DS bug 全扫除计划

> 🚩 大家好这是一篇总结数据结构编程技巧的文档，建议大家多读一读，如果做作业的时候遇到 bug 了也可以看一看，或许会给大家带来启发。
> 也欢迎大家在群聊中对其中的内容进行讨论和补充，提出一些有价值的问题。那么就祝大家学习进步，成绩 ++，bug--。
> 感谢每一位对此项目进行过帮助的同学（可能会有小礼物～）

一、一般问题

1.1 同样的数据本地是对的，评测却错误？

A：一般是由于 Windows 机器和 Linux 机器对 C 某些语言的行为处理不同，可能的原因：

1.1.1 局部变量的初始化问题

~~局部变量 Windows 机器上帮你初始化好了，但是 Linux 不一定会；~~ 非也，这并不是一个普遍现象，Linux 上运行程序未初始化的局部变量也可能为 0：

C/C++

```c
#include <stdio.h>
int main()
{
    int a, b, c;
    printf("%d %d %d\n", a, b, c);
    return 0;
}
```

这段代码在我的 Windows 系统下的输出结果为 `0 0 32759`，而在 Linux 下输出结果为 `32765 0 0`。根据我的一些不严谨测试，可能 Windows 下较早出现的两个变量未初始化值更可能为 0，而 Linux 则不是，这导致了较多的错误（原理可能与 Linux 和 Window 的栈增长方向不同有关）。实际局部变量的原理可参考 C 语言未初始化的局部变量必懂

但是否初始化也为一个重要的可能错因。

1.1.2 字符串结尾 `\0` 问题

跟上一条类似，字符串初始化没赋 0，导致后续操作字符串找不到结尾了：

C/C++

```
char str1[10005];
char str2[10005];
int i = 0;
scanf("%s", str1);
int len = strlen(str1);
for (i = 0; i < len; i++) {
    str2[i] = str1[i];
}
```

注意这样处理后 `str2` 不一定以 `'\0'` 结尾，可能会出错，而这种错误在本地可能不会被发现。

1.1.3 同一段字符串上的复制

C/C++

```
strcpy(str, str + i);
```

这个错误我本人犯过，Win 上死活发现不了，后来租一台 Linux 服务器才发现的。

1.1.4 `\r` 问题

在 Windows 环境下， 操作系统会把文本文件的 `\r\n` 自动处理为 `\n`，写文本文件时也会把 `\n` 变为 `\r\n` 再进行写入。而在 Linux 下，默认的换行只有 `\n`，Win 下文件的 `\r\n` 在 Linux 下不会被自动处理，所以在平台（OJ，Judge 等）上你的程序就可能会读到 `\r`。

一种简单的检测文件输入下你的处理是否正确的方式为：将文件打开模式的 `"r"` 改为 `"rb"`，因为 Win 仅会对文本文件进行如上处理，以二进制模式打开就能读到文件中的 `\r` 了。

1.1.5 `strcmp`、`isdigit` 等函数的返回值

详见本文 2.2

1.1.6 函数参数写自增语句

C/C++

```
scanf("%d%d", &stack[++top].id, &stack[top].val)
```

乍一看好像没什么问题，对……吗？

（怎么看都有问题，一眼 ub）

> 📌 你以为的 top 先自增，再给栈里的成员赋值一点问题没有，电脑这样跑也没问题。但是评测机可能不是这么认为的。因为不同的编译器，**有的参数是从左往右读，有的是从右往左读**，当从右往左读的时候就会发生先赋值 `stack[top].val` 再 `++top` 的情况。

一定不要这么写了。

1.2 不定行读入的问题

C/C++

```
while (scanf("%d", &a) != EOF) {
    // do something
}

while (gets(s) != NULL) {
    // do something
}
```

> 🔔 记住 `scanf` 读不到返回 `EOF`（值为 -1，类型为 `int`）
>
> `gets` 读不到返回 `NULL`（值为 0，类型为 `void *`）
>
> 可参考 Red 的 repo 中 ref 内容
>
> 更好的方法：**根据返回值的类型判断**

1.3 `scanf` 和 `gets` 混用的问题（严重）

这个问题不熟悉的话会觉得特别玄学，**一定要引起重视**！！！

C/C++

```
int a;
char s[105];
scanf("%d", &a);
gets(s);            // s啥也没读进去，被置为空"\0"
```

