C语言 宝盖版

上机前：基础部分

- ```c
  #include<stdio.h>
  int  main()
  {
      int  a[]={5,8,7,6,2,7,3};
      int y,*p=&a[1];
      y=(*--p)++;
      printf("%d ",y);
      printf("%d",a[0]);
      return 0;
  } //输出5 6
  ```

- ```c
  int i = 5;
  int j = ++i;  // 前置自增
  ```

  - **结果**：`i = 6`, `j = 6`

  ```c
  int i = 5;
  int j = i++;  // 后置自增
  ```

  - **结果**：`i = 6`, `j = 5`
- **自增在循环中**
  - `for (initialization; condition; increment) { // loop body  }`
    - `increment` 部分在循环体执行完之后执行。`for (; i <= j; ++i, --j)` 和 `for (; i <= j; i++, j--)` 的效果是相同的，最终结果也相同。
    - `condition`中`i++ < 5`：在每次循环迭代之前执行。
      - `for (i = 0; i++ < 5; ) { printf("i = %d\n", i); }`输出12345。i++,++i就不一样了。
  - `while`中就和`condition`中一样。
    - ```c
      char s[] = "language";
      char *p = s;while (*p++ != 'u') {
         printf("%c", *p - 'a' + 'A');
       }//ANGU
      ```
  - `if(i++ ==i)`->0.
- **静态，全局，局部**
      ```

```c
int ff(int n) {
    static int f = 1;  // 静态变量 f，初始值为 1
    f = f * n;         // f 乘以 n，更新 f 的值
    return f;          // 返回 f 的值
}
void main() {
    int i;
    for(i = 1; i <= 5; i++) {
        printf("%d\n", ff(i));  // 调用 ff(i)，并打印返回值
    }
}
//尽管static int f = 1;但每次f不变，最后一次120
```

```c
void f(int y, int *x) {
    y = y + *x;  // 这里 y = y + *x
    *x = *x + y;  // 这里 *x = *x + y
}
int main() {
    int x = 2, y = 4;
    f(y, &x);
    printf("%d %d\n", x, y);
    return 0;
}
//8,4
```

  - 动态全局变量的作用域是整个程序文件，从声明开始到文件结束。如果使用 `extern` 关键字声明，可以在其他文件中访问。
  - 静态全局变量的作用域是声明它的文件，从声明开始到文件结束。不能在其他文件中访问，即使使用 `extern` 关键字也无法访问。
  - 动态局部变量的作用域是声明它的函数或代码块，从声明开始到函数或代码块结束。

- **C 语言合法常量**
  - 八进制数必须以数字0开头，仅包含0到7之间的数字
    - `int i=010;` `i` 被初始化为 `010`，这是一个八进制数，等于十进制的 `8`。
  - 十六进制数必须以 `0x` 或 `0X` 开头
  - 二进制 0b
  - `e` 是用于表示科学记数法的标志，它表示 10 的幂。正确的科学记数法应当是 `m * 10^n`，其中 `m` 是一个数字，`n` 是整数（例如：`1.0e2` 表示 100）(n!=1.0)
  - `printf("%o\n", x << 1);` 中使用了 `%o` 格式说明符，它用于打印**八进制**数值。
  - 八进制转义字符以反斜杠 `\` 开头，后跟一到三位八进制数字（0-7）。输出字符。`\101`
- 变量名：字母，下划线在开头
  - `scanf`可以作为变量；`case`不能作为变量。
- **sizeof**
  - `x=0;y=sizeof(x++);`之后x=0。sizeof不是函数是运算符。
  - 对字符串，len+1
- `x = -1`，由于 `x` 是非零的（负数也是非零），条件 `if (x)` 为 `true`。
- 逗号运算符（`,`）用于在一个表达式中顺序执行多个子表达式，并返回最后一个子表达式的值。

- **运算符优先级表**

优先级	运算符	描述	结合性
1	`()` `[]` `->` `.`	括号、数组下标、结构体指针成员、结构体成员	左到右
2	`++` `--`	后缀自增、自减	左到右
3	`++` `--` `+` `-` `!` `~` `*` `&` `sizeof`	前缀自增、自减、正负号、逻辑非、按位取反、指针、取地址、大小	右到左
4	`*` `/` `%`	乘、除、取余	左到右
5	`+` `-`	加、减	左到右
6	`<<` `>>`	左移、右移	左到右
7	`<` `<=` `>` `>=`	小于、小于等于、大于、大于等于	左到右
8	`==` `!=`	等于、不等于	左到右
9	`&`	按位与	左到右
10	`^`	按位异或	左到右
11	`|`	按位或	
12	`&&`	逻辑与	左到右

优先级	运算符	描述	结合性
14	`?:`	条件运算符	右到左
15	`=` `+=` `-=` `*=` `/=` `%=` `<<=` `>>=` `&=` `^=` `|=`	赋值及复合赋值运算符	
16	`,`	逗号运算符	左到右

- `m=a>b;` 就是m=0或1。
- 有一些计算（&&，||）中，因为短路，可能不会执行一些句子。
- int n=20,m=15; 执行语句n=(0<m<n<10);后，n的值是1.
- `*q.id==...` 是一个指针。是 `*q.id` 还是 `(*q).id` !!!
- `++p->str` 、 `(++p)->str`
- ```
  #define MAX(x,y) (x)>(y)?(x):(y)
  t=MAX(7,6)*10; //输出7.（：优先级小于*，相当于7,60）
  ```

- `a <<= b` 等价于 `a = a << b`
  - `a` 的初始值是 `5`，二进制表示为 `0000 0101`。`a = a << 1` 将 `a` 左移1位，结果是 `0000 1010`，即 `10`。`a = a << 2` 将 `a` 再左移2位，结果是 `0010 1000`，即 `40`。左移运算可以用于快速乘以2的幂次，例如左移1位相当于乘以2，左移2位相当于乘以4。
  - 设int b=2；表达式(b>>2)/(b>>1)的值是0。(b>>2)=0.
- **字符串函数**
  - `sizeof` 和 `strlen` 的区别！！
  - 字符串 "The" 小于字符串 "the"
  - `strcmp` 比较两个字符串　第一个字符串小于第二个字符串<0
  - `strcpy(dest, src);` 将 src 的内容复制到 dest。确保目标缓冲区有足够的空间来存储复制的内容，以避免缓冲区溢出。
  - `strncpy(ss, s1, n);` 将s1的前n个字符复制到ss中
    - 在字符串s1中的指定位置n处插入字符串s2

    ```
    strncpy(ss, s1, n);
    strcpy(ss + n, s2); // ss是指针，所以要+n
    strcpy(ss + n + len2, s1 + n);
    ```

