士疑解惑串讲

> Date: 2023-11-19
> Author: kai_Ker

现在是私货时间!

今天讲的主要课件在线阅读：士疑解惑程设串讲（2023-11-19）（空格翻页）

今天讲的后面一个课件：第一次程设串讲增补内容-2022（空格翻页）

Dev 调试教程：Dev-C++ 安装、配置与调试功能

Scanset 集合匹配：scanf() 中的跳过与字符匹配

调试技巧与方法

白盒测试：观察代码细节，找出程序 Bug

这种调试方式也称为 “走查”，就是用眼睛去看代码，当然了，用眼睛看，肯定也是有技巧的。

首先不能走马观花，一定要**边看边思考**，最好可以假想一个人出来，你给他一行一行解释，按照代码的执行顺序，解释每一行代码的作用。

不仅如此，你还应该多问问自己，这里这个数字写对了吗？数组大小够不够？这里的判断是这个逻辑吗？循环条件有没有遗漏？等等。

> **[CODE]** `sqrt-bi-bug.c` Bug 代码示例
> **[CODE]** `sqrt-bi-bug-1.c` 输入输出的 Bug 查出来了
> **[CODE]** `sqrt-bi-bug-2.c` 修复后的代码

黑盒测试：构造测试数据，关注边界情况

这种方式适用于样例过了，但是交上去却没有 AC 的情况，并且你暂时找不出代码中的其他问题。

也许你需要再仔细读一遍题，再仔细看一看题目的数据范围，尝试构造一些特殊用例。

比如 $n=0$，$n=1$；比如题目说元素可重复，就构造几个重复两次、重复三次的例子。

很多时候，对于题目不要太想当然，对于题目没有说明的关系，就要考虑一下有没有存在的可能性，但是也不要去钻牛角尖。

> 给定一个长度为 n 的正整数数组，以及一个 d，让你求数组中**连续不超过** d 个元素之和的最大值。
> 数据范围：$1 \leqslant n, d \leqslant 1000, 0 \leqslant a_i \leqslant 100$

重点还要关注一下数据范围，有时候对范围的估算可能不太准。

> 给定两个数组，将它们中的元素**两两相乘**，求总和。
> 数据范围：$1 \leqslant n, m \leqslant 10^6, 0 \leqslant a_i, b_i \leqslant 100$

单步调试：把握细节，人机共算

单步调试，利用现有工具，比如 Dev-C++ 中的调试功能，让代码一步一步运行，观察变量的实时变化。

整个过程需要实时跟进演算，判断每一步的执行是否正确。

关于单步调试的方式，这里不再赘述，相信大家一定能够通过各种渠道学习、熟悉。

打印法调试：插入“探针”，阶段推进

有的时候，尤其是当代码比较复杂的时候，特别是循环较多时，单步调试可能会比较消耗时间，也不太容易准确把握每一步的结果。

在这种时候，可以尝试一下打印法调试，在关键位置，将**关键变量**（尤其是循环控制变量）的信息打印出来，而隐藏了其他细节。

无论是循环还是函数调用，还是普通的顺序执行，这实际上是人为将代码的执行过程划分阶段，然后判断每一阶段执行的正确与否。

> [CODE] `sqrt-bi-bug-3.c` 添加了一些打印语句
> [CODE] `sqrt-bi-bug-4.c` 输入输出的 Bug 查出来了
> [CODE] `sqrt-bi-bug-5.c` 死循环，看不清变量变化过程，插入暂停语句

空白符与空白符的读入

空白符有哪些?

不同于我们通常所认识的那样，这里所指的 **空白符** 并不单指 “空格”，而是包含三大类：

- 制表符 `\t`
- 空白符 `\n` （换行） `\v` （垂直制表符） `\f` （分页符） `\r` （回车）
- 空格 `' '`

> [CODE] `backslash-v.c` 探究：垂直制表符是什么?

