程序分组设计训练

课程知识汇总

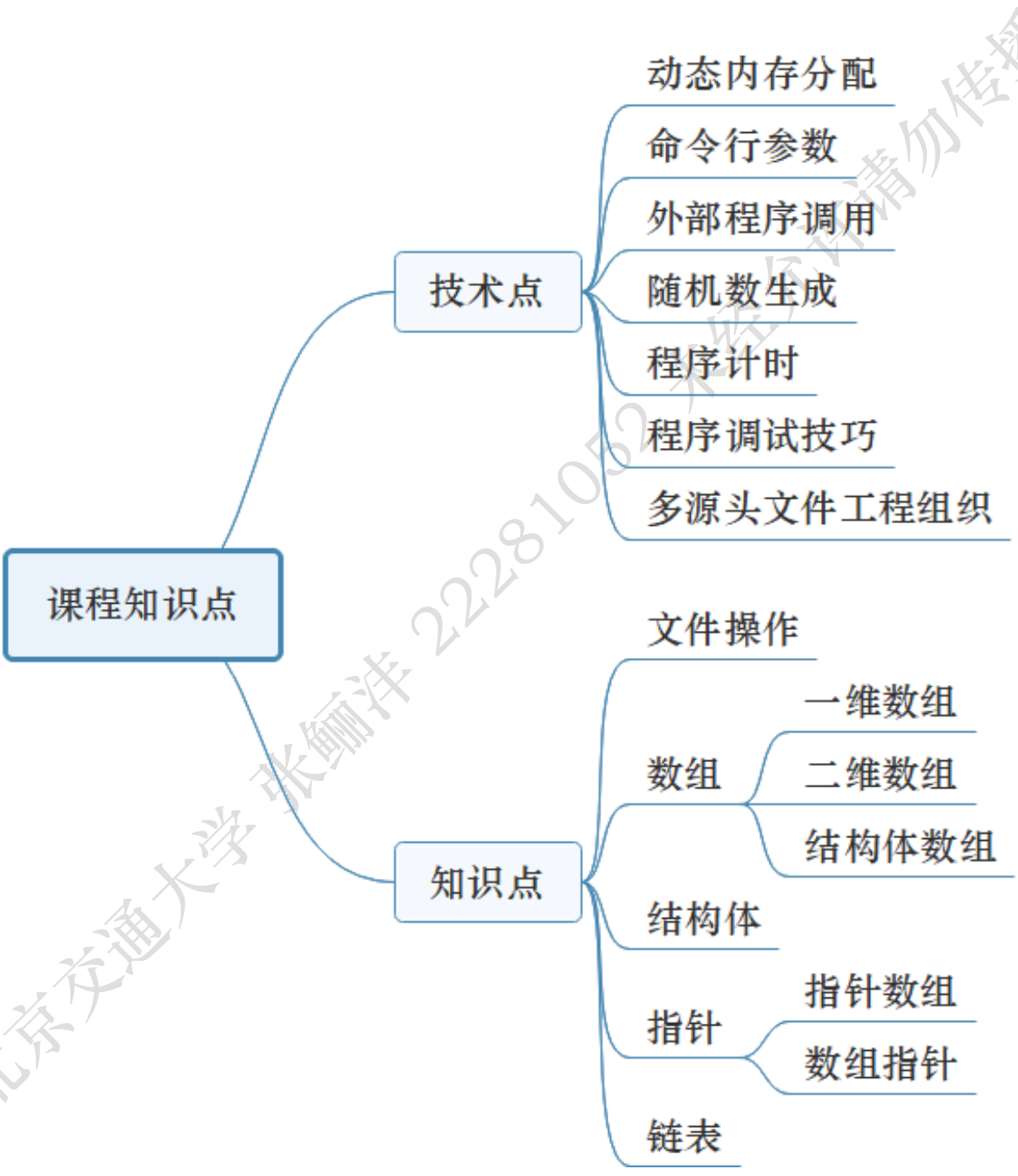
（第二版）

知识汇总中有个人编写的代码，会有部分地方使读者难以理解。代码部分仅供参考，最好结合自己写的代码进行复习。

张鳴沣

2023 年 6 月 26 日

课程体系



程序分组设计训练

课程知识汇总

目录

1 实验 1 部分

1.1 数组与指针

1.1.1 一维数组

格式：

类型说明符 数组名 [常量表达式];

