rbp-0x30] 423 4015d3: 48 89 c1 mov rcx,rax 424 4015d6: e8 05 15 00 00 call 402ae0 <puts> 425 4015db: 90 nop 426 4015dc: 48 83 c4 50 add rsp,0x50 427 4015e0: 5d pop rbp 428 4015e1: c3 ret </pre> <p>size=33/栈大小 =0x50/*input=%rbp-48</p>

我们可以得到如下的规律：

size	栈大小	传入的参数*input
n (n∈0~16)	0x30+n/16*0x10	%rbp - n
n (n > 0~16)	0x30+n/16*0x10	%rbp - (n/16+1) *0x10

- 我们可以看到在 $n > 16$ 的时候，可以越界写入的字节数不仅是栈之间的空闲位置，还要加上 `test()` 函数栈中的未使用的地址。因此，我终于得到了一个最终的结论，如下表所示：

size	gets()可以正常写入的字符数 (constant表示一个常数)
n (n∈0~16)	n + constant
n (n > 0~16)	(n/16+1) *16 + constant

经过我的反复尝试，这个 constant 值为7，带入上面的表格就可以得到 `input[size]` 为n时 `gets()` 可以正常写入的字符数

2.6 其他编译器的比较

在上面的过程中，我一直使用的都是GCC的编译器，所以我想探索一下在不同的编译器上，程序的栈的设置，于是我使用 `Compiler Explorer` 网站来模拟其他编译器的处理

2.6.1 clang

clang编译器生成的汇编代码如图2.6.1所示，clang的机制与GCC基本一致，只是在栈大小的设置上较小，从最初的16→32→48.....并且经过我的测试（这里就省略过程了，和前面基本一样），clang中 `gets()` 可以正常写入的字符数没有GCC里的 `constant`（栈之间没有可写的字节），其余与GCC相同

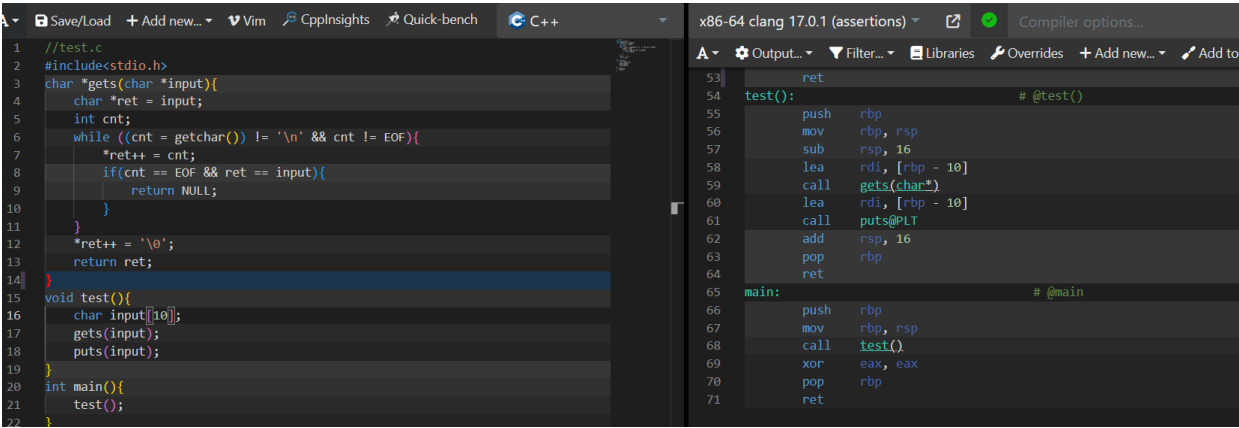
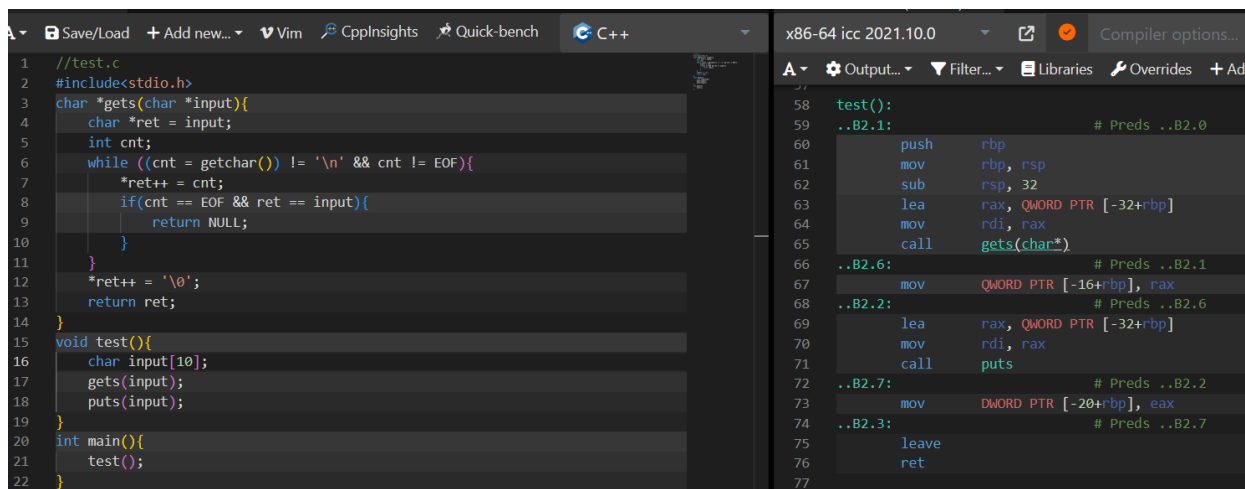