👋 原理解析：

scanf 读字符串的原理是，遇见空白符（空格或换行）会停止识别，并将读头指针指在空白符前，而 gets 遇见换行会直接认为识别到字符串尾。所以例子中输入一个数字回车后，scanf 读完会扔下一个换行在后面，因此后面的 gets 会以为自己任务完成了，直接给 s 读了一个空串进去。

详细原理可参考：scanf 用法及 scanf 中有 \n 的问题

⏰ 注意：避免 `scanf` 和 `gets` 混用，尽量统一形式。千万不要让 `gets` 出现在 `scanf` 的后面。实在实在迫不得已，要多写一个 `gets` 。例如：

C/C++

```
int a;
char s[105];
scanf("%d", &a);
gets(s);            // 为了消耗那个多出来的换行
gets(s);            // s可以正常读了
```

二、字符串专题

2.1 字符串以 `'\0'` 结尾的问题

在对字符数组赋值创建字符串的时候注意，例如：

C/C++

```
char str1[105];
str1[0] = 'A';
puts(str1);          // 错误，大概率打印出来乱码

char str2[105];
str2[0] = 'A';
str2[1] = '\0';
puts(str2);          // 输出：A
```

同时，在题目交代字符串长度的情况下，**千万不要**只开大小恰好为最长长度的字符数组，这个错误不常出现，但是容易被忽视。在读取时，末尾添加的 `'\0'` 可能会越界到其他变量的“地盘”，甚至在后续的操作中被覆盖，导致字符串无法正常结束。

C/C++

```
struct info_t {
    char phone_number[11];       // 众所周知，电话号码 11 位
    char name[25];
} foo;

scanf("%s", foo.phone_number);   // 读取电话号码，'\0' 被挤到后面去了
scanf("%s", foo.name);           // 读取姓名的时候，覆盖了电话号码的 '\0'
printf("%s's phone is: %s\n", foo.name, foo.phone_number);
    // 比如会这样输出： Mike's phone is: 12345678901Mike
```

> 简单情况下，结构体内各成员变量，会**按照顺序，一个挨着一个**排在内存中，上例中，两个成员变量占据了连续的 36 个字节，`phone_number` 的 `'\0'` 会越出 11 字节的边界，到 `name[0]` 的位置上，又在读取 `name` 时，被覆盖。但很多情况，`struct` 的内存分布没有这么简单，有余力、想了解的同学可以参阅这篇文章： C/C++ struct 字节对齐那些事儿，总之，用 `sizeof` 就好啦。

2.2 关于 `strcmp` 的返回值

不要想当然地认为 strcmp 的返回值就是 -1 、1 、0，它的结果应该是负数、正数、0。

C/C++

```
if (strcmp(s1, s2) == -1) // 错误的
    swap(s1, s2);

if (strcmp(s1, s2) < 0)   // 正确的
    swap(s1, s2);
```

更致命的是，有的 Windows 机器上本地调试还就是返回正负一，上评测就错了，因此一定要注意这一点。

> 同理 `ctype.h` 中的 `isalpha`、`isdigit`、`isupper`、`islower` 等的返回值也是 0 或非零数，不要想当然认为就是 0 或 1 了。

C/C++

```
if (isdigit(s[i])) {       // 推荐这样写，比较符合语言表达的习惯
    ......                 // 当然 isdigit(s[i]) != 0 也不会出错
}
```

三、链表专题

3.1 Q：链表插入函数，它真的插了吗？

C/C++

```c
#include <stdlib.h>

typedef struct Node{
    int val;
    struct Node * next;
} *List;

void insert(List head, int val) {
    List p = (List) malloc(sizeof(struct Node));
    p->val = val;
    p->next = NULL;
    if (head == NULL) {
        head = p;
    } else {
        List r = head;
        while (r->next != NULL) r = r->next;
        r->next = p;
    }
}

int main() {
    List head = NULL;
    insert(head, 114514);
    return 0;
}
```

例如这段代码是错误的，我们知道函数内部参数改变不影响外面的值，因此 `head` 为空时，`insert` 的执行不会影响外面 `head` 的值，插入后链表仍是空的。解决方法：

