关于 scanf() 输入函数及键盘缓冲区的问题

耿楠

西北农林科技大学信息工程学院，陕西 · 杨凌，712100

2016 年 10 月 19 日

摘要

在 C 语言中，scanf() 函数是读取数据的一种有效但不理想的方法，它能保证在读取单一数据时不出错，但当有不同类型数据的交叉读取，或者同其他输入函数混用时却非常容易出错。针对这一问题，本文通过对计算机输入缓冲区及输入流的简单分析，探讨了 scanf() 函数的基本操作原理和使用 scanf() 函数时需要注意事项以及 DEBUG 技术在分析使用 scanf() 函数时所出现的错误的方法。通过本文的分析，期望能够对大家的学习提供参考。

一、输入流和输入缓冲区的概念

在 C 语言中，术语流 (stream) 表示任意输入的源或任意输出的源。小型程序都是通过一个流 (通常与**键盘**相关) 获得全部输入，并且通过另一个流 (通常与**屏幕**相关) 输出全部的输出。较大规模的程序可能会需要额外的流。这些流常常表示存储在不同介质 (如硬盘、CD、DVD、闪存等) 上的文件，但也很容易和不存储文件的设备 (网络端口、打印机等) 相关联。在<stdio.h>中，定义了大量的流处理操作函数，**这些函数可以处理各种形式的流，而不仅仅是键盘、屏幕和存储介质**。

缓冲区又称为缓存，它是内存空间的一部分，是系统预留的一定大小的存储空间，这些存储空间用来缓冲输入或输出的数据，这部分预留的空间就叫做缓冲区。缓冲区用于在输入输出设备和 CPU 之间进行缓存数据，它使得低速的输入输出设备和高速的 CPU 能够协调工作，避免低速的输入输出设备占用 CPU，解放出 CPU，使其能够高效率工作。

缓冲区根据其对应的是输入设备还是输出设备，分为输入缓冲区和输出缓冲区。例如，在从输入流读取数据时，需要先把读出的数据放在缓冲区，CPU 再直接从缓冲区中取数据，等缓冲区的数据取完后再去磁盘中读取，这样就可以减少磁盘的读写次数，再加上 CPU 对缓冲区的操作远远快于对输入流的操作，故应用缓冲区可大大提高计算机的运行速度。又比如，使用打印机打印文档，由于打印机的打印速度相对较慢，需要先把文档输出到打印机相应的缓冲区，打印机再自行逐步打印，同时，CPU 可用于处理别的事务。

缓冲区分为全缓冲、行缓冲和不带缓冲三种情况。

1) **全缓冲**是当填满标准 I/O 缓存后才进行实际 I/O 操作，其典型代表是对磁盘文件的读写。

2) **行缓冲**是当输入和输出中遇到换行符时，执行真正的 I/O 操作。这时，输入的字符先存放在缓冲区，等按下回车键换行时才进行实际的 I/O 操作，其典型代表是通过键盘输入数据。

3) **不带缓冲**也就是不进行缓冲，标准出错情况 stderr 是典型代表，这使得出错信息可以直接尽快地显示出来。

在 C 语言中，scanf()、getchar()、gets() 等输入函数通常是通过键盘输入数据，因此采用的是**行缓冲**技术，以 scanf() 为例，其逻辑结构如 图 1所示。

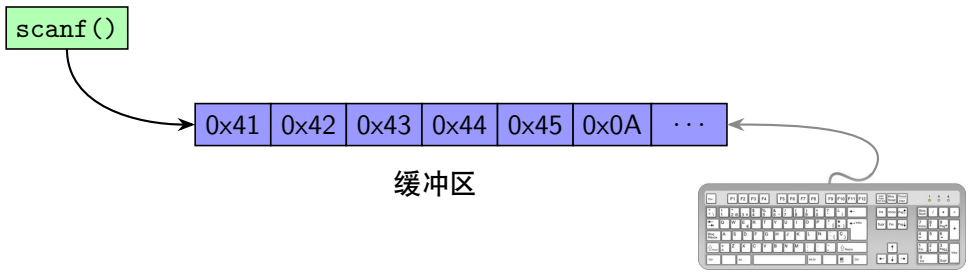


图 1　scanf() 函数的行缓冲

需要说明的是：当从键盘输入“字符串”的时候需要输入**回车键**才能够将这个字符串**推送**到缓冲区中，输入的这个回车键（“\r”）会被转换为一个换行符“\n”，**这个换行符“\n”也会被存储在缓冲区中，并且被当成一个字符来处理！**例如，当输入字符串“ABCDE”后，输入回车键（“\r”）时，则将“\n”这个字符串送入了缓冲区中，那么此时缓冲区中的字节个数是 6，而不是 5。

键盘缓冲区用来缓存“按键”的“**ASCII 码**”，例如，scanf() 每次从键盘缓冲区中读取一个字符（ASCII 码），然后根据格式字符串进行“模式匹配”，直至匹配失败或键盘缓冲区为空。再如，getchar() 每次从键盘缓冲区中读取一个字符（ASCII 码），直至键盘缓冲区为空。

