

## 结构体数组的排序

- 结构体

结构体的定义: 结构体类型的定义、结构体变量的定义

结构体的初始化: 定义结构体类型的同时定义结构体变量并赋初值、定义结构体变量时赋初值, 缺省值

结构体的操作: 作为整体复制与赋值, 通过&取地址, 通过"."访问成员

- 简单排序算法(升序)

- 交换排序

```
for(i=0;i<n-1;i++)
    for(j=i+1;j<n;j++)
        if(a[i]>a[j]){
            swap(s[i],a[j]);
        }
```

- 选择排序

```
for(i=0;i<n-1;i++){
    k=i; //i代表当前要确定的第i小的数, 显然, 第0、1、...、i-1小的数都已确定
    for(j=i+1;j<n;j++)
        if(a[k]<a[j])
            k=j;
    if(k!=i){
        swap(a[k],a[i])
    }
}
```

- 冒泡排序

```
for(i=0;i<n-1;i++)
    for(j=0;j<n-i-1;j++)
        if(a[j]>a[j+1]){
            swap(a[j],a[j+1])
        }
```

若已经是升序了, 依旧会进行 $n*n$ 次比较, 白白浪费时间。

冒泡排序的改进方式:

```
for(i=0, change=True; i<n-1 && change; i++){
    change=False;
    for(j=0; j<n-i-1; j++)
        if(a[j]>a[j+1]){
            change=True;
            swap(a[j],a[j+1])
        }
}
```

## ◦ 插入排序

插入排序将原本的元素集合U划分成两个子集，一个为有序集合A，一个为无序集合B， $A+B=U$ 。初始时， $A=\{a[0]\}$ ，这是因为一个元素显然是有序的。当外层循环为i时，0、1、2、...、i-1已经排好序，此时考察下标为i的元素应该插入在哪个位置才不会破坏升序的性质，故名为“插入排序”。

```
for(int i=1;i<n;i++){
    k=a[i];
    j=i-1;
    while(j>=0 && a[j]>k){
        a[j+1]=a[j];
        j--;
    }
    a[j+1]=k;
}
```

## 寻找长度最长的单词

- 二维数组存储words，maxi记录当前最长的单词的下标，这样一次遍历完成之后即可找到最长的单词，也即选做要求的 $O(N)$ 的时间复杂度

## 计算斜率

- 输入两个点的坐标，判断点的关系(防止重合或者斜率不存在的情况)

## 子串查找

- 一种很坏的情况 text = aaaaaaaaaab(9a1b), pattern = aab，会发生什么情况？比较 $3*8$ 次， $O(m*n)$
- 感兴趣的同学可以了解一下KMP算法，最坏情况也是 $O(m+n)$ 的时间复杂度

## 字符串逆序输出

- 用指针处理数组元素

## 打印日历 P110 T2

输入指定年份与元旦那天是星期几，然后输出这一年的日历

- 判断是否是闰年
  - 能被4整除、但不能被100整除，或能被400整除的年份为闰年
- 闰年和平年的区别是二月份的天数，闰年二月份29天（一年365天），平年二月份28天（一年366天）
- 格式化输出
- 编写函数print\_one\_month(int num\_days, int first\_day);实现给定每个月的天数和这个月第一天是星期几，输出这个月的日历
- 主函数
  - 根据输入的年判断是否为闰年
  - 求出每个月的天数和第一天是星期几
  - 假设一月一日为星期一，那么如何求得二月一号是星期几？

$$(1 + 31) \% 7 = 32 \% 7 = 4$$

那么二月一号就是星期4，依此类推。

## 学生信息管理系统

- 有序链表的建立
- 链表排序

这里只对值进行交换而不交换指针，交换指针的操作比较繁琐，有兴趣的同学可以自己实现一下

```
//结构体定义
struct student {
    int id;
    char name[20];
    char gender;
    float score;
    struct student* next;
};
//使用typedef重命名
typedef struct student student;
//选择排序
void SelectSort(student* head) {
    //类比数组排序，使用两个变量i和j分别进行外层和内存的循环
    //head为头指针
    student *i, *j, *min;
    i = head -> next;
    while(i -> next != NULL){
        min = i;
        j = i -> next;
        while(j){
            if(min -> score > j -> score)
                min = j;
        }
        if(min != i){
            swap(min -> id, i -> id);
            swap(min -> name, i -> name); //字符串的交换可能得用到字符串处理的相关函数
            swap(min -> gender, i -> gender);
            swap(min -> score, i -> score);
        }
        i = i -> next;
    }
}
```