数组名表示内存首地址，它是地址常量。

1.1.2 二维数组

格式：

数组类型名 数组名[数组元素个数] [数组元素个数]

例：

double a[5] [4]; 二维实数数组, 5 行、4 列

int b[5] [6]; 二维整数数组, 5 行、6 列

char str[3][2][4]; 三维字符数组

数组元素的存放顺序

二维数组：按行序优先

多维数组：最右下标变化最快

原因：内存是一维的

对于二维数组 a[x][y]：

a 是开始位置（数组 a 的首地址）也是 a[0]的开始位置，也是 a[0][0]的位置。

1.1.3 一维数组与指针

数组名可以看作是一个地址常量；计算机系统在处理数组时，在一个内存区域中，为数组分配一片连续的空间，并把这片区域的首地址值存入数组名中。数组名相当于地址常量。

数组名可以看作是一个**基类型为数组元素的基类型的地址常量**。

定义指针类型变量指向数组元素：

首先定义一个一维数组 a；再定义一个指向数组元素类型的指针变量 p；此时 p 就可以指向 a 数组的数组元素了。

例：int *p；int a[5]；则：p = &a[0]；或 p = a

C 语言中**数组名代表数组的首地址；**

p = &a[0]与 p = a 等价；

这里 a 为数组名，代表数组的首地址。

指针的类型必须等同于数组元素的类型；可以在定义指针的同时赋予数组的首地址。

例如：int a[8]； int *p = a;

理解指针位移：例如： int a[6]; int *p; p = a;

元素值			元素地址		
*p	a[0]	*a	a	p	&a[0]
*(p+1)	a[1]	*(a+1)	a+1	p+1	&a[1]
*(p+2)	a[2]	*(a+2)	a+2	P+2	&a[2]

C 规定，数组名在参与大部分的表达式运算时，将被转换成指向数组首元素的指针 (array-to-pointer)。

例如：int a[5]; int *p; p = a;

不发生转换的情况：**sizeof 和&操作符**；

sizeof：

sizeof(a)返回 20 个字节，此时 a 是 int [5]。sizeof(int *)返回 4 个字节。

sizeof 返回整个数组的长度，而不是指向数组的指针的长度。

可以用 sizeof(p)测试指针的大小

可以用 sizeof(*p)测试指针指向的数据类型的大小

&操作符：

p = &a;

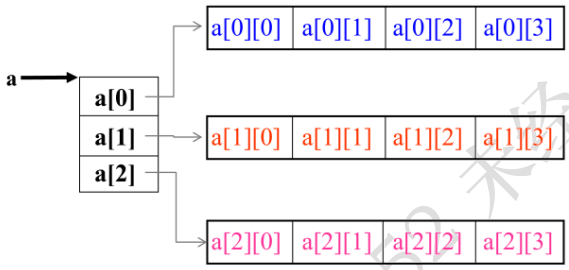
取一个数组名的地址所产生的是一个指向数组的指针，而不是一个指向某个指针常量的指针。