图2.6.1

2.6.2 icc

icc编译器生成的汇编代码如图2.6.2所示，可以看到icc在 `input[size] < 16` 的时候就将 `rbp - 32` 作为参数传入 `gets()`，这里的32就是 `test` 栈的大小，也就是说 `gets()` 可以正常写入的字符数应该就是 2.5 中的结论不分类讨论，直接用 $n > 16$ 的情况（由于icc编译器并不免费且配置较复杂，这里只是一个猜测，没有验证）



3. 效果分析

通过上面的探索过程，我们可以得到：

1. 在C语言调用函数的时候，会为函数分配对应的栈，这个栈根据不同编译器的处理，大小也不同，例如在前面的程序运行时栈的大小 gcc > icc > clang
2. 在为 gets() 函数传入参数的时候不同编译器的处理也不同，例如在 input[size] < 16 时 clang 和 gcc 都是按需分配，将 rbp-size 传入，而 icc 直接传入 rsp
3. 在 gets() 函数进行读写的时候，实际上并没有对它的写入地址做足够的限制，导致字符数即使超过缓冲区的大小，依然可以正常读写，gets() 函数在这种情况下会写入还未使用的栈空间甚至越过栈空间，这也许会导致一些错误。
4. 对于这种越界写入，GCC 的表现：

size	gets()可以正常写入的字符数 (constant表示一个常数)
n (n≤16)	n + constant
n (n > 16)	(n/16+1) * 16 + constant

5. 但是不同的IDE可能会对这种越界进行保护，比如当我使用 clion 进行运行时，只要超过缓冲区范围，就会抛出异常，如图3.1所示，当缓冲区为5时，输入6个字符就会显示 stack smashing detected 表示检测到了检测到栈破坏 (stack smashing) 或缓冲区溢出 (buffer overflow)。经过查阅资料了解到，

当编译器启用了栈保护 (stack protection) 功能时，它会在堆栈上放置一些特殊的辅助值（称为"canary"），并在函数返回前检查这些值是否被修改。如果发现这些值被改变，就表明发生了栈破坏。²

