最后的挣扎

结构体初始化

定义

```
struct InitMember
{
    int first;
    double second;
    char* third;
    float four;
};
```

方法一：定义时赋值

```
struct InitMember test = {-10,3.141590, "method one", 0.25};
```

方法二：定义后逐个赋值

```
struct InitMember test;

test.first = -10;
test.second = 3.141590;
test.third = "method two";
test.four = 0.25;
```

方法三：定义时乱序赋值（C++风格）

```
struct InitMember test = {
    second: 3.141590,
    third: "method three",
    first: -10,
    four: 0.25
};
```

方法四：构造函数

```
//定义图的定点
typedef struct Vertex {
    int id,inDegree,outDegree;
    vector<int> connectors;    //存储节点的后续连接顶点编号
    Vertex() : id(-1),inDegree(0),outDegree(0) {}
    Vertex(int nid) : id(nid),inDegree(0),outDegree(0) {}
} Vertex;

//定义Graph的邻接表表示
typedef struct Graph {
    vector<Vertex> vertexs;   //存储定点信息
    int nVertexs;             //计数：邻接数
    bool isDAG;               //标志：是有向图吗

    Graph(int n, bool isDAG) : nVertexs(n), isDAG(isDAG) {
vertexs.resize(n); }
    Graph() : nVertexs(1), isDAG(1) { vertexs.resize(1); }
    //向图中添加边
    bool addEdge(int id1, int id2) {
            ...
            ...
            ...
        return true;
```

```
    }
} Graph;

Graph g(8, false);
```

c++new的使用

常规

```
int *x = new int;          //开辟一个存放整数的存储空间，返回一个指向该存储空间的地址(即指针)
int *a = new int(100);   //开辟一个存放整数的空间，并指定该整数的初值为100，返回一个指向该存储空间的地址
char *b = new char[10]; //开辟一个存放字符数组(包括10个元素)的空间，返回首元素的地址
float *p=new float (3.14159);//开辟一个存放单精度数的空间，并指定该实数的初值为//3.14159，将返回的该空间的地址赋给指针变量p
```

动态申请列大小固定的二维数组

```
//列值固定
const int MAXCOL = 3;
cin>>row;
//申请一维数据并将其转成二维数组指针
int *pp_arr = new int[nRow * MAXCOL];
int (*p)[MAXCOL] = (int(*)[MAXCOL])pp_arr;
//此时p[i][j]就可正常使用
```

动态申请大小不固定的二维数组

```

cin>>row>>col;
int **p = new int*[row];
for (int i = 0; i < row; i ++)
{
    p[i] = new int[col];
}
```

