

数据结构整理——数据结构基础

摘自数据结构PPT及作业 并且根据个人需要进行一些删改

Homework 1

字符串的赋值

定义的字符数组只能在定义阶段使用赋值符号 `char str[] = "China";`, 或者 `char *s, s="abcdefg";`, 在之后只能使用strcpy函数 `strcpy(str, "China")`, 并且要注意长度是否满足。

切记不能不能!!!

```
char str[100];
str[100]="China";
```

即不能在赋值语句中通过赋值运算符"="对字符数组整体赋值, 并且数组之间也不能直接赋值, $a=b$ 。

删除字符串中特定字符操作

```
for(i=j=0; s[i]!='\0'; i++){
    if(s[i]!='char')//char 是要删除的字符, 随需要更改
        s[j++]=s[i];
}
s[j]='\0'; /*一定要注意给'\0', 否则这不是一个正确的字符串*/
```

字符串数字转换

```
for (a = 0; ch[a] >= '0' && ch[a] <= '9'; a++){
    s = 10 * s + ch[a] - '0';
}
```

区别指针数组和数组指针

指针数组: 是数组, 每一个元素是指针类型, 定义形式 `int/char/... *name[number], *`
(`name[number]`)也可, 例:

```
char *language[] = {"FORTRAN", "BASIC", "PASCAL", "JAVA", "C"};
```

```
int a[5], *num[5] = {&a[0], &a[1], &a[2], &a[3], &a[4]};
```

数组指针: 是指针, 指向数组类型, 定义形式 `int/char/... (*name)[number]`, 例:

```
{
    int a[4][5];
    int (*p)[5]=a; //括号里为二维数组的列数
}
```

p是一个指针变量, 它指向包含5个int元素的一维数组, 此时p的增量以它所指向的一维数组长度为单位(相当于一行一行的移位);

$p+i$ 是一维数组 $a[i]$ 的地址，即 $p+i=\&a[i]$ ；对该式两边作取内容运算（*）得 $*(p+i)=a[i]$ ，由于二维数组中 $a[i]=\&a[i][0]$ ，则 $*(p+i)$ 表示 $a[i][0]$ 的地址，即 $*(p+i)=\&a[i][0]$ ；

$*(p+2)+3$ 表示 $a[2][3]$ 地址（第一行为0行，第一列为0列）， $((p+2)+3)$ 表示 $a[2][3]$ 的值。

函数指针

函数指针的调用

```
int (*fun)()=strlen;
//可以fun()
//也可以 (*fun)()
```

十进制转十六进制

```
do{
    h=n%16;
    s[i++]=(h<=9)?h+'0':h+'A'-10;
}while((n/=16)!=0);
```

最后记得把字符串颠倒过来

整行转换为字符串

```
void itoa(int n,char s[])
{
    int i,sign;
    if((sign=n)<0)
        n=-n;
    i=0;
    do{
        s[i++]=n%10+'0';
    }while((n/=10)>0);
    if(sign<0){
        s[i++]='-';
    }
    s[i]='\0';//一定要加字符串的结束标志
    reverse(s);
}
```

strstr()的原型函数

$O(m*n)$ 的时间复杂度

函数`index(char s[],char t[])`检查字符串 s 中是否包含字符串 t ，若包含，则返回 t 在 s 中的开始位置（下标值），否则返回-1。

```
int index(char s[],char t[])
{
    int i,j,k;
    for(i=0;s[i]!='\0';i++)
    {
        for(j=i,k=0;t[k]!='\0'&& s[j]==t[k];j++,k++);
        if(t[k]=='\0')
            return i-k;
    }
    return -1;
}
```

```

    return i;
}
return -1;
}

```

或者模糊处理（忽略大小写）

```

int index(char s[],char t[])
{
    int i,j,k;
    for(i=0;s[i]!='\0';i++)
    {
        for(j=i,k=0;t[k]!='\0'&&tolower(s[j])==tolower(t[k]);j++,k++);
        if(t[k]=='\0')
            return i;
    }
    return -1;
}

```

改进算法1:

```

int index(char s[], char t[])
{
    int i, j, k,n,m;
    n = strlen(s);
    m = strlen(t);
    for(i =0; n-i >= m; i++){//当剩余字符串的长度小于查找字符串的长度时，停止查找，节省时间
        for(j=i,k=0;t[k]!='\0'&&s[j]==t[k]; j++,k++)
            ;// 注意分号!
        if(t[k] == '\0')
            return ( i);
    }
    return ( -1);
}