- # 赋值！！！！！
    ```

- 已知ch是字符型变量，则ch="";是错误的赋值语句。字符型！=字符串型
- ```
  char s[8];s = "asfghjk";  // 错误：不能将字符串常量直接赋值给字符数组
  char s[8];strcpy(s, "asfghjk");  // 使用 strcpy 函数复制字符串
  char *s; s = "asfghjk";  //正确
  char s1[] = "Hello, World!";//正确
  ```
- int *p 中p是指针变量名；同时命名两个指针变量时两个都要加上**
- `char (*p)[10];`
  - `p` 是一个指向包含 10 个 `char` 类型元素的数组的指针。
  - 这是一个指针，指向一个数组，该数组包含 10 个 `char` 类型的元素。
  - ```
    int a[10][20]={1,2,3,4,5};
    as(a,200);
    float as(int (*a)[20], int size){/*...*/} //ok.
    ```
- `char *p[6];`
  - `p` 是一个包含 6 个 `char` 类型指针的数组。
  - 这是一个数组，数组中的每个元素都是一个指向 `char` 类型数据的指针。
- 若有定义`char *p1,*p2,*p3,*p4,ch;`则不能正确赋值的程序语句为`p3=getchar();`因为`p3`没有被初始化，指向一个有效的内存位置，因此解引用 `p3` 会导致未定义行为。`p4=&ch;*p4=getchar();`ok.
- ```
  void soer(int a[], int size);
  void fun(int *p, void(*q)(int a[], int size)); //(*q)是函数指针
  ```
- `char *name; name = malloc(100); scanf("%s", name);` 这段代码中的 `name` 是一个 **指向字符的指针**，而 `name` **并不是字符串本身，而是指向一个字符数组的指针。**
- `char name[100]; scanf("%s", name);` **数组名（`name`）会被隐式转换为指向数组首元素的指针**，即 `name` 就是 `&name[0]`。因此，`scanf("%s", name)` 实际上是传递了 `name` 的首地址（即 `&name[0]`）。
- ```
  int main() {
      const char *str = "hello";  // 字符串常量
      printf("%s\n", str);  // 输出 hello
      // 尝试修改字符串常量（这是非法的，会导致运行时错误）
      // str[0] = 'H';  // 错误：不能修改字符串常量
      return 0;
  }

  int main() {
      char arr[6] = "hello";  // 字符数组
      printf("%s\n", arr);  // 输出 hello
      // 修改字符数组的内容
      arr[0] = 'H';
      printf("%s\n", arr);  // 输出 Hello
      return 0;
  }
  ```
- 命令行参数：在程序运行的时候，通过命令行参数从外部输入数据

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

`argc`： argument count， `argc` 为 `argv` 字符串数组长度

`argv`： argument values， `argv[0]` 一般为文件名， `argv[1] ~ argv[argc-1]` 为输入字符串， 一般以空格分割

例如在命令框中输入 `main.exe bian cheng hao nan`， `argc` 就是5， `argv[0]` 就是main.exe

```
char a[3][3]={"ad","ce","fb"};
char *s=(char *)a;//这里做强制类型转换，原本的a类型为char **，是一个二级指针类型，这里转为一级
```

- 如果for的循环体语句中没有使用continue语句，则以下for语句和while语句等价。

```
for (表达式1; 表达式2; 表达式3)
    for的循环体语句
表达式1;
while (表达式2) {
    for的循环体语句;
    表达式3;
}
```

- 自定义函数

```
int fmax(int i,int s,int q[110][110]){ ...}
fmax(i,s,q);   //对数组，定义时 q[110][110]，调用时q;
```

- **按位运算符**:
  - `^`: 按位异或运算符。只有当两个位不同，结果才为1，否则为0。 `ch = ch ^ 'x';` 和 `ch = ch ^ 120;` 是等价的。
    - 92 (十进制) = 01011100 (二进制)
      92 (十进制) = 01011100 (二进制)
      01011100 ^ 01011100 = 00000000 (二进制) = 0 (十进制)
  - `&`: 按位与运算符
  - `~`: 按位取反运算符
  - `|`: 按位或运算符
-
- for(;;);是正确的.
- 数组说明 int a[3][]={1,2,3,4,5}; 是错误的.
- 静态局部变量如果没有赋值，其存储单元中将是0。
- 以下选项中合法的用户标识符是 `_2Test`
- 指针不能相加，指针相减可以算出两指针间有多少元素，可以赋值和比较相等。对基本类型相同的指针变量不能进行运算的运算符是 `+`

- 对于定义 `char str[] = "abc\000def\n";`，字符串的长度（`len`）是 3，数组的大小（`sizeof`）是 9。`\000` 是八进制里的 `\0`
- `char s[]="Hello World";`可以的。
- ```
  int x=10, y=20, t=0;
  if (x==y)  t=x; x=y; y=t;
  printf("%d,%d",x,y);    //输出20,0
  ```
- ```
  char ch;
  while((ch = getchar()) !='#'){
      putchar(ch);
      ch = getchar();
  }
  //输入123456#，输出135   2个getchar
  ```
转义字符	含义
`\n`	换行符
`\t`	水平制表符（Tab）
`\r`	回车符
`\b`	退格符
`\f`	换页符
`\a`	响铃符
`\\`	反斜杠字符
`\'`	单引号字符
`\"`	双引号字符
`\?`	问号字符
`\0`	空字符（NULL）
`\xhh`	以十六进制表示的字符
`\ooo`	以八进制表示的字符

排序

- ```
  for(int i=0;i<a-1;i++){  //小红花
          for(int j=i+1;j<a;j++){
              if(a1[i]>a1[j]){
                  int q=a1[i];a1[i]=a1[j];a1[j]=q;
              }
          }
      }
  for(int i = 0; i < len; i++){  //冒泡
  ```
  ```

```
        for(int j = 0; j < len - 1 - i; j++){
            if(s[j] > s[j+1]){
                char c = s[j];s[j] = s[j+1];s[j+1] = c;
             }
         }
    // 插入排序函数
void insertionSort(int arr[], int n) {
    for (int i = 1; i < n; i++) {
        int key = arr[i];
        int j = i - 1;
        // 将 arr[i] 插入到已排序的序列 arr[0..i-1] 中
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;
    }
}
```