1.1.4 二维数组与指针

以 int a[3][4]; 为例

数组名 a 为指针常量（**二级指针**），二维数组的首地址

行名 a[0]、a[1]、a[2]也是指针常量（**一级指针**）



对二维数组 int a[3][4],有

a-----二维数组的首地址，即第 0 行的首地址

a+i-----第 i 行的首地址

a[i] ⇔ *(a+i)------第 i 行第 0 列的元素地址

a[i]+j ⇔ *(a+i)+j -----第 i 行第 j 列的元素地址

***(a[i]+j) ⇔ *(*(a+i)+j) ⇔ ⇔ a[i][j]**

a+i=&a[i]=a[i]=*(a+i) =&a[i][0]，值相等，含义不同

a+i ⇔ &a[i],表示第 i 行**首地址**，指向行（**二级指针**）

a[i] ⇔ *(a+i) ⇔ &a[i][0]，**表示第 i 行第 0 列元素地址**，指向列（**一级指针**）

表示形式	含义
a	二维数组名，数组首地址
a[0],*(a+0),*a	第 0 行第 0 列元素的地址
a+1, &a[1]	第 1 行首地址
a[1], *(a+1)	第 1 行第 0 列元素的地址
a[1]+2, *(a+1)+2， &a[1][2]	第 1 行第 2 列元素的地址
*(a[1]+2), *(*(a+1)+2), a[1][2]	第 1 行第 2 列元素的值

数组的首元素地址：表示数组的首个元素的地址。

数组的首地址：表示整个数组的地址

*降级　　&升级

注：区分(*(a+x))[y]与*(a+x)[y]

“(*(a+x))[y]” 相当于 a 为二级指针，之后向后移动 x 行后转换为一级指针，[y]相当于在一级指针表示“a[0+x][0]”的基础上向后移动 y 位，取值得到最终数字。

“*(a+x)[y]” 相当于 a 为二级指针，之后向后移动 x 行后，[y]相当于在二级指针表示在 x 的基础上向后移动 y 行，之后转换为一级指针。

1.1.5 指针的顺时针右旋法则

规则：

从变量名开始，先右后左，遇到括号折返，如此反复。

记下此过程中遇到的符号顺序，然后解读：这些符号都是变量名的修饰语，修饰语的位置与记录的符号位置一致。[1]

范例如下：

int (*daytab)[13] 符号顺序：daytab * [13] int daytab 是指针 daytab 是指向长度为 13 的数组的指针 daytab 是指向类型为整数且长度为 13 的数组的指针	void (*f[10]) (int, int) 符号顺序：f [10] * (int, int) void f 是长度为 10 的数组 f 是长度为 10 的指针数组（请注意，不是数组指针） f 是长度为 10 的（带两个整形参数的）函数指针数组 f 是长度为 10 的（带两个整形参数，返回值为 void 的）函数指针数组
char (*(*x( ))[ ]) ( ); 符号顺序：x ( ) * [] * () char x 是不带参数的函数 x 是返回值为指针的不带参数的函数 x 是返回值为数组指针的不带参数的函数 x 是返回值为指针数组指针的不带参数的函数 x 是返回值为函数指针数组指针的不带参数的函数 x 是返回值为（返回值为字符的函数指针数组）指针的不带参数的函数	**char (*(*x[3])( ))[5]** 符号顺序：x [3] * () * [5] char x 是长度为 3 的数组 x 是类型为指针长度为 3 的数组，或者说是长度为 3 的指针数组 x 是长度为 3 的函数指针数组 x 是长度为 3 的（返回值为指针）的函数指针数组 x 是长度为 3 的返回值为（长度为 5 的数组）指针的函数指针数组 x 是长度为 3 的返回值为长度为 5 的（字符）数组指针的函数指针数组[1]

注：优先级由“()” “[]” “*”依次递减

详细参见：https://yangwujie.github.io/slides/c/01.html

1.1.6 数组指针与指针数组

数组指针：

定义为：int(* p)[n];

（int (*p)[5]定义了一个指向含有 5 个元素的一维数组的指针）

对于一维数组：

```
int (*p)[n];   // 定义了指向含有 n 个元素的一维数组的指针
int a[n];      // 定义数组
p = a;         // 将一维数组首地址赋值给数组指针 p
```

对于二维数组：

```
int (*p)[4];   // 定义了指向含有 4 个元素的一维数组的指针
int a[3][4];
p = a;         // 将二维数组的首地址赋值给 p，也可是 a[0]或&a[0][0]
p++;           // 表示 p 跨过行 a[0][],指向了行 a[1][]
```

此时数组指针也成为指向一维数组的指针，也就是行指针。

指针数组：

定义为：int* p[n];

p 是数组，是一个由 n 个指针类型元素组成的指针数组，或者说**这个当一个数组里含有的元素为指针类型的时候，它就被成为指针数组。**当 p+1 时，则 p 指向下一个数组元素。