空白符可以使用函数 `isspace` 进行判断，若 `isspace(ch)` 的值为 **非 0**，则字符 `ch` 为空白符。

scanf 对空白字符的处理

也许我们都知道，读入一个整数的时候，会自动**忽略**前面的空白字符，因此，当题目中只出现了整数的读入时，不管它们分在几行之中，我们都不需要去管这些乱七八糟的空格或者换行。

注意，这里的 “忽略”，指的是**消耗并舍弃**所有前导空白字符。

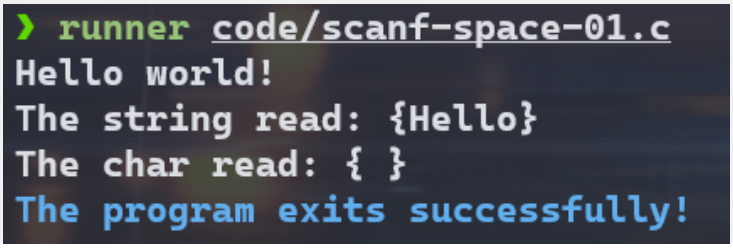
事实上，除了 `%d` ，**绝大部分**的 “转换指示符” 都会这样做，也即消耗并舍弃所有前导空白字符，直到遇到需要的字符为止。

%s 作转换指示符的情形

当然， `%s` 也符合上面所说的特性，当使用 `%s` 作为转换指示符时， `scanf` 会丢弃所有的空白字符，直到第一个非空白字符，开始读取，一直读到**下一个空白字符**为止。

> 注意，这个紧随其后的空白字符并不会被读取，也**没有被丢弃**！
> **[CODE]** `scanf-space-01.c` 演示：会保留紧随其后的空白字符

```
scanf("%s", str);
scanf("%c", &ch);
```



%c 作转换指示符的情形

`%c` 用于匹配一个字符，且不会舍弃空白字符，因此，在使用 `%c` 读取字符时，应该仔细考虑一下可能存在的空白字符的干扰。

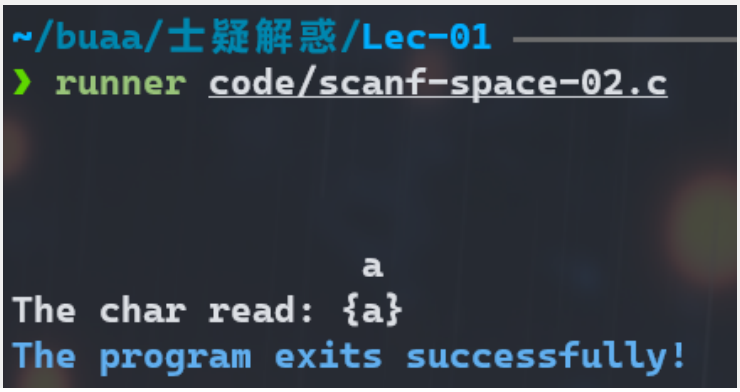
格式字符串中含有空白字符串的情形

大家也许会学到，可以这样做来过滤字符前的空格：`" %c"`，通过在 `%c` 前增加一个空格，可以确保 `%c` 不会读取到换行。

那么，这个格式字符串中的空格，是只能过滤一个空格吗？我们可以来试一试！

> **[CODE]** `scanf-space-02.c` 演示：空格的过滤能力

```
scanf(" %c", &ch);
```



通过演示可以看出，这个空格不仅可以吸收并消耗空格，还可以过滤换行、制表符，而且没有数量限制！

任何格式字符串中的**单个空白符**会处理**所有**来自输入的可用**连续空白符**，因此，格式字符串中连续的换行符就没有意义了。

格式字符串末尾的空白符所带来的问题

有时候，同学们可能会想，在格式字符串的末尾增加一个空格或者换行来过滤行末的换行，比如：

```
scanf("%s\n", str);
```

但有的时候会带来一些问题：

[CODE] `scanf-space-03.c` 演示：格式字符串末尾的空格

```
scanf("%s\n", str);
```

注意到，后面按了很多回车，也都没有停止读入，因为这个 `\n` 会处理**连续**的空白字符，要**等到**一个非空白字符才会停止读入，当然，强制终止读入也会让它停下来。

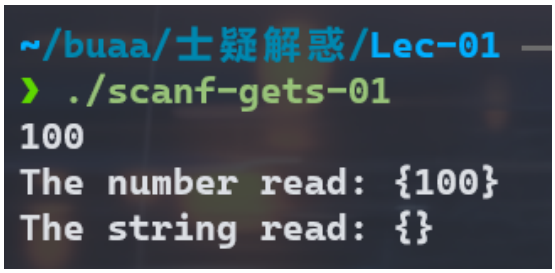
使用 gets 读取包含空白符的字符串

`gets` 会读取当前缓冲区内的字符，**直到遇到换行符**或者文件尾，`gets` 会读取并丢弃换行符，同时会在读取到的字符串尾部增加一个空字符 `\0`。

下面的代码出了什么问题？

```
// scanf-gets-01.c
#include <stdio.h>

int main() {
    int num;
    char str[1024];
    scanf("%d", &num);
    gets(str);
    printf("The number read: {%d}\n", num);
    printf("The string read: {%s}\n", str);
    return 0;
}
```