- 在循环中什么时候跳出循环，临界值有没有改变。很容易错的！！
- **快速排序**
-

```
//将数组分成两部分，左边的元素小于或等于基准元素，右边的元素大于基准元素，并返回基准元素的最终位置。
int partition(int arr[], int low, int high) {
    int pivot = arr[high]; // 选择最后一个元素作为基准
    int i = low - 1; // i 用于指向小于等于 pivot 的元素

    for (int j = low; j < high; j++) {
        if (arr[j] <= pivot) {
            i++; // 增加小于等于 pivot 的元素的指针
            swap(&arr[i], &arr[j]); // 交换元素
        }
    }
    // 最后将 pivot 放到正确的位置
    swap(&arr[i + 1], &arr[high]);
    return i + 1; // 返回基准元素的位置
}
void quickSort(int arr[], int low, int high) {
    if (low < high) {
        int pivotIndex = partition(arr, low, high); // 获取基准元素的位置
        quickSort(arr, low, pivotIndex - 1);  // 对基准左边的部分排序
        quickSort(arr, pivotIndex + 1, high); // 对基准右边的部分排序
    }
}
```

week11. 结构体

- 在定义结构类型时，关键字`struct`和它后面的结构名共同组成一个新的数据类型名。

- 分号！！！
- 结构体类型本身不占用内存空间，结构体变量占用内存空间。 系统分配给它的内存空间大小是各成员所需内存量的总和。
- `.`被称为成员或者分量运算符。`->`被称为指向运算符。
- 可以整体赋值。
- **共用体（union）**
  - 是C语言中的一种数据结构，它允许在同一内存位置存储不同类型的数据。共用体中的所有成员共享同一块内存，因此在任何时刻，共用体只能存储一个成员的值。
  - ```
    //共用体 Data 可以存储一个整数、一个浮点数或一个字符串，但在任何时刻只能存储其中一个。
    union Data {
        int i;
        float f;
        char str[20];
    };
    ```
- **枚举类型定义**：

  `enum weekday {sun, mon=3, tue, wed, thu};`这段代码定义了一个枚举类型`weekday`，其中包含以下枚举常量：
  - `sun`的值为 0（默认值）。
  - `mon`的值为 3（显式赋值）。
  - `tue`的值为 4（因为它在`mon`之后，自动递增）。
  - `wed`的值为 5（因为它在`tue`之后，自动递增）。
  - `thu`的值为 6（因为它在`wed`之后，自动递增）。
- **结构嵌套**：
  - 在定义嵌套的结构类型时，必须先定义成员的结构类型，再定义主结构类型。
  - 调用时`s.birth.year = 1984`(有2个)
- **结构指针**：
  - 使用结构指针作为函数参数只要传递一个地址值,因此,能够提高参数传递的效率。
  - `stu.num=100`，`(*p).num=100`，`p->num=100`等价。
  - `.`的优先级大于其他。一定要有括号。
  - “以下`scanf`函数调用语句中不正确的是”这种题，注意scanf后面假如不是%s，就是指针。
- ```
  struct student{ //1等价2,3
     int num;
     char name[20];
  } s;

  struct student{ //2  student是结构体类型
    int num;
    char name[20];  // num,name是结构成员
  };
  struct student s;  //s是结构体变量

  typedef struct student{ //3
    int num;
    char name[20];
  };
  student s;
  ```

```
17
18 typedef struct student{ //3
19   int num;
20   char name[20];
21 }s; //s是struct student的类型别名，而不是变量。因为有typedef在。
```

- **结构数组**
  - ```
    1 struct person{
    2       char name[10];
    3       int age;
    4 } c[10] = { “John”, 17, “Paul”, 19, “Mary”, 18, “Adam”, 16 };
    ```
  - 经常一个数+一个指针，然后弄晕我。这里指针就可以链表了。
    - ```
      1  struct stu{
      2    int x;
      3    int *y;
      4  } *p;
      5  int dt[4] = {10, 20, 30, 40};
      6  struct stu a[4] = {50, &dt[0], 60, &dt[1], 70, &dt[2], 80,
         &dt[3]};
      7
      8  int main( )
      9  {
      10    p=a;
      11    printf("%d,", ++p->x); // ->优先级高
      12    printf("%d,", (++p)->x);
      13    printf("%d", ++(*p->y));
      14
      15    return 0;
      16 }
      17 //输出51,60,21
      ```
    - ```
      1 static struct {
      2     int x, y[3];
      3 } a[3] = {{1,2,3,4},{5,6,7,8},{9,10,15,12}}, *p;
      4 p = a+1;
      5 // *((int *)(p+1)+2)=15.
      6 //二维指针，跳了一级，还是指针。
      ```

week12. 函数进阶与程序结构

函数递归

- 要素：边界条件、递归方程
- 允许直接递归调用也允许间接递归调用。
- 汉罗塔，逆序输出，week12-7-1求迷宫最短通道……
      ```

- **C语言的编译预处理功能：文件包含，宏定义，条件编译**

宏定义

- `#define`+宏名+宏定义字符串。C语言中，编译预处理后，所有的符号常量名和宏名都用相应的字符串替换。
- `#define S(a,b) { int t = a; a = b; b = t; }` 合法!
- `#define S(a,b) t=a;a=b;b=t`也合法。宏不存在类型问题,宏名其参数也无类型。
- 宏定义与变量定义不同,它只作字符替换不分配内存空间。
- 可以嵌套使用。
- 宏定义只是一种简单的字符替代,不进行语法检查,只有在编译已被宏展开后程序时才会发现语法错误并报错。例如:若将#define SIZE 20的零写成英文字号o， x=SIZE+15;会替换为x=2o+15;,再对其进行编译时系统就会报错。（不报错！）
- 宏展开不占运行时间,只占编译时间。不是一条C语句。
- 宏名不是必须用大写字母表示。
- 注意不带括号的情况!

```c
#define f(a,b,x) a*x+b
printf("%d,%d\n", f(1,2,3), f(f(1,2,3),4,2)); /* 中间没有空格 */
//输出5,11
```

```c
#define N 2
#define M N+1
#define NUM (M+1)*M/2 //8.5
```

```c
#define  MAX(x,y)  (x)>(y)?(x):(y)
main()
{  int a=4,b=5;
printf("%d\n",8*MAX(a,b));
}//4
```

条件编译

```c
#if...
#else...
#endif //标志着条件编译的结束(必须有)
#ifdef //"if defined" 的缩写,它用于检查某个宏是否已经定义。如果该宏已经定义（不管它是否有值，是否为0），则编译对应的代码块，否则跳过。
#ifndef // "if not defined" 的缩写
```

