C语言中的输入输出功能和原理

I/O abilities and secrets in C

化学与材料科学学院 2023级 罗嘉宏

luojh@mail.ustc.edu.cn

中国科学技术大学

University of Science and Technology of China

接口简单而功能多样的I/O

printf/scanf函数族

丰富的功能主要体现在格式字符串中。

printf和scanf函数相信大家都很熟悉。
我们主要着重于介绍一些可能不是那么常见但是在解决某些问题上很“优雅”的功能。

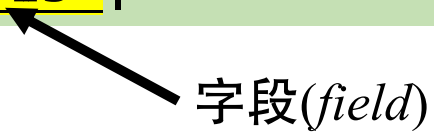
接口简单而功能多样的I/O

printf/scanf函数族

定宽输出时的对齐问题?

试想你有下面这样的需求：用ASCII字符打印表。这在Fortran语言中称为“表控模式”，具有很好的支持。一般这种表里面的field都是定宽的，所以需要用空格补齐长度不足的字符位置。

```
STUDENTS IN C PROGRAMMING COURSE
================================
NAME         FAMILY NAME    AGE     STUDENT NO.|
-------------------------------------------------|
JIAHONG      LUO              18  PB23030713  |
```



接口简单而功能多样的I/O

printf/scanf函数族

定宽输出时的对齐问题?

```
AGE     STUDENT NO.  |
---------------------|
  18 PB23030713      |
```

右对齐　　左对齐

一般来说，你会这样写代码：

```
printf(" %6d %11s|\n", age, studentID);
```

输出默认都是右对齐的：

```
AGE     STUDENT NO.  |
---------------------|
    18   PB23030713|
```

于是就有各种复杂操作。例如作者之前用过自己写的fixlen()函数，用于将一个字符串补齐到特定的长度，同时支持左右对齐。

练习：怎样实现一个fixlen()函数?

问：如何找空间容纳结果？怎样分配内存?

接口简单而功能多样的I/O

printf/scanf函数族

定宽输出时的对齐问题?

```
AGE     STUDENT NO. |
--------------------|
     18 PB23030713  |
```

右对齐　　左对齐

然而还是有更加优雅的方式:

```
printf(" %6d %-11s|\n", age, studentID);
```

输出就是这样:

```
AGE     STUDENT NO. |
--------------------|
     18 PB23030713  |
```

%-*length*X可以用于将定长输出转为左对齐。

这里的负号控制左对齐

接口简单而功能多样的I/O

printf/scanf函数族

有的时候，数值输出也要对齐。

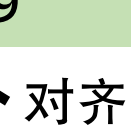
数值主要是分两类，整数和浮点数。

整数的对齐主要是符号位和field长度的问题

符号位的解决方法有两种：

1. 空格或负号
2. 正号或负号

```
THE MARTIX A = [
    0         1         2
----|---------|---------|---------
0   |+12345   |+1958    |-920
1   | 6360    |+2023    |+230307
2   |-13      | 1919    |-810     ]
```



接口简单而功能多样的I/O

printf/scanf函数族

其他进制?

程序员最爱(?)——八进制和十六进制(无符号)

八进制：**%o**
十六进制：**%x，%X**

最好在输出中标明一下输出的进制(例如，**0x**cccccccc)，特别是八进制。否则，这些输出很可能与十进制混淆。后面还会提到怎么自动加上0x这种前缀。

```
int a = 2023;
printf("%o %x %X\n", a, a, a);
printf("0%8o 0x%8x 0x%8X\n", a, a, a);
printf("0%08o 0x%08x 0x%08X\n", a, a, a);
```

```
3747 7e7 7E7
03747 0x     7e7 0x     7E7
0000003747 0x000007e7 0x000007E7
```

接口简单而功能多样的I/O

printf/scanf函数族

有的时候，数值输出也要对齐。

关于整数有3种格式说明符：

%+*n*d、% *n*d、%0*n*d。第一个表示无论正负都输出符号，第二个表示正号时输出空格，第三个表示用0补齐。

```
int a = -2023;
printf("%d\n", a);
printf("% d    %+d    %d\n", a, a, a);
printf("% d    %+d    %d\n", -a, -a, -a);
```

输出：

```
-2023
-2023    -2023    -2023
 2023    +2023    2023
```

```
int a = 2023;
printf("%d\n", a);
printf("%6d\n", a);
printf("% 6d\n", a);
printf("%+6d\n", a);
printf("%06d\n", a);
```

```
2023
  2023
  2023
 +2023
002023
```

接口简单而功能多样的I/O

printf/scanf函数族

有的时候，数值输出也要对齐。

浮点数采用科学记数法是比较好的选择(避免前导0影响field宽度)
一般而言，浮点数输出主要是下面几种格式说明符：

%e、%f、%g

主要是怎样的功能？请看下面的代码：

```
double ustc = 1958.0920;
printf("%e %f %g\n", ustc, ustc, ustc);
printf("%e %f %g\n", ustc * 1000.0, ustc * 1000.0, ustc * 1000.0);
printf("%e %f %g\n", ustc / 1e9, ustc / 1e9, ustc / 1e9);
```

```
1.958092e+03 1958.092000 1958.09
1.958092e+06 1958092.000000 1.95809e+06
1.958092e-06 0.000002 1.95809e-06
```

接口简单而功能多样的I/O

printf/scanf函数族

%e，%f，%g的使用(注：加上l，例如%lf，可以使用double浮点)

%e 强制使用科学记数法 用**%E**可以使用大写的E

%f 强制不使用科学记数法

%g 自动选择是否使用科学记数法

怎样控制小数点后位数？ %.*n*X 注：会四舍五入！

```
printf("%.2f %.4f %.10f\n", ustc, ustc, ustc);
```

```
1958.09 1958.0920
1958.0920000000
```

怎样控制总宽度(一般用于制表对齐)？ %*m*.*n*X

```
printf("%8.2f %8.4f %16.10f\n", ustc, ustc, ustc);
```

```
 1958.09  1958.0920
1958.0920000000
```

接口简单而功能多样的I/O

printf/scanf函数族

综合练习：用printf输出某位同学的信息，原始数据和格式分别为：

```
原始数据:
firstName = "Mingming"        不超过11位，char*
lastName = "Wang"             不超过11位，char*
studentNo = 7030123           PB+8位，前面补0，int
gpa = 3.1                     2位小数，右对齐，float
```

```
First name    Last name    Student No.    GPA
-------------------------------------------
Mingming      Wang         PB07030123       3.10
```

接口简单而功能多样的I/O

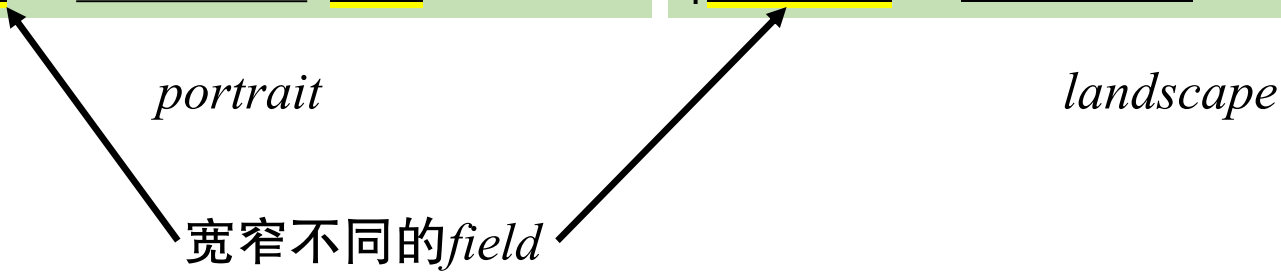
printf/scanf函数族

动态的格式?

有人会问，如果要动态地修改printf的格式字符串该怎么办?
例如，某个表格有宽窄两个版本，分别对应landscape和portrait的打印输出：

```
Undergraduate Students in USTC
Year   Major       Admission
-----------------------------
2023   Total       1966
```

```
|Undergraduate Students in USTC          |
|Year       Major           Admission    |
|----------------------------------------|
|2023       Total           1966         |
```



接口简单而功能多样的I/O

printf/scanf函数族

动态的格式?

此时，除了用sprintf(后面会提到)构造可变的格式串之外，更好的方法是使用符号*来表示格式说明中待定的数值。

```
int fieldLength;
scanf("%d", &fieldLength);
printf("Variable width field: [%*d]\n", fieldLength, 19580920);
```

```
10
Variable width field: [  19580920]
```

```
15
Variable width field: [       19580920]
```

当宽度不足？会补齐宽度(相比Fortran直接输出一堆星号好很多)

```
6
Variable width field: [19580920]
```

接口简单而功能多样的I/O

printf/scanf函数族

再谈大小

到底要多大？printf/scanf中整数的大小问题 %d，%x，%X(注：这个PPT中的*斜体X*表示某个格式说明符)，%u/x/X/o等是类型说明，而大小则是用下面4个：

%l：增大一级，例如%lld=*long long* %d, %lf=double (%f是float)
%h：缩小一级，例如%hd=*short* %d(也就是short int) hhd=*char*
%z：size_t或者说机器字长，x64上是__int64。
%L(**f**/**e**/**g**)：表示使用long double

注：对于要频繁处理或构造定宽数据的程序(例如用结构体构造WAV、BMP文件头)，建议采用Windows的方案：使用机器架构(例如i386, x64, ia64, PowerPC等)相关的头文件并定义DWORD这种已知宽度的数据类型和它对应的各种常量(例如DWORD对应%x等等)。

接口简单而功能多样的I/O

printf/scanf函数族

打印指针？打印机器字长(size_t那么长)的十六进制数？

比如说，需要打印一个指针指向的地址，怎么做才能适用于各种机器？

%u(%x)吗？在32位机器上也许可以，64位不行。
%llu吗？32位上可能有警告。

所以，稳妥的方法是%p和%zu(%zx)。刚才已经说过，%z*X*表示size_t大小的整数。

```
int a = 2023;
int* p = &a;
printf("The value of a is %d, which is stored at 0x%p.\n", a, p);
```

```
The value of a is 2023, which is stored at 0x0000004bcfdffb24.
```

接口简单而功能多样的I/O

printf/scanf函数族

0x等前缀?

加上一个#即可。例如：

```
printf("The value of a is %#x.\n", a);
```

```
The value of a is 0x7e7.
```

```
printf("The value of a is %#o.\n", a);
```

```
The value of a is 03747.
```

浮点数输出时，#保证输出小数点

```
printf("The value of a is %#8.0f, or
%8.0f\n", 1958.0, 1958.0);
```

```
The value of a is    1958., or
1958
```

接口简单而功能多样的I/O

printf/scanf函数族

字符串只想输出一部分?

%[n].***m***s格式说明符用于要求只输出m个字符

```
char* str = "USTC1958";
    printf("[%s]  [%.5s]  [%10s]  [%10.5s]  [%-10.5s]",
        str, str, str, str, str);
```

```
[USTC1958]  [USTC1]  [  USTC1958]  [     USTC1]  [USTC1     ]
```

练习：如何输出str中从第2个字符开始的3个字符?

接口简单而功能多样的I/O

printf/scanf函数族

输出了多少个字符?