```

改进算法2:

```

int index(char s[], char t[])
{
    int i = 0, j=0;

    while ( s[i]!='\0' && t[j]!='\0') {
        if (s[i] == t[j]) {
            i++;
            j++;
        }
        else {
            i = i-j +1;//i回退到上次匹配的起始位置的下一个位置
            j = 0;//原字符串回到开头
        }
    }
    if ( t[j] == '\0')
        return i-j; //返回字符串的开始查找位置
    else
        return -1;
}

```

KMP算法

$O(m+n)$ 的时间复杂度

源串称为主串，定义为S，当前匹配位置为i；目标串称为子串，定义为T，当前匹配位置为j。当前匹配在找到不匹配的字符后，重新开始匹配时：

- 主串当前位置i不回溯，即不重置为上次匹配开始位置的一下位置；
- 子串当前位置j视情况回溯至起始串位置（0），或子串中某一位置。

根据子串T当前匹配的规律：“ $T_0 \dots T_{k-1} = T_{j-k} \dots T_{j-1}$ ”
由当前失配位置j(已知)，可以归纳计算下次匹配起点k的表达式。

令 $k = \text{next}[j]$ （函数next用子串当前位置j来计算下次开始匹配位置k，k与j显然具有函数关系），则

$$\text{next}[j] = \begin{cases} -1 & \text{当 } j=0 \text{ 时} \\ \max \{ k \mid 0 < k < j \text{ 且 } 'T_0 \dots T_{k-1}' = 'T_{j-k} \dots T_{j-1}' \} & \\ 0 & \text{其他情况} \end{cases}$$

取当前位置j前序串首尾最大的相同子串长度

注意：

- (1) k值仅取决于子串本身而与相匹配的主串无关。
- (2) k值为子串从头向后及从j向前的两分子串最大相同子串的长度。
- (3) 这里的两分子串可以有部分重叠的字符，但不可以全部重叠，即k最大为j-1。

```
int KMPindex(char S[], char T[])
{
    int i = 0, j = 0, *next;
    next = (int *)malloc(sizeof(int)*(strlen(T)+1)); //包含了'\0',所以要+1
    getnext(T, next);
    while (S[i]!='\0' && T[j]!='\0') {
        if (S[i] == T[j]) {
            i++;
            j++;
        }
        else {
            (j == 0) ? i++ : (j = next[j]); //j回退到相应位置开始匹配，i值不变
        }
    }
    free(next);
    if (T[j] == '\0') //匹配成功，返回匹配位置
        return i-j;
    else
        return -1;
}

void getnext(char T[], int next[])
{
    int i = 0, j = -1;
    next[0] = -1;
    while (T[i]!='\0') {
        if (j == -1 || T[i] == T[j]) { //i为后缀位置；j为前缀位置
            i++;
            j++;
        }
    }
}
```

```

        next[i]=j;
    }
    else
        j = next[j]; //若字符不同，则j值回溯
    }
}

```

reverse函数

倒置字符串

```

version1:
int first=0;
int end=strlen(s)-1; //一定要注意-1
while(first<end){
    tmp=s[first];
    s[first]=s[end];
    s[end]=tmp;
    first++,end--;
}
=====
version2:
for(i=0,j=strlen(str)-1;i<j;i++,j--)
{
    k=str[i];
    str[i]=str[j];
    str[j]=k;
}

```

inverp函数

将字符串反向输出但不改变原字符串

```

void inverp(char *a)
{
    if ( *a=='\0')
        return;
    inverp(a+1); //运用了递归的思想，一直递归到结束再反向输出
    printf("%c", *a );
}

```

升序合并函数

将已按升序排好的两个字符串a和b中的字符按升序并归到字符串c中

以升序为例

```

version1 数组版本:
char a[]="acegikm";
char b[]="bdfhjlnpq";
char c[80],*p;
int i=0,j=0,k=0;
while(a[i]!='\0'&&b[j]!='\0')
{
    if(a[i]<b[j]){c[k++]=a[i++]}
}

