结构体和联合体

结构体：是一种自定义的数据类型，使不同类型的变量组合成一个整体。

结构体的声明：

```
struct 结构体名{
    数据类型 成员名;
    数据类型 成员名;
    ......
}结构体变量名;
```

使用结构体时，应先声明。比如：

```
struct data{
    int x;
    double y;
    char z;
};
```

之后可以定义结构体变量。

```
struct data a;
```

就定义了一个叫a的data类型的结构体变量，a中含有一个int型变量，一个double型变量和一个char型变量。

结构体变量的声明要在主函数前或主函数中，不能在主函数之后。

结构体不能进行强制类型转换，结构体指针可以。

结构体中的变量在存储时互不影响，所有的成员同时存在。

各个成员按照被声明的顺序在内存中存储，第一个成员的地址是整个结构体的地址。

结构体的内存长度：

1.结构体中成员是按照声明顺序一个一个放到内存中去的，但并不是紧密排列的。从结构体存储的首地址开始，每一个成员放置到内存中时，它都会认为内存是按照它自己的大小来划分的。因此每个成员放置的位置是自己宽度的整数倍（以结构体变量首地址为0计算）。

2.满足1的条件后，检查计算出的内存长度是否为结构体中内存最大的成员内存的整数倍。是，则结束；若不是，则补齐为它的整数倍。

下面来举两个例子：

```
struct data
{
    char x;//0
    double y;//8-15
    int z;//16-19
    //20size不是8的倍数，补齐为24
}a;//内存为24
```

```
struct data
{
    int x;//0-3
    char y;//4
    double z;//8-15
}a;//内存为16
```

其他的情况：

1.结构体中含有指针变量，指针变量所占内存为4size（32位编译器）或8size（64位编译器）。

2.伪指令#pragma pack (n)，编译器按照n个字节对齐。#pragma pack ()取消自定义对齐方式。
每个数据成员的对齐按照 #pragma pack指定的数值 和 这个数据成员自身长度 中比较小的那个进行。
结构体整体对齐将按照 #pragma pack指定的数值 和 结构体最大数据成员长度 中比较小的那个进行。

默认对齐值：
Linux 默认#pragma pack(4)
window 默认#pragma pack(8)

推论：如果想让结构体的大小等于结构体中各个变量大小的和，那么只需要让对齐当长度等于1即可。

3.如果一个结构体里有结构体成员，则结构体成员要从其内部最大成员大小的整数倍地址开始存储。

下面是一个复杂点的例子：

```
struct EE
{
    int a;      //长度4 < 8 按4对齐；偏移量为0；存放位置区间[0,3]
    char b;     //长度1 < 8 按1对齐；偏移量为4；存放位置区间[4]
    short c;    //长度2 < 8 按2对齐；偏移量由5提升到6；存放位置区间[6,7]
    //结构体内部最大元素为int,由于偏移量为8刚好是4的整数倍，所以从8开始存放接下来的struct FF
    struct FF
    {
        int a1;     //长度4 < 8 按4对齐；偏移量为8；存放位置区间[8,11]
        char b1;    //长度1 < 8 按1对齐；偏移量为12；存放位置区间[12]
        short c1;   //长度2 < 8 按2对齐；偏移量为13,提升到2的倍数14；存放位置区间[14,15]
        char d1;    //长度1 < 8 按1对齐；偏移量为16；存放位置区间[16]
    }f;
    //整体对齐系数 = min((max(int,short,char), 8) = 4，将内存大小由17补齐到4的整数倍20
    char d;         //长度1 < 8 按1对齐；偏移量为21；存放位置区间[21]
    //整体对齐系数 = min((max(int,short,char), 8) = 4，将内存大小由21补齐到4的整数倍24
}e;
```

```
struct EE
{
    short a;//0-1
    char b;//2
    short c;//4-5

    struct FF
    {
        int a1;//8-11
        char b1;//12
    }e;

    char d;//16
    //补到20
}f;
```

再介绍一下malloc函数的应用：

malloc动态内存分配函数，用于申请一块连续的指定大小的内存块区域，以void*类型返回分配内存区域的地址。

函数原型：

```
extern void* malloc(unsigned int num_bytes);
```

分配长度为num_bytes字节的内存块。

头文件为#include<malloc.h>

分配成功则返回指向被分配内存的指针，否则返回空指针NULL。

因为返回值为无类型指针，使用时要强制转换为需要的类型。指针=（需要的类型*）malloc（sizeof申请的空间大小）

```
int *p=NULL;
p=(int*)malloc(sizeof(int)*10);
```