（需注意，p=a；这种赋值方法是错的，因为 p 是一个不可知变量，只存在 p[0]，p[1]，p[2]，但可以这样 p=a; 这里 p 表示指针数组第一个元素的值，a 的首地址的值）[2]

```
int *p[3];       // 定义指针数组
int a[3][4];
for (i = 0; i < 3; i++)
    p[i] = a[i]; // 通过循环将 a 数组每行的首地址分别赋值给 p 里的元素
```

1.2 断点与调试

1.2.1 条件断点

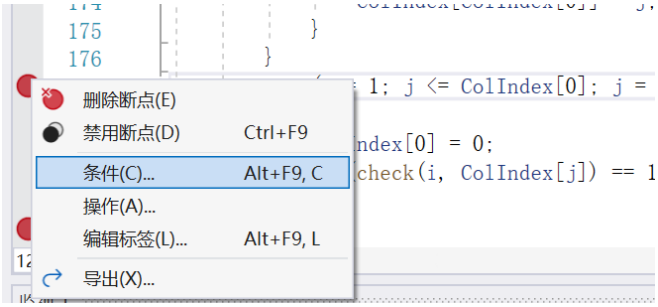


图 1　条件断点设置 1

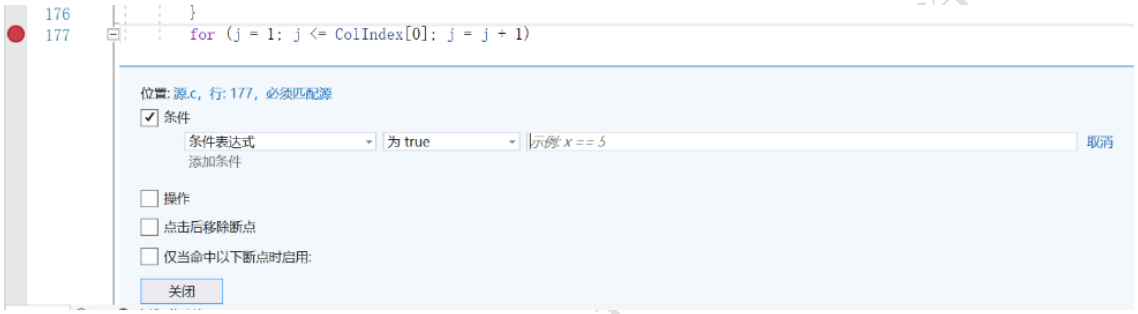


图 2　条件断点设置 2

（1）条件断点是当设置某一条件后，当符合此条件时运行断点，否则略过此断点。

（2）当设置好普通断点后，右键点击断点，选择“条件(C)”，即会弹出图 2 的对话框，之后根据需求进行设置即可。

1.2.2 step in 、step over、step out

step-in：进入函数里，点击后执行下一个语句。

step-over：如果当前行是一个函数调用，则调试器将在函数调用之后的下一条语句停止。调试器不会进入函数体。（不常用）

step-out：跳出函数。

图 3　continue、step in 、step over、step out（从左到右）

2 实验 2 部分

2.1 随机数

2.1.1 伪随机数函数 rand( )

标准库提供随机数功能，需要包含文件<stdlib.h>。

随机数生成函数：int rand(void)

无参数，得到 0 和符号常量 RAND_MAX 间的随机整数。不同系统的 RAND_MAX 可能不同，至少为 32767。

根据下面公式可以得到所需范围内的随机数：

n = a + rand () % b

其中 a 为位移因子，是所需连续整数范围的第一个数，b 是比例因子，是所需连续整数范围的宽度，则希望产生 1-6 之间随机数的公式为：

n = 1 + rand () % 6

2.1.2 srand 函数与 time 函数

普通情况下我们在平时使用 rand()函数的时候必须要种种子，也就是种随机数种子，确保每一次产生的数值不一样；使用 srand ()的函数，然后使用 time()函数在一定的时间里产生不同的序列；一般在写的时候是这样 **srand (time(NULL))** 这样就相当于种种子。