在读入时，还没有来得及输字符串，程序就结束了，`scanf` 读取了一个整数，并**遗留了末尾的一个换行**，这个换行被 `gets` 读取了，因此带来了错误。

如果使用 `%d\n` 来处理换行，可能会干扰下一行的字符串（万一下一行的字符串由空格开头呢），因此建议的做法是使用 `getchar` **函数**来吸收换行符。

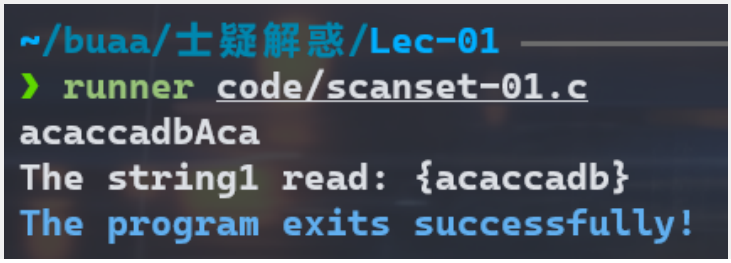
补充：集合匹配

在格式字符串中，使用 `%[集合]` 可以进行集合匹配，会匹配一个来自 `集合` 的字符的**非空字符序列**。

例如，`%[abcd]` 会匹配**连续的** `abcd` 中的任何一个字符，直到遇到一个集合外的字符为止。

[CODE] `scanset-01.c` 演示：集合匹配的基本用法

```
scanf("%[abcd]", str);
```



如果集合由 `^` 开头，则表示 “排除”，则匹配所有不在集合中的字符（不包括这个用来表示 “排除” 的 `^`）。

但是很多时候，我们需要匹配的东西有点多，如果全部写出来可能有点复杂。我们常用的编译器都支持 “范围匹配”，即使用 `-` 来表示范围。

例如，`%[0-9]` 等价于 `%[0123456789]`，而 `%[a-zA-Z]` 可以匹配全部的英文字母。

[CODE] `scanset-02.c` 演示：排除匹配与范围匹配

```
scanf("%[0-9]", str1);
scanf("%[^0-9]", str2);
```

```
~/buaa/士疑解惑/Lec-01
❯ runner code/scanset-02.c
00100200abcde110
The string1 read: {00100200}
The string2 read: {abcde}
The program exits successfully!
```

补充阅读：scanset 与指定字符匹配（欢迎来参观我的博客）

总结一下

- 对于 `scanf` 的转换指示符中，只有 `%c`，`%[集合]`，`%n`（不常用）**不会自动忽略**空白字符。
- `scanf` 的格式字符串中的**单个空白字符**就会将**连续的任意空白字符**吸收掉。
- 请谨慎地将 `scanf` 与 `gets` 混用，最好不要这样做！可以使用 `getchar` 吸收换行。
- 集合匹配的用法

sprintf 与 sscanf

这两个函数同样在 `<stdio.h>` 中被声明，并具有与 `printf` 和 `scanf` 相似的用法，不同之处在于，`sprintf` 和 `sscanf` 是以字符串为载体进行读取、写入的。

它们的函数原型为：

```
int sprintf( char *buffer, const char *format, ... );
int sscanf( const char *buffer, const char *format, ... );
```

第一个参数为待处理字符串，第二个参数为格式字符串，后续为接收参数。

`sprintf` 将内容**写入字符串** `buffer`，通过格式字符串 `format` 对后续参数进行格式化拼接。

例如：

```
char str[1024];
sprintf(str, "%d;:%06d", 10, 13);
```

得到的 `str` 的结果为 `10;:000013`。

`sscanf` 会按照格式字符串，**从字符串中读取**内容，并将内容写入变量。

例如：

```
int n;
sscanf("1234", "%d", &n);
```

可以很灵活地将字符串转成整数，得到的 `n` 为 `1234`。

上述只是一些简单的实例，它们也可以有很多更加灵活的应用。

多关键词排序与 qsort

简单排序相信大家已经很熟悉了，无论是冒泡排序还是直接使用快排函数，都可以取得不错的效果。但我今天想介绍一下的是，多关键词排序。

首先考虑一下数据的存储方式，针对每一种信息，使用一个数组进行维护，利用索引进行关联。

比如，针对学生的上机成绩与罚时进行存储排序，如果还要存储姓名的话，我们可以开三个数组。

```
#define MAXN 1001
int score[MAXN];
long long time[MAXN];
char name[MAXN][10];
```