```

```

    else{c[k++]=b[j++]}
```

```

}
c[k]='\0';//为strcat做准备
if(a[i]=='\0')//a字符串已经排列完，只剩下b字符串的剩下部分
    p=b+j;//移位，也可以是p=&b[j];
else
    p=a+i;//p=&a[i];
strcat(c,p);
=====
version2 链表版本：
/*lista,listb为两个升序链表的头指针*/
LinkList MERGELIST(LinkList lista,LinkList listb)
{
    LinkList listc,p=lista,q=listb,r;
    if(lista->data<=listb->data){
        listc=lista;
        r=lista;
        p=lista->link;
    }
    else{
        listc=listb;
        r=listb;
        q=listb->link;
    }
    while(p!=NULL&&q!=NULL){
        if(p->data<=q->data){
            r->next=p;
            r=p;
            p=p->next;
        }
        else{
            r->next=q;
            r=q;
            q=q->next;
        }
    }
    r->link=p?p:q;
    return listc;
}

```

Homework2

几个重要文件操作函数

`in=fopen(FILEname,mode)`

`fclose(FILE *)`

`fputs(char *,FILE *)`

`fgets(char *,MAXSIZE,FILE *)` 错误时的返回值为NULL

`c=fgetc(FILE *)` 返回值为读入字符,否则为EOF

`fputc(char ,FILE *)`

`fscanf(FILE *,Format,...)`

```
fprintf(FILE *,Format,...)
```

```
freopen(FILEname,mode,FILE *)
```

swap函数的定义和使用

原理：要对传入的**指针的值**进行交换，交换指针的指向(**地址**)**不能**使a,b的值交换，因为在函数里面指针变量是形参，地址交换是局部的，不会改变实参指针变量。

c语言实参变量和形参变量之间的数据传递是单向的“值传递”方式。用指针变量做函数参数时同样要遵循这一规则。不可能通过执行调用函数来改变实参指针变量的值，但是可以改变实参指针变量**所指变量**的值。

```
void swap(int *m,int *n)
{
    int t;
    t=*m;
    *m=*n;
    *n=t;
}
int a=2,b=3;
swap(&a,&b);
//结果 a=3,b=2
```

数组的传入是**一对共享同一数据区**的数组，即可以传入后改变数组中的元素。

函数中变量的传出

数组

如果在函数中定义了一个数组，**无法传出**

```
char *fun(){
    char s[50];
    ...
    return s;//返回值是空的 最后的返回出来s为s的首地址，没有返回数组的内容
}
```

改进方法

- 1.使用malloc函数，这样它的空间不会被释放，可以传出，只有在**free**时才会释放空间
- 2.传入一个数组，作为保存的地方

struct 结构体定义

没有名字的结构体定义只能使用一次

自己不能嵌套自己，只能嵌套自己的指针

```
struct //少了结构名
{
    int num;
    float age;
} student;
struct student std1;//错误样例
```

strcat的原型函数

```
void strcat(char *s, char *t)
{
    int n;
    n=strlen(s);
    while( *(s+n)= *t )//其实隐含了一个判断 *t!= '\0'
    {
        s++;
        t++;
    }
}
```

结构体的元素获取

```
struct student
{
    int age;
    int num;
}std, *p;
p = &std;
对于非指针, 用 '.' 运算符: std.age
对于指针, 可以p->age或者(*p).age
```

计算结构体的元素数目

```
#define Cnt (sizeof(struct xxx[])/sizeof(struct xxx))
```

结构体数组的qsort函数

```
int cmp(const void *p, const void *q)
{
    struct xxx *m=(struct xxx *)p;
    struct xxx *n=(struct xxx *)q;
    if(m->element1<n->element1)
        return -1;
    if(m->element1>n->element1)
        return 1;
}
```

!!! ++ * ->的优先级和判断执行顺序

- *和++具有相同的运算优先级, 但是单目运算符, 按照从右向左的顺序
- ->的优先级高于*和++的优先级
- 注意后++和前++, 后++的值为改变之前的值

操作符的结合性

- 等号: 从右向左
- 后++: 整个语句执行完, 再++
- 前++: 先++, 再执行语句中的其他操作 //注意和* ++的优先级区别

p++和(p++)相同, 先取指针p指向的值,再将指针p自增1;

(*p)++, 先取指针p指向的值, 再将该值自增1(数组的第一个元素自增1);

*++p, 先将指针p自增1(此时指向数组的第二个元素), 操作再取出该值;

++*p, 先取出指针p指向的值, 再将值自增1, 返回值是自增之后的值;

++(*p), 先将指针指向的值自增1, 并取出改变后的值;