二、数据输入实例分析

本小结将以通过几个例子，结合 DEBUG 工具深入分析和理解有相关的知识的技术。

1、读入整型数据存入字符型变量

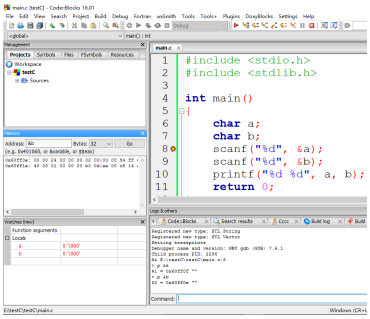
假设有如下代码：

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

int main()
{
    char a;
    char b;
    scanf("%d", &a);
    scanf("%d", &b);
    printf("%d %d", a, b);
    return 0;
}
```

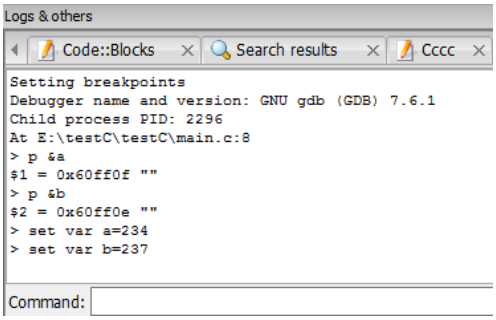
如果在命令行分别输入了 67Enter和 66Enter，则结果是什么呢？如果不仔细分析，则会认为输出的结果是：67 66，很遗憾，这可能是一个错误的结果。下面，采用 DEBUG 的方式对程序进行剖析。

在程序的第 8 行加入断点，如图 2a所示。仔细查看此时程序中变量 a 的地址是 0x60ff0e，b 的地址是 0x60ff0f 分别是 1 个字节 (char类型)。在 Memory 和 Watches 窗口中可以查看这两个变量的值，此处，a 和 b 的值皆为 0。

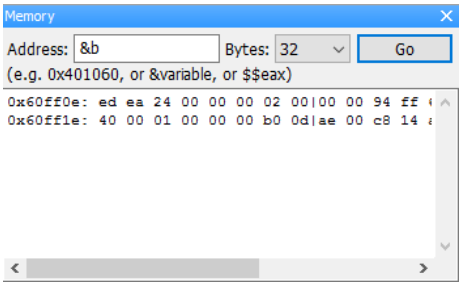
为便于对比分析，可以用“`set var`”命令分别修改变量 `a` 和 `b` 的值 (如图 2b)，单击 Memory 窗口中的Go，在 Watches 窗口中右键快捷菜单中选Update可分别更新 Memory 和 Watches 窗口中显示的数据，如图 2c和 图 2d所示。



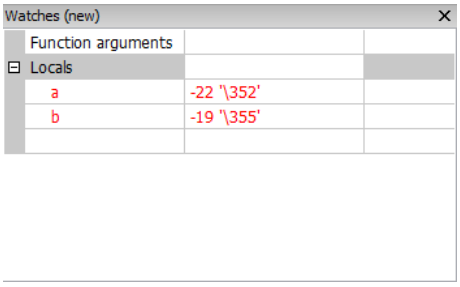
(a) 启动程序调试



(b) 用“`set var`”修改变量的值



(c) Memory 窗口

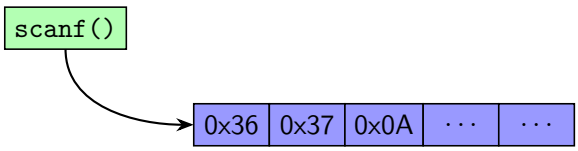


(d) Watches 窗口

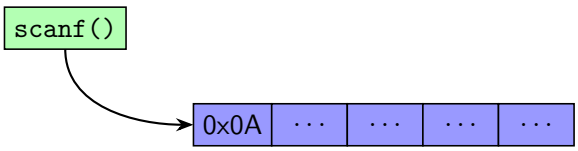
图 2　整型数赋给字符型变量的 DEBUG 过程

单步执行当前行 `scanf("%d", &a);` 程序，此时，程序会等待用户的输入，在控制台输入 67Enter，键盘缓冲区中的数据如图 3a所示。注意，键盘缓冲区中存储的是字符的 ASCII 码值，字符`'6'`的 ASCII 值为 0x36(十六进制)、字符`'7'`的 ASCII 值为 0x37、字符`'\n'`的 ASCII 值为 0x0A。

执行 `scanf()` 函数时，其格式字符串为%d，也就是要读入整型数据，此时，`scanf()` 函数首先寻找正号或负号，后读取数字，直到读到一个非数字时才停止。在当前输入缓冲区中，第 1 个非空的字符是字符`'6'`，紧接着是字符`'7'`，再下来是字符`'\n'`，这是一个非数字空白字符，因此，得到整数 67(0x00000043)，并将其赋给内存中以&a(0x60ff0f) 开始的连续 4 个字节中，由于字符`'\n'`不属于当前项%d，所以它会被放回原处。此时，缓冲区的数据如图 3b所示。



(a) 输入数据的缓冲区

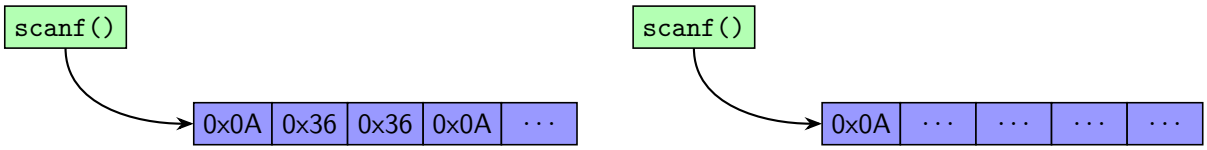