1. 使用全局变量

C/C++

```c
#include <stdlib.h>

typedef struct Node{
    int val;
    struct Node * next;
} *List;

List head = NULL;        // 全局变量

void insert(int val) {
    List p = (List) malloc(sizeof(struct Node));
    p->val = val;
    p->next = NULL;
    if (head == NULL) {
        head = p;        // 这个函数里直接改head，不需要传参
    } else {
        List r = head;
        while (r->next != NULL) r = r->next;
        r->next = p;
    }
}

int main() {
    insert(114514);
    return 0;
}
```

2. 给 `insert` 加上返回值

C/C++

```c
#include <stdlib.h>

typedef struct Node{
    int val;
    struct Node * next;
} *List;

List insert(List head, int val) {
    List p = (List) malloc(sizeof(struct Node));
    p->val = val;
    p->next = NULL;
    if (head == NULL) {
        head = p;
    } else {
        List r = head;
        while (r->next != NULL) r = r->next;
        r->next = p;
    }
    return head;                    // 把新的head返回出去
}

int main() {
    List head = NULL;
    head = insert(head, 114514);   // 外面的head在这里更新
    return 0;
}
```

3.2 链表为空时的特殊判断

链表插入、删除时一定要判断链表空了怎么办，尤其是删除，很容易忽略。还有**循环链表**到最后只剩一个结点，应该怎么删？

3.3 `malloc` 是什么，有什么用

`malloc` 函数的作用是在内存中开辟一块区域，并返回这个区域首地址。因此只有当你需要**创建新的链表结点**时才需要 `malloc`，例如：

C/C++

```c
typedef struct _Node {
    int val;
    struct ListNode* next;
} Node, *pNode;

/*
 * 从数组创建一个链表
 * @param a 数组
 * @param n 数组中元素的个数
 * @return  链表表头
 */
pNode create_list(int a[], int n) {
    pNode head;
    pNode p, r;
    for (int i = 0; i < n; i++) {
        p = (pNode) malloc(sizeof(Node));
        // for 循环每轮创建一个新的结点，因此每一轮只需要 malloc 一次
        p->val = a[i];
        p->next = NULL;
        if (head == NULL)
            head = p;
        else
            r->next = p;
        r = p;
    }
    return head;
}
```

3.4 `malloc` 的常见错误

假设链表结构体定义如下：

C/C++

```
typedef struct _Node {
    int val;
    struct Node *next;
} Node, *pNode;
```

C/C++

```
pNode l1 = (pNode) malloc(sizeof(_Node));        // 结构体类型错误
pNode l2 = (pNode) malloc(sizeof(pNode));         // 指针大小跟结构体不是一个意思
pNode l3 = malloc(sizeof(struct _Node));        // 没有类型转换
pNode l4 = (pNode) malloc(sizeof(struct _Node); // 正确
pNode l5 = (pNode) malloc(sizeof(Node));         // 正确
```

3.5 `free` 函数的使用

我的建议是，**平时做题不用**，因为 ds 题不会卡你的空间大小，而且数据量很小，不删除也不会造成内存泄漏。至于规范的编程方法，多读读 PPT 或教材理解一下吧。

⏰ 如果你的指针已经被 `free` 掉了，就不要再访问那一块内存了，会被认为是非法访问。

C/C++

```
p = (pNode)malloc(sizeof(Node));
free(p);
p->next = NULL; // 非法的访问，程序崩溃

struct _Node node;
pNode q = &node;
free(q);         // 同样是非法的访问（free了 非 malloc 的内存）
```

四、栈和队列

4.1 栈的板子

大道至简，够用就行，别整太复杂容易出错。

C/C++

```
#define MAX 1000005

int s[MAX];
int top = -1;

void Push(int x) { s[++top] = x; }
void Pop() { top-- }
int Top() { return s[top]; }
int Empty() { return top == -1; }
```

如果需要栈里存结构体的话开一个结构体就好了 ~

4.2 总结一下 bank

总结已经塞在 Red 第四次的题解中了，并附上了一些易错点的提示

阿蓉的总结版本

T5 银行排队模拟（生产者-消费者模拟） - 分类别

和上一题一样是一个模拟，不过这里直接使用队列模拟就好，上个题要用栈和队两种思想。

问题是，是一个队列还是n个队列?
根据生活常识，显然是一个队列，真实情况下，大家都知道，银行医院等都是摇号排队，一旦有窗口，就呼叫序号最小的人，和此题从1开始编号如出一辙