- 链表的销毁

链表的销毁要逐个结点的释放内存

- 文件的打开、读取、关闭

## 取指定区间的字符串

• 动态分配内存

函数名	函数原型	函数功能	返回值
calloc	void *calloc(unsigned n, unsigned size);	分配n个数据项的连续内存空间，每个数据项的大小为size	返回起始地址，若不成功返回NULL
malloc	void *malloc(unsigned size);	分配大小为size字节的内存区	返回起始地址，若不成功返回NULL
realloc	void *realloc(void *p,unsigned size);	将p所指的已分配内存区的大小改为size，size可比原来的size或大或小	返回执行该内存区域的指针
free	void free(void *p);	释放p所指的内存区	无返回值

常用的是malloc和free，且通常配对出现，在学习链表和结构体的内容时使用较多

• 动态内存分配和变长数组

变长数组（variable-length array），C语言术语，也简称VLA。是指用整型变量或表达式声明或定义的数组，而不是说数组的长度会随时变化，变长数组在其生存期内的长度同样是固定的。 --百度百科

可变长数组是指在计算机程序设计中，数组对象的长度在运行时（而不是编译时）确定。 -- 维基百科

具体的例子如下：

```
int main(){
    int n;
    scanf("%d",&n);
    int a[n];
}
```

这是很多同学在学习动态分配内存之前喜欢使用的一种定义方式，然而并不是所有的编译器都支持可变长数组的。所以在学习了动态分配内存之后，尽量避免使用变长数组。

两个多项式的和

- 假定输入有序（无序则增加一个排序的步骤，参考学生管理系统中对链表的排序）
- 有序链表的合并，时间复杂度为O(m+n)（头节点）

```
//给定两个按e升序排列的链表a和b（a，b为头结点），将它们合并成为一个新的按e升序排列的链表
//定义结构体如下
struct node{
    int c;
    int e;
    struct node *next;
};
typedef struct node node;

node* merge(node *ha,node *hb){
    node *p1 = ha->next;
    node *p2 = hb->next;
```

```

//使用ha作为合并后的链表的头结点
node *tail = ha;
node *temp;
free(hb);
while(p1 && p2){
    if(p1->e < p2->e){
        //p1指数较小, 将p1从第一个链表中取下来, 接在tail后面
        tail -> next = p1;
        p1 = p1 -> next;
        //tail后移, 使得tail始终指向新链表的最后一个结点
        tail = tail -> next;
    }
    else if(p1->e > p2 -> e){
        //p2指数较小, 将p2从第二个链表中取下来, 接在tail后面
        tail -> next = p2;
        p2 = p2 -> next;
        tail = tail -> next;
    }
    else{
        //p1 p2的指数相同, 系数叠加
        p1 -> c += p2 -> c;
        //将p1取下来, 接在tail后面, 同时, 释放p2
        tail -> next = p1;
        p1 = p1 -> next;
        temp = p2;
        p2 = p2 -> next;
        free(temp);
        tail = tail -> next;
    }
}
}
}

```

可以引入一个时间复杂度为 $O(n\log n)$ 的排序算法, 归并排序。

## 整理输出学生成绩链表

- 文件的打开、读取
- 结构体链表的排序

## 比较两个文件是否相同

- 命令行参数

## main函数的参数

```
int main(int argc, char *argv[]);
```

- 第一个参数 int argc, 表示命令行参数的个数。
- 第二个参数 char \*argv[], 是一个指向命令行参数的指针数组:

- 每一参数又都是以空字符（null）结尾的字符串。
- 首个字符串 argv[0]，标识程序名本身

### 举个例子

假设程序test.exe需要从命令行读入两个参数代表两个文件名，分别为text1.txt, text2.txt

在黑框中输入：

```
test.exe text1.txt text2.txt
```

test.cpp 代码如下：

```
#include<stdio.h>
#include<stdlib.h>
int main(int argc, char *argv[]){
    printf("argc=%d\n",argc);
    for(int i=0;i<argc;i++){
        printf("argv[%d]=%s\n",i,argv[i]);
    }
}
```

则运行结果如下：

```
argc=3
argv[0]=test.exe
argv[1]=text1.txt
argv[2]=text2.txt
```

一些同学打不开文件，可能是由于目录问题导致，这里提供两个比较简单的解决方式

- exe文件和需要打开的文件都使用绝对目录
- 将exe文件和需要打开的文件放在同一目录下,然后在当前目录下打开黑框或者先打开黑框之后通过cd命令进入该目录，在当前目录下，就可以简单的使用形如：

```
test.exe text1.txt text2.txt
```