(b) 读取数据后的缓冲区

图 3　处理整型数 67 时的缓冲区

接下来，继续执行前行 `scanf("%d", &b);` 程序，此时，程序会等待用户的输入，在控制台输入 66Enter，此时，键盘缓冲区中的数据如图 4a所示。

scanf() 函数在寻找数据的起始位置时，会忽略**空白字符**（**包括空格符、水平与垂直制表符、换页符和换行符**）。在当前输入缓冲区中，第 1 个非空的字符是字符'6'，紧接着是字符'6'，再下来是字符'\n'，这是一个非数字空白字符，因此，得到整数 66(0x00000042)，并将其赋给内存中以&b(0x60ff0e) 开始的连续 4 个字节中，由于字符'\n'不属于当前项%d，所以它会被放回原处。此时，缓冲区的数据如图 4b所示。

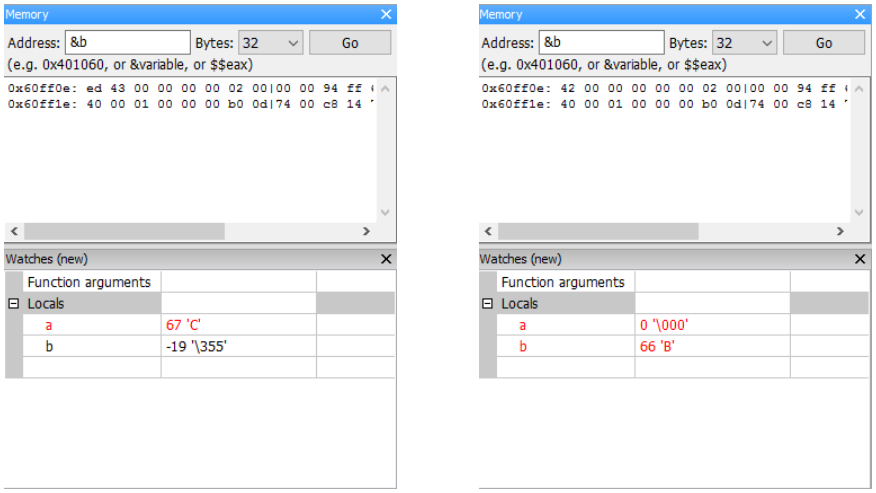


(a) 输入数据后的缓冲区　　(b) 读入数据后的缓冲区

图 4　处理整型数 66 时的缓冲区

需要注意的是在程序中，a 被声明为char类型，占 1 个字节，而读取的是整数，将占 4 个字节，因此会将 0x00000043 赋给内存中以&a(0x60ff0f) 开始的连续 4 个字节中 (图 5a)，但后续 3 个字节并不属于变量 a，这 3 个字节也许可以访问，也许不可以访问。本例中，碰巧可以访问，因此能够将数据存入。**若不可访问，当程序执行到此时，一定会发生崩溃**。

同样，b 被声明为char类型，占 1 个字节，而读取的是整数，将占 4 个字节，因此会将 0x00000042 赋给内存中以&b(0x60ff0e) 开始的连续 4 个字节中 (图 5b)，注意此时会**覆盖**了变量 a 的 3 个低字节，但后续 3 个字节并不属于变量 b，这 3 个字节也许可以访问，也许不可以访问。本例中，碰巧可以访问，因此能够将数据存入。**若不可访问，当程序执行到此时，一定会发生崩溃**。



(a) 整数 a　　(b) 整数 b

图 5　读取整数后的后 DEBUG 的结果

由图 5b所示的最终结果可以看出，a 的值此时为 0，b 的值此时为 66(**各占 1 个字节**)。

在接下来的 printf("%d %d", a, b); 中，由于两个输出格式串都为 %d，因此在输出 a 时，会读取从&a 开始的 1 个字节的数据 (0x00)，并隐式转换为整型数输出。在输出 b 时，会读取从&b 开始的 1 个字节的数据 (0x42)，同样隐式转换为整型数输出。程序输出结果如图 6所示。

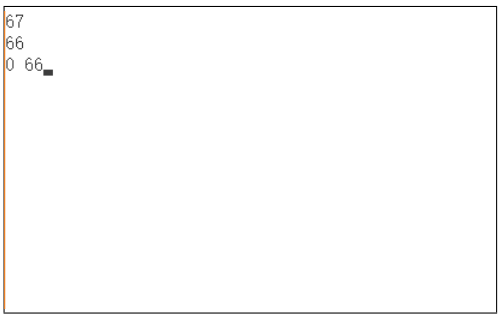


图 6　程序输出的结果

在 scanf() 函数的使用中，一定要注意，前一次读入时，虽然最后的 '\n'也是空白字符，但 scanf() 函数并没有丢弃这一字符，而是“放回原处”，留作为下一次读入时缓冲区中的第 1 个字符。

显然，通过 DEBUG 剖析可知，该程序由于数据类型与输入格式不匹配，在程序运行过程中必然会造成非法内存的使用，但碰巧的是，此时非法使用的内存是可访问的，程序可以运行。若是这些内存不可访问，则必然会造成程序的崩溃。

2、读入字符型数据存入整型变量

假设有如下代码：

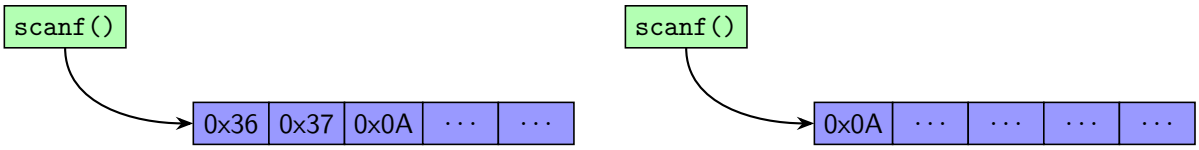
```
#include <stdio.h>
#include <stdlib.h>