函数 srand 用 seed 值设种子值：void srand(unsigned seed)，默认初始种子值是 1；rand 根据当时种子值生成下一随机数并修改种子值。

使用 time 函数返回值做种子，防止随机数重复。（时间是不重复的）

```
#include <time.h>
#include <stdlib.h>
srand((int)time(0)); //  设置随机数种子，#include<time.h>
data_3[i][0] = rand() % 1000;
```

注意：最好不要把种子设计到循环内。设置种子时，将种子设置在了循环内，导致了时间不足而造成了所有行的数字都是相同的，所以在设置种子时需要将种子设置在循环外，以防产生的每一行数字都是相同的。

2.2 命令行参数调用

2.2.1 入口设计

以工程方式组织程序开发，将其他程序拆分为多个源代码文件，实验入口函数要求由一条语句实现，实验的所有功能均实现在 run( )函数内。实验主入口函数代码如下：

```
int main(int argc, char* argv[])
{
    run(argc, argv); //  调用程序主功能实现函数
    return 0;
}
```

其中，argc 是输入参数个数，argv 是输入参数的内容（字符串!!!）。

工程方式组织程序开发，将其他程序拆分为多个源代码文件，实验入口函数要求由一条语句

2.2.2 调用 cmd 窗口

进行程序相关测试，首先用“Win+R”键，调用运行窗口，输入 cmd。

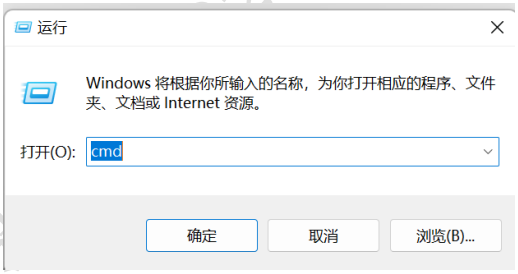


图 4　运行窗口输入 cmd

如果程序在 D 盘，输入“d:”，cmd 窗口就会转到 D 盘进行后续操作。用户输入相关文件地址，之后就可以开始相关程序运行。此时输入地址为第一个参数（argv[0]）。

