

Computer Science & Engineering 150A

Problem Solving Using Computers

Mini Lecture – “Problems” With scanf

Christopher M. Bourke

Spring 2009

Consider the following code:

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5      char myChar;
6
7      printf("Please enter a character:");
8      scanf("%c", &myChar);
9      printf("Your character was %c, thanks.\n",myChar);
10
11     printf("Please enter a character:");
12     scanf("%c", &myChar);
13     printf("Your character was %c, thanks.\n",myChar);
14
15     printf("Please enter a character:");
16     scanf("%c", &myChar);
17     printf("Your character was %c, thanks.\n",myChar);
18
19     return 0;
20 }
```

- The first character read works fine
- The second character read doesn't work: it doesn't wait for you to enter it!
- The third character read works as well

Why the code fails requires an understanding of *buffers*

- Buffers are stores of memory used to temporarily hold data
- Video streaming buffers: may hold the next 10 seconds worth of video for smoother playback on inconsistent data networks
- The standard input is also a buffer: each and every character you type is stored into a buffer
- Certain C functions (`scanf`, `getc`, `gets`) read from this buffer
- The newline (`\n`) character is still a character!

- 1 Type `a`
- 2 Type `\n` (enter)
- 3 `scanf` reads `a`, stores it in `myChar` and prints it out
- 4 The program then prompts the user for the next character, but `\n` is still in the buffer
- 5 `scanf` reads the *endline character!* and prints it out!

- Some C functions read until the next endline character but leave it in the buffer
- Others read until the next endline character and remove it from the buffer
- Others may read up to and *including* the next endline character and remove it from the buffer
- How to tell which ones do what? Read the documentation!

Solution 1: Flushing the buffer

CSCE150A

Introduction

Solution 1

Solution 2

Solution 3

Summary

- The Standard C library has a function, `fflush(FILE *stream)` that “flushes” buffers
- flushing a buffer is to empty it of its contents, disregarding them
- We could “flush” the input buffer: `fflush(stdin)`;
- Works on *some* systems, but the ANSI C specification defines `fflush(stdin)` as “undefined behavior”

Solution 2: Chomp the newline character

Since `scanf` starts executing when an endline character is reached, we can simply read another character, expecting it to be `\n` and disregard it.

```
1  #include <stdio.h>
2
3  int main(void) {
4      char myChar;
5
6      printf("Please enter a character:");
7      scanf("%c", &myChar);
8      printf("Your character was %c, thanks.\n", myChar);
9      scanf("%c"); //read and disregard
10
11     printf("Please enter a character:");
12     scanf("%c", &myChar);
13     printf("Your character was %c, thanks.\n", myChar);
14     scanf("%c"); //read and disregard
15
16     printf("Please enter a character:");
17     scanf("%c", &myChar);
18     printf("Your character was %c, thanks.\n", myChar);
19     scanf("%c"); //read and disregard
20
21     return 0;
22 }
```

Solution 2: Chomp the newline character

CSCE150A

Introduction

Solution 1

Solution 2

Solution 3

Summary

- Doesn't work if (say) we quickly enter in a bunch of newlines
- Doesn't work if we enter in other white-space characters (tab, space)
- Doesn't work if we enter multiple characters

Another possible solution: place spaces around the %c placeholder.

```
1  #include <stdio.h>
2
3  int main(void) {
4      char myChar;
5
6      printf("Please enter a character:");
7      scanf(" %c ", &myChar);
8      printf("Your character was %c, thanks.\n",myChar);
9
10     printf("Please enter a character:");
11     scanf(" %c ", &myChar);
12     printf("Your character was %c, thanks.\n",myChar);
13
14     printf("Please enter a character:");
15     scanf(" %c ", &myChar);
16     printf("Your character was %c, thanks.\n",myChar);
17
18     return 0;
19 }
```

- This tells `scanf` to *expect* and ignore white space around a non-whitespace character
- Still doesn't solve the problem if we give it many non-whitespace characters

- `scanf` was designed for structured input (*scan formatted*).
- As long as the format is understood and *followed*, everything's peachy
- `scanf` *can* tell you if it failed, but not how or why
- In real, critical systems, it should not be used, but its good enough for our purposes.
- For more details, see the `comp.lang.c` FAQ:
<http://c-faq.com/stdio/scanfprobs.html>