int main()
{
    int a;
    int b;
    scanf("%c", &a);
    scanf("%c", &b);
    printf("%d %d", a, b);

    return 0;
}
```

该例中，分别使用了两个 scanf() 函数以%c 读入字符，分别存入 a 和 b。那么，如果在命令行输入67Enter，则结果分别是什么？

此时，输入缓冲区中的数据如图 7a所示。

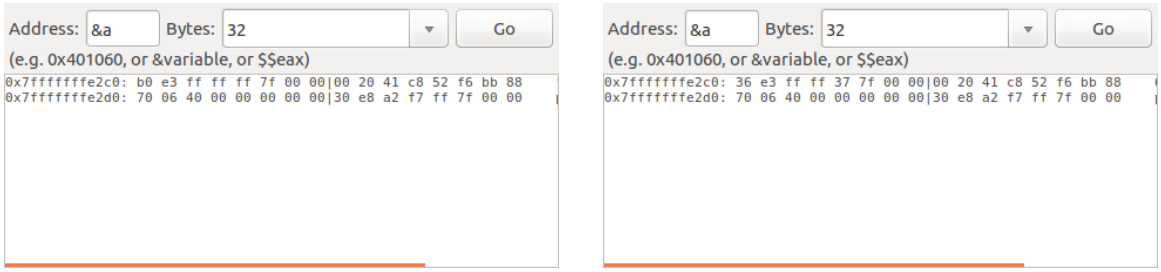
第 1 个 scanf() 函数将按格式字符串%c 读入字符'6'，并将其 ASCII 码存入变量 a 中，紧接着第 2 个 scanf() 函数将按格式字符串%c 读入字符'7'，并将其 ASCII 码存入变量 b 中。此时，缓冲区中的数据如图 7b所示 (**注意此时'\n'仍然在缓冲区中**)。

需要注意的是，scanf("%c",&a); 和 scanf("%c",&b); 分别读入的是一个字节的 ASCII 值，将此值存入地址&a 和&b 时，仅存入了 1 个字节的数据 (而**并没有改写 a 和 b 的 3 个高字节**数据)。关于这一细节，可以从图 8a所示的读入数据前的 DEBUG 状态和图 8b所示的读入数据后的状态的比较进行分析。



(a) 输入数据后的缓冲区　　(b) 读入数据后的缓冲区

图 7　处理字符'6'和'7'时的缓冲区



(a) 读入 a 和 b 前　　(b) 读入 a 和 b 后

图 8　读入字符数据后内存的状态

程序的运行结果如图 9所示 (**此结果可能因不同时间、不同机器的运行结果而不完全相同，这是因为 a 和 b 的初始值可能不一致造成的**)。

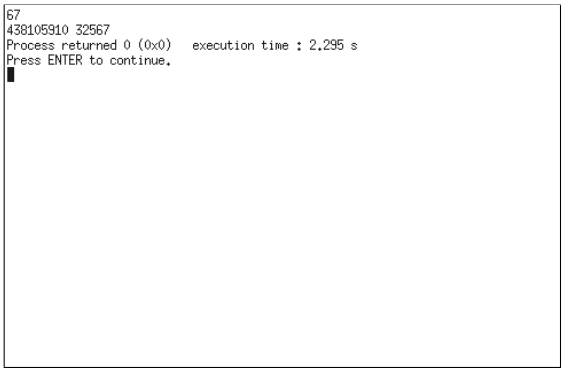


图 9　读入字符赋给整型变量的运行结果

显然，通过 DEBUG 剖析可知，该程序也是由于数据类型与输入格式不匹配，在程序运行过程中虽然未造成非法内存的使用，但其结果却与预期的结果完全不同，并且由于 C 语言在声明变量时并不自动对变量进行初始化，因此，此时程序运行结果也是不确定的。

3、读入字符型数存入字符型变量

假设有如下代码:

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
```

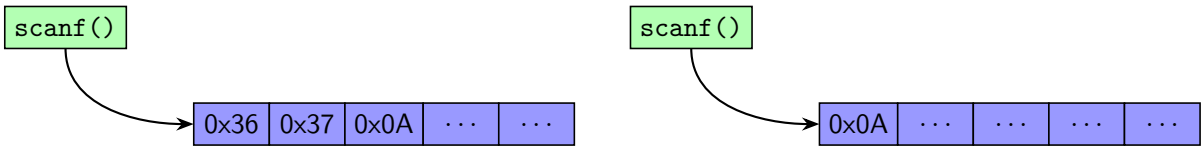
```c
int main()
{
    char a;
    char b;
    scanf("%c", &a);
    scanf("%c", &b);
    printf("%d %d", a, b);
    return 0;
}
```

如果在命令行分别输入了67Enter、6␣6 Enter和5Enter，则结果分别是什么？[1]

2.3.1 输入67

此时，输入缓冲区中的数据如图 10a所示。

变量 a 和 b 被声明为char类型，各占 1 个字节。第 1 个 scanf() 中用的格式是"%c"，则字符'6'与之匹配，将其 ASCII 码值存入 a 中。第 2 个 scanf() 中用的格式是也是"%c"，则字符'7'与之匹配，将其 ASCII 码值存入 b 中。至此，缓冲区中的数据如图 10b所示 ('\n'仍然在缓冲区中)。