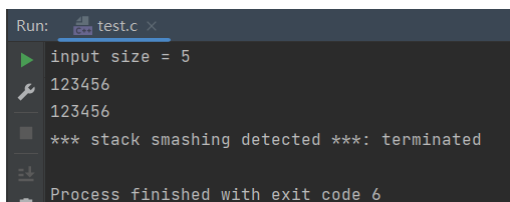


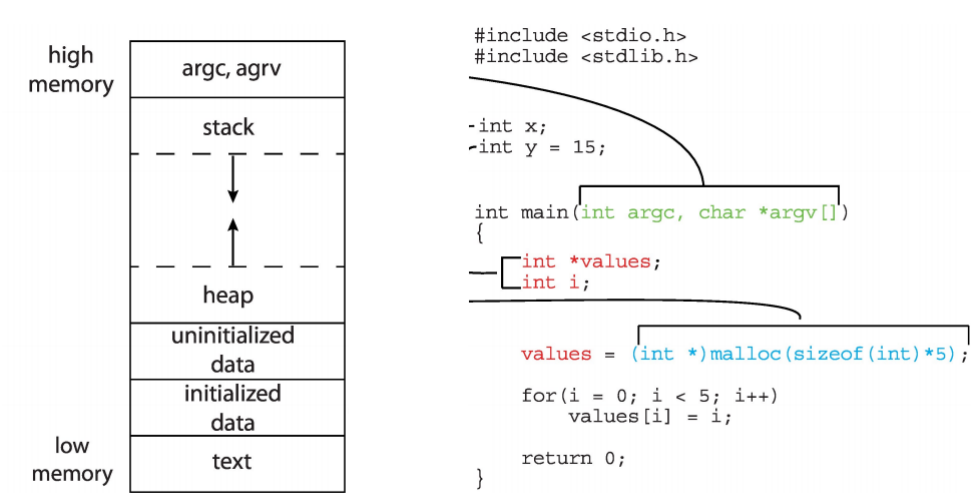
图3.1

4.实验体会

这次实验的过程对我来说是一个全新的体验，因为以前的实验从本质上来讲都是大致知道答案或者方向来一步步接近，但这次实验在探索时我并不知道最后的答案应该是什么，在网上搜索到的解答或讨论都十分模糊甚至是错误的，并且由于不同的编译器、不同的硬件配置等也可能影响得到的结果。因此我只能根据汇编代码和调试工具自己探索代码运行的时候到底发生了什么，比如函数调用的时候栈是怎么分配的、`gets()` 函数具体是怎么运行的、读到的数据可以写到哪里等等，从汇编角度去分析才能知道程序的很多步骤到底在干什么。

此外，我还体会到汇编与高级语言之间一个很大的不同点就在于很多准备工作是在高级语言程序中看不到的，比如栈的分配、地址传参等等，尤其在不同的编译器也许会有不同的处理，这更加体现出理解汇编代码的重要性。

在探索过程中，对于之前学过的一些课程知识的应用也使我感受到课程内容之间的联系和结合，比如对于函数的调用以及栈的分配正好是操作系统刚讲过的内容，如下图所示：



而缓冲区溢出则让我想到了网络安全导论课程大作业对于 Heartbleed 漏洞的复现，如下图所示：

一、“Heartbleed”漏洞简介及原理

OpenSSL 程序包可以使用户方便地对通信进行 SSL 加密，达到安全通信的效果，因此，OpenSSL 被广泛用于服务器上。在 OpenSSL 中，有一个名为“Heartbleed”的拓展来检测对方服务器的状态是否正常在线，检测方法是发起 TLS 请求，即 ClientHello 消息，若对方服务器发回“Server hello”，即表明正常树立 SSL 通讯。而在每次 TLS 请求中会附加一个问询的字符长度 pad length，如果这个 pad length 大于实际的长度，服务器仍会回来相同规模的字符信息，于是形成了内存里信息的越界访问，攻击者可以构造异常的心跳数据包，即心跳包中的长度字段与后续的数据字段不相符合，来获取心跳数据所在的内存区域的后续数据。因此，在每次检测中服务器就能泄露一写些数据，通过逐次累加，这些数据里可能包括 post 请求数据，会话 cookie 和密码等信息，也可能并没有包含这些信息，但攻击者可以不断利用“心跳”来获取更多的信息。

最后，虽然我这次探索实验只是研究了一个很简单的函数 `gets()`，也只分析了一些很简单的汇编代码，但是复杂问题或错误往往来自简单的漏洞，缓冲区溢出漏洞的原理就可以在 `gets()` 函数的溢出上体现，当你成功地捕捉到这种溢出并且设置攻击代码覆盖返回地址的话，就可以注入病毒、错误信息等等对程序运行乃至系统运行产生威胁的东西来造成巨大的破坏。因此，在探索的过程中我更加理解了缓冲区溢出、栈溢出的机制，获益匪浅。

5.经验教训

在实验过程中，我也遇到了一些问题，总结如下：

1. 不要迷信书本，我在上网搜索上发现《深入理解计算机系统》书中有对这个函数的分析，但通过对汇编代码的分析，我发现现在的编译器的处理与书中的内容并不相同，书中的结论也和我运行程序后的结果不同，因此由于编译器等的改变，具体的情况还是要以自己得到汇编代码和运行结果为准
2. 在分析汇编代码时，一定要进行调试，查看一些寄存器和地址的值，才能得到结果，因为汇编是在操纵硬件，知道各寄存器的值可以更好地理解代码运行过程
3. 在探索过程中，我还有很多问题没有具体解决，比如GCC在栈之间的一些地址为什么可以写入；icc编译器是否提供栈保护，还是也可以越界访问（了解到icc编译器更倾向于性能而导致很多bug，我倾向于可以）等等问题还没有解决
4. 最后就是既然我们知道了缓冲区溢出会导致很多问题，那么在之后编写代码的时候就要尽量避免，不再用 `gets()` 等函数，对栈进行更好的保护

6.参考文献

1. `gets`函数是如何实现的：<https://bbs.csdn.net/topics/100041558> 
2. 浅析栈保护机制：<https://blog.csdn.net/najdhdfh/article/details/109202138> 