更进一步，为什么选择一个队列，因为按照题意，开关窗口是看的等待人数(而不包含正在办理人数)，而且如果预先分好，在增删窗口时就会改变分配，逻辑十分的复杂

所以我们就确定了模拟的模型，一个排队队伍(队列)，一些乘客(结构体，包含排队号码，处理时间，入队时间)，一次次的处理周期(大循环)

关于增删窗口逻辑
只在周期最初(新增人之后)考虑增加窗口，在周期最末(前面处理完成之后，新的人上去处理，队伍人数减少后)考虑减少窗口

考虑这么一个问题
要是缩减窗口之后，有人在对公窗口办理业务怎么办，根据实践，将其赶到对私窗口是能过的，那么如果用常规数组去模拟窗口，那么会变得麻烦。

- 第一种方法是5个数组存放当前办理时间模拟正在办理逻辑，这会遇到上面的问题，需要不停转移用户，保证都在对私窗口
- 第二种是没有实际的窗口，有一个处理队列指针，将客户信息设计为用户节点，队列也是链表队列，每次就查询完成业务的客户出处理链表，当链表长度小于窗口数目是出队入链
- 第三种是抽象的窗口，换个脑子，第一种是5个窗口记时间，这边3个时间记窗口数目就好了，每次高时间窗口向低的转化，最低的就是办完的，三者总数就是正在办理的窗口数目

强力推荐方法2、3，arong写的第一种，遇到颇多问题

关于坑点

- 何时结束模拟的大循环：循环次数比用户波次大并且无人等待(有人办理无所谓，我们只输出等待时间)，这就牵扯到另一个坑点只在获取新客户（不管到达新客户数是否为0）时，才按策略调整服务窗口数，也就是没新人来了(循环次数大于用户波次之后就不再考虑窗口增加了)，至于有客户去接受服务（即等待客户减少），银行将根据策略及时减少服务窗口，每次大循环结束前都判就好了，有人办理自然会减，没人去办理减也减不了
- 末尾(或者某段时间)客户少于3，部分做法会超出队列范围
- 增减窗口看的是等待人数，正在办理者不予考虑

（模型三）参考框架如下

```
while i<用户波次 and 空等待队列
    # 新人入队等待逻辑
    if i<用户波次 then
        初始化用户信息并进入等待队列最后考虑加窗口

    # 办理业务逻辑
    time[0] = time[1] # 办理时间2的变1(顺带清空时间1的窗口数)
    time[1] = time[2] # 办理时间3的变2(顺带清空时间2的窗口数)
    time[2] = 0       # 清空办理时间3的窗口数
```

```
        # 新人出队办理逻辑
        while 开放窗口数 > time[0]+time[1]+time[2] and 队列没空
            出队并将对应时间的窗口数目加一(time[x]++)，还要在这里输出
        end while

        # 末尾减少窗口逻辑
        考虑减少窗口

    end while
```

🔔 **一周期内**

1. 加入本周期内所有新的顾客到队列
2. 如果有新的顾客到队列（第一步执行），就检查是否需要增加窗口
3. 空闲且激活的窗口和窗口的顾客已经服务完成的，从队列弹出顾客进入窗口服务
4. 窗口服务顾客（窗口服务顾客指的是所有有顾客的窗口，包括不被激活但是有顾客的窗口）
5. 队列中的顾客等待
6. 检查是否需要减少窗口（只需要看减少前的平均等待人数是否小于 7，不需要考虑减少后平均等待人数是否小于 7，也就是说不管怎么样只要平均人数小于 7 就减少窗口）
7. 周期结束

也可以参考 ghgg 的博客，讲得很好。

五、树

六、查找和排序

七、图

八、大作业

> 大作业很少有人愿意给你一行一行看

可以尝试使用群公告 Repo 中的 debug 辅助工具

九、debug 教程

大家出现 bug 一般是以下几种情况：本地编译错误、本地答案错误、本地程序崩溃、本地死循环、评测错误。下面分别讨论这几种情况：

9.1 本地编译错误

我知道这个错误是最最基础的，但是也要提一下。比如当你编译发现下面出现了一大堆看不懂的英文请千万不要慌：

```
D:/Codefield/c/std/main.c:66:25: error: 'str4' undeclared (first use in this function); did you mean 'str3'?
   66 |             scanf("%s", str4);
      |                         ^~~~
      |                         str3
```