%n格式说明符要求向对应的地址中写入目前已经输出的长度(本身不占宽度)。printf的返回值是总共输出的长度。

```
int x = 1958;
int llen, rlen;
int totalLen = printf("USTC was %nfounded in %d%n.\n", &llen, x, &rlen);
printf("llen = %d, rlen = %d, total = %d.\n", llen, rlen, totalLen);
```

```
        9              24 26
        ↓              ↓  ↓
USTC was founded in 1958.\n
llen = 9, rlen = 24, total = 26.
```

接口简单而功能多样的I/O

printf/scanf函数族

练习：解释scanf("(%d)%d", &a, &b);的工作过程。

而scanf呢?

scanf的工作过程？例如，scanf("%d", &a);

1. scanf首先跳过任何的空白字符
2. 你需要scanf读取%d，那么它遇到第一个非空白字符时，就会判断它是否可能是整数的一部分(0-9,+,-)
3. 然后，scanf读取这个整数，直到遇到第一个非数字字符*ch*
4. 由于字符*ch*已经被读取，为了不影响后面的读取，scanf将其放回in缓冲区内
5. %c基本等价于getchar，不考虑空白字符(简单地读取in缓冲区中下一个字符)
6. 格式串中非格式项的部分，scanf会匹配它们并跳过去

接口简单而功能多样的I/O

printf/scanf函数族

而scanf呢?

大部分与printf是类似的，注意以下几点：

- scanf接受的空白字符包括空格，换行符，制表符
- %s字符串输入不接受空格
- %*n*X表示最多在当前的*X*项中读入*n*个字符(从第一个有效字符开始计算)
- *符号可以让scanf读取这一项但是不写入到变量中(假读取，例如%*d)
- %n读回目前已经读取的字符数。

接口简单而功能多样的I/O

printf/scanf函数族

而scanf呢?

练习：从下面这样的输入中提取浮点数部分，并输出这一部分的长度。

```
The USTC was founded in (1.958e3).
```

提示：scanf会尝试匹配格式字符串中不是格式描述符的部分。用%n获得与长度有关的信息。

接口简单而功能多样的I/O

printf/scanf函数族

而scanf呢?

scanf的返回值?

输入的数据与格式字符串中格式项的匹配次数。(也就是赋值成功的数据项数)
出错的情况下返回EOF(-1)。

控制台上怎么给出文件中的文件末端指示符(EOF)? 按Ctrl+Z两次得到^Z^Z然后回车Enter即可。

接口简单而功能多样的I/O

printf/scanf函数族

而scanf呢?

scanf吞换行符的问题
常见于这样的代码:

```
printf("Input to a: ");
scanf("%d", &a);          ①
printf("Input to ch: ");
scanf("%c", &ch);         ②
```

用户肯定在输入a的值之后回车，然后由于scanf的机制，这个回车符仍然是在in缓冲区里面的(语句①中的格式串没有\n，故scanf不会读入\n，回车只是指示scanf开始匹配而已)。
于是下一次②中用%c就会把缓冲区的下一个字符读出来，读取到的就是\n了。
%c不受空白分隔字符的影响。它就是简单地读取下一个字符。

接口简单而功能多样的I/O

printf/scanf函数族

而scanf呢?

怎么解决？下面三个代码，哪(个)些是可行的?

```
printf("Input to a: ");
scanf("%d\n",&a);
printf("Input to ch: ");
scanf("%c",&ch);
```

```
printf("Input to a: ");
scanf("%d",&a);
printf("Input to ch: ");
scanf("\n%c",&ch);
```

```
printf("Input to a: ");
scanf("%d",&a);
getchar();
printf("Input to ch: ");
scanf("%c",&ch);
```

接口简单而功能多样的I/O

printf/scanf函数族

而scanf呢?

怎么解决？第一个不行，在于它要读到**一个整数+一个\n再回车**才能启动读取。

```
printf("Input to a: ");
scanf("%d\n", &a);
printf("Input to ch: ");
scanf("%c", &ch);
```

```
printf("Input to a: ");
scanf("%d", &a);
printf("Input to ch: ");
scanf("\n%c", &ch);
```

```
printf("Input to a: ");
scanf("%d", &a);
getchar();
printf("Input to ch: ");
scanf("%c", &ch);
```

吞掉回车

接口简单而功能多样的I/O

printf/scanf函数族

而scanf呢?

怎么解决? 下面还有3个有趣的实现方法:

```
printf("Input to a: ");
scanf("%d", &a);
printf("Input to ch: ");
scanf("%c%c", &dummy, &ch);
```

```
printf("Input to a: ");
scanf("%d", &a);
printf("Input to ch: ");
scanf("%*c%c", &ch);
```

```
printf("Input to a: ");
scanf("%d", &a);
printf("Input to ch: ");
scanf("%c%c", &ch, &ch);
```

接口简单而功能多样的I/O

printf/scanf函数族

而scanf呢?

换一种定界符(原来是空白字符作为定界符)?

接受读取a,b,c中的任何一个字符，其他字符视为空格一样的分隔符

```
char buf[10];
scanf("%[abc]", buf);
scanf("%[^abc]", buf); 接受除a,b,c外的任何一个字符
```

[*characters*]	Scanset	Any number of the characters specified between the brackets. A dash (-) that is not the first character may produce non-portable behavior in some library implementations.
[^*characters*]	Negated scanset	Any number of characters none of them specified as *characters* between the brackets.

接口简单而功能多样的I/O

printf/scanf函数族

练习：用scanf实现读取一整行。

提示：用%[*chars*]，应该接受某些*chars*还是不接受某些*chars*？*chars*怎样选取？

接口简单而功能多样的I/O

有关in缓冲区的有趣函数：ungetc：将一个字符放回输入流里面

```
int ungetc(int char, FILE *stream)
```

scanf在读到不合适的字符(例如%d读了一些整数位后遇到了个空格)时，就是使用类似的方法将这个字符放回到缓冲区里面供后续读取。

接口简单而功能多样的I/O

别人是怎么做的？D. E. Knuth的TEX排版引擎和catcode

The mascot of TEX

Category	*Meaning*	
0	Escape character	(\ in this manual)
1	Beginning of group	({ in this manual)
2	End of group	(} in this manual)
3	Math shift	($ in this manual)
4	Alignment tab	(& in this manual)
5	End of line	(⟨return⟩ in this manual)
6	Parameter	(# in this manual)
7	Superscript	(^ in this manual)
8	Subscript	(_ in this manual)
9	Ignored character	(⟨null⟩ in this manual)
10	Space	(␣ in this manual)
11	Letter	(A, . . . , Z and a, . . . , z)
12	Other character	(none of the above or below)
13	Active character	(~ in this manual)
14	Comment character	(% in this manual)
15	Invalid character	(⟨delete⟩ in this manual)

Knuth在设计TEX时，采用了一种很优雅的方式处理混合的输入：字符类别码(*category code*, 简写为*catcode*)。

128个标准ASCII字符被分为语法上的16种类别，并按照一定的规定读取。

The TEXBook, Chapter 7, p. 37

接口简单而功能多样的I/O

别人是怎么做的？D. E. Knuth的TEX排版引擎和catcode

Category	*Meaning*	
0	Escape character	(\ in this manual)
1	Beginning of group	({ in this manual)
2	End of group	(} in this manual)
3	Math shift	($ in this manual)
4	Alignment tab	(& in this manual)
5	End of line	(⟨return⟩ in this manual)
6	Parameter	(# in this manual)
7	Superscript	(^ in this manual)
8	Subscript	(_ in this manual)
9	Ignored character	(⟨null⟩ in this manual)
10	Space	(␣ in this manual)
11	Letter	(A, . . . , Z and a, . . . , z)
12	Other character	(none of the above or below)
13	Active character	(~ in this manual)
14	Comment character	(% in this manual)
15	Invalid character	(⟨delete⟩ in this manual)

一般而言，TEX中的命令是下面这样的：

```
{\hskip 36 pt}
```

TEX接受两种token(语法单元)，分别是单个的字符和一个控制串(Conteol Sequence)，也就是跟在***Escape char*$_0$**后面的一串***Letter*$_{11}$**。

那么，上面这个语句就会被分解成

{$_1$ hskip$_{\text{token}}$ 3$_{12}$ 6$_{12}$ ␣$_{10}$ p$_{11}$ t$_{11}$ }$_2$

The TEXBook, Chapter 7, p. 38

接口简单而功能多样的I/O

别人是怎么做的？D. E. Knuth的TEX排版引擎和catcode

一般而言，TEX中的命令是下面这样的：

```
{\hskip 36 pt}
```

TEX接受两种token(语法单元)，分别是单个的字符和一个控制串(Conteol Sequence)，也就是跟在***Escape char*$_0$**后面的一串***Letter*$_1$**。

那么，上面这个语句就会被分解成

$\{_1$ hskip$_{\text{token}}$ 3$_{12}$ 6$_{12}$ ␣$_{10}$ p$_{11}$ t$_{11}$ $\}_2$

因此，TEX内部就可以用状态机的方式读取源文件并将其分解和解释。这种方法非常适合于读取复杂且有结构的文本(例如程序源代码)。

The TEXBook, Chapter 7, p. 38

接口简单而功能多样的I/O

别人是怎么做的？D. E. Knuth的TEX排版引擎和catcode

- 此外，这套catcode很便于扩展功能。例如，某人希望用<和>代替{和}，就可以令catcode$_{<}$=1，catcode$_{>}$=2。这样，TEX就可以把<和>作为定界符了。
- 又比如说，就像C/C++中常用下划线_表示内部函数一样，TEX中也有类似的需求，但是TEX中下划线是用来表示下标的命令(例如x_1=10会得到x_1=10)。

怎么办呢？有一个符号@。TEX中一般带@的命令是内部命令，这些命令如果在外部访问就会出现错误。这是怎么做到的呢？

一般情况下的用户代码中，@会在12类中(Other)，此时使用@就会因为控制串中只能有Letter$_{11}$而报错。而在内部代码中，将@调为Letter$_{11}$，此时@就是一个和一般字母没有区别的字符，自然就可以用在控制串中了。

接口简单而功能多样的I/O

别人是怎么做的？D. E. Knuth的TEX排版引擎和catcode

练习：用scanf实现读取Windows Config File (*.ini)。

Windows Config File是很常见的配置文件格式。其格式如下：

[组$_1$名称]
键$_1$=值$_2$
键$_2$=值$_2$
…
[组$_2$名称]
键$_n$=值$_n$
…

接口简单而功能多样的I/O

printf/scanf函数族还可以打印到哪里？有哪些变种？

除了标准的`printf`/`scanf`打印到屏幕上外，还有下面两种以及它们的变形：

- `fprintf`/`fscanf`：打印到某个流(注意也不一定是文件)
- `sprintf`/`sscanf`：打印到某个内存区域(例如`char[]`)
- v*X*`printf`/v*X*`scanf`：vararg版本的*X*`printf`/*X*`scanf`
- *X*`printf_s`/*X*`scanf_s`：安全的*X*`printf`/*X*`scanf`
- *X*`wprintf`/*X*`wscanf`：宽字符(`wchar_t`)版本的*X*`printf`/*X*`scanf`
  - 注：似乎是Windows限定款？毕竟Linux系上已经普遍支持utf-8了(Linux上有但少用)
  - 破cp936!!
- 还有*libicu*这种国际化库(笑)
- 还有好多好多的变形，例如locale等等，请看MSDN…

后面我们再讨论这些函数的使用。

程序员的锅(代码本身缺陷，并非外部问题)

看看下面几种常见的错误写法

```
int a;
scanf("%d", a);
```

忘了加上&

警告

printf/scanf这种基于可变参数实现的函数和编译器**没有任何义务检查格式串和可变参数中的任何问题**！**不提供类型转换**！

```
int a;
a = scanf("%d");
```

scanf的结果不在返回值(Python程序员经常这样)！

```
int a, b;
scanf("%d%d", &a);
```

漏写参数，常见于后期增补的代码

(一般而言，多写参数不会有什么不好的后果，但是要避免)

```
int a;
scanf("d", &a);
```

格式串漏了%

程序员的锅(代码本身缺陷，并非外部问题)

看看下面几种常见的错误写法

```
int a;
scanf("%lld", &a);
```

写错格式说明符，**严重者可能导致读取的数据有问题或者缓冲区溢出！**

```
char str[10];
scanf("%s", &str);
```

scanf中str不需要加&

```
int a = 1958;
printf("%d", &a);
```

printf中写了&

```
int a = 1958;
printf("%d%d", &a);
```

试图读取超过指定个数的变量：可能导致内存地址上相邻的敏感数据被输出！

程序员的锅(代码本身缺陷，并非外部问题)