- **文件包含**
  - `#include"文件名"`
    - 编译程序首先到当前工作文件夹寻找被包含的文件，若找不到，再到系统include文件夹中查找文件，一般适用于编程者自己的包含文件
    - `#include “stdio.h”` 是正确的预处理指令。
  - `#include<文件名>`
    - 编译程序到C系统中设置好的include文件夹中把指定的文件包含进来.
    - `#include <文件名>`：通常用于包含标准库头文件，编译器只在标准目录中搜索文件。
    - `#include <路径/文件名>`：编译器会直接使用提供的路径来查找文件，而不会进行标准目录的搜索。但是母路径改不了，一直是核心文件的路径，然后你再添加的话只是在这个路径后面加东西.
  - `.h` 头文件，`.c` 主函数文件。
  - 一个完整的程序只能包含一个 `main()` 函数。
  - c程序中注释语句可以这样写：/`*`注释`*`/和//注释。

week13. 指针进阶

- **定义问题**
  - 语句 `int *p[5];` 表示p是一个指针数组，它包含5个指针（指向个体）变量元素。
  - `int (*p)[4]` 表示 `p` 是一个指向包含 4 个整数的数组的指针。
  - ```
    int* *p(); //p 是一个函数，返回类型是 int*，即返回一个指向 int 的指针。
    int *p(); //*p() 表示 p 是一个返回 int*（指向 int 的指针）的函数。
    char *s; s= “ABCDE”;//可以的，s指向A
    ```
  - ```
    char *p[10],str[10][20];
    for (i=0;i<10;i++)p[i]=str[i];
    for (i=0;i<10;i++)scanf("%s",str[i]);
    sort(p);
    ```
    - `p[i] = str[i];`:
      - `str[i]` 是一个字符数组（字符串）的首地址。
      - `p[i]` 是一个指向字符的指针。
      - 这条语句将 `str[i]` 的首地址赋值给 `p[i]`，使 `p[i]` 指向 `str[i]` 的首地址。
      - 这种方式适用于将指针数组的每个元素指向一个字符串。
    - `p[i] = &str[i];`:
      - `&str[i]` 是一个指向字符数组（字符串）的指针。
      - `p[i]` 是一个指向字符的指针。
      - 这条语句将 `str[i]` 的地址赋值给 `p[i]`，使 `p[i]` 指向 `str[i]` 的地址。
      - 这种方式会导致类型不匹配，因为 `p[i]` 是 `char*` 类型，而 `&str[i]` 是 `char (*)[20]` 类型（指向包含20个字符的数组的指针）。

- `->`被称为指向运算符。

- `int* const p = 0;`中`p`是指向`int`类型的常量指针，而不是指向常量型整数。若`int`是常量，则应该是`const int* p`。
- `char *p = t;return ( p + 1 );`这样是输出p+1及其之后的所有。不是只输出一个。
- ```
  const char *st[] = {"bag", "good", "This", "are", "Zoo", "park"};
  const char *smin;
  smin = st[0];  //不用=*st[0];
  ```
  - `const char *st[]`是一个**指向字符串的指针数组**。每个元素`st[i]`是一个`const char*`，它指向一个字符串的起始位置（即字符串的第一个字符）。
- `p = &s[0]; printf("%d\n", ++*p->b);`
- **二维数组**:
  - `a[i][j]`=`*(*(a+i)+j)`=`*(a[i]+j)`　一定要有括号!
  - `*(*(iArray+4)+3)`,`*(iArray[0]+4*COL+3)`等价
  - 第一个`*`是行，第二个是列
  - 练习一下
    - 若有语句"int a[3][4]={{1,3,5,7},{2,4,6,8}};"，则`*(*a+1)`的值为3。
    - ```
      const char *st[] = {"Hello", "world", "!"}, **p = st;
      p++;
      printf("%s-%c\n", *p, **p);  //world-w
      (*p)++;
      printf("%s-%c-%c\n", *p, **p, (**p)+1);  //orld-o-p
      ```

动态内存分配

- 定义在`<stdlib.h>`头文件中

```
# include <stdio.h>
# include <stdlib.h>
int main()
{
    int i, n, sum;
    int *p;
    scanf("%d", &n);

// 动态分配内存
p = (int*)malloc(n * sizeof(int));  // 为 n 个整数分配内存
if (p == NULL) {
    printf("Not able to allocate memory.\n");
    exit(1);  // 如果内存分配失败，则退出程序
}
for (i = 0; i < n; i++) {
    scanf("%d", &p[i]);  // 将输入的整数存储到动态分配的内存中
}

for (i = 0; i < n; i++) {
    sum += p[i];  // 累加每个整数
}
```

```
printf("%d\n", sum);
// 释放动态分配的内存
free(p);
return 0;}
```

- `p = (int*)malloc(n * sizeof(int));`：使用 `malloc` 动态分配 `n` 个 `int` 类型的内存。`sizeof(int)` 确保分配的内存足够容纳 `n` 个整数。p是指针型。
- 通过 `if (p == NULL)` 来检查内存是否成功分配。如果 `malloc` 返回 `NULL`，则说明内存分配失败，打印错误信息并退出程序。
- `free(p);`：释放通过 `malloc` 分配的内存，防止内存泄漏。
  - `p` 必须是一个指针，通常是指向堆（动态内存）中分配的内存块。
  - 假如在`free(p);`之后输出p，或前面a=p;输出a，都会乱码。
- ```
  int* arr = (int*)malloc(5 * sizeof(int));  // 为 5 个 int 类型的数组分配内存
  if (arr == NULL) {
      printf("Memory allocation failed.\n");
      return 1;
  }
  ```
- ```
  //建立一个长度为n的动态指针数组，用于保存n个字符数组的内存地址。在读入每个字符串时，用一个长度为1000的字符数组作为缓冲数组，将字符串读入并求出长度后，再动态分配空间，将缓冲数组中的字符串复制到新分配的动态空间中，并将动态空间的首地址保存到指针数组中。
  scanf("%d", &n);
  getchar();
  char **strings = (char **)malloc(n * sizeof(char *)); //二维
  char buffer[1000];
  for (i = 0; i < n; i++) {
      gets(buffer);
      strings[i] = (char *)malloc(strlen(buffer) + 1);
      strcpy(strings[i], buffer);
  }
  //做一些事情
  for (int i = 0; i < n; i++) {
     printf("%s\n", strings[i]);
     // 释放每个字符串的内存
     free(strings[i]);
  }
  // 释放指针数组的内存
  free(strings);
  ```

- **Pointers to pointers**
  - ```
    int k = 5;
    int *p = &k;
    int **m  = &p;
    printf("%d%d%d\n", k, *p, **m);
    5 5 5
    ```
    - 修改`**m`就是修改`k`
    ```

```c
int main() {
    int a[3] = {1, 2, 3};
    int *p = a;
    int **r = &p;
    // 打印指针 r 解引用后的值和数组 a 的地址
    printf("%p %p", (void*)*r, (void*)a);
    return 0;
}
//输出将是两个相同的地址值,比如0x7ffeedc0a5c0 0x7ffeedc0a5c0
```