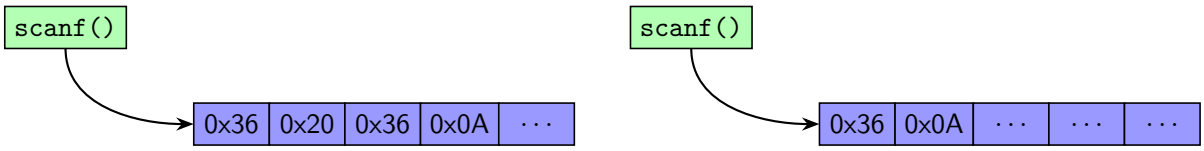
(a) 输入数据后的缓冲区　　(b) 读入数据后的缓冲区

图 10　处理字符 67 时的缓冲区

2.3.2 输入6␣6

此时，输入缓冲区中的数据如图 11a所示。

第 1 个 scanf() 读入字符'6'，将其 ASCII 码值存入 a 中。第 2 个 scanf() 读入字符'␣'，将其 ASCII 码值存入 b 中。至此，缓冲区中的数据如图 11b所示 ('6'和'\n'仍然在缓冲区中)。



(a) 输入数据后的缓冲区　　(b) 读入数据后的缓冲区中

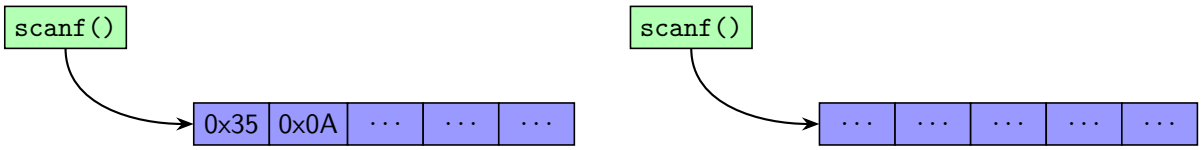
图 11　处理字符6␣6时的缓冲区

[1]限于篇幅，后续的示例分析，不再给出 DEBUG 剖析程序的过程，请大家自己根据第1小节和第2小节的例子完成程序剖析。

2.3.3 输入5

此时，输入缓冲区中的数据如图 12a所示。

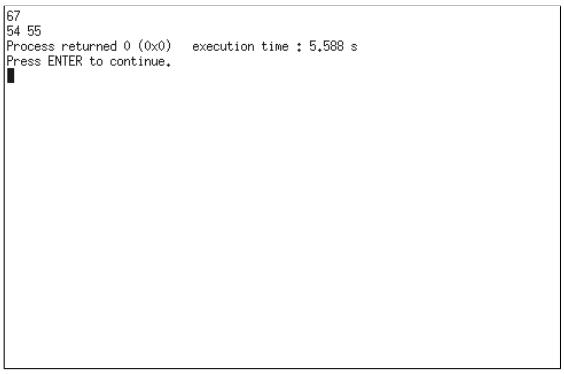
第 1 个 `scanf()` 函数读入字符`'5'`，将其 ASCII 码值存入 `a` 中。第 2 个 `scanf()` 函数读入字符`'\n'`，将其 ASCII 码值存入 `b` 中。至此，缓冲区中的数据如图 12b所示 (**此时缓冲区中无数据**)。



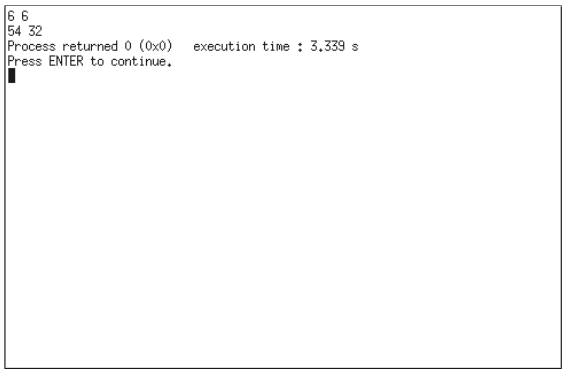
(a) 输入数据后的缓冲区　　(b) 读入数据后的缓冲区

图 12　处理字符 5 和回车时的缓冲区

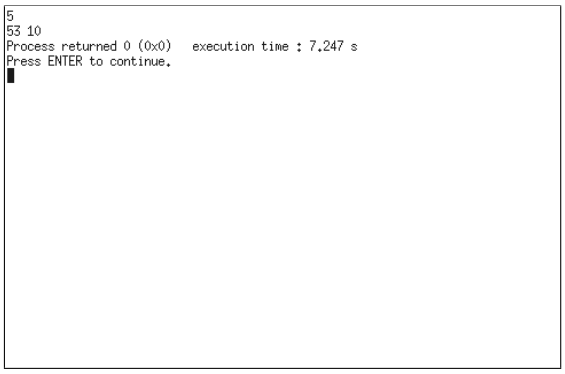
因此，在命令行分别输入了67Enter、6␣6 Enter和5Enter后，其结果分别如图 13a-图 13c所示。



(a) 输入67的结果



(b) 输入6␣6的结果



(c) 输入5的结果

图 13　处理字符的输出结果

4、格式串中的空格

假设有如下代码：

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

int main()
{
    char a;
    char b;