因此，为了降低出错的可能性：

- 注意格式串的跨平台性
- 避免使用可变的格式串
- 尤其是禁止使用用户提供的格式串

stdio的潜在安全问题

stdio在设计时并没有考虑到某些可能的安全问题(早期Unix开发的普遍现象)。最常见的安全漏洞就是字符串缓冲区溢出(buffer overflow)

要了解这个漏洞是如何产生的，请先看函数调用方式(calling convention)

怎样的代码可能有安全问题?

```
int fence = 0x19580920;
char password[10] = { 0 };
int another = 0x20230920;
scanf("%s", password);

if (strcmp(password, "ustc") == 0)
    printf("Access OK!");
else
    printf("Permission denied.");

return 0;
```

鉴于msvc实在是过于安全(加入了很多security cookie之类的无关代码)，我们用gcc编译这段代码并用objdump看一看其对应的汇编代码。

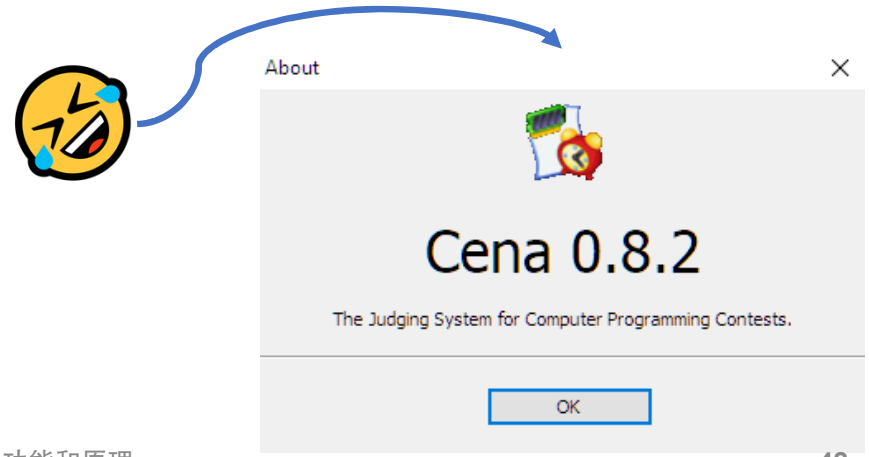
怎样的代码可能有安全问题?

编译命令:
```
bin\gcc -S -masm=intel v1.c -o v1.asm
```

-S：只编译，不汇编和链接

-masm=intel：输出Intel格式的汇编代码(只是个人习惯Intel式的代码)

鉴于msvc实在是过于安全(加入了很多security cookie之类的无关代码)，我们用gcc (x86且旧版本，可以从广为人知的Cena中找)编译这段代码。



gcc -v:

```
...
Thread model: win32
gcc version 3.4.2 (mingw-special)
```

前面是字符串部分…

```
    push    ebp
    mov     ebp, esp
    push    edi
    sub     esp, 84
    and     esp, -16
    mov     eax, 0
    add     eax, 15
    add     eax, 15
    shr     eax, 4
    sal     eax, 4
    mov     DWORD PTR [ebp-60], eax
    mov     eax, DWORD PTR [ebp-60]
    call    __alloca
    call    ___main
    mov     DWORD PTR [ebp-12], 19580920h   ①
    lea     edi, [ebp-40]       起始地址
    cld                         rep stosb从低到高
②   mov     ecx, 10             填充的个数
    mov     al, 0               填充内容为\0
    rep     stosb               执行填充
    mov     DWORD PTR [ebp-44], 20230920h   ③
```

ESP=(EBP-84)&(-16)？位与(-16)有什么用？

$(-16)_{DWORD}$=FF FF FF F0h

位与(-16)相当于清除最低的4位
可以使得ESP对齐16字节的边界。

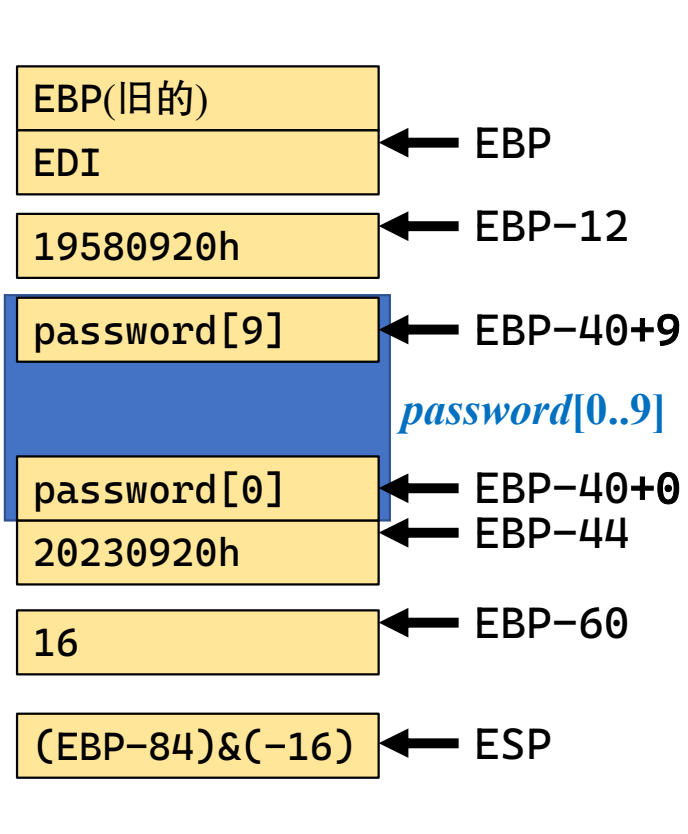
```
int fence = 0x19580920;       ①
char password[10] = { 0 };    ②
int another = 0x20230920;     ③
scanf("%s", password);

if (strcmp(password, "ustc") == 0)
    printf("Access OK!");
else
    printf("Permission denied.");

return 0;
```

(接上一页)

```
lea     eax, [ebp-40]                          ④
mov     DWORD PTR [esp+4], eax
mov     DWORD PTR [esp], OFFSET FLAT:LC0
call     _scanf
lea     eax, [ebp-40]                          ⑤
mov     DWORD PTR [esp+4], OFFSET FLAT:LC1
mov     DWORD PTR [esp], eax
call     _strcmp
test     eax, eax                              if
jne     L2
mov     DWORD PTR [esp], OFFSET FLAT:LC2       ⑥
call     _printf
jmp     L3
L2:
mov     DWORD PTR [esp], OFFSET FLAT:LC3       ⑦
call     _printf
L3:
mov     eax, 0                                 ⑧
mov     edi, DWORD PTR [ebp-4]
leave
ret
```

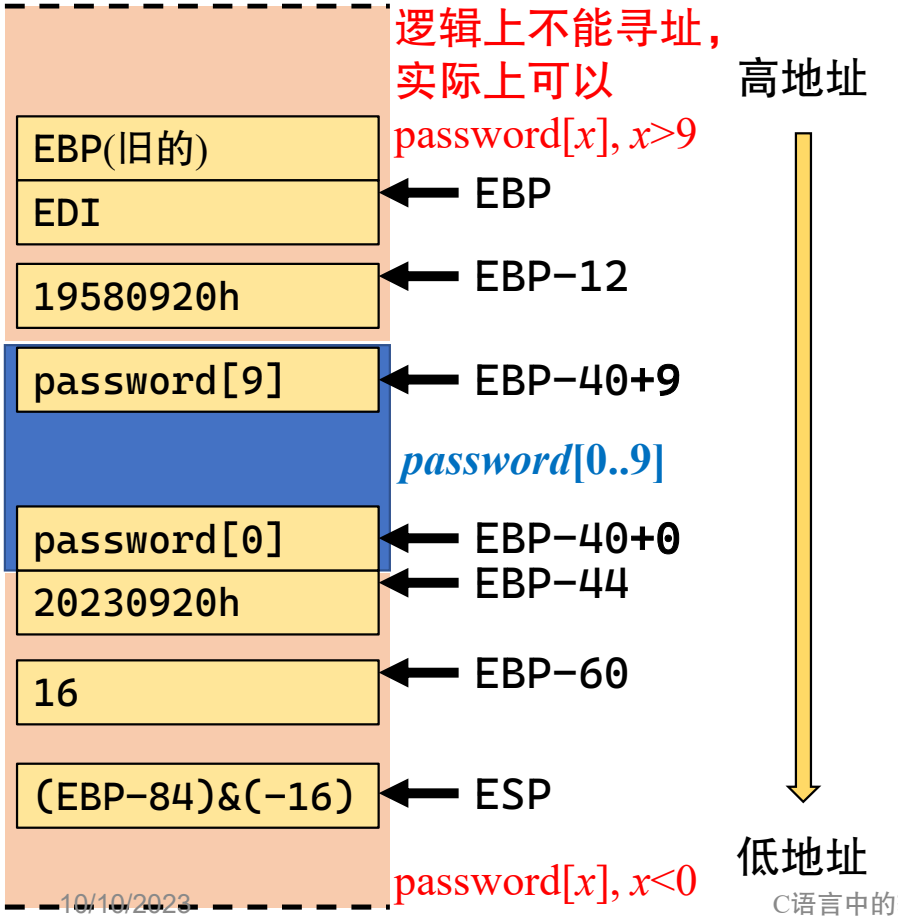
```
int fence = 0x19580920;
char password[10] = { 0 };
int another = 0x20230920;
scanf("%s", password);                         ④

if (strcmp(password, "ustc") == 0)             ⑤
    printf("Access OK!");                      ⑥
else
    printf("Permission denied.");              ⑦

return 0;                                      ⑧
```

怎样的代码可能有安全问题?



我们看到：password[10]是在栈上分配的空间。根据上面的汇编代码，推断出其栈的分布应该如图。

怎样的代码可能有安全问题?