```c
#include <stdio.h>
int main() {
    int a = 1, b = 2, c = 3;
    int *ptr1 = &a, *ptr2 = &b, *ptr3 = &c;
    int **sptr = &ptr1; // sptr 是一个指向 ptr1 的指针

    *sptr = ptr2; // 通过 sptr 修改 ptr1，使其指向 b
    // 这相当于 ptr1 = ptr2。*sptr看成一个整体。

    return 0;
}
```

- **函数指针（恶心死我了什么抽象东西）** p281
  - 函数名本身就是一种地址，从而找到函数执行的入口。在函数调用时，函数名被隐式地转换为函数地址，因此， fun 和 &fun 都表示函数 fun 的地址
  - pFun = &fun 或者 pFun = fun
  - 执行： pFun(...) 或者 (*pFun)(...)
  - p指针已经指向func函数，`p(2,3)` 和 `(*p)(2,3)` 都等价于 `func(2,3)`。

```c
#include <stdio.h>
void first() {
    printf("first");
}
void second() {
    first();
}
void third() {
    second();
}
int main() {
    void (*ptr)();  //定义函数指针 ptr 并将其指向 third 函数。通过函数指针 ptr 调用 third 函数。
    ptr = third;
    ptr();
    return 0;
}
```

- `void (*ptr)();` 定义了一个函数指针 `ptr`，指向返回类型为 `void` 且无参数的函数。
- 函数指针的调用语法允许省略解引用操作符 `*`。因此，`ptr();` 和 `(*ptr)();` 都是有效的
- `ptr = third;`不能`*ptr = third;`但是`ptr = &third;` 是可以的，并且与 `ptr = third;` 等效

```c
#include <stdio.h>
// 声明一个函数f，参数是一个指向返回int类型且参数为 int 类型的函数指针
void f(int (*x)(int));

int myfoo(int);

// 声明一个函数指针 fooptr，指向返回 int 类型且参数为 int（括号里的） 类型的函数
int (*fooptr)(int);

// 声明一个函数 foo，参数为 int 类型，返回一个指向返回 int 类型且参数为 int 类型的函数指针
int ((*foo(int)))(int);

int main() {
    // 调用 foo 函数，传入参数 0，返回一个函数指针并赋值给 fooptr
    fooptr = foo(0);
    // 调用 fooptr 指向的函数，传入参数 10
    fooptr(10);
    return 0;
}

// 定义 foo 函数，参数为 int 类型，返回一个指向返回 int 类型且参数为 int 类型的函数指针
int ((*foo(int i)))(int) {
    // 返回 myfoo 函数的地址
    return myfoo;
}

// 定义 myfoo 函数，参数为 int 类型，返回 int 类型
int myfoo(int i) {
    // 打印 i + 1 的值
    printf("%d\n", i + 1);
    return i + 1;
}
```

- 假如参数是一个指向返回 int 类型且参数为 int 类型的函数指针 `void f(int (*x)(int));`

```c
void f(int (*x)(int));
3.      int myfoo(int i);
4.      int (*foo)(int) = myfoo;
5.      int main()
6.      {
7.          f(foo(10));     // foo（10）：返回值是int，而f（）中的参数是要求是函数指针的!
8.      }
报错!
```

week14. 链表（朱老师win！

- 无聊的理论
  - 链表是一个**非连续**的数据结构
  - 链表中的元素（结点的数据部分）可以是任意类型的数据。
  - 单向链表中的每个结点都需要动态分配内存空间。单向链表是一种动态数据结构，它的结点在程序运行时根据需要逐个分配内存，而不像数组那样预先分配固定的内存空间。
  - 通常使用结构的嵌套来定义单向链表结点的数据类型。
  - 链表在插入和删除操作上的效率较高。链表的查询效率较低（链表不支持随机访问）。
  - 链表在某些情况下可以节省内存，但并不是总是如此。链表结点需要额外存储指针，这就意味着链表相较于数组需要更多的内存空间。
- 写在前面：一定要分清数据类型。p还是*p还是**p。
- **链表建立**
  - ```c
    //单链表结构体定义
    struct aa
    {
        int a;
        struct aa *pa;
    };//假如没有typeof,一定有struct！！
    ```
  - ```c
    // struct stu *head,*tail,*p;
    // 初始 head=tail=NULL;都是指针！
    scanf("%d%s%d",&num,name,&score);
    while(num!=0){
        p=(struct stu*)malloc(size);
        p->num=num;
        strcpy(p->name,name);
        p->score=score;
        p->next=NULL;
        if(head==NULL) head=p;
        else tail->next=p;
        tail=p;
        scanf("%d%s%d",&num,name,&score);
    }
    ```
    - 最后一个节点的指针是null。不是-1.
    - `strcpy(p->name,name);`字符串不能直接等于。
    - 判断链表为空：`head->next==NULL`

```
con* creatList() {  //头插法创建联系人链表
    int n;
    scanf("%d", &n);
    con *head = NULL;
    con *newNode;
    for (int i = 0; i < n; i++) {
        newNode = (con*)malloc(sizeof(con));//会了吗？？
        scanf("%d %s %s", &newNode->xh, newNode->name, newNode->tel);
        newNode->next = head;
        head = newNode;  // 更新头指针
    }
    return head;  // 返回链表的头指针
}
```

```
void generate(struct node **head)
{
    int num, i;
    struct node *temp;

    printf("Enter length of list: ");
    scanf("%d", &num);
    for (i = num; i > 0; i--)
    {
        temp = (struct node *)malloc(sizeof(struct node));
        temp->a = i;
        if (*head == NULL)
        {
            *head = temp;
            (*head)->next = NULL;
        }
        else
        {
            temp->next = *head;
            *head = temp;
        }
    }
}
```

- **链表遍历**
  - `p=p->next;`
  - 遍历结束条件是`p2!=NULL`还是`p2->next!=NULL`！！

```
51. void display(struct node *head)
52. {
53.     if (head != NULL)
54.     {
55.         printf("%d  ", head->a);
56.         display(head->next);      用递归来打印链表的结点信息
57.     }
58. }
```