一般情况下, 结构只对其成员引用, 但必须给出它的“全称”(全路径)。在表示结构成员引用时, 要特别注意运算符的优先级关系。如下例:

```
struct {
    int x;
    int *y;
} *p;
```

说明下面表达式的含义:

++p->x;	对成员x增量
++(p->x);	同上
(++p)->x;	先对p增量, 再取成员x
(p++)->x;	先取成员x, 再对p增量
p++->x;	同上
*p->y;	取成员y所指对象内容
*p->y++;	先取成员y所指对象后, 对y增量
(*p->y)++;	对y所指对象增量
*p++->y;	取成员y所指对象内容后, 再对p增量

注意: 在C语言中, 初等量运算符 ((), [], *, ->) 高于单目运算符 (*, ++, --, ...)。

例: 给出下面程序的运行结果。

```
main()
{
    #include <stdio.h>
    struct {
        int x;
        char *y;
    } *p, A[ ] = {
        1, "for",
        2, "while",
        3, "do_while",
        4, "switch"
    };
    p = A;
    printf("%d", ++p->x);
    printf("%d", ++(p->x));
    printf("%d", (p++)->x);
    printf("%d", p++->x);
    printf("%c", *p->y);
    printf("%c", *p->y++);
    printf("%c", *(p->y++));
    printf("%c", *p++->y);
    for(i=0; i<4; i++)
        printf("\n%d, %s", A[i].x, A[i].y);
}
```

初始

1	→	"for"
2	→	"while"
3	→	"do_while"
4	→	"switch"

结果:

```
2 3 3 2 d d o _
3, for
2, while
3, _while
4, switch
```

Homework 3

数据结构基础

指针的使用

指针可以比大小，可以减（相减结果为指针所差的元素个数），但不可以相加

任何使用指针变量之前一定要给它开辟一个空间 `malloc` 或者置空 `NULL` 或者有所指向

对于一个指针，要先有指向，才能继续赋值

```
int x;
int *px;
*px=x; //这个是常见错误，因为这个px没有指向，是野指针！！
```

为字符串开辟空间

```
p=(char *)malloc(strlen(s)+1); //一定要加1，为'\0'留位置
```

遍历一个数组

```
for(pi=&a[0],pj=&a[N-1];pi<=pj;pi++){
}
```

链表的操作

其他代码或者算法

逆波兰表达式的计算顺序

中缀转后缀

规则：从左至右遍历中缀表达式中每个数字和符号：

- 若是数字直接输出，即成为后缀表达式的一部分；
若是符号：
- 若是`)`，则将栈中元素弹出并输出，直到遇到`(`，`(`弹出但不输出（左、右括号都不输出，右括号不如入栈）；
- 若是`(`，则直接入栈；
- 若是`+, -, *, \`等符号，则从栈中弹出并输出优先级不低于(高于或等于)当前的符号(不包括左括号`(`)，直到遇到一个优先级低的符号；然后将当前符号压入栈中。
(优先级`+`，`-`最低，`*`，`/`次之，`(`最高)
- 遍历结束，将栈中所有元素依次弹出，直到栈为空。

逆波兰表达式作用：

1. 消除括号
2. 将嵌入在表达式各处的无序优先级关系转换为从左到右的顺序形式

后缀算值

规则：从左至右遍历后缀表达式中每个数字和符号：

- 若是数字直接进栈；
- 若是运算符 (+, -, *, /) ，则从栈中弹出两个元素进行计算（注意：后弹出的是左运算数），并将计算结果进栈。
- 遍历结束，将计算结果从栈中弹出（栈中应只有一个元素，否则表达式有错）。

枚举(enum)

相当于给每一个元素赋值 给一个字符串赋予一个值

```
enum color {red, yellow, green, black, white, grey};  
###结果 red=0,yellow=1,green=2,black=3,white=4,grey=5;
```

对齐输出

左对齐：%-[[lenth]][type] 注意有负号

右对齐：%[[lenth]][type]

动态开辟二维数组空间

```
char **lines = (char **)malloc(sizeof(char *)*n);  
for (i = 0; i < n; i++) {  
    lines[i] = (char *)malloc(sizeof(char) * 100);  
} //开辟二维数组空间
```