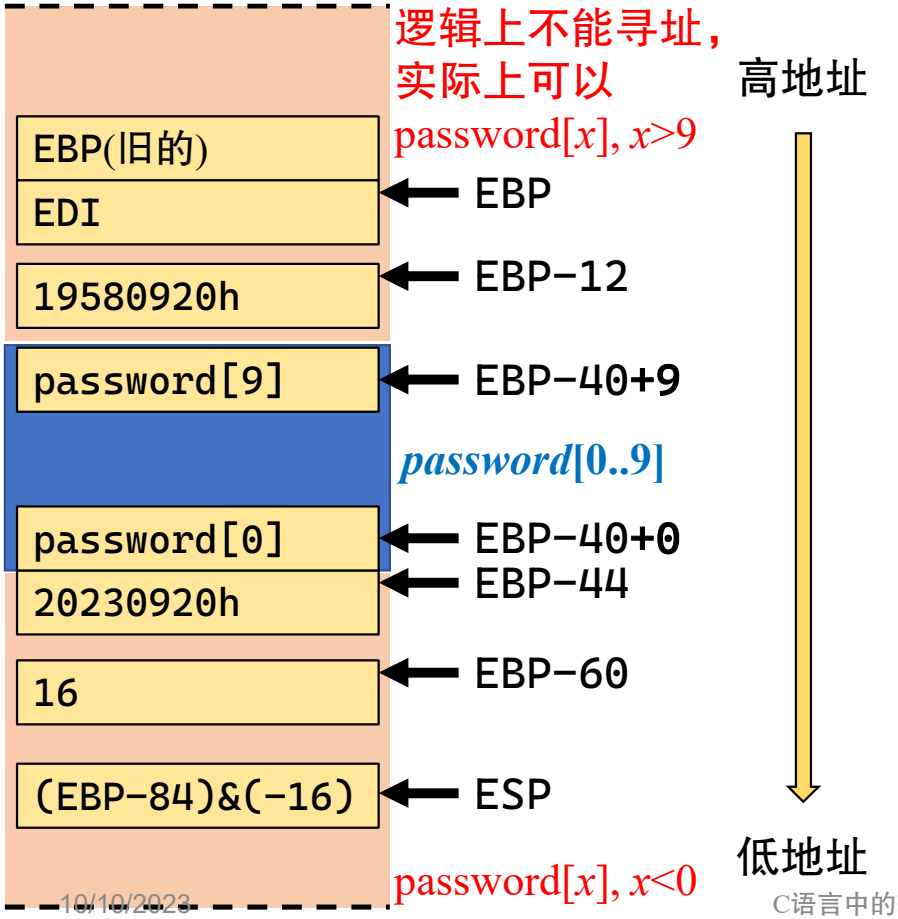


实际上，password[x]就是一个语法糖，它也可以改写成下面的代码：

password[x] ≡ (&(password[0]))+x

所以可以看到，利用超出[0,10)范围的x实际上可以对超出password[0..9]的内存进行寻址。

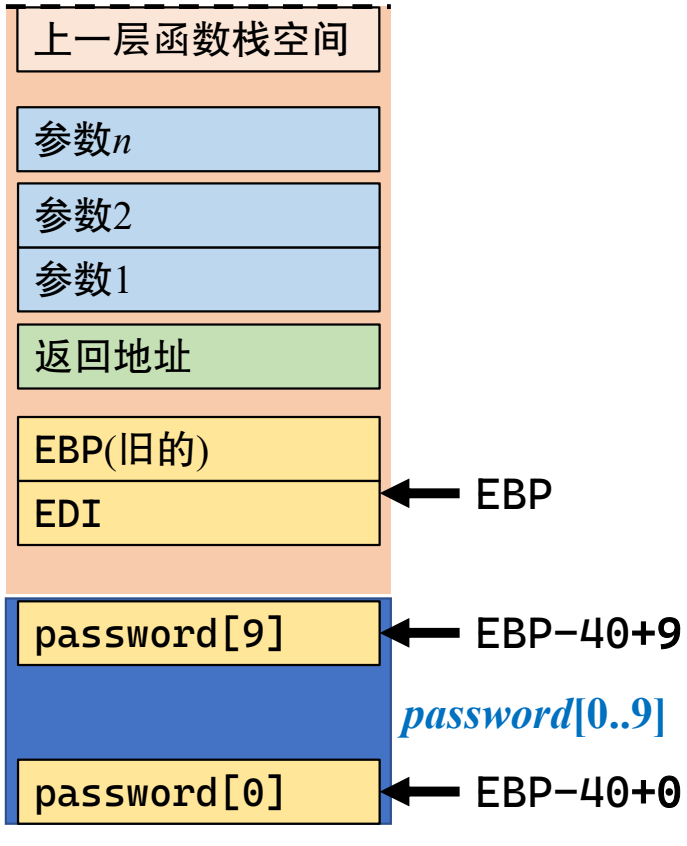
怎样的代码可能有安全问题?



printf在使用%s格式说明符输入字符串时，只接受字符串的首字符地址，并不关心可以写入的长度。

所以，当用户输入一个超过password长度10的字符串时，就会导致printf将password[10]以及以后的内存数据覆盖为输入的字符串。
这种覆盖会导致什么？来看看高地址区域都有什么。

怎样的代码可能有安全问题?



参考**x86调用和返回机制**(在handout上)，你可以看到高地址区域的栈结构。

某些特定的区域，例如**返回地址**、**参数**以及**其他局部变量**，一旦被修改就有可能改变程序的执行路径(例如修改返回地址就可以修改返回时跳转到的代码位置)，从而达到执行任意代码的目的(shellcode等等)。

怎样的代码可能有安全问题?

类似的，像gets这样不检查缓冲区大小就直接写入数据的函数也是不安全的。

对于print类函数，也要注意像%n这样可能导致本来应该是被显示出来的数据被理解成了指针，而导致这些数据指向的地址被写入的格式说明符(printf/scanf只是根据格式说明符解释并访问其参数，并不能知道代码中这个参数具体是什么类型)。

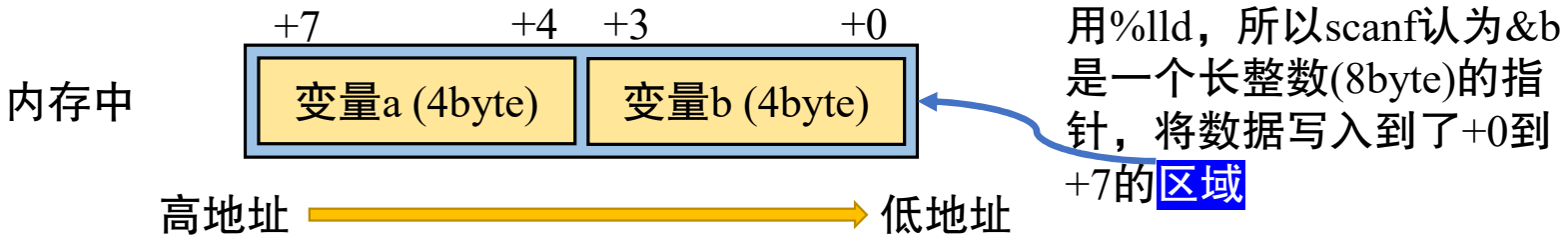
别以为只有%s可能导致问题，看起来很没有问题的代码也可能有问题。请看：

```
int a = 2023; // Correct password
int b = -1;
scanf("%lld", &b);
if(b == a)
    printf("Correct password!\n");
printf("a = %d, b = %d.\n", a, b);
```

```
8409545967526          (键盘输入)
Correct password!
a = 1958, b = 1958.
```

怎样的代码可能有安全问题?

啥? **840954596752**6怎么造出来的?
首先，07a6h=1958。然后，0000 07a6 0000 07a6h=840954596752**6**



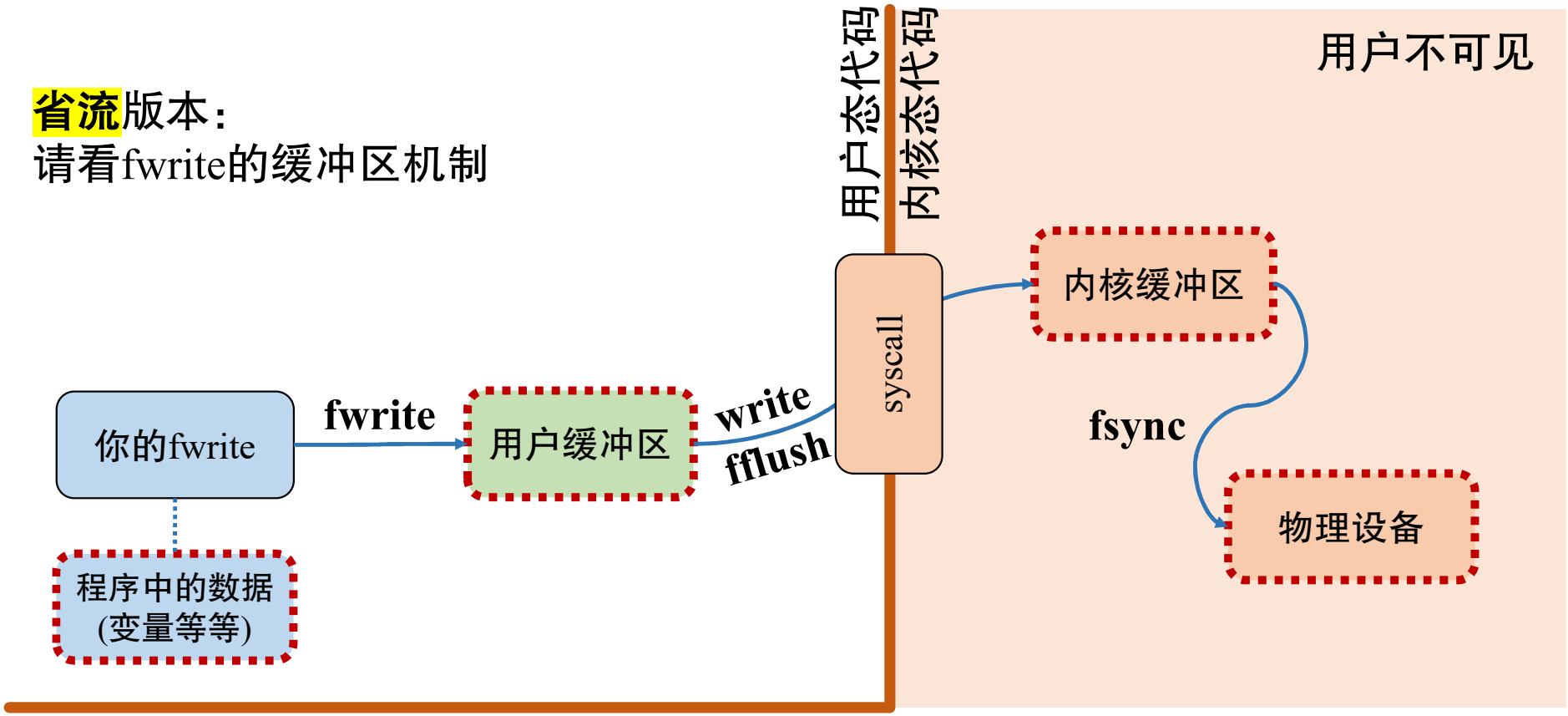
结果就是，a=000007a6h(原来的2023被覆盖掉了)，b=000007a6h。

一定程度上预防stdio安全问题的方法

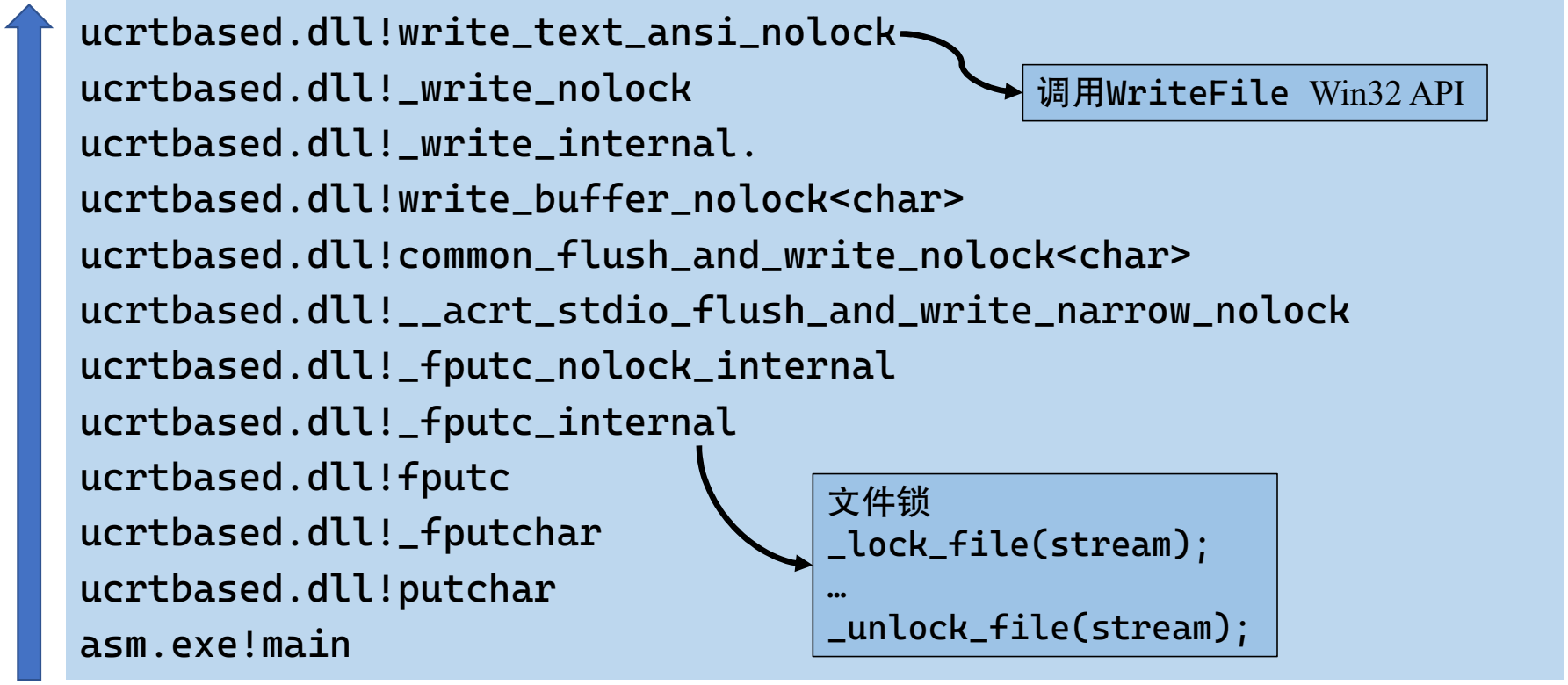
1. 禁止使用用户输入/提供的格式字符串
2. 避免使用可变的格式字符串(例如`printf(format_string);`)
3. 避免在`printf`中使用`%n`等可能导致意外数据写入的格式说明符
4. 当`scanf`如果遇到要输入字符串(`%s`)，请换用`scanf_s`
5. 禁止使用`gets`等有安全问题的旧式函数，请使用`fgets` (*GJB 8114-2013*)
6. 禁止使用三字母词(这是啥?!没听过就更好了! *GJB 5369-2005*)