    //注意这两句 scanf 的格式串的% 前有一个空格
    scanf(" %c", &a);
    scanf(" %c", &b);

    printf("%d %d", a, b);

    return 0;
}
```

该例中，scanf() 函数的格式串中有一个前导空格，那么，如果在命令行分别输入了67Enter、6␣6Enter、5Enter6Enter和 ␣5␣6Enter，则结果分别是什么？

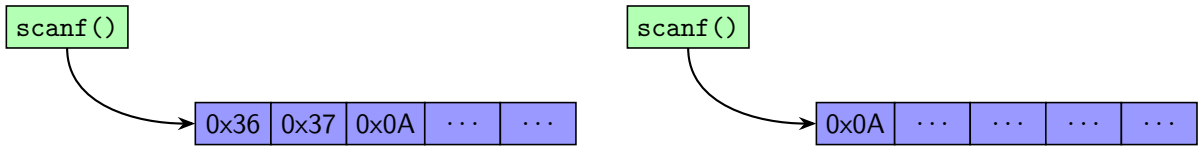
在格式串中遇到一个或多个连续的空白字符时，scanf() 函数会从输入中重复读空白字符直到遇到一个非空白字符 (把该字符 “放回原处”)，格式串中空白字符的数量无关紧要，格式串中的一个空白字符可以与输入中**任意数量的空白字符相匹配**[2]。

在格式串中使用空白字符，有吸收回车 (0x0A) 和空格 (0x20) 的 “神奇功效”，吸收之后要求缓冲区的字符不是回车 (0x0A) 和空格 (0x20) 时，把这个字符交给下一个格式字串。

2.4.1 输入67

此时，输入缓冲区中的数据如图 14a所示。

变量 a 和 b 为char类型，各占 1 个字节。第 1 个 scanf() 函数中用是”%c”，则字符'6'与之匹配，将其 ASCII 码值存入 a 中。第 2 个 scanf() 函数中仍用”%c”，则字符'7'与之匹配，将其 ASCII 码值存入 b 中。至此，缓冲区中的数据如图 14b所示 (**注意此时'\n'仍然在缓冲区中**)。



(a) 输入数据后的缓冲区　　(b) 读入数据后的缓冲区

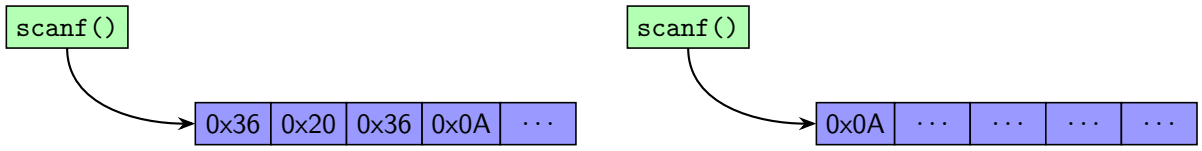
图 14　格式串有空格处理字符 67 时的缓冲区

[2]需要注意的是：在格式串中包含空白字符并不意味着输入中必须包含空白字符。格式串中的一个空白字符可以与输入中**任意数量**的空白字符相匹配，包括零个。

2.4.2 输入6␣6

此时，输入缓冲区中的数据如图 15a所示。

第 1 个 scanf() 函数读入字符'6'，将其 ASCII 码值存入 a 中。第 2 个 scanf() 函数将会跳过空格 (0x20)，直到遇到'6'，将其 ASCII 码值存入 b 中。至此，缓冲区中的数据如图 15b所示 (**注意此时'\n'仍然在缓冲区中**)。



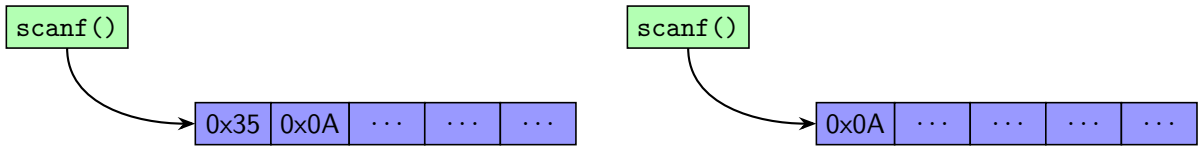
(a) 输入数据后的缓冲区　　(b) 读入数据后的缓冲区

图 15　格式串有空格处理字符6␣6时的缓冲区

2.4.3 输入5< 回车 >6< 回车 >

输入5Enter，输入缓冲区中的数据如图 16a所示。

第 1 个 scanf() 函数读入字符'5'，将其 ASCII 码值存入 a 中。此时，缓冲区中的数据如图 16b所示 (**注意此时'\n'仍然在缓冲区中**)。