不妨耐心读一下这句话是什么意思，相信你一定能从英文的含义中获得启发的。

> 🔔 不要忽视**编译警告**！
>
> 当你在编译代码的时候发现 Warning，请仔细阅读警告信息并理解，一般警告虽然不会直接导致程序无法运行，但是可能引发潜在的 WA 以及 RE。

> 💡 **通过编译器找到本地的一些未定义行为与优先级错误**
>
> 当 gcc 编译时加入 `-Wall` 参数时，编译器会给出绝大部分可能的未定义行为的 warning，可以参考其进行 Debug。开启方式为：
>
> - Dev_c：编译选项 - 代码生成 / 优化 - 代码警告 - 显示最多的警告信息
> - Vscode：.vscode 文件夹下的 tasks.json 中，tasks 值下的 args 项中添加 `"-Wall"`
> - Visual Studio：项目 -（项目名） 属性 -C/C++ - 常规 - 警告等级，选择“启用所有警告 (/Wall)”

9.2 本地答案错误

一般来说，如果你的代码在本地跑不对测试样例，这其实是比较容易解决的。本地答案错误一般表现为：

- 程序的输出不是你想要的；

- 返回代码是 0。

为了解决这个问题你可以参考以下**调试**思路：

9.2.1 将程序划分为几个单元

数据结构的作业一般都很繁琐，而且有明显的分阶段处理。例如在数据结构第二次作业的图书管理系统中，程序让你实现以下几个功能：

- 从文件中读取书库信息；
- 在控制台交互式操作；
- 将书库的结果输出。

写代码的时候可以很容易想到分为三个部分，那么调试的时候也是这个思路，**依次调试三个部分，保证每一部分的正确性**。

9.2.2 善用打印调试法

打印调试法一般有以下几个作用：

- 检查中间变量的正确性；
- 检查程序是否正常执行到某段代码。

以图书管理系统一题为例：

首先你要保证你录入书库是正确的，否则下面的正确性无从谈起，那么你应该在代码中加入：

C/C++

```
int main() {
    ... // 读入书库并排序的操作

    /***** 加入的调试语句（如果你是数组）*****/
    int i = 0;
    for (i = 0; i < len; i++) {
        printf("%s\n", books[i].name);
    }

    /***** 加入的调试语句（如果你是链表）*****/
    List t = head;
    while (t != NULL) {
        printf("%s\n", t->name);
        t = t->next;
    }

    // 以下是第二部分：交互式控制台操作
    while (1) {
        int op;
        scanf("%d", &op);
        if (op == 0) break;
        else if (op == 1) {
            ...
    }
}
```

这样就可以检查你的读入操作是否符合期望，如果结果不对，就需要检查前面的代码。例如在读入后排序前加入调试打印语句，判断问题是出在读入还是排序上。如果结果符合期望，就可以继续加入调试语句检查后面的部分。

9.2.3 掌握 IDE 的调试功能

一言以蔽之，IDE 的调试非常的细致好用。你需要掌握的技巧包括：

- 跟踪变量的值、添加变量；
- 为代码打断点；
- 单步执行代码；

- 进入函数执行；

同时在调试的过程中如果发现卡住了，要注意是否要在控制台输入信息。

针对常用的 IDE，给出以下使用教程：

- Dev C++：dev c 调试教程
- 小熊猫 Dev；小熊猫 dev c 调试教程
- CLion：CLion 调试教程
- VSCode：VSCode 调试教程
- Visual Studio：Visual Studio 调试教程

9.3 本地运行崩溃

表现为，代码输入以后运行不输出，返回代码是一大长串。具体原因主要有以下三类：

- 0 作除数【返回码结尾 5620】；
- 内存错误（数组越界、非法指针）【返回码结尾 5477】；
- 无限递归爆栈【返回码结尾 5725】。

快速定位错误的方法：

1. 在程序中插入打印语句：

C/C++

```
int main() {
    ...... // 阶段1

    puts("Stage 1 finished");

    ...... // 阶段 2

    puts("Stage 2 finished");

    ...... // 阶段 3
    return 0;
}
```