stdio的缓冲区机制

省流版本：
请看fwrite的缓冲区机制



缓冲区是什么？请看putchar()的调用过程

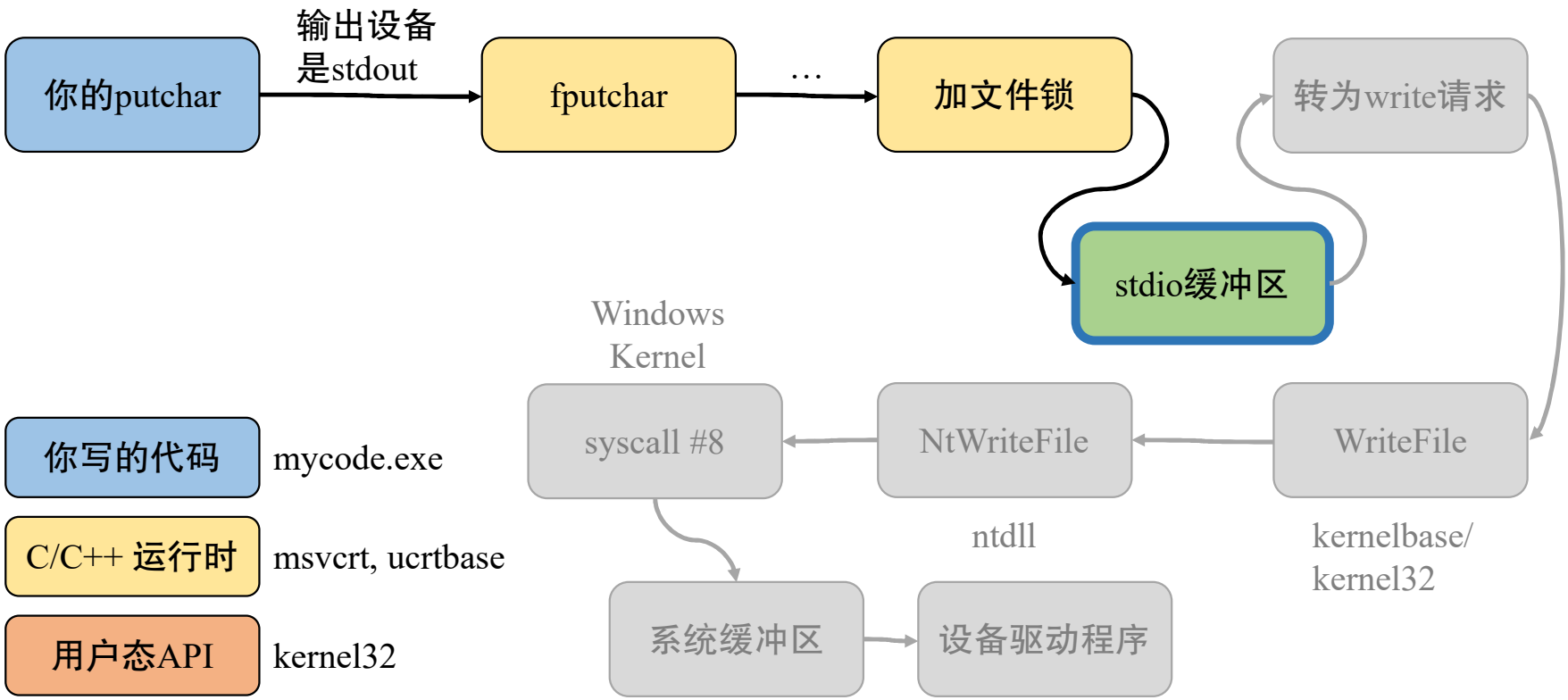


缓冲区是什么？请看putchar()的调用过程

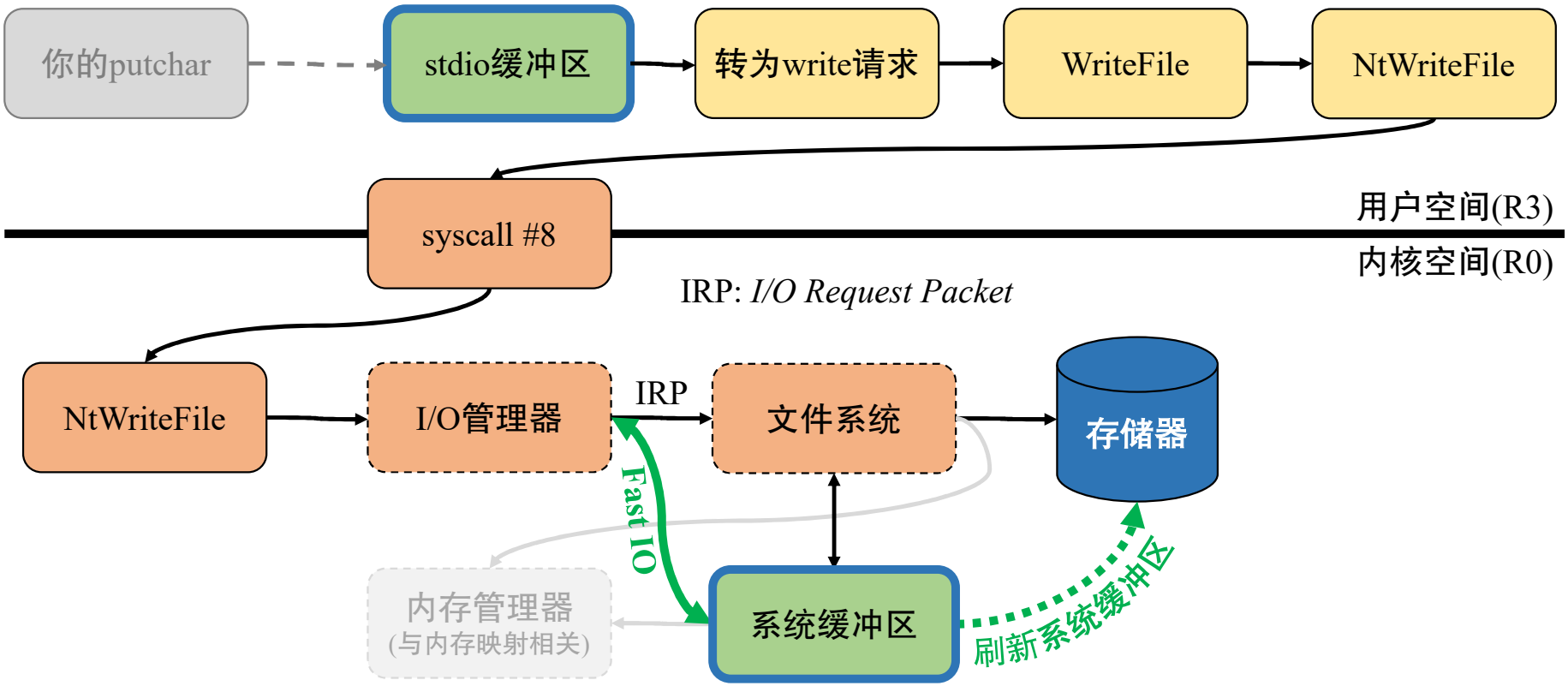
如何找到相关系统调用实现的细节信息？

1. Windows: Windows SDK源代码中VC Runtime实现 + Windows Research Kernel (WRK, 基于Windows Server 2003 / Windows XP)中的内核源代码
2. Linux: glibc (GNU C Runtime Library)源代码 + Linux内核源代码

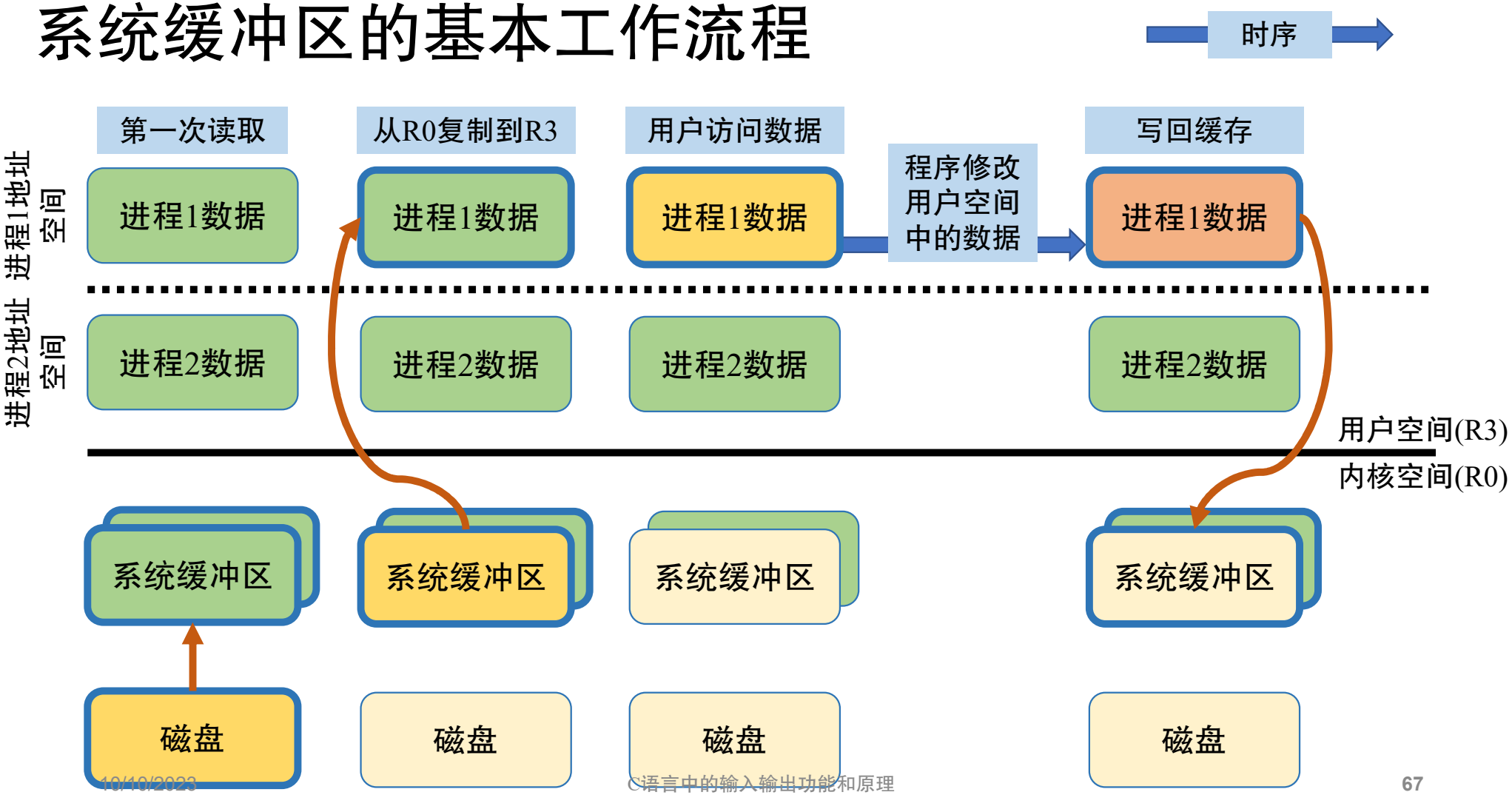
R3层面的缓冲区(用户缓冲区)