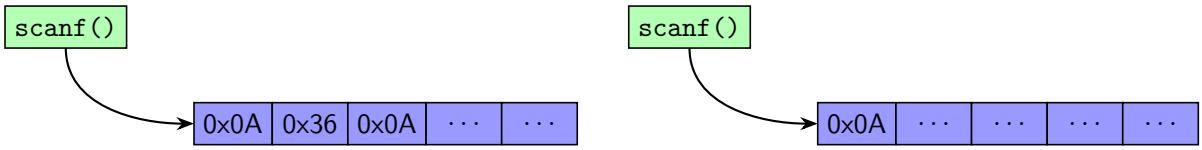


(a) 输入数据后的缓冲区　　(b) 读入数据后的缓冲区

图 16　格式串有空格处理字符'5'< 回车 > 的缓冲区

再输入6Enter后，输入缓冲区中的数据如图 17a所示。

第 2 个 scanf() 函数跳过上次 scanf() 函数留下的"'\n'"，读入字符'6'，将其 ASCII 码值存入 b 中。此时，缓冲区中的数据如图 17b所示 (**注意此时'\n'仍然在缓冲区中**)。



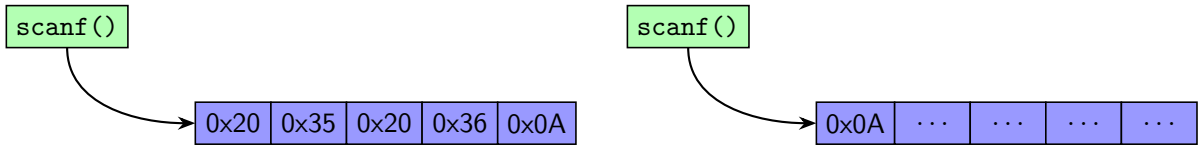
(a) 输入数据后的缓冲区　　(b) 读入数据后的缓冲区

图 17　格式串有空格处理字符'6'< 回车 > 的缓冲区

2.4.4 输入 ␣5␣6

此时，输入缓冲区中的数据如图 18a所示。

第 1 个 `scanf()` 函数会跳过空格 (0x20)，读入字符`'5'`，将其 ASCII 码值存入 `a` 中。第 2 个 `scanf()` 函数继续会跳过空格 (0x20)，直到遇到`'6'`，将其 ASCII 码值存入 `b` 中。至此，缓冲区中的数据如图 18b所示 (**注意此时`'\n'`仍然在缓冲区中**)。

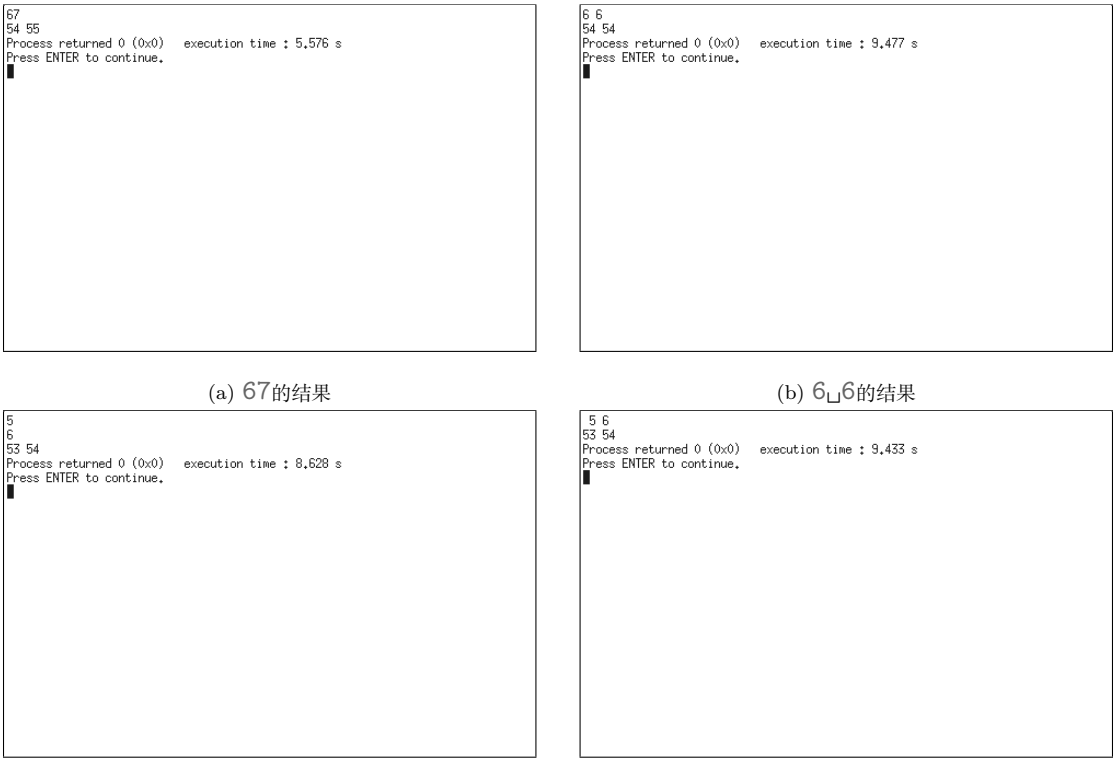


(a) 输入数据后的缓冲区　　(b) 读入数据后的缓冲区

图 18　格式串有空格处理字符 ␣5␣6 的缓冲区

因此，在命令行分别输入了67Enter、6␣6 Enter、5Enter6Enter和 ␣5␣6Enter后，其结果分别如图 19a-图 19d所示。

在这种情况下，由于格式串中的空格的存在，当使用%c 读取数据时，永远不可能读取到空白字符。此时，缓冲区中只要有空白字符，则会符合空格格式，从而就会被赋 (或叫吸收) 给空格格式。**其实所有的格式控制符都吸收的是满足格式要求的字符，但长度一般是有限度的，而空格格式控制符长度是无限制的所以他可以吸收所有连续的空白字符**。



(a) 67的结果　　(b) 6␣6的结果

(c) 5和6的结果　　(d) ␣5␣6 的结果

图 19　格式串中有空格的处理结果

5、scanf() 与其它输入函数混合使用

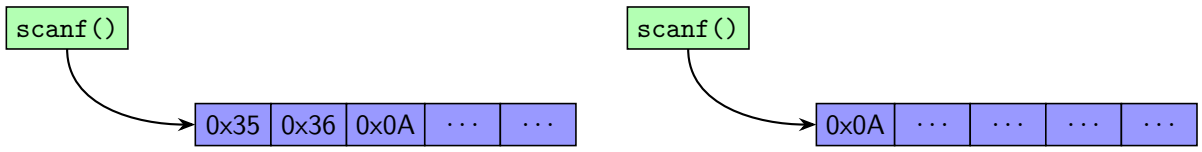
假设有如下代码：

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

int main()
{
    int a;
    int b;

    scanf("%d", &a);
    b = getchar();

    printf("%d %d", a, b);

    return 0;
}
```