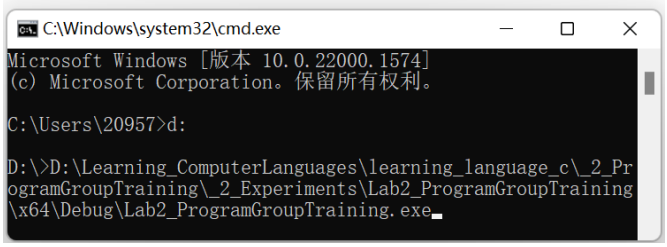


图 5　cmd 窗口

输入空格，之后输入数据，不打回车，则是继续输入了参数 argv[1]、argv[2]。

3 实验 3 部分

3.1 文件路径检查

逻辑如下：

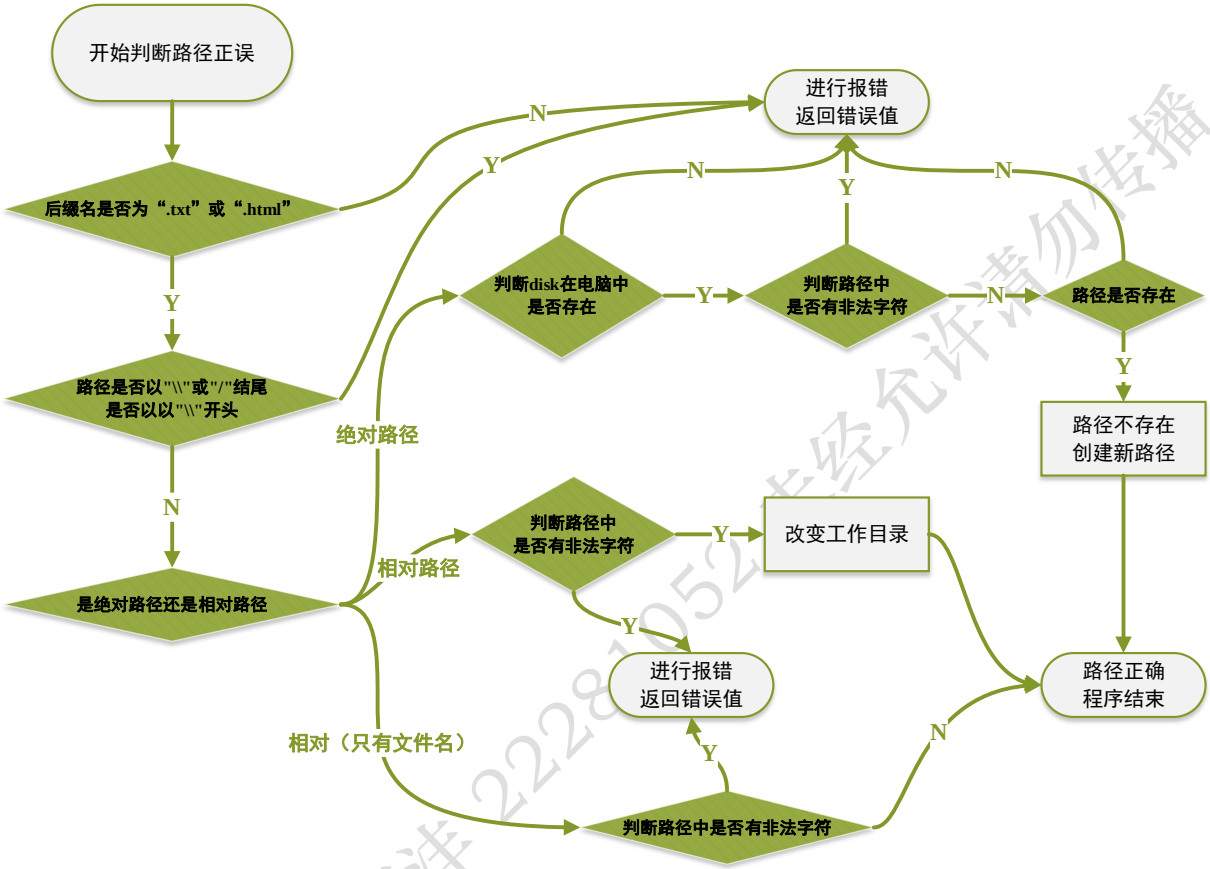


图 6　判断路径合法函数流程图

代码如下：

```
/*
 *函数名称：check_seper_file
 *函数功能：对argv[1]或argv[2]数据是否合法（为正确文件名）进行检查
 *          判断路径为相对路径还是绝对路径，之后切割路径与文件名，进行储存
 *输入参数：char* str:文件路径与文件名     CONF data:结构体数组地址
 *返回值：    int,如果检查后合法，返回值为1,否则为0
 *版本信息：create by Lifeng Zhang, 2023-04-01
 *          repair by Lifeng Zhang, 2023-04-04
 */
int check_seper_file(char* str, CONF* data)
{
  int len = strlen(str);
  //_MAX_DRIVE、_MAX_DIR、_MAX_FNAME、_MAX_EXT 在配置中已有定义
  //_MAX_DRIVE==3，_MAX_DIR==_MAX_FNAME==_MAX_EXT==256
  char disk[_MAX_DRIVE];
  char dir[_MAX_DIR];
```

```
char fname[_MAX_FNAME];
char ext[_MAX_EXT];

char way[MAX_STR_LEN];
char file[MAX_STR_LEN];
char filename_t[MAX_STR_LEN] = "";

// 实验要求文件以.txt结尾
if (str[len - 1] != 't' && str[len - 2] != 'x' && str[len - 3] != 't' && str[len - 4] != '.')
    return 0;

if (*str == '\\') // 不能以"\\"开头
    return 0;

// 不能以"\\"或"/"结尾，若以此结尾则认为文件名为空
char* index1 = strrchr(str, '\\');
char* index2 = strrchr(str, '/');
if (index1 - str + 1 == strlen(str) || index2 - str + 1 == strlen(str))
    return 0;

// 进行合并，以便后续处理
char* index;
if (index2 > index1)
    index = index2;
else
    index = index1;

// 判断输入路径为绝对路径还是相对路径
char* p1 = strstr(str, ":\\");
char* p2 = strstr(str, ":/");

if (p1 || p2) // 输入的是绝对路径
{
    // void _splitpath( const char *path, char *drive, char *dir, char *fname, char *ext);
    // 第一个是待处理的完整的文件名路径
    // 四个参数分别代表四个需要从原始文件路径中截取的字符串，有驱动器盘符(drive)，中间的路径（dir），文件名(fname)，和后缀名（ext）
    _splitpath(str, disk, dir, fname, ext);

    // 判断disk在电脑中是否存在
    if (access(disk, 0) != 0)
    {
        printf("%s is no exist! disk在电脑中不存在 \n", disk);
        return 0;
    }
```

```
        // 判断路径中是否有非法字符
        if (strstr(dir, "*") || strstr(dir, "?") || strstr(dir, "<") || strstr(dir, ">") || strstr(dir, "|") ||
strstr(dir, ":"))
            return 0;
        if (strstr(fname, "*") || strstr(fname, "?") || strstr(fname, "<") || strstr(fname, ">") ||