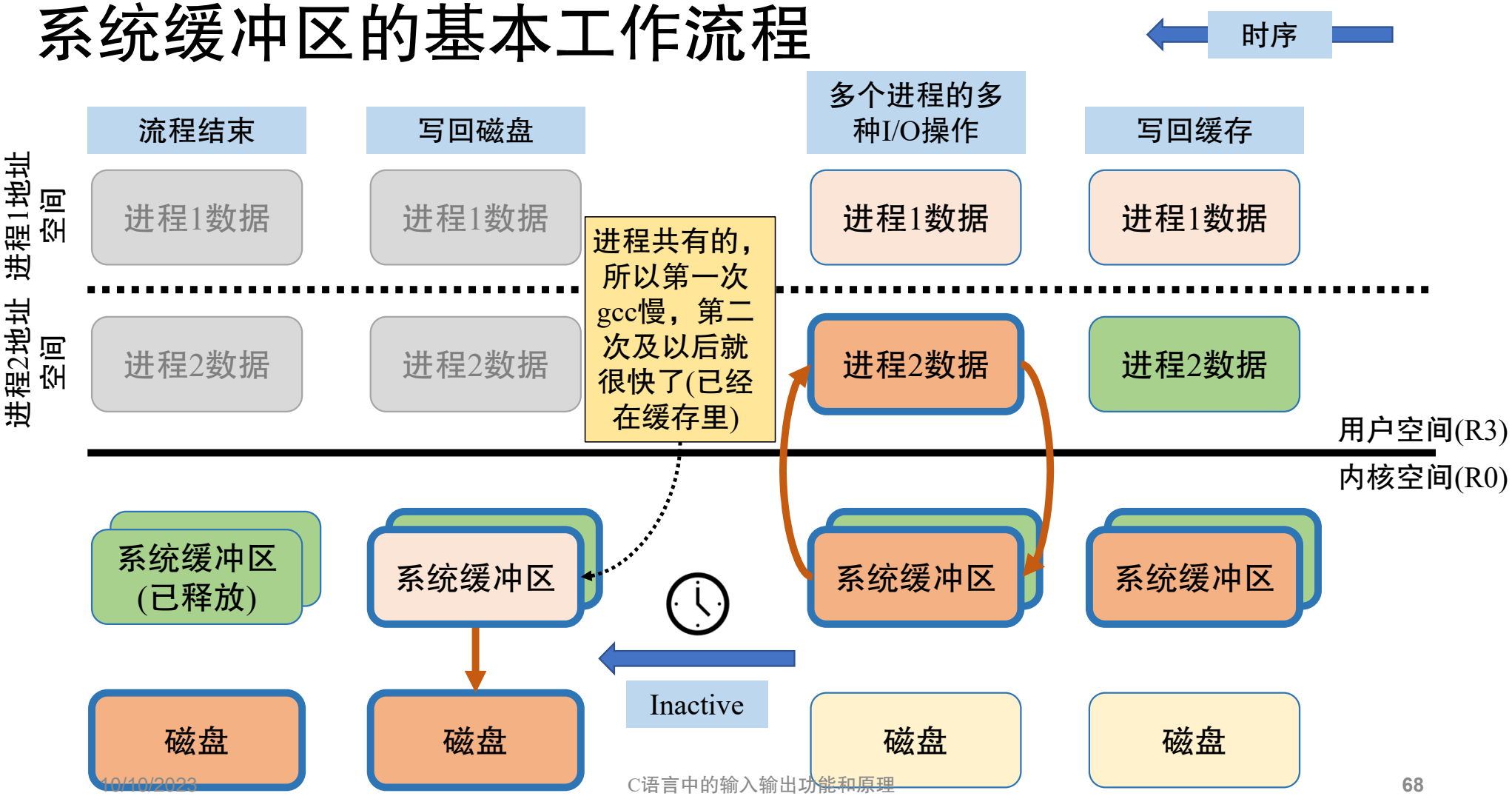
R0层面的缓冲区(系统缓冲区)



系统缓冲区的基本工作流程



系统缓冲区的基本工作流程



系统缓冲区的控制方式

- 关闭缓存(读取和写入都直接操作)。FILE_FLAG_NO_BUFFERING标志。
- 写入操作立即刷新到磁盘(一般是写入)。FILE_FLAG_WRITE_THROUGH标志。数据仍会写入缓存，但缓存管理器会立即将数据写入磁盘，而不会产生延迟。
- 调用FlushFileBuffers函数强制刷新已打开的文件。
- 为了将元数据更改存储到磁盘，必须刷新文件或使用FILE_FLAG_WRITE_THROUGH标志。
- 文件在关闭后最终OS会保证将所有数据刷新到磁盘。

如何控制缓冲区

Part 0. 标准控制台I/O流的缓冲设定(一般情况，不做保证)?

- stderr一般是不缓冲的，所以写入到stderr的数据一般情况下会立即显示出来;
- stdin和stdout是行缓冲(line-buffer)，也就是一行一次flush(如果缓冲区满了的话也会提前flush);
- 也有full-buffer，就是单纯的满了就flush。

C中如何控制缓冲区

Part 1. 何时刷新?

- 一般情况下，系统会很好地帮我们处理好缓冲区的问题。
- 即使不手动操作缓冲区，也可以确保正常I/O。
- 只有在特殊需要的情形下才手动控制缓冲区。

- **适时使用fflush(原始I/O则为write函数)，刷新用户缓冲区的数据到内核缓冲区。**
  - 为什么“适时”？频繁进行系统调用可能影响I/O性能

- **关键数据(例如重要的日志)使用fsync确保写入!**
  - 同理，频繁访问磁盘不是好事情。所以请只在需要确保写入的情况下才调用**fsync**。相当于将内核缓冲区中的数据写入磁盘。

C中如何控制缓冲区

Part 2. 怎样的情况下会出现数据丢失?

- 正常情况下，进程退出时系统会保证数据被刷新和写入。
- 在某些进程突然中止的情况下可能丢失未写入的数据。
  - 例如，进程**抛出异常**，然后在未处理异常的情况下终止;
  - 例如，**调试器**挂载在进程上的时候，进程抛出异常(此时调试器立即接管这个进程的控制流和内存)，进程来不及刷新缓冲区(最常见的现象是某个本该显示的错误信息没有显示出来);
  - 又例如，**突然掉电**的情况下，内存中缓冲区里面未及时写入的数据会丢失。

如何控制缓冲区

对于关键的控制台输出，可使用setbuf并传入NULL来禁用缓冲区。这样，所有输出的内容都能保证显示出来。

Part 3. 如何调整缓冲区?

- 使用函数setbuf可以精调stdio缓冲区(用户缓冲区)的位置。

```
setbuf(文件流, 缓冲区地址);
```

- 缓冲区应该有多大? <stdio.h>中的BUFSIZ宏给出了缓冲区应有的大小。
- 缓冲区地址取NULL的含义是不用缓冲区。

错误范例:

```
char buf[BUFSIZ];
setbuf(stdout, buf);
```

最后一次刷新缓冲区是在main()返回后，此时buf已经被释放!
所以，建议用**static** char buf[…];

如何控制缓冲区

练习：通过setbuf，你已经可以让用户缓冲区对你可见。现在，请想一个办法，使得程序在多次调用系统库函数输出数据之间对这个缓冲区做snapshot(拍快照)，并利用这些snapshot的内容研究C的缓冲区机制。

注1： setbuf的内部调用了这个函数(*libio/setbuf.c*#L31)：

```
_IO_setbuffer (fp, buf, BUFSIZ);
```

你可以看到，_IO_setbuffer是更高级的版本，它可以指定buf的长度。然而，我们不建议使用这个接口(可移植性考虑)。

注2：setvsize则是_IO_setbuffer的类似物，且为C标准中给出的函数，可以通用。

注3： 系统缓冲区对R3程序(用户态程序)不可见。Microsoft(包括Linux内核的设计方)保留在任何时候以任何方式并不经通知修改系统内核实现的权利。**不要试图用任意方式猜测系统内核的实现并使用这些猜测设计程序。DO NOT play with anything unpredictable.**

文件I/O

与控制台I/O类似，文件I/O也有一套库函数(有一些扩充)：
打开：fopen / 重定向：freopen
缓冲区：fflush, setbuf, setvbuf(前面已经提到过)
关闭：fclose
读取：fscanf, fgetc, fgets, fread… 原始输入：read
写入：fprintf, fputc, fputs, fwrite… 原始输出：write
定位：fseek, ftell, rewind
错误处理：feof, ferror, clearerr, perror(显示错误，这次不涉及，还有strerror)
文件操作：remove, rename(请自行查阅文档，这次不涉及，注意非空目录操作是需要遍历目录的)
临时文件相关：tmpfile(不涉及), tmpnam(不涉及), Linux上有fmemopen(内存文件)

哪些不同？主要在于：文件是**有限长度**并且可以**随机访问**的。

文件I/O

什么是流？谈到文件I/O就必须强调流的观点。

控制台流I/O是最“正统”的流式I/O。特点：顺序写入、读取，一旦写入/读入就不能撤回(先别管ungetc)。你不可能知道控制台输入有多长。

文件I/O就区别于字面的“流式I/O”，文件是已经存在的实体，它的长度是确定的，它是可以寻址的(广义的寻址，意思是可以随机访问某个位置)。

文件I/O提供流式I/O的接口是为了方便修改程序和移植。

控制台I/O的内部是以stdin, stdout, stderr(一般情况下分别是0, 1, 2)作为文件id调用文件I/O的接口的。因此，文件I/O接口是更底层的接口。

文件I/O提供了很多文件读写特有的接口。这些接口可能对于控制台I/O是无效的。

文件I/O

忽略系统的底层设计(磁盘分配、对齐等等细节问题)，文件在程序看来就是一个大的“字节数组”。

fopen：打开文件，并将一个FILE*对应到这个文件。

```
FILE *fopen(const char *filename, const char *mode)
```

mode是什么？主要是下面几种：

- r：读取，文件不存在就会失败。
- w：写入，文件存在则先清空，在文件不存在时创建文件。
- a：追加，在文件不存在时创建文件，写入指针在文件末尾。
- r+：读写，文件不存在就会失败。
- w+：读写，文件存在则先清空，在文件不存在时创建文件。
- a+：读写，在文件不存在时创建文件，写入指针在文件末尾。