```
//  模版  单链表head中偶数值的结点删除
struct ListNode {    //链表结点定义
   int data;
   struct ListNode *next;
};

struct ListNode *deleteeven( struct ListNode *head )
{
    struct ListNode *p1,*p2;
    while(head != NULL && head->data % 2 == 0) {  // 1.假如头满足条件
        p2 = head;
        head=head->next;
        free(p2);//释放内存！！
    }
    if(head == NULL) return head;                        //1.1 全满足
    p1 = head;
    p2 = p1->next;
    while(p2!=NULL) {                                 // 2.普通遍历
        if(p2->data % 2 == 0) {                          // 2.1满足：跳
            p1->next=p2->next;
            free(p2);
            p2=p1->next;
        }
        else{                                            //2.2不满足：继续
            p1=p2;
            p2=p1->next;
        }
    }
    return head;
}
```

- **链表插入**
  - 在p1和p2中间插入p3 `p3->next=p2; p1->next=p3;`或者`p3->next=p1->next; p1->next=p3;`
  - **先建立新联系，再修改旧联系**

- **节点删除**
  - `p1->next=p2->next; free(p2);`
  - 函数中有free()函数的调用，其功能一定是要删除某些结点。

```
//单链表的逆序1
void Convert(LNode *H) {
```

```
    LNode *p, *q;
    p=H;
    H->next=NULL;                           //这里和下面不一样!
    while (p != NULL) {
        q = p;
        p = p->next;
        q->next = H;  // 反转当前结点
        H = q;        // 更新H，使其指向新链表的头结点
    }
}

//单链表逆置2
void Convert(LNode *H)
{
    LNode *p,*q;
    p=H->next;
    H->next=NULL;//说明H是哨兵节点，H里面没有东西
    while(p!=NULL)
    {
        q=p;
        p=p->next;
        q->next=H->next;//H始终不更新，作为哨兵节点
        H->next=q;
     }
}

//单链表逆置3
//使用一个临时节点来保存当前节点的下一个节点，然后将当前节点的下一个节点指向其前驱节点，然后前驱节点和当前节点都向后移动一个节点，直到当前节点为空。
struct Node* reverseList(struct Node* head) {
    struct Node* prev = NULL;
    struct Node* curr = head;
    while(curr != NULL) {
        struct Node* nextTemp = curr->next;
        curr->next = prev;
        prev = curr;
        curr = nextTemp;
    }
    return prev;
}

//单链表逆置4（递归）
void stack_reverse(struct node **head, struct node **head_next){
        struct node *temp;
        if (*head_next != NULL){
            temp = (*head_next)->next;
            (*head_next)->next = (*head);
            *head = *head_next;
            *head_next = temp;
            stack_reverse(head, head_next);//就是递归代替 while(curr != NULL)的过程
        }
    }

```

```c
struct ListNode {
   int data;
   struct ListNode *next;
};

struct ListNode *mergelists(struct ListNode *list1, struct ListNode *list2)
{
    struct ListNode *head=list1,*p1,*p2,*p;
    if(head==NULL) {
        head=list2;
        return head;
    }
    p2=head;
    while(list2!=NULL)
    {
        p=list2->next;
        while(list2->data > p2->data && p2->next!=NULL)
        {
            p2->next=list2;//?
            p2=p2->next;
        }
        if(list2->data <= p2->data)
        {
            if(p2==head) head=list2;
            else p2->next=list2->next;
            list2->next=p2;
            p1=list2;
        }
        else{
            p2->next=list2;
            break;
        }
        list2=p;
    }
    return head;
}
```

```c
//没有耐心看下去。但现在的宝盖一定要看下去。
//程序的功能是：逆序创建一个键值为整数的链表 L，编程实现将其中绝对值重复的键值结点删掉。即对每个键值 K，只有第一个绝对值等于 K 的结点被保留。同时，所有被删除的结点须按照原来顺序保存在另一个链表中。例如给定链表L的各键值21→-15→-15→-7→15，则输出去重后的链表： 21→-15→-7，以及被删除的结点链表： -15→15。
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

struct ListNode {
    int data;
    struct ListNode *next;
};

struct ListNode *Createlist(int n);
struct ListNode *Del_absrepeat(struct ListNode **head);
```

```c
void Printlist(struct ListNode *head);

int main() {
    struct ListNode *head = NULL, *head2 = NULL;
    int n;
    scanf("%d", &n);
    head = Createlist(n);
    printf("原始链表：");
    Printlist(head);

    head2 = Del_absrepeat(&head);
    printf("删除重复结点的链表：");
    Printlist(head);

    printf("被删除的结点组成的链表：");
    Printlist(head2);
    return 0;
}

void Printlist(struct ListNode *head) {
    struct ListNode *p;
    for (p = head; p != NULL; p = p->next)
        printf("%d ", p->data);
    printf("\n");
}

struct ListNode *Createlist(int n) {  //新建（逆序）
    struct ListNode *head = NULL, *newNode;
    int value;
    for (int i = 0; i < n; i++) {
        scanf("%d", &value);
        newNode = (struct ListNode *)malloc(sizeof(struct ListNode));
        newNode->data = value;
        newNode->next = head;
        head = newNode;
    }
    return head;
}

struct ListNode *Del_absrepeat(struct ListNode **head) {
    if (*head == NULL) return NULL;

    struct ListNode *current = *head;
    struct ListNode *prev = NULL;
    struct ListNode *deletedHead = NULL;
    struct ListNode *deletedTail = NULL;
    int absValues[1000] = {0}; // 假设数据范围在 -999 到 999 之间

    while (current != NULL) {
        int absValue = abs(current->data);
        if (absValues[absValue] == 0) {
            absValues[absValue] = 1;
            prev = current;
            current = current->next;
        } else {
```

```
            if (prev != NULL) {
                prev->next = current->next;
            } else {
                *head = current->next;
            }
            struct ListNode *temp = current;
            current = current->next;
            temp->next = NULL;
            if (deletedHead == NULL) {
                deletedHead = temp;
                deletedTail = temp;
            } else {
                deletedTail->next = temp;
                deletedTail = temp;
            }
        }
    }
    return deletedHead;
}
```

- ```
//双链表
#include <stdio.h>
#include <stdlib.h>

// 定义双向链表节点结构
struct Node {
    int data;
    struct Node *prev;
    struct Node *next;
};

// 定义链表结构，包含头节点和尾节点
struct DoublyLinkedList {
    struct Node *head;
    struct Node *tail;
    struct Node **insertedNodes; // 用于记录插入的节点
    int insertCount; // 记录插入的节点数量
};

// 初始化链表
struct DoublyLinkedList* initList(int maxOperations) {
    struct DoublyLinkedList *list = (struct DoublyLinkedList
*)malloc(sizeof(struct DoublyLinkedList));
    list->head = NULL;
    list->tail = NULL;
    list->insertedNodes = (struct Node **)malloc(maxOperations *
sizeof(struct Node *));
    list->insertCount = 0;
    return list;
}