的命令来执行（只用名称即可）

### 统计出现次数最多的单词

每读出一个词，在结构体链表中查找，找到则count++，没找到则建立新的结点

文件读取结束之后，对结构体链表进行排序，输出前十个单词

```
struct node{
    char a[100];
    int count;
    struct node* next;
};
```

- 判断文件结束

(引用自隔壁班陈助教)

### feof的常见错误用法

```
fp = fopen("stardata.txt", "r");
if (fp == NULL) { printf("打开stardata.txt文件出错了\\n"); break; }
else {
while (!feof(fp)) putchar(fgetc(fp));
fclose(fp);
}
```

**feof** 并非是判断是否读到文件末尾 EOF，而是判断当前光标后面是否有内容。当光标在 EOF 前时，其后仍有内容 EOF，故而函数返回 1。当读取了 EOF 后，光标后面无内容，此时才返回 0。

修改如下：

```
char ch;
while((ch=fgetc(fp))!=EOF) putchar(ch);
```

## 一些小复习

### 指针

- 行指针和指向元素的指针

(引用自隔壁班陈助教)

我们通俗理解下：

行指针和指向元素指针的区别是指向的内容不同，表现在类型不同。

假设类型为 int。

行指针是说，指针指向的内存中放的是一个整形数组 int a[n]。将这个 int 数组 a 想象成一个数据点，那么其类型是 int[n]，所以要定义指针 int[n]\* p = &a。

指向元素的指针是说，指针指向的内存中放的是一个 int 变量 b，容易定义指针 int \*q = &b。

**但显然int[n]\* p = &a 这样写编译是不会通过的。所以有两种正确写法；**

```
//第一种写法
int (*p)[n] = &a; // 常用写法，这里不做详细说明

//第二种写法
typedef int T[3];
int main(){
    T* a;
    int(*b)[3];
    int c[3][3] = {1,2,3,4,5,6,7,8,9};
    b = c;
    a = c;
}
```

使用VS对上面第二段代码进行调试，跟踪变量，结果如下，可以看到，a、b都是int [3]\* 类型的变量

▸ a	0x0000006cc571fb58 {1, 2, 3}	int[3] *
▸ a+1	0x0000006cc571fb64 {4, 5, 6}	int[3] *
▸ b	0x0000006cc571fb58 {1, 2, 3}	int[3] *
▸ b+1	0x0000006cc571fb64 {4, 5, 6}	int[3] *

那对行指针进行加1的操作会发生什么？

结果也在上图中给出，可以看到，a+1和b+1之后，指向的是c这个二维数组的下一行（指向元素的指针很简单，这里就不赘述了）

### • 指针数组和数组指针

首先理解[]和\*这两个运算符的优先级和结合性

从教材245页的附录C可以知道，[]的优先级高于\*的优先级，可以得到如下等式：

```
int *p[3] = int *(p[3])
```

既然p先跟[3]结合，那么p[3]定义的就是一个数组，可以理解为p的类型是一个数组，这个数组有三个元素，那么每个元素是什么类型呢？显然就是int \*类型了，也就是说，**int \*p[3]**定义了一个数组，这个数组有三个元素，每个元素为一个指向int型的指针。可以记为**type(p) = array(pointer(int),3)**

那么，如果用括号将\*p括起来呢？

```
int (*p)[3];
```

p先跟\*结合，那么意味着p是一个指针，那么这个指针指向什么？显然就是**int [3]**这样的东西，也就是说p是一个指向一个包含3个元素的数组的指针，记为**type(p)=pointer(array(int,3))**，这样的解释跟上面“行指针和指向元素的指针”这一小节的内容也是吻合的。

## 短路定理

- ```

a && b && c
//求a、b、c、d的值
int main()
{
    int a,b,c,d;

    a = 0;
    b = 1;
    c = 2;
    d = a++ && b++ && --c;

    printf("a=%d b=%d c=%d d=%d\n",a,b,c,d);
}
//a++ -> ++a
int main()
{
    int a,b,c,d;

    a = 0;
    b = 1;
    c = 2;
    d = ++a && b++ && --c;

    printf("a=%d b=%d c=%d d=%d\n",a,b,c,d);
}