文件I/O

mode的可能写法归纳起来：

mode	读	写	文件存在	文件不存在	指针位置
r	○	–		失败	开头
w	–	○	清空	创建	开头
a	–	○		创建	末尾
r+	○	○		失败	开头
w+	○	○	清空	创建	开头
a+	○	○		创建	末尾

还有**特殊的b字符**，例如mode=**wb**等，表示要求访问的是一个二进制数据文件。系统不会在你读写的东西上加上任何的额外数据。(文本文件就不一样)

文件I/O

忽略系统的底层设计(磁盘分配、对齐等等细节问题)，文件在程序看来就是一个大的“字节数组”。

fclose：关闭文件。关闭后的文件不可访问，访问的行为不可预测。

```
int fclose(FILE *stream);
```

返回值：

- 0：成功关闭
- EOF：错误

文件I/O

忽略系统的底层设计(磁盘分配、对齐等等细节问题)，文件在程序看来就是一个大的“字节数组”。

freopen：重定向。将一个流连接到另一个流(信竞同学肯定很熟悉)。

```
FILE *freopen(const char *filename, const char *mode, FILE *stream);
```

功能：将stream关闭后按照`filename`和`mode`打开另一个文件(与fopen类似，参数含义都一样)，并**使用同一个stream**。

文件I/O

忽略系统的底层设计(磁盘分配、对齐等等细节问题)，文件在程序看来就是一个大的“字节数组”。

读取：fscanf, fgetc, fgets, fread…
写入：fprintf, fputc, fputs, fwrite…

这些函数与控制台I/O的函数基本上是相同的，此处不再多说。注意，读/写操作的前提是这个文件流有对应的功能，出错的情况基本为下面两种：

- 流的特性决定其不能读/写(例如没人去写入stdin)
- 流在打开时没有对应的要求(例如写入用mode=r打开的文件)
- 文件权限问题(但这种情况在打开时就会出错)

文件I/O

原始I/O(无用户缓冲区) Linux专用，在unistd.h中

```
读取: ssize_t read(int fd, void *buf, size_t count);
写入: ssize_t write(int fd, const void buf[.count], size_t count);
```

文件I/O

重点：文件中的读写指针

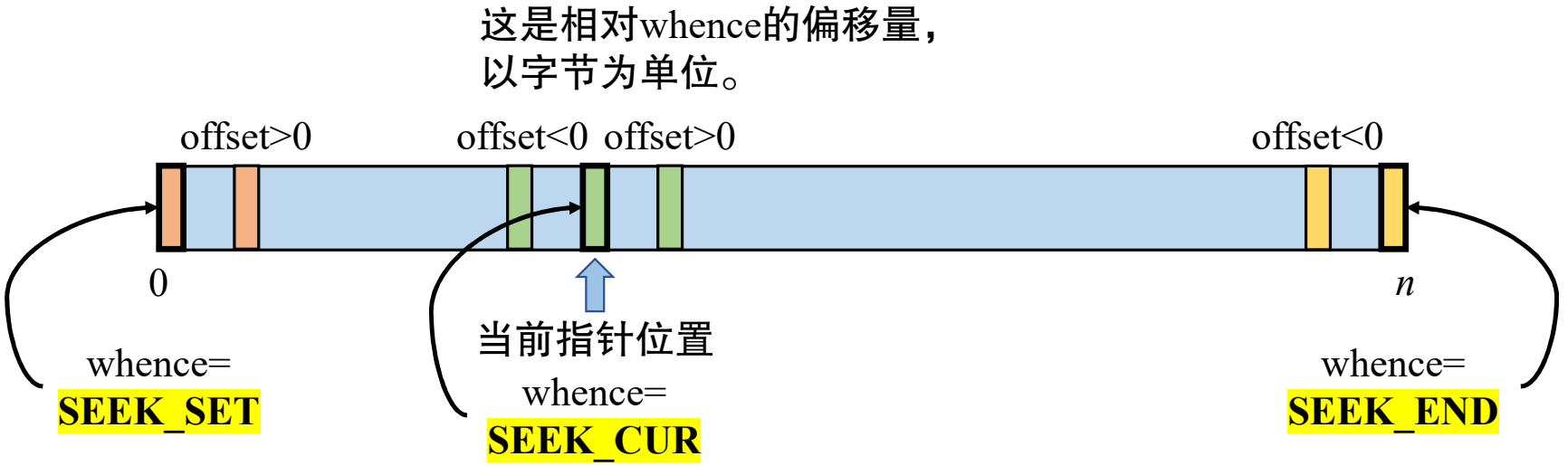
前面我们提到过，文件的长度一般是可知的，而且可以随机访问(逻辑意义上，非物理意义上)。因此，我们可以通过文件读写指针控制文件中的读写位置。

提示：在某些情况下，某些系统中可能存在读写操作不能交替进行的情况。在这种情况下，使用一个空的fseek调用(例如，从当前位置移动0个字符这种空调用)。

文件I/O

重点：文件中的读写指针

`int fseek(FILE *stream, long int offset, int whence);` 调整读写指针位置。



文件I/O

重点：文件中的读写指针

rewind的功能是将文件指针置于文件开头，类似下面这样：

```
fseek(stream, 0, SEEK_SET);
```

rewind的特性：

- 同时清除eof标志和错误标志(后面会讲到)
- 不返回是否成功

此外，可以对stdin调用rewind来清除键盘缓冲区。

文件I/O

重点：文件中的读写指针

```
long int ftell(FILE *stream)
```

相对的，ftell可以用来读取当前的文件指针位置。

注：追加模式下(mode=a)，文件在被打开后的读取位置是随C语言实现变化的，不能确定。

练习：给出一个文件流f，请确定这个文件的长度。

文件I/O

文件中的错误处理

feof：文件指针是否达到文件末尾，0为未到末尾，非0为达到末尾。

坑：For example, if a file contains 10 bytes and you read 10 bytes from the file, feof will return 0 because, even though the file pointer is at the end of the file, you haven't **attempted to read beyond the end**. Only after you try to read an 11th byte will feof return a nonzero value.

这有点类似于我们所谓的“左闭右开区间”，这种特性一定程度上也是因为C运行库内部只能在程序调用C库函数的时候运行，而不能在后台自动检查文件指针的位置。调用读取函数时，函数内部最开头处有更新eof状态的代码，所以如果是一次读取完成后才达到文件末尾，并不会设置eof。

eof标志和!(pointer<end_file)是分开的两个东西！

文件I/O

文件中的错误处理

ferror：文件是否有相关错误发生，0为无错误，非0为有错误。

errno是C中常见的指示错误的方式。一般来说，errno的行为像是全局变量。其值可以与一系列常量比较(例如EINVAL等)来确定错误类型。
在某个需要被检查errno的系统调用前清除errno（使用_set_errno(0)），调用后请立即检查errno的值。系统库函数在没有错误的时候不会清除errno中的错误代码。

clearerr：清除文件流上的错误标志。

一旦有前面的操作设置了错误标志，这个标志就不会被其他一般操作重置，直到clearerr, fseek, fsetpos, rewind被调用。

字符串(内存)的I/O

C中提供的内存I/O方式

1. Linux上提供的fmemopen
2. **sprintf/sscanf方式(跨平台)**

字符串(内存)的I/O

sprintf和sscanf

函数形式与printf/scanf很类似，区别在于其向一个字符数组(更宽泛地说是一个内存区域)进行I/O。
区别在于：

1. 内存毕竟是内存，不是流，所以没有流指针。sscanf只能从头开始读(无论是对一个区域的第几次调用)。
2. 需要特别注意可能的缓冲区溢出问题，注意限制各个格式项的宽度。

When reading a string with sscanf, always specify a width for the %s format (for example, "%32s" instead of "%s"); otherwise, improperly formatted input can easily cause a buffer overrun.

字符串(内存)的I/O

sprintf和sscanf

sprintf/sscanf的一大作用是构造一些内存数据，包括printf的格式项(虽然在有可能的替代的情况下我们不建议这样做)。下面这个练习展示了这一点：

C中有asctime函数用于将tm结构转换成下面这样的字符串：

```
Sun Feb  3 11:38:58 2002
```

编写一个程序，将tm结构和这样的字符串互相转换(值得注意的是，这种字符串中的每一个field都是定宽的)。

高级主题

不知道你是否有好奇过：

1. 某些cmd小游戏(例如贪吃蛇)中，键盘按键可以直接被读取(不需要按回车)。
2. MSVC总是报错“Unsecure C Library functions, use xxx instead”！这是什么？带_s的函数又如何安全了?
3. 64位的I/O接口是什么样子?
4. 彩色的控制台输出？更加丰富的例如彩虹进度条和光标定位。
5. Linux上的/dev/null、/dev/zero这样的文件怎么用？Windows有对应物吗?
6. 怎么让自己的程序摆脱GB2312和cp936，实现真正的国际化?
7. 出错了怎么办？errno能给我带来多少信息？有没有别的办法不用errno这种“全局变量”呢？(已经在2中回答过了，_s函数等等，perror可以用来显示错误信息)
8. 多线程的应用程序中，标准输入输出如何做到互斥？(使用加锁的方式防止多个线程同时向控制台输出，造成输出错乱)
9. 同步I/O太浪费进程的wall clock time了，异步的怎样？(Windows: I/O完成端口, iopc；Linux: aio是最原始的选择，通过io_submit等接口)
10. 到底什么能被当作文件？(请看虚拟文件系统) Socket呢？(一定程度上也是文件，但是这个抽象不够彻底)进程之间如何通讯？(Socket, IPC等)

高级主题

1. 键盘按键可以直接被读取(不需要按回车)。

实现方式:
(Windows) 使用conio.h中的getch()和getche()实现，kbhit也是可以的(直接读按键)。
(Linux) 使用gnu-conio项目(请看github)或者通过termio

```
struct termios tm, tm_old;
int fd = 0, ch; // fd = 0为stdin
if (tcgetattr(fd, &tm) < 0) { // 保存现在的设置
    return -1;
}
tm_old = tm;
cfmakeraw(&tm); // 设置为原始模式
if (tcsetattr(fd, TCSANOW, &tm) < 0) {
    return -1;
}
ch = getchar(); // 读取字符
if (tcsetattr(fd, TCSANOW, &tm_old) < 0) { // 恢复term
    return -1;
}
```

高级主题

2. 不安全的I/O？怎样的才是安全的？

(Windows特定)：

- **printf_s**：printf_s会检查格式字符串是否包含正确的格式项。
- **scanf_s**：在需要读入字符串的时候，目标缓冲区地址后面要加上它的可用大小(用来防止缓冲区溢出，注意是可用的字符数，不含末尾的\0字符)。例如：

```
char s[10];
scanf_s("%9s", s, (unsigned)_countof(s));
```

  注意：表示大小的参数的类型是unsigned，不是size_t。
- 类似的，操作缓冲区的函数(例如strcpy)多数都有安全版本(strncpy或者strcpy_s)。
- 其他带有_s的函数通常是将原来的结果换成了一个指针参数，而返回错误代码(不用errno)：

```
FILE *fopen(
   const char *filename,
   const char *mode
);
```

```
errno_t fopen_s(
   FILE** pFile,
   const char *filename,
   const char *mode
);
```

高级主题

3. 64位的I/O接口是什么样子?