// 创建新节点
struct Node* createNode(int data) {
    struct Node *newNode = (struct Node *)malloc(sizeof(struct Node));
    newNode->data = data;
```
```

```c
    newNode->prev = NULL;
    newNode->next = NULL;
    return newNode;
}

// 在最左侧插入节点
void insertLeft(struct DoublyLinkedList *list, int data) {
    struct Node *newNode = createNode(data);
    if (list->head == NULL) {
        list->head = newNode;
        list->tail = newNode;
    } else {
        newNode->next = list->head;
        list->head->prev = newNode;
        list->head = newNode;
    }
    list->insertedNodes[list->insertCount++] = newNode;
}

// 在最右侧插入节点
void insertRight(struct DoublyLinkedList *list, int data) {
    struct Node *newNode = createNode(data);
    if (list->tail == NULL) {
        list->head = newNode;
        list->tail = newNode;
    } else {
        newNode->prev = list->tail;
        list->tail->next = newNode;
        list->tail = newNode;
    }
    list->insertedNodes[list->insertCount++] = newNode;
}

// 删除第 k 个插入的节点
void deleteNode(struct DoublyLinkedList *list, int k) {
    struct Node *nodeToDelete = list->insertedNodes[k - 1];
    if (nodeToDelete->prev != NULL) {
        nodeToDelete->prev->next = nodeToDelete->next;
    } else {
        list->head = nodeToDelete->next;
    }
    if (nodeToDelete->next != NULL) {
        nodeToDelete->next->prev = nodeToDelete->prev;
    } else {
        list->tail = nodeToDelete->prev;
    }
    free(nodeToDelete);
}

// 在第 k 个插入的节点左侧插入节点
void insertLeftOf(struct DoublyLinkedList *list, int k, int data) {
    struct Node *node = list->insertedNodes[k - 1];
    struct Node *newNode = createNode(data);
    newNode->next = node;
    newNode->prev = node->prev;
```

```c
    if (node->prev != NULL) {
        node->prev->next = newNode;
    } else {
        list->head = newNode;
    }
    node->prev = newNode;
    list->insertedNodes[list->insertCount++] = newNode;
}

// 在第 k 个插入的节点右侧插入节点
void insertRightOf(struct DoublyLinkedList *list, int k, int data) {
    struct Node *node = list->insertedNodes[k - 1];
    struct Node *newNode = createNode(data);
    newNode->prev = node;
    newNode->next = node->next;
    if (node->next != NULL) {
        node->next->prev = newNode;
    } else {
        list->tail = newNode;
    }
    node->next = newNode;
    list->insertedNodes[list->insertCount++] = newNode;
}

// 打印链表
void printList(struct DoublyLinkedList *list) {
    struct Node *current = list->head;
    while (current != NULL) {
        printf("%d ", current->data);
        current = current->next;
    }
    printf("\n");
}

int main() {
    int M;
    scanf("%d", &M);

    struct DoublyLinkedList *list = initList(M);

    for (int i = 0; i < M; i++) {
        char command[3];
        int x, k;
        scanf("%s", command);
        if (command[0] == 'L') {
            scanf("%d", &x);
            insertLeft(list, x);
        } else if (command[0] == 'R') {
            scanf("%d", &x);
            insertRight(list, x);
        } else if (command[0] == 'D') {
            scanf("%d", &k);
            deleteNode(list, k);
        } else if (command[0] == 'I' && command[1] == 'L') {
            scanf("%d %d", &k, &x);
```

```
            insertLeftOf(list, k, x);
        } else if (command[0] == 'I' && command[1] == 'R') {
            scanf("%d %d", &k, &x);
            insertRightOf(list, k, x);
        }
    }

    printList(list);

    return 0;
}
```

- ```
//函数参数list1和list2是用户传入的两个按data升序链接的链表的头指针；函数mergelists将两个链表合并成一个按data升序链接的链表，并返回结果链表的头指针。
struct ListNode *mergelists(struct ListNode *list1, struct ListNode *list2)
{
    struct ListNode *head=list1,*p1,*p2,*p;
    if(head==NULL) {
        head=list2;
        return head;
    }
    p2=head;
    while(list2!=NULL)
    {
        p=list2->next;
        while(list2->data > p2->data && p2->next!=NULL)
        {
            p1=p2;
            p2=p2->next;
        }
        if(list2->data <= p2->data)
        {
            if(p2==head) head=list2;
            else p1->next=list2;
            list2->next=p2;
            p1=list2;
        }
        else{
            p2->next=list2;
            break;
        }
        list2=p;
    }
    return head;
}
```

- ```
//找到两个链表 A 和 B 有且仅有的一个最大值节点，分别删除节点后，将链表 A 的前半段与链表 B 的后半段相连，B 的前半段与 A 的后半段相连。
#include <stdio.h>
#include <stdlib.h>

typedef struct Node {
    int data;
```

```
    struct Node* next;
} Node;

void printLink(Node* head) {
    /* -- 定义省略，用于打印链表数据 -- */
}

Node* createNode(int data) {
    Node* newNode = (Node*)malloc(sizeof(Node));
    newNode->data = data;
    newNode->next = NULL;
    return newNode;
}

Node* createLink() {
    Node* head = createNode(0);
    Node* tail = head;
    int data;
    while (1) {
        scanf("%d", &data);
        if (data == -1) break;
        Node* newNode = createNode(data);
        tail->next = newNode;
        tail = newNode;
    }
    return head;
}

void findPreviousMaxNode(Node* head, Node** pre) {
    Node* current = head;
    *pre = head;
    while (current->next != NULL) {
        if (current->next->data > (*pre)->next->data) {
            *pre = current;
        }
        current = current->next;
    }
    return;
}

void reLink(Node* A, Node* B) {
    Node *preMaxA, *preMaxB;
    Node *temp;
    findPreviousMaxNode(A, &preMaxA);
    findPreviousMaxNode(B, &preMaxB);

    temp = preMaxA->next->next;
    preMaxA->next = preMaxB->next->next;
    preMaxB->next = temp;
    printLink(A);
    printLink(B);
}

int main() {
    Node* A = createLink();
```

```
    Node* B = createLink();
    reLink(A, B);
    return 0;
}
```

week15. 文件

- 文件指针和位置指针都随着文件的读写操作而不断变化，自动向后移动。f
- 文件指针指向的是一个struct类型，并不是文件缓冲区和外部存储区，在这个struct类型当中有成员是指向文件缓冲区的
- 随机操作不仅仅适用于文本文件，也适用于二进制文件。随机操作是指在文件中的任意位置进行读写，而不是从文件的开头开始顺序地进行操作。
- ```
  fopen(...);
  fprintf(...);
  fclose(fp);
  //打开一个文件进行读取：（当fopen函数打开文件失败时，返回给文件指针的值是NULL）
  FILE *fp = fopen("file.txt", "r");
  //打开一个文件进行写入（清空文件内容）：
  FILE *fp = fopen("file.txt", "w");
  //打开一个文件进行追加（在文件末尾写入，文件指针指向文件尾）：
  FILE *fp = fopen("file.txt", "a");
  //打开一个文件进行读写操作：
  FILE *fp = fopen("file.txt", "r+");
  //打开一个文件进行二进制读取：
  FILE *fp = fopen("file.bin", "rb");
  ```
  - 与r相关的，不能用来打开不存在的文件。
  - `"w+"`：建立新文本进行读写。w,w+都会清空文件内容。

