计算机系统基础
实验报告

实验 **1** 【C 编程】

实验目的：

检验、练习C语言代码编写技术，包括：

- 显式管理内存
- 创建与操控基于指针的数据结构
- 正确使用字符串
- 通过附加字段来改进数据结构的性能
- 提升代码健壮性，正确应对非法输入

通过编写一个以单链表实现的线性表来实践上述要求。

实验分析：

给定单链表表元类型struct list_ele，包含char *value和struct list_ele* next两个字段。此即单链表常见实现之一。也可以把next指针放在前，并令第二个字段为长度为0的字符数组，创建表元时分配sizeof(next)+strlen(s)+1字节的内存；可以节省一个指针空间、节省一次内存申请和释放的性能开销，访问时也可以减少一次解引用。

接下来依次实现各函数：

0x0

```
queue_t *queue_new(void) { return calloc(1, sizeof(queue_t)); }
```

该函数分配内存给一个新的队列，失败则返回NULL，成功则返回分配到的地址。

Calloc在分配失败时也返回NULL，成功时还会将分配到的内存置0，可以便捷地实现该函数。

0x1

```
void queue_free(queue_t *q) {
    if (NULL == q) return;
    for (list_ele_t *p = q->head, *t; p; p = t)
        t = p->next, free(p->value), free(p);
    free(q); // q should be set to 0 MANUALLY.
}
```

该函数释放参数q所指队列所占用的全部内存。由于入参为queue_t *q，咕只能做到将所指队列q释放；函数调用者应当手动将释放后的队列置NULL。

0x2

为方便实现头插、尾插函数，先编写一个创建新表元的函数:

```
static list_ele_t *ele_new(const char *s) {
    list_ele_t *ele = malloc(sizeof(list_ele_t));
    if (ele) {
        if ((ele->value = malloc(strlen(s) + 1)))
            strcpy(ele->value, s), ele->next = NULL;
        else
            free(ele), ele = NULL;
    }
    return ele;
}
```

先为新表元本身分配内存，分配失败则直接返回NULL。

（新表元分配成功时）再为字符串s分配strlen(s)+1字节的内存，并将原字符串拷贝到此处，next设为NULL;

若这次分配失败，须将新表元本身释放再返回NULL，否则会引发内存泄漏。

0x3

```
bool queue_insert_head(queue_t *q, const char *s) {
    if (NULL == q)
        return false;

    list_ele_t *newh = ele_new(s);
    if (NULL == newh)
        return false;

    if (q->head == NULL)
        q->tail = newh;
    newh->next = q->head;
    q->head = newh;
    q->size++;
    return true;
}
```

q为空指针、新表元分配失败时均返回NULL;

q所指队列为空时，令q->tail指向新表元。

0x4

为了快速实现尾插，在queue_t内添加tail字段。其余逻辑与头插类似。

```
bool queue_insert_tail(queue_t *q, const char *s) {
```

```
    if (NULL == q)
        return false;

    list_ele_t *newt = ele_new(s);
    if (NULL == newt)
        return false;

    if (q->tail)
        q->tail = q->tail->next = newt;
    else
        q->head = q->tail = newt;
    q->size++;
    return true;
}
```

0x5

```
static size_t min(size_t a, size_t b) { return a < b ? a : b; }

bool queue_remove_head(queue_t *q, char *buf, size_t bufsize) {
    if (!q || !q->head)
        return false;

    if (buf && bufsize) {
        size_t len = min(bufsize - 1, strlen(q->head->value));
        memcpy(buf, q->head->value, len);
        buf[len] = '\0';
    }
    if (q->head == q->tail)
        q->tail = NULL;
    list_ele_t *t = q->head->next;
    free(q->head->value), free(q->head);
    q->head = t;
    q->size--;
    return true;
}
```

为了程序的健壮性（鲁棒性），须先测试buf非NULL且bufsize非0（以防bufsize-1变为SIZE_T_MAX）。

0x6

以上函数均会或设定或改变队列大小，可以在queue_t内添加size字段并维护，这样size函数便可以快速完成其工作：

```
size_t queue_size(queue_t *q) { return q ? q->size : 0; }
```

0x7

```
void queue_reverse(queue_t *q) {
    if (!q || !q->head)
        return;
    list_ele_t *pre = NULL, *cur = q->head;
    while (cur) {
        list_ele_t *tmp = cur->next;
        cur->next = pre, pre = cur, cur = tmp;
    }
    q->tail = q->head, q->head = pre;
}
```

要不借助额外空间（即不调用malloc或free）来翻转链表，须使用两个指针cur和pre，分别指向原链表中当前遍历到的元素、已翻转的链表的头元素。

每次将cur所指元素插到已翻转的链表的头部，并更新cur为原链表中下一个元素，如此循环直至cur为NULL。

Debug：

实现了大体逻辑后提交评测只得到21分。修改如下错误后分数依次增加：

21pts → 48pts

```
ele->value = malloc(strlen(s))
```

字符串测试。字符串结尾的’\0’需要额外的存储空间。

```
ele->value = malloc(strlen(s) + 1)
```

48pts → 55pts

健壮性测试。需要先检验缓冲区buffer的有效性。

```
bool queue_remove_head(queue_t *q, char *buf, size_t bufsize) {
    ...
    if (buf && bufsize) { ... }
    ...
}
```

55pts → 86pts

```
bool queue_remove_head(queue_t *q, ...) {
    if (!q || !q->head)
        return false;
    ...
    return true; // rather than “return --(q->size)”;
}
```

当且仅当 q 或 q->head 为 NULL 时才返回 false，否则就返回 true。

86pts → 100pts

内存泄漏。个人认为这是比较严重的疏漏。

原来的写法为：

```
static bool ele_construct(list_ele_t *ele, const char *s) {
    return (ele->value = malloc(strlen(s) + 1))
               ? (strcpy(ele->value, s), 1)
               : 0;
}

bool queue_insert_head(queue_t *q, const char *s) {
    ...
    list_ele_t *newh = calloc(1, sizeof(list_ele_t));
    if (NULL == newh || !ele_construct(newh, s))
        return false;
   ...
}
```

在 newh->value 分配失败时，未释放 newh 即从 queue_insert_head 返回，导致 newh 指向的内存泄漏。

发现问题后，直接的修改方法如下：

```
static bool ele_construct(list_ele_t *ele, const char *s) {
    if ((ele->value = malloc(strlen(s) + 1)))
        return strcpy(ele->value, s), true;
    else
        return free(ele), false;
}
```

不过忘记释放内存反映了内存管理模式存在问题。相较于先为对象分配内存，再显式在所分配的内存上构建对象（可能产生异常），不如将分配内存和构建对象的工作放到同一个底层函数内。此即“先 allocate 再 construct”与“直接 new”的区别。

借鉴于 C++的 new 关键字，设计一个创建新元素的 ele_new 函数，在内部处理各种可能的异常后再将结果（或异常）反映到返回值。这样可以大大减轻调用者的负担。该函数的实现见 0x2。

心得体会：

- 手动管理内存务必注意变量的生存周期，malloc（或calloc、realloc）必须与free成对出现，保证各个条件分支下内存资源都被正确地分配和释放。否则就会出现内存泄漏、多次释放同一块内存等问题。
- 使用C风格字符串时需要多分配一个字节用来存储结束符’\0’。
- 一个健壮的程序使用他人提供的指针时，需要先检验其有效性。