显示文件的最后n行

读入命令，简便版本和复杂版本

实现操作

可以通过循环链表或者数组实现

数组版本

```
int main() {  
    FILE *in, *out;  
    int linecnt = 0, n, i; //记录文件行数和要输入的行数  
    char line[100]; //每次读入一行  
    scanf("%d", &n);  
    char **lines = (char **)malloc(sizeof(char *)*n);  
    for (i = 0; i < n; i++) {  
        lines[i] = (char *)malloc(sizeof(char) * 100);  
    } //开辟二维数组空间  
    in = fopen("in.txt", "r");  
    out = fopen("out.txt", "w");  
    i = 0;  
    while (fgets(line, 100, in) != NULL) {  
        linecnt++;  
        strcpy(lines[i], line);  
        i = (i + 1) % n; //循环的过程  
    }  
}
```

```

int start = i; //i++了之后到达最后一个位置的下一位，即回到了最后n行的开头
if (linecnt < n) { //防止要输出的行数多于文件本身的行数
    for (i = 0; i < linecnt; i++) {
        fputs(lines[i], out);
    }
} else {
    for (i = 0; i < n; i++) {
        fputs(lines[start], out);
        start = (start + 1) % n;
    }
}
fclose(in);
fclose(out);
}

```

循环链表版本

```

int main(){
    //创建循环链表
    if((fp = fopen(filename, "r")) == NULL){
        printf(" Can't open file: %s !\n", filename);
        return (-1);
    }
    first = ptr = (struct Node *)malloc(sizeof ( struct Node));
    first->line = NULL; // 一定要有置空操作，为了对付没有填满的情况
    for(i=1; i<n; i++){
        ptr->next = (struct Node *)malloc(sizeof ( struct Node));
        ptr = ptr->next;
        ptr->line = NULL;
    }
    ptr->next = first;
    ptr = first;

    while(fgets(curline, MAXLEN, fp) != NULL){
        if(ptr->line != NULL) /*链表已经满了，需要释放掉不需要的行*/
            free(ptr->line);
        ptr->line = (char *) malloc ( strlen(curline)+1);
        strcpy(ptr->line, curline);
        ptr = ptr->next;
    }
    for(i=0; i<n; i++) {
        if(ptr->line != NULL) //防止要输出的行数多于文件本身的行数
            printf("%s", ptr->line);
        ptr = ptr->next;
    }
    fclose(fp);
    return 0;
}

```

折半查找

有序数字的折半查找

找到返回位置，没找到返回-1

```
//以递增数组为例
int bsearch(int item, int array[], int len)
{
    int low=0, high=len-1, mid;
    while(low <= high){
        mid = (high + low) / 2;
        if(( item < array[mid])
            high = mid - 1;
        else if ( item > array[mid])
            low = mid + 1;
        else
            return (mid);
    }
    return -1;
}
```

有序字符串的折半查找

找到返回位置或者字符串计数值加一，否则插入字符串

对于字符串的排序还可以是先插入再排序

Bubble sort

```
void sortinform(inform *p, int n) {
    int i, j;
    inform tmp;
    for (i = 1; i < n; i++) {
        for (j = 0; j < n - i ; j++) {
            if (strcmp(p[j].name, p[j + 1].name) > 0) {
                tmp = p[j];
                p[j] = p[j + 1];
                p[j + 1] = tmp;
            }
        }
    }
}
```

简单选择排序

```
void sortByname(struct Student array[], int n)
{
    int i, j;
    struct Student tmp;
    for(i=0; i<n; i++)
        for(j=i; j<n; j++){
            if(strcmp(array[i].name,array[j].name)>0){
                tmp = array[i];
                array[i] = array[j];
                array[j] = tmp;
            }
        }
}
```

```

void sortByScore(struct Student array[], int n)
{
    int i, j;
    struct Student tmp;
    for(i=0; i<n; i++)
        for(j=i; j<n; j++){
            if(array[i].score < array[j].score){
                tmp = array[i];
                array[i] = array[j];
                array[j] = tmp;
            }
        }
}

```

qsort

```

struct students{
    char *s;
    int score;
}info[n];
int cmp(const void *p,const void *q)
{
    struct students *m,*n;
    m=(struct students *)p;
    n=(struct students *)q;
    if(m->score<n->score){
        return -1;
    }
    else if(m->score>n->score){
        return 1;
    }
    else{
        if(strcmp(m->s,n->s)<0)
            rerurn -1;
        if(strcmp(m->s,n->s)>0)
            return 1;
    }
}

}
qsort(info,n,cmp);

```