找出u到v的所有路径-邻接表

```c
#include<stdio.h>
#include<stdlib.h>

#ifndef BASE
#define BASE
#define TRUE 1
#define FALSE 0
#define OK 1
#define ERROR 0
#define INFEASIBLE -1
#define OVERFLOW -2
typedef int Status;
typedef int bool;
#endif

#define VertexType char //点类型
#define VRType int //边类型
#define maxSize 100
void Visit(VertexType e) {
    printf("%c", e);
}

#define MAX_VERTEX_NUM 20
typedef enum{DG, UDG} GraphKind;
typedef struct ArcNode{
    int adjV; //边指向的顶点
    VRType weight; //权重
    struct ArcNode *next;
}ArcNode; //边
typedef struct VNode{
    VertexType data;
    ArcNode *firstarc;
}VNode, AdjList[MAX_VERTEX_NUM]; //顶点
typedef struct{
    GraphKind kind;
    int vernum,arcnum;
    AdjList vers;
}ALGraph;


/*------------------------
 |7.14 创建有向图的邻接表|
```

```
 ------------------------*/
Status InitGraph_AL(ALGraph *pG) { //初始化
    int i;
    pG->arcnum = 0;
    pG->vernum = 0;
    for (i=0; i<MAX_VERTEX_NUM; ++i)
        pG->vers[i].firstarc = NULL; //VC++6.0中指针初始化为
0xcccccccc
    return OK;
}
int LocateVex_AL(ALGraph G, VertexType e) { //定位值为e的元素下标
    int i;
    for (i=0; i<G.vernum; ++i) {
        if (G.vers[i].data == e) {
            return i;
        }
    }
    return -1;
}
Status CreateDG_AL(ALGraph *pG) { //创建有向图的邻接表
    //输入规则：顶点数目->弧的数目->各顶点的信息->各条弧的信息
    int i,a,b;
    char tmp[MAX_VERTEX_NUM];
    char h,t;
    ArcNode *p, *q;

    InitGraph_AL(pG); //VC++6.0中指针初始化为0xcccccccc，如果不将指针初
始化为NULL，会出错
    //图的类型
    pG->kind = DG;
    //顶点数目
    scanf("%d", &i); if (i<0) return ERROR;
    pG->vernum = i;
    //弧的数目
    scanf("%d", &i); if (i<0) return ERROR;
    pG->arcnum = i;
    //各顶点信息
    scanf("%s", tmp);
    for (i=0; i<pG->vernum; ++i) pG->vers[i].data=tmp[i];
    //弧的信息
    for (i=0; i<pG->arcnum; ++i) {
        scanf("%s", tmp);
```

```
        h = tmp[0]; t = tmp[2];
        a = LocateVex_AL(*pG, h);
        b = LocateVex_AL(*pG, t);
        if (a<0 || b<0) return ERROR;
        p = (ArcNode *)malloc(sizeof(ArcNode)); if (!p)
exit(OVERFLOW);
        p->adjV=b;p->next=NULL;
        if (pG->vers[a].firstarc) { //已经有边了
            for (q = pG->vers[a].firstarc; q->next; q=q->next) ; //
找到最后一条
            q->next = p;
        } else { //第一条边
            pG->vers[a].firstarc = p;
        }
    }
    return OK;
}

/*-----------------------------------------------------------------
 |7.28 有向图-从u-v的所有简单路径                                  |
 -----------------------------------------------------------------*/
int visit[MAX_VERTEX_NUM]; //前面定义了
VertexType paths[maxSize][MAX_VERTEX_NUM]; //存放路径
int path[MAX_VERTEX_NUM]; //路径
int pathnum=0; //当前是第几条路径
void FindAllPath(ALGraph G, int u,int v,int k) { //u->v当前是第k个位
置
    int i;
    ArcNode *p;
    visit[u]=1; //走到了u
    path[k]=u; //添加到路径->下标位置为k的结点是u（第k+1个是u）
    if (u==v) { //找到了
        for (i=0; i<=k; i++) {//复制到paths
            paths[pathnum][i] = G.vers[path[i]].data;
        }
        paths[pathnum][i]='\0'; //结束符
        pathnum++; //找下一条路径
    } else {
        //u的邻边开始找
        for (p=G.vers[u].firstarc; p; p=p->next) {
            if (visit[p->adjV]==0)
                FindAllPath(G, p->adjV, v, k+1); //去这个邻接点找
```

```
        }
    }
    // 回溯到上一个结点
    // 注意：回溯应该写在外面-->也就是不管有没有找到都要回溯
    visit[u]=0;
    path[k]=0;
}


int main() {
/*7.28
6
11
ABCDEF
B,A
B,D
C,B
C,F
D,C
D,E
D,F
E,A
F,A
F,B
F,E
B->A
A->B
D->A
*/
    int i,j;
    int cnt;
    ALGraph G;
    char tmp[20];

    CreateDG_AL(&G);

    while (1) {
        scanf("%s", tmp); //A->B
        i = LocateVex_AL(G, tmp[0]);
        j = LocateVex_AL(G, tmp[3]);
        for (cnt=0; cnt<MAX_VERTEX_NUM; cnt++) visit[cnt]=0;
        pathnum=0;
```

```
        printf("7.28 输出所有 %c 到 %c 的路径\n", tmp[0], tmp[3]);
        FindAllPath(G, i, j, 0);
        if (pathnum==0) {
            printf("\t- 走不通\n");
        }
        for (i=0; i<pathnum; i++) {
            printf("\t%d %s\n", i+1, paths[i]);
        }
    }
    return 0;
}
```

CCF 编译出错原因： 不允许C++STL容器嵌套（需要满足相应的格式）

就是要在后面的“>”之间，必须得有一个空格，如果有多层，那每层都得有一个空格。

```
map<string,list<string> > user;
```

STL底层说明

C++ STL 的实现：

```
1.vector       底层数据结构为数组 ，支持快速随机访问

2.list              底层数据结构为双向链表，支持快速增删

3.deque        底层数据结构为一个中央控制器和多个缓冲区，详细见STL源码剖析P146，支持首尾（中间不能）快速增删，也支持随机访问
deque是一个双端队列(double-ended queue)，也是在堆中保存内容的.它的保存形式如下：
[堆1] --> [堆2] -->[堆3] --> ...
每个堆保存好几个元素,然后堆和堆之间有指针指向,看起来像是list和vector的结合品.

4.stack          底层一般用list或deque实现，封闭头部即可，不用vector的原因应该是容量大小有限制，扩容耗时

5.queue       底层一般用list或deque实现，封闭头部即可，不用vector的原因应该是容量大小有限制，扩容耗时
```

（stack和queue其实是适配器,而不叫容器，因为是对容器的再封装）

6.priority_queue　　的底层数据结构一般为vector为底层容器，堆heap为处理规则来管理底层容器实现

7.set　　底层数据结构为红黑树，有序，不重复

8.multiset　　底层数据结构为红黑树，有序，可重复

9.map　　底层数据结构为红黑树，有序，不重复

10.multimap　　底层数据结构为红黑树，有序，可重复

11.hash_set　　底层数据结构为hash表，无序，不重复

12.hash_multiset 底层数据结构为hash表，无序，可重复

13.hash_map　　底层数据结构为hash表，无序，不重复

14.hash_multimap 底层数据结构为hash表，无序，可重复