该例中，先使用 scanf() 函数用%d 读入一个整数，赋给 a，再用 getchar() 函数读入一个字符，将其 ASCII 码赋给 b。那么，如果在命令行输入56Enter，则结果分别是什么？

此时，输入缓冲区中的数据如图 20a所示。

第 1 个 scanf() 函数将按格式字符串%d 整型数据 56，存入变量 a 中，此时，缓冲区中的数据如图 20b所示 (**注意此时'\n'仍然在缓冲区中**)。



(a) 输入数据后的缓冲区　　(b) 读入数据后的缓冲区

图 20　不同数据输入函数混用的缓冲区

当执行到 b = getchar(); 时，由于前一次 scanf() 函数留下了一个 '\n'，因此，不再要求用户输入数据，而是直接读入上一次 scanf() 函数留下的'\n'(图 20b)，并将其 ASCII 码值存入变量 b 中，此时，缓冲区中的数据如图 21所示 (**注意此时缓冲区为空**)。

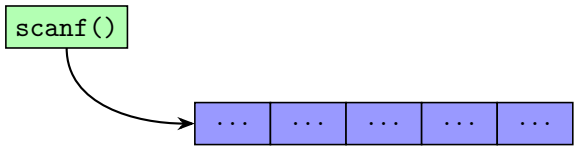


图 21　执行 getchar() 的缓冲区剩余数据

如果在命令行输入67Enter后，其结果如图 22所示。

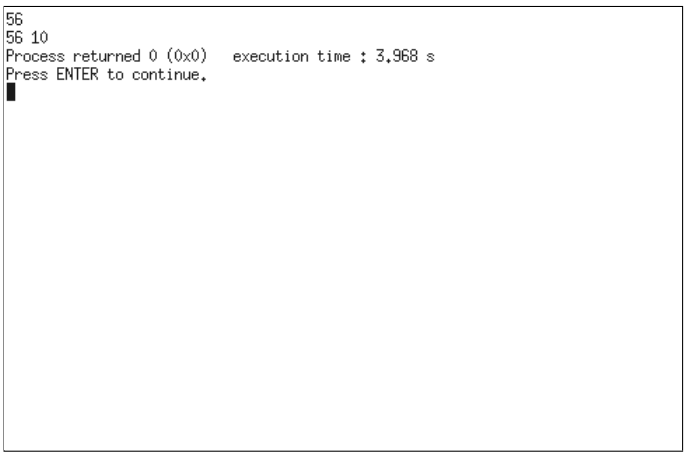


图 22　混合读入数据的运行结果

三、删除 scanf() 函数留下的'\n'

通过以上分析，可以看出执行 scanf() 函数后，会在缓冲区中留下一个'\n'，如果后续代码中使用的是%c 或是其它的输入函数，则会产生误操作，从而造成**不可预知的错误**。为此，应该在执行下一个读入操作前清空前一个 scanf() 在缓冲区所留下的不需要的数据，常用的方法有两个：

1、使用循环删除

可以通过一个循环，不断使用 getchar() 读取每个字符，当不是'\n' 时，继续读，相当于“吞掉了”不需要的字符，最终也“吸收”了 '\n'，其代码如下：

```
while (getchar () != '\n')
{
  continue;
}
```

2、使用正则表达式删除

可以根据“模式匹配”的规则，采用“正则表达式”“吞掉”不需要的字符，最终也“吸收”了'\n'，其代码如下：

```
scanf("%*[\t\n\r]");
```

四、结论

综上所述，在程序设计中数据类型与输入格式的匹配是及其重要的事，否则就有可能造成内存的非法使用或是数据的不完整，从而**造成程序的崩溃或不可预知的错误**。同时，scanf() 函数也有可能在缓冲区中留下多余的一个'\n'，如果后续代码中使用的是%c 或是其它的输入函数，则会产生误操作，从而造成**不可预知的错误**。

由此可见，scanf() 函数是读数据的一种有效但不理想的方法。许多专业 C 程序员会避免用 scanf() 函数，而是采用字符格式读取所有数据，然后再把它们转换成数值形式。但 scanf() 提供

了一种读入数据的简单方法，不过需要注意的是，如果用户录入了非预期的输入，那么许多程序都将无法正常执行

`scanf()` 函数看似是一个小问题，却引出一大堆问题，**编程无小事**呀。

在此，借用毛主席的的《沁园春 · 雪》填词一首，以期大家能领悟 “**编程无小事**” 的意境。

《沁园春 ·C》
编程代码，千行自在，万行逍遥。
望 CPU 内外，数据莽莽；
总线上下，顿失滔滔。
C 舞银蛇，内驰 01，欲与天公试比高。
Debug，看代码素裹，分外妖娆。

程序如此多娇，
引无数英雄竞折腰。
惜秦皇汉武，略输文采；
唐宗宋祖，稍逊风骚。
一代天骄，成吉思汗，只识弯弓射大雕。
俱往矣，数编程人物，还看今朝。