```



```
a=1 b=1 c=2 d=0
a=1 b=2 c=1 d=1
```

- a || b || c

```
int main()
{
    int a,b,c,d;

    a = 0;
    b = 1;
    c = 2;
    d = a++ || b++ || --c;

    printf("a=%d b=%d c=%d d=%d\n",a,b,c,d);
}

int main()
{
    int a,b,c,d;

    a = 0;
    b = 1;
    c = 2;
    d = ++a || b++ || --c;

    printf("a=%d b=%d c=%d d=%d\n",a,b,c,d);
}
```

```
a=1 b=2 c=2 d=1
```

```
a=1 b=1 c=2 d=1
```

- ||和&&同时出现，注意优先级

## 链表的常规操作

- 链表的建立
  - 头插法
  - 尾插法
- 链表的遍历
- 删除节点 (p->next或者伴随指针)
- 插入节点 (q插在p的后面)
- 特殊情况 (ppt第十章p36)

## 链表的使用细节

头节点和首元结点：

- 首元结点：链表中存储第一个数据元素的结点。
- 头节点：它是在首元结点之前附设的一个节点，其指针域指向首元结点。头节点的数据域可以不存储任何信息，也可以存储与数据元素类型的其他附加信息，例如，当数据元素为整数型时，头节点的数据域中可存放该线性表的长度。
- 简单说，头节点是不存储实际元素的结点，设在首元结点之前，首元结点是第一个存储有实际数据元素的结点。

很多时候，引入头节点可以简化代码，也可以减少一些麻烦的对头节点的特殊处理

(建议大家使用头节点简化编码，但是没有头节点的也要会处理)

## 文件处理

- 打开文件fopen，关闭文件fclose

```
//fopen
FILE *fopen(char *name,char *mode)
//name为文件名,mode为打开模式

//fclose
int fclose(FILE *fp) //成功返回0值，否则非0
```

| 文件使用方式        | 含义                  |
|---------------|---------------------|
| "r/rb" (只读)   | 为输入打开一个文本/二进制文件     |
| "w/wb" (只写)   | 为输出打开或建立一个文本/二进制文件  |
| "a/ab" (追加)   | 向文本/二进制文件尾追加数据      |
| "r+/rb+" (读写) | 为读/写打开一个文本/二进制文件    |
| "w+/wb+" (读写) | 为读/写建立一个文本/二进制文件    |
| "a+/ab+" (读写) | 为读/写打开或建立一个文本/二进制文件 |

- 文件读写fscanf和fprintf

```
//fscanf
int fscanf(FILE* fp, char* format, args,...);
//从指定的文件流fp中按照指定的格式format读取数据，并将读取到的数据存储到对应的变量中。它可以用于从文件中读取各种类型的数据，例如整数、浮点数、字符串等
//例如：fscanf(stdin,"%s%d",s,&a); 从键盘读取
//fprintf
int fprintf(FILE* fp, char* format, args,...);
//将格式化的数据按照指定的格式format写入到指定的文件流fp中。它可以用于向文件中写入各种类型的数据，例如整数、浮点数、字符串等
//例如：fprintf(fp,"%s %d",s,a);
```

- 字符读写fgetc和putc

```
int fgetc(FILE *p);
//从fp指定的文件中取得下一个字符
//返回所取得的字符，若读入错误则返回EOF

int fputc(char ch, FILE *fp);
//将字符ch输出到fp指向的文件中
//成功则返回该字符，否则返回EOF(-1)
```

- 字符串读写fgets和fputs

```
char* fgets(char *buf,int n ,FILE*fp);
//fgets从fp所指文件读n-1个字符送入s指向的内存区,并在最后加一个'\0' （若读入n-1个字符前
遇换行符或文件尾(EOF)即结束）
//成功则返回指向该串的指针，出错或遇到文件结束符则返回空指针

int fputs(char *s,FILE *fp)
//fputs把s指向的字符串写入fp指向的文件
//成功则非负数，错误则返回EOF(-1)
```

- 数据块读写fread和fwrite

```
int fread(char*buffer,unsigned size,unsigned count,FILE*stream);
/*
1.buffer: 是读取的数据存放的内存的指针，
           (可以是数组，也可以是新开辟的空间)
   ps: 是一个指向用于保存数据的内存位置的指针（为指向缓冲区
        保存或读取的数据或者是用于接收数据的内存地址）
2.size: 是每次读取的字节数
3.count: 是读取的次数
4.stream: 是要读取的文件的指针
   ps: 是数据读取的流（输入流）
*/
//若成功，返回实际读取的数据项个数，若失败则返回0

int fwrite(char *ptr,unsigned size,unsigned n,FILE *fp);
//将ptr指向的n*size个字节输出到fp指向的文件中
//返回写到fp中的数据项个数
```