有些函数涉及寻址很大的对象(例如NTFS中的大文件)会提供64位的版本，这样就可以方便地操作大于2GB或者4GB的文件：

```
int _fseeki64(
   FILE *stream,
   __int64 offset,
   int origin
);
```

```
__int64 _ftelli64(
   FILE *stream
);
```

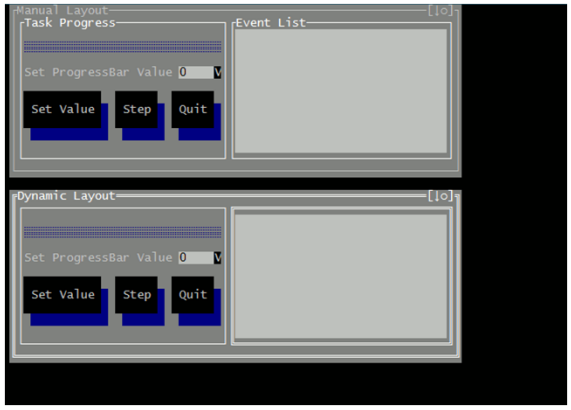
然而，这些接口并不是标准定义的。所以，如果你需要跨平台，且不需要处理大文件，就使用C标准库函数。如果需要跨平台，且需要处理大文件，则使用OS提供的文件API是更可靠的选择。

高级主题

4. 彩色的控制台输出？更加丰富的例如彩虹进度条和光标定位。

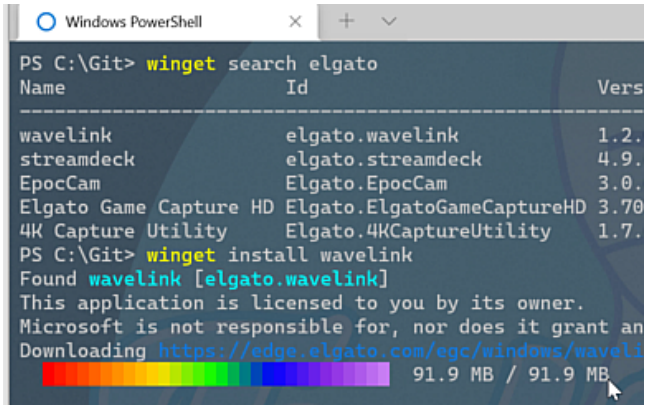
这个问题涉及到使用的终端类型。

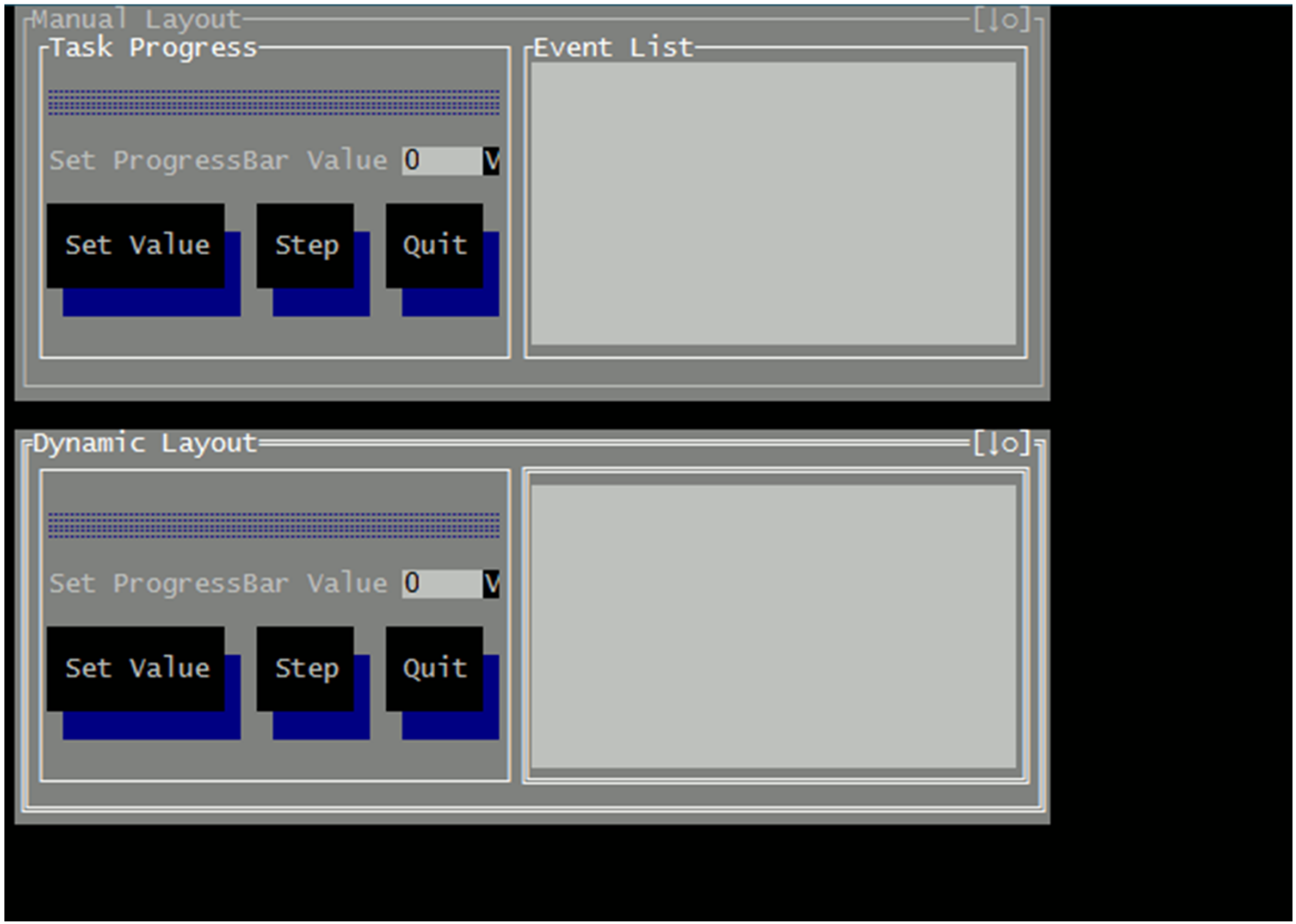
- 在传统的Windows终端里面(cmd.exe, conhost.exe)，只有16种颜色可选，可以通过**SetConsoleTextAttribute**和**SetConsoleCursorPosition**这类函数来解决。
- 在Linux系OS上的终端以及Windows的终端模拟器(例如mintty)中，可以使用**Escape Code**来解决颜色输出和移动光标的问题。参考：https://www.man7.org/linux/man-pages/man4/console_codes.4.html
- 彩色控制台可能还取决于硬件实现和终端种类。例如，在串行端口上的终端(例如COM1上)就一般不考虑颜色的问题。



左图：用字符画出来的UI，常见于DOS程序。用过Free Pascal的同学一定很熟悉。

右图：这个就是彩虹进度条





用字符画出来的UI，常见于DOS程序(~~例如我曾经在某个阶段用过的Visual Basic for DOS，~~笑)。用过Free Pascal的同学一定很熟悉。

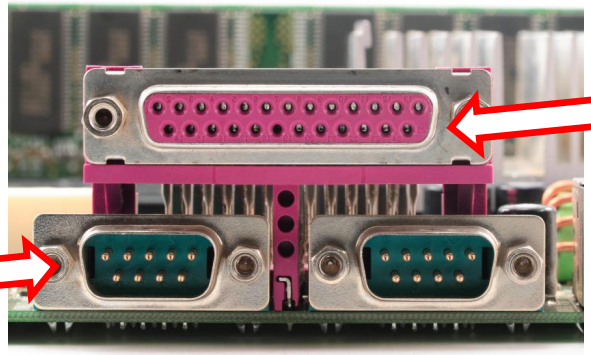
高级主题

5. Linux上的/dev/null、/dev/zero这样的文件怎么用？Windows有对应物吗？

/dev/null等等“文件”是Linux虚拟文件系统的一部分。/proc/…也是类似的。这是Linux所谓“一切皆为文件”的体现。
Windows有类似的对应物。下面是一个列表。因此，这些文件名称是被保留的，不要使用。

CON(控制台)、AUX、COM1(串行端口)、COM2、COM3、COM4、LPT1(并行端口)、LPT2、LPT3、PRN(打印机)、NUL(空文件，类似/dev/null)

COM
串行端口



LPT
并行端口

高级主题

5. Linux上的/dev/null、/dev/zero这样的文件怎么用？Windows有对应物吗？

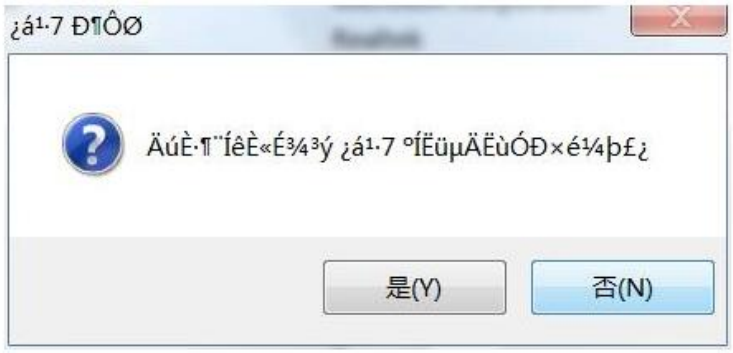
freopen之后如何“还回去”？巧妙地使用CON文件即可：

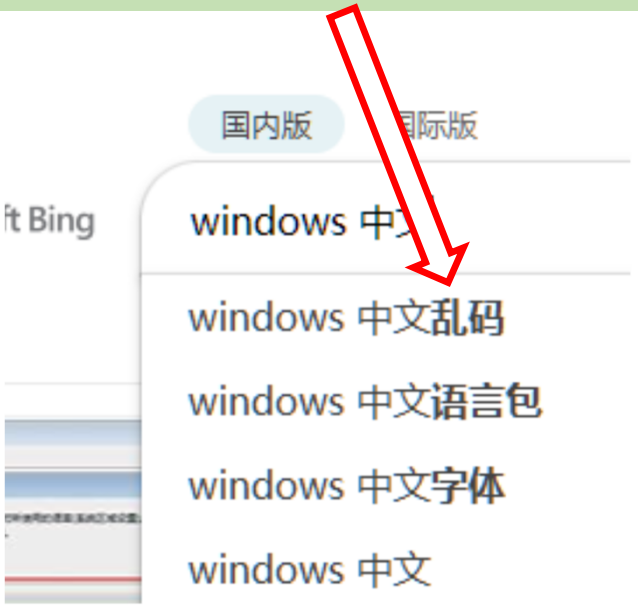
```
freopen("CON", "r", stdin);
freopen("CON", "w", stdout);
```

高级主题

6. 怎么让自己的程序摆脱GB2312和cp936，实现真正的国际化？

GB2312和cp936，以及宽窄字符集的转换是Windows程序员曾经的噩梦。





高级主题

6. 怎么让自己的程序摆脱GB2312和cp936，实现(真正的)国际化？

方法1：统一使用utf-16和宽字符集；
方法2：统一使用utf-8和cp65001。
对应的文件要做一些设置，例如写cgi程序，就要在meta标签里面加上encoding=utf-8。

Linux用户：基本上都是支持utf-8的。

icu(International Components for Unicode)是支持Unicode的有力工具。icu4c是在C/C++中使用icu库的方式。

https://github.com/unicode-org/icu/blob/main/icu4c/readme.html
https://unicode-org.github.io/icu-docs/apidoc/released/icu4c/

值得一看的几句话

Programs must be written for people to read, and only incidentally for machines to execute. (*Abelson / Sussman*)
程序必须写得能供人阅读，机器执行只是附带。

It’s hard enough to find an error in your code when you’re looking for it; it’s even harder when you’ve assumed your code is error-free. (*Steve McConnel*)
你的目的是找bug的时候，bug已经很难找了；更不要说当你认为自己的代码中没有错误的时候。

值得一看的几句话

从 Windows 10 版本 2004（内部版本 19041）开始，printf 系列函数根据 IEEE 754 的舍入规则输出可精确表示的浮点数。 在早期的 Windows 版本中，以“5”结尾并且可精确表示的浮点数总是向上取整。 IEEE 754 规定它们必须舍入到最接近的偶数（也称为“**四舍六入五成双**”）

分析化学：？？？

最后：谢谢大家~

NEVER `/w0`, or it may fail without warning.