显然，如果你的代码在崩溃前无输出，表示阶段 1 出错；输出 Stage 1 finished 表示阶段 2 出错；两句话都输出则可以定位错误在阶段 3。

2. 使用 IDE 的调试器：

一般来说如果你的 IDE 不打断点直接**调试**，是会标识出崩溃的位置的。发现死在哪一行可以为 debug 提供思路。

9.4 本地死循环

表现为运行没反应，也不停止（或者满屏输出）。排查方法：如果你的代码没有输出也不停止，可以考虑在所有循环中加入打印语句：

C/C++

```
int main() {
    for (i = 0; i < j; i++) {
        while (flag) {
            ...
            puts("1");      // 加上
        }
        for (t = head; t->next != NULL; t = t->next) {
            ...
            while(...) {
                ...
                puts("3");  // 加上
            }
            puts("2");      // 加上
        }
        puts("4");          // 加上
    }
    return 0;
}
```

运行后只需看你的输出是满屏的 1 、2、3 还是 4 即可定位错误原因。

9.5 评测错误

> 有没有一种可能，你应该先考虑一下，你交上去的是不是你**没有保存**的，或者是**旧版本**的代码呢？尤其是需要提交文件的现在。

你想要提交代码，一定要保证代码在本地运行样例是正确的，否则交上去大概率还是徒劳。但是如果本地对的，交上去是错的，那么就比较头疼了。一般解决方法有以下几种：

9.5.1 自己编造样例测试（最推荐的）

仔细阅读题面，一定要理解清楚题意。然后自己构造出一些样例来在本地运行，需要注意以下几点：

- 首先可以先随意构造几组，力求覆盖较多的情况；
- 注意边界条件，例如链表删空、栈满栈空等。

> 📢 我知道你评测不过很急，但你先别急，耐心构造几组样例总会有办法的。

发现错误之后请看 4.2~4.4 继续排查。

9.5.2 走查（静态分析）

说白了就是硬看，看自己写的每一个函数，仔细想想自己当初为什么要这样写，这种写法到底对不对，可能会出现什么问题。尤其要关注那些自己写的时候感觉迷迷糊糊很复杂的函数。

Red：其实上学期我反而是使用这种方式比较多，一是觉得直接拿测试数据帮同学 de 有一种投机取巧的感觉，二是也想更好地帮同学定位 bug 以及训练自己的分析能力，因此我也对这种方法有些许了解：

1. 首先确保自己对题意的理解是正确的，使用的方法也是正确的，否则不可能通过静态分析找到你的问题。
2. 从头到尾理一遍程序运行的逻辑，理清每一步自己想做什么，而代码实际做什么，往往能发现很多想法与实际自己写出来的东西不符的；而你不理解代码实际在做什么的部分，也更可能是出问题的部分，可以单独把这部分代码拎出来进行测试。
3. 仔细检查所有的循环条件与循环变量的更改，检查变量的初始化与多组输入时每组结束变量的重置，检查循环与递归结束的边界条件，检查变量类型、函数传参与返回类型，检查数组大小。总而言之，检查自己遇到过的所有易错的地方和遇到过的普适性 bug。
4. 直觉 ~

上面的调试方法都需要你找到一组错误的样例，而当你实在找不到一组致错的样例时（比如数组开小了导致的错误），不妨尝试一下分析自己的代码逻辑与可能的错误行为。

十、更新记录（2023/4/19 起）

- 2003/4/15：[Red] 增加了 4.1.1 开启编译警告与 4.5.2 静态分析的一些经验
- 2023/4/19：[张君豪]新增 4.1 、4.2、十等等内容，重新组织了序号。

  [Red] 加入了阿蓉对 Bank 的解析，修整了部分格式

  [Red] 新增了 1.1.4 `\r` 的解析
- 2023/4/20：[张君豪] 更新了 1.1.6 函数参数自增导致的 bug

  [DoveTao]更新了 9.1 通过编译器找到未定义行为中 VS 的设置方法

贡献者：张君豪、Red、沈锎、DoveTao

鸣谢：赵博远、刘贯恒、白楠、岳伯禹、何欣航、马昊阳、朱玉林、田一然、DoveTao

还有 Red（为什么大家都是实名喵）