对冒泡排序进行改造

常规的冒泡排序相信大家已经很熟悉了，如果想进行多关键字排序的话，也可以仿照冒泡排序的思路。

```
for (int i = 0; i < n - 1; ++i) {
    for (int j = 0; j < n - 1 - i; ++j) {
        if (需要交换(j, j + 1)) {
            交换元素(j, j + 1)
        }
    }
}
```

这里有两点需要关注：大小判断与元素交换。

这里我们使用两个关键词：成绩为主要排序依据，降序排列，若成绩相同，则按照罚时升序排列。若成绩与罚时均相同，则**保留输入顺序**。

因此，若同学 j 和同学 $j+1$ 满足：

```
(score[j] < score[j + 1]) ||
(score[j] == score[j + 1] && time[j] > time[j + 1])
```

则，他们的位次**需要交换**。

交换的时候，有两点需要注意：**全部信息**都需要交换，即使没有参与比较；字符串不可以使用通常的方式交换。

譬如，要交换同学 j 和同学 $j+1$ 的位次，可以这么写：

```
int t_score = score[j];
score[j] = score[j + 1];
score[j + 1] = t_score;

long long t_time = time[j];
time[j] = time[j + 1];
time[j + 1] = t_time;

char t_name[10];
strcpy(t_name, name[j]);
strcpy(name[j], name[j + 1]);
strcpy(name[j + 1], t_name);
```

于是我们便完成了这样的一个排序。

对索引进行排序

上述的排序看上去没啥问题，就是繁琐了一点，尤其是交换元素的部分，不仅麻烦，还影响性能。

于是，我们就在想，是不是有一种方式，可以代替这些真实的数据。那就给每位同学编号吧！

这样，我们需要存储的信息就又变多了，需要额外记录一下编号。

```
/* 初始化：index[i] = i */
int index[MAXN];
```

在排序的时候，我们只需要交换学生们的编号就可以了。

> **[DEMO]** `multi-sort.drawio.png` 演示交换编号

```
for (int i = 0; i < n - 1; ++i) {
    for (int j = 0; j < n - 1 - i; ++j) {
        int stu_j = index[j], stu_j1 = index[j + 1];
        if (
            (score[stu_j] < score[stu_j1]) ||
            (score[stu_j] == score[stu_j1] && time[stu_j] > time[stu_j1])
        ) {
            index[j] = stu_j1;
            index[j + 1] = stu_j;
        }
    }
}
```

使用快排函数

首先，`qsort` 函数似乎不要求大家掌握，但学会了一定可以十分方便。

`qsort` 的函数原型为：

```
void qsort( void *ptr, size_t count, size_t size,
            int (*comp)(const void *, const void *) );
```

一共四个参数，第一个为**指向待排序的数组的指针**，第二个为**元素个数**，第三个为**数组每个元素的字节大小**，第四个为**比较函数**。

比较函数的签名为：

```
int cmp(const void *a, const void *b);
```

如果**首个参数小于第二个，那么返回负整数值**，如果首个参数大于第二个，那么返回正整数值，如果两个参数等价，那么返回零。

函数签名强制要求了比较函数**不修改**数组内元素的值，并且比较的结果与元素在数组中的位置无关。

使用比较函数可能存在的问题：

```
int cmp(const void* a, const void* b) {
    int arg1 = *(const int*)a;
    int arg2 = *(const int*)b;

    return arg1 - arg2;  // 通常正确，但可能会溢出！
}
```

利用 `qsort` 对编号进行排序，可以仿照类似于冒泡排序中的思路，重点关注比较函数的写法。

```
int cmp(const void* a, const void* b) {
    int index1 = index[*(const int*)a];
    int index2 = index[*(const int*)b];

    if (score[index1] == score[index2]) {
        return time[index2] - time[index1];  // 错啦！减反啦！同时，防止溢出
    }

    return score[index1] - score[index2];  // 错啦！减反啦！同时，防止溢出
}
```