使用malloc函数开辟空间时，系统会在空间前做一个标记。（原本是0，标记后是1）malloc函数遇到标记为0的空间就会在此开辟。若标记为1说明此空间正在被使用，这样就不会重复利用空间。

使用完成后要释放空间，否则会造成内存泄露。

```
int *p=NULL;
p=(int*)malloc(sizeof(int)*10);
free(p);
p=NULL;
```

使用free函数可以释放malloc函数给指针变量分配的内存。释放后要将指针重新指向NULL，避免出现野指针。

结构体变量的初始化

对结构体变量初始化时，要对结构体中成员一一赋值。不能跳过前面的变量直接给后面赋值，但可以只赋值前面的部分。

对于未赋值的变量，数值型自动赋值为0，字符型自动赋值为/0。

介绍几种赋值方法：

1.定义时直接赋值。

```
struct Student
{
 char name[20];
 char sex;
 int number;
}stu1={"wushiyuan",'W',1};

struct Student
{
 char name[20];
 char sex;
 int number;
```

```
};
struct Student stu1={"wushiyuan",'W',1};
```

2.先定义结构体再赋值。

```
struct Student
{
 char name[20];
 char sex;
 int number;
}stu1;

stu1.name="wushiyuan";
stu1.sex='W';
stu1.number=1;

//也可用strcpy函数进行赋值
strcpy(stu1.name,"wushiyuan");
```

定义之后可以对任意变量赋值，不一定按照声明中的顺序，也不一定要都赋值。

```
struct Student
{
 char name[20];
 char sex;
 int number;
};

struct Student stu1={
  .name="wushiyuan",
  .number=1,
  .sex='W',
 };
```

如果定义结构体变量的时候没有初始化，之后就不能全放在一起初始化了。

结构体变量的访问

.是运算符，用来访问结构体中的变量。在所有运算符中优先级最高。

如果访问的结构体中的变量也是一个结构体，就需要继续用.访问直到最低一级的变量。

举一个简单的例子：

```
#include <stdio.h>

struct A
{
    int x;

    struct B{
        int y;
        char z;
    }w;

}a;
```

```
int main()
{
    a.w.y=1;
    a.w.z='x';
    printf("%d\n%d\n%c\n",a.x,a.w.y,a.w.z);

    return 0;
}
//输出的结果为0 1 x
```

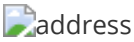
也可以访问结构体变量和结构体变量中成员的地址，举例说明：

```
struct A
{
    int x;

    struct B{
        int y;
        char z;
        double u;
    }w;

}a;
```

对于这个结构体，我们可以访问a的地址，也可以访问a.w.y的地址，还可以访问a.w的地址。

这里要注意，可以访问a.w的地址，但是不能访问a.w。

```
printf("%d\n%d\n%d\n%d\n%d\n%d\n",&a,&a.x,&a.w,&a.w.y,&a.w.z,&a.w.u);
```



结构体变量的地址就是结构体变量中第一个变量的地址。

“为什么a.w的地址是4215864而不是4215860？”

*x*存放位置区间为*[0,3]*，下一个变量应从*4*开始存。但结构体变量*w*中，内存最大的*u*是*double*类型变量，占据*8size*。

所以*w*中第一个变量应从*8*开始存，也就是*4215864*。

结构体数组：具有相同类型的结构体变量组成的数组就是结构体数组。

结构体数组的定义：

```
struct Student
{
 char name[20];
 char sex;
 int number;
}stu[10];
```

结构体数组在定义时可以直接初始化，但定义后就不能一起赋值了。

```
struct Student
{
 char name[20];
```

```
 char sex;
 int number;
}stu[2]={
    {"wushiyuan",'W',1},
    {"liushiyuan",'M',2}
};//是正确的

struct Student
{
 char name[20];
 char sex;
 int number;
}stu[2];

stu[2]={
    {"wushiyuan",'W',1},
    {"liushiyuan",'M',2}
};//是错误的
```

指向结构体变量的指针

定义的格式为：struct 结构体名* 指针名

```
struct Student
{
 char name[20];
 char sex;
 int number;
}stu;

struct student* p=&stu;
//p指向stu这个结构体变量的地址
```

```
struct Student
{
 char name[20];
 char sex;
 int number;
}stu[10];

struct student* p=stu;
//p指向stu这个结构体数组的首地址，也就是stu[0]的地址
```