- 判断文件是否结束(二进制文件)feof

```
int feof(FILE *fp);
//检查文件是否结束
//遇到文件结束符则返回EOF，非0，否则返回0
```

- 文件指针处理fseek、rewind、ftell

```
int fseek(FILE*fp,long offset,int base);
//将fp指向的文件的位置指针移到以base为基准，以offset为偏移量的位置
//offset为位移量（从base开始移动的字节数）正数向后移动，负数向前移动
//base的几个取值：文件开始 SEEK_SET 0 文件当前位置 SEEK_CUR 1 文件末尾 SEEK_END 2

//成功则返回0，否则返回非0

void rewind(FILE *fp);
```

```
//将fp指向的文件的位置指针置于文件开头。这意味着之后对该文件进行的读写操作将从文件的开头开始。常用于在对文件进行多次读取操作之前，将文件指针重置到起始位置，以便重新读取文件内容
//无返回值

//由于使用rewind没有返回值，无法确定是否rewind成功，因此可以使用下面的fseek替换
if (fseek(fp, 0L, SEEK_SET) != 0)
{
    //不为0意味着没有成功
    //可以在此输出错误提示
}
long ftell(FILE *fp);
//返回位置指针当前位置(用相对文件开头的位移量表示)
//成功，返回当前位置指针位置；失败，返回-1
```

## 自增和自减

```
int i = 1;
int a = i ++ 1;
printf("%d",a);
//2
```

```
int i = 1;
int a = ++i + 1;
printf("%d",a);
//3
```

- 左值和右值

简单来说，左值是指可以出现在赋值运算符左边的东西，它是可以被改变的，它是存储数据值的那块地址的内存（但不一定是我们常规意义上的指针），比如变量名，右值是指可以出现在赋值运算符右边的东西，它是不可以被改变的，比如数值、常量、表达式等。也可以理解为右值为变量的地址，左值为变量的数据。左值可以是右值，但右值不能是左值。

由此可以进一步理解 `i++` 和 `++i`

`i++`只能是右值，它实际上是返回*i*原本的值，再将*i*加1

`++i`可以是左值也可以是右值，它实际上是先对*i*对应的内存中存储的值进行加1，然后返回*i*，这个*i*是变量的地址，所以可以继续对它赋值

```
//一道有意思的题
int i = 1;
int a = i ++;
printf("%d\t",a);
++i = 1;
printf("%d\t",i);
++i = i++;
printf("%d\t",i);
//1 1 3
```

```
i ++ = 1 // 可以这样写吗？
```

|     | 代码    | 说明             |
|-----|-------|----------------|
| ✖   | C2106 | "=": 左操作数必须为左值 |
| abc | E0137 | 表达式必须是可修改的左值   |

## 逗号表达式和三目运算符

- 逗号表达式

```
(expression1, expression2, ..., expressionn)
```

逗号表达式的求值顺序是从左到右，依次计算每个表达式，并返回最后一个表达式的值。

但是计算式不能直接看最后一个表达式，因为前面的表达式可能改变一些变量的值。

- 三目运算符

```
condition ? expression_if_true : expression_if_false;
//example
int max = (a > b) ? a : b;
```

## 二分查找

```
int binarysearch(int arr[], int low, int high, int key) {
    while (low <= high) {
        int mid = low + (high - low) / 2;

        if (arr[mid] == key) {
            return mid; // 找到关键字，返回索引
        }
        else if (arr[mid] < key) {
            low = mid + 1; // 关键字在右半部分
        }
        else {
            high = mid - 1; // 关键字在左半部分
        }
    }

    return -1; // 关键字不存在，返回 -1
}
```

有趣的地方：对于有序列表，它可以将原本的，通过遍历去查找的时候，可能需要进行n次的比较，降为了

$$\log_2(n)$$

比如 $n=2^{30}$ ，原本需要比较的次数的数量级高达 $10^9$ ，使用二分查找仅需要30次比较！！

## 一些个复习Tips

- 考前仔细复习教材和老师的PPT
- 完成老师提供的复习资料
- 刷往年真题

*Good luck to everyone on the exam!*