但是，问题又来了，`qsort` 并不保证排序后数据的稳定性！那么这种情况应该怎样处理呢？其实也很简单，把初始的位次也当成排序的条件就可以了！

初始位次不是编号，还需要再开一个数组！

```
/* 初始化：order[i] = i */
int order[MAXN];
```

那么现在就是三关键词排序了：

```
int cmp(const void* a, const void* b) {
    int index1 = index[*(const int*)a];
    int index2 = index[*(const int*)b];

    if (score[index1] == score[index2]) {
        if (time[index1] == time[index2]) {
            return order[index1] - order[index2];  // 升序排列，这个对了
        }
        return time[index2] - time[index1];  // 错啦！减反啦！同时，防止溢出
    }
    return score[index1] - score[index2];  // 错啦！减反啦！同时，防止溢出
}
```

还有一种复合数据类型，叫做 **`struct`** ，有余力的同学可以学习一下，使用起来会比较方便。

需要特别强调一点，如果不使用 `typedef` 的话，需要在类型前加上 `struct` 关键字！

二分查找

二分查找的思想很好理解，但是在实践时可能会遇到很多细节上的问题，今天我就来和大家分享一下二分查找的书写策略。

利用区间相交关系确保二分查找的不重不漏

首先确定查找区间，什么意思呢，就是当 `left` 与 `right` 给定时，哪些元素是我们需要考察的。比如就取 `[left, right)` 这一个区间。

那么，**初始情况** 便是：

```
left = 0, right = size;
```

因为右边界我们不要，而数组是从 0 开始的。当然你可以从 1 开始使用数组下标，相应调整即可。

查找的**循环条件**是：

```
while (left < right) {
    // ... ...
}
```

为什么不取等？因为 `left == right` 时，区间 `[left, right)` 已经没有元素了，已经查完了！

判断下标为 `mid` 的元素后，**`left` 和 `right` 如何调整**？

```
// 假定数组递增
if (arr[mid] > target) {
    // arr[mid] 大了，mid 及右边的都不要
    right = mid;
    // 为什么不要加 1？因为 [left, right) 本身就取不到 right
} else if (arr[mid] < target) {
    // 为什么这里加 1？因为 mid 不合要求，不能要
    left = mid + 1;
} else {
    return mid;
}
```

因此，二分查找的关键在于选定一个查找区间，要让初始值恰好覆盖整个数组、循环条件为查找区间内还有元素、边界下标转移要能够恰好将不满足的值排除。

查找左右边界

如果要查找一个值第一次出现的位置，可以这样操作。

还拿上面的区间为例，当 `arr[mid] == target` 时，将 `right` 设为 `mid`，继续循环，直到 `left == right` 时退出循环。

找到一个值时，将 `right` 移过来，本质上是将其排除在外，但并不会丢失这个值，因为如果这个 `right` 恰好就是要找的，那么 `right` 在接下来的循环中便不会移动！

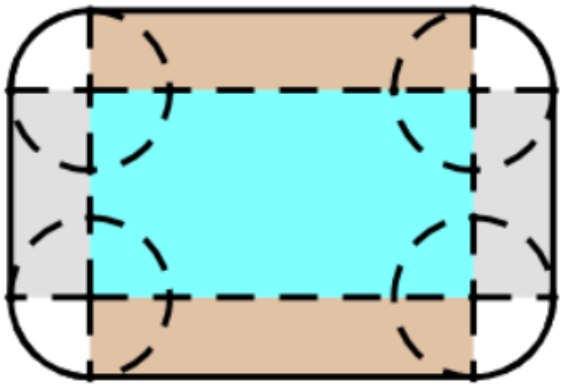
计算几何

计算几何是这样的一类题：需要你编程进行一些几何题的计算。

比如：判断三角形的类型，判断四边形类型，计算三角形面积、周长等等。

草稿纸的重要作用

假设一个题目：让你计算如下图圆角矩形的面积，告诉你四个圆的半径，以及两组直边的长度，你会怎么算？大家可以想一想



数学库的使用

```
#include <math.h>
```

三角函数、反三角函数。

算术平方根、立方根（`cbrt`）。

π 如何获取？`acos(-1.0)` 精度足够高。

绝对值 `fabs` 类型适用于 `double`！