用结构体指针变量访问结构体变量中的成员，有两种方式：

1.(*p).number

2.p->number

```
struct student stu[10];
struct student* p=stu;//p指向stu[0]的地址
(++p)->number;//访问的是stu[1]的number
```

一种节省内存的方式：

```
struct Student
{
 char* name;//可以防止名字长度不一样造成内存浪费
 char sex;
 int number;
};
```

一些补充：

1.同类型的结构体可以直接互换值

```
struct student stu1=...;
struct student stu2=...;

struct student stu3;
stu3=stu1;
stu1=stu2;
stu2=stu3;
```

2.用memset函数清空结构体

```
struct Student
{
 char name[20];
 char sex;
 int number;
}stu;

memset(&stu,0,sizeof(struct student));
```

```
struct Student
{
 char name[20];
 char sex;
 int number;
}stu[10];

memset(stu,0,sizeof(struct student)*10);
```

联合体：是一种自定义的数据类型，使不同类型的变量共同占有一段内存。

联合体的声明：

```
union 联合体名{
    数据类型 成员名;
    数据类型 成员名;
    ......
}联合体变量名;
```

使用联合体时，应先声明。比如：

```
union data{
    int x;
    double y;
    char z;
};
```

之后可以定义联合体变量。

```
union data a;
```

就定义了一个叫a的data类型的联合体变量，a中含有一个int型变量，一个double型变量和一个char型变量。

一个联合体的内存长度是联合体中内存最大的变量的长度。

联合体中的变量在存储时是相互覆盖的，同一时刻只能存放一个，最后一次存放的变量起作用。

对一个联合体赋值的时候会发生什么？对其中已经赋值的变量重写。

```
#include <stdio.h>

union data
{
    struct node
    {
        int x;
        int y;
        int z;
    }u;//这个结构体的内存长度为4×3=12size
    int k;
}a;//这个联合体的内存长度为其中内存最大的变量u的内存12size

int main()
{
    a.u.x=1;
    a.u.y=2;
    a.u.z=3;//xyz在内存中按声明顺序从低到高排序，赋值时x的位置为1，y的位置为2，z的位置为3
    a.k=0;//对k赋值时，从union的首地址开始存放，也就是x的位置
    //这样x位置上原来的数值就被k所赋的值替代了，而y和z位置的值没有被改变
    printf("%d\n%d\n%d\n",a.u.x,a.u.y,a.u.z);//输出结果为0 2 3

    return 0;
}
```

联合体中变量的访问：

```
struct data
{
    int x;
    char *y;

    union node
    {
        int i;
        char *j;
    }b;

}a[10];
```

访问结构体变量a[1]中联合b中的成员i，写作a[1].b.i

访问结构体变量a[2]中联合b中的字符串指针j所指的第一个字符，写作*a[2].b.j（注意不能写作a[2].b.*j）

结构体的应用

上学期的一道题：多关键字排序

题目描述

给定 n 个物品，从 1–n 编号，每个物品有 k 个权值（即关键字），现在请你对这 n 个物品排序。

排序规则如下：

- 第一关键字升序排列，第一关键字相同时第二关键字降序排列，第二关键字相同时则第三关键字升序排列，以此类推。（升降序交替）
- 如果 k 个关键字都相同，则按照编号升序排列。

输入格式

第一行，两个正整数n，k（n≤1000,k≤10）。

接下来 n 行，每行 k 个整数，第 i 行第 j 列的数 wij 表示 i 号物品的第 j 个权值，保证其在`int`范围内。

输出格式

一行共n个数，表示按规定排序之后物品的编号顺序，两数之间用一个空格隔开。

参考代码

```
//摸了，进行一个粘代码
#include<stdio.h>
#include<stdlib.h>

int n,k;

struct node{
    int id;
    int w[11];
}a[1010];

int cmp(const void* a,const void* b){
    struct node c=*(struct node*)a;
    struct node d=*(struct node*)b;
    int i;
```

```
    for(i=0;i<k;i++){
        if(c.w[i]==d.w[i]){
            continue;
        }
        else return i%2?(d.w[i]>c.w[i]?1:-1):(d.w[i]>c.w[i]?-1:1);
        return c.id>d.id?1:-1;
    }
}

int main(){
    scanf("%d%d",&n,&k);
    int i,j;
    for(i=0;i<n;i++){
        a[i].id=i+1;
        for(j=0;j<k;j++){
            scanf("%d",&a[i].w[j]);
        }
    }
    qsort(a,n,sizeof(struct node),cmp);
    for(i=0;i<n;i++){
        printf("%d ",a[i].id);
    }
    return 0;
}
```