    `"a+"`：以读写追加模式打开文件。如果文件不存在，将创建一个新文件。如果文件已存在，文件指针将定位到文件末尾。
  - 文件打开方式 `"w+"` 和 `"a+"` 有同样效果的情况是：要打开的文件存在且为空； 要打开的文件不存在
  - 将结构体数组写入文件时，通常使用二进制模式。like `(fp = fopen("course.dat", "wb/wb+")) == NULL`
- **文件读写函数**
  - 字符：`ch=fgetc(fp);  fputc(ch,fp);`
  - 字符串：`fgets(c,n,fp);`最多读取n-1个字符。 `fputs(s,fp);`

    `fgets()` 函数从文件中读取字符串时，并不是读取到字符 `'\0'` 或 `EOF` 为止，而是读取到换行符 `'\n'`、文件结束符 `EOF` 或达到指定的最大字符数（包括终止符 `'\0'`）为止。
  - `fscanf(fp,"%d %f",&n,&x);  fprintf(...);`
  ```

    - 数据块(二进制)：`fread(buffer,size,count,fp);  fwrite(buffer,size,count,fp);`其中 buffer 是一个指针，在函数fread()中，它表示存放输入数据的首地址;在函数fwite(中，它表示存放输出数据的首地址。size表示数据块的字节数。count表示要读写的数据数。f表示文件指针。
    - 读到末尾：
        - `while ((ch = fgetc(fp)) != EOF)`
        - `while(!feof(fp))`
        - `fscanf(file,"%d",&num)!=EOF`
        - `fscanf(file, "%d", &num) == 1`
        - `! fscanf(file, "%d", &num)`
- **其他函数**
    - `feof` 函数用于检查文件指针是否到达文件末尾。只有在尝试读取文件末尾之后才会返回非零值。因此，通常在读取操作之后使用 `feof` 函数来检查是否到达文件末尾。
    - `rewind(fp)`将文件内部的位置指针重置为文件开始处
    - `fseek(fp,偏移量，开始位置)`控制指针移动
    - `ftell(文件指针)`获取文件开头的位移量
    - `ferror(文件指针)`
    - `clearer(文件指针)`
- 文件的读函数是从输入文件中读取信息，并存放在内存中。
- c语言源程序是文本文件，目标文件和可执行文件（exe）是二进制文件。
    - 当我们运行编译器时（例如 `gcc`），它将 C 语言源程序转换为目标文件。这个过程称为编译。
- 从文件的逻辑结构上看，c语言把文件看作数据流，并将数据按顺序以一维方式组织存储。
- **缓冲文件系统**
    - 在进行文件操作时，系统自动为每一个文件分配一块文件内存缓冲区（内存单元）。 标准库函数如 `fgetc()`、`fputc()`、`fgets()`、`fputs()` 等都在缓冲模式下工作。通过使用 `FILE*` 类型的文件指针，C 语言会为每个打开的文件分配一个缓冲区。 文件指针指向文件缓冲区中文件数据的存取位置。
    - **内存与磁盘的交互**：
        - **读取操作**：当从文件读取数据时，操作系统会先从磁盘读取一部分数据到内存缓冲区中。如果缓冲区已加载数据，则数据可以直接从内存中读取，而不需要每次都访问磁盘。
        - **写入操作**：当向文件写入数据时，数据会首先被写入到缓冲区。当缓冲区满时，或者文件关闭时，缓冲区中的数据会一次性写入到磁盘。
- C系统的标准输出文件stdout是指显示器。 C系统的标准输入文件stdin，通常对应终端的键盘。
- 一般不能用任何一个文本编辑器打开二进制文件进行阅读。但二进制文件名能用`.txt`作为扩展名。关键在于打开文件时使用正确的模式（rb，wb）。
-
```c
//存储课程信息的结构体数组，从键盘输入n(n≤10)们课程信息，并将其逐个写到一个文件中去。
#include <stdio.h>
typedef struct _Course {
    int cNo;
    char cName[30];
} COURSE;

int main()
```

```
{
    int i, n;
    FILE *fp;
    COURSE cArr[10];
    scanf ("%d", &n);
    if ((fp = fopen("course.dat", "w")) == NULL) {
        printf ("Can not open the file!");
        exit(0);
    }
    for (i = 0; i < n; i++){
        scanf("%d %s", &cArr[i].cNo, cArr[i].cName);
        fwrite(&cArr[i], sizeof(COURSE), 1, fp);
    }

    fclose(fp);
    return 0;
}
```

-

问题

-
```
//week13 2-8  有无输出？
#include<stdio.h>
void main()
{
    int a[3][2]={0},(*p)[2],i,j;
    for(i=0;i<2;i++)
    {
        p=a+i;
        scanf("%d",p);
        p++;
    }
    for(i=0;i<3;i++)
    {
        for(j=0;j<2;j++)
            printf("%2d",a[i][j]);
        printf("\n");
    }
}
```

- week12 1-5
  - 若有宏定义：#define S(a,b) t=a;a=b;b=t 由于变量t没定义，所以此宏定义是错误的。 错了。
- week12 2-5格式 1:已解决

```
#include"文件名"
```

格式 2:

```
#include<文件名>
```

下面 4 个结论, 正确的是 ( )。c错了?

A.格式 1 中的文件名能带路径, 而格式 2 不能B.格式 1 中的文件名不能带路径, 而格式 2 可以C.如果被包含文件在当前目录下, 两种格式都能用D.如果文件名中不带路径, 格式 1 能搜索当前目录和编译环境指定的标准目录,而格式 2 不行

- week11-4-3
- ```
  // week15 3-6 错误的使用 feof 作为循环条件
     while (!feof(fp)) {
          ch = fgetc(fp);
          if (ch != EOF) {
              putchar(ch);
          }
     }
  ```
- week14
  - 2-4 B
- week15
  - 1-1 文件指针和位置指针都是随着文件的读写操作在不断改变，f 1
  - 1-9 文件指针指向文件缓冲区中文件数据的存取位置。F 1
  - 3-5 C 1
  - 3-6 AEF
  - 4-10 1
  - 4-11 fwrite(cArr[i],sizeof(COURSE)，1，fp 1
  ```