strstr(fname, "|") || strstr(dir, ":"))
            return 0;

        // 储存路径、文件名
        // void _makepath( const char *path, char *drive, char *dir, char *fname, char *ext);
#include <direct.h>
        // 第一个是要储存的的完整的文件名路径
        // 四个参数分别代表四个需要从结合字符串，有驱动器盘符(drive)，中间的路径（dir），文件名(fname)，和后缀名（ext）
        _makepath(way, disk, dir, NULL, NULL);
        _makepath(file, NULL, NULL, fname, ext);

        strcpy(data->filesavepath, way);
        strcpy(data->filename, file);

        if (access(data->filesavepath, 0) == -1) // 路径不存在，创建新路径
            _mkdir(data->filesavepath);
    }



    else if (p1 == NULL || p2 == NULL) // 仅仅是文件名
    {
        if (strstr(str, "*") || strstr(str, "?") || strstr(str, "<") || strstr(str, ">") || strstr(str, "|") ||
strstr(str, ":"))
            return 0;
        _splitpath(str, disk, dir, fname, ext);

        _makepath(way, disk, dir, NULL, NULL);
        _makepath(file, NULL, NULL, fname, ext);

        strcpy(data->filesavepath, way);
        strcpy(data->filename, file);

        if (access(data->filesavepath, 0) == -1) // 路径不存在，创建新路径
            _mkdir(data->filesavepath);

    }
```

```
        else // 相对路径
        {
            if (strstr(str, "*") || strstr(str, "?") || strstr(str, "<") || strstr(str, ">") || strstr(str, "|") ||
    strstr(dir, ":"))
                return 0;

            // 储存路径、文件名
            _splitpath(str, disk, dir, fname, ext);

            _makepath(way, disk, dir, NULL, NULL);
            _makepath(file, NULL, NULL, fname, ext);

            strcpy(data->filesavepath, way);
            strcpy(data->filename, file);
            _chdir(dir);
        }
        return 1;
}
```

3.2 配置文件

配置文件存储操作系统和应用程序的设置（**配置参数**和**初始设置**）。

配置文件可用于存储配置信息：变量，缺省值，偏好和其他详细信息的状态。

配置文件一般对用户是**隐藏**的，在更改程序首选项时由相应的应用程序修改。配置文件也可以称为首选项文件或设置文件。

常用配置文件扩展名：.INI、.CFG、.CONF。

3.3 工作目录与路径

3.3.1 工作目录

工作目录概念：

与每个程序相关联的一个**动态目录，**也叫**当前工作目录**（current working directory）。程序通过**文件名**或者**相对路径**(未指明根目录)访问文件时，文件名或相对路径被解释为**相对于程序的当前工作目录**。

在 VS 软件中调试运行时：

IDE 环境将 **cpp 文件所在目录**设定为默认的工作目录。

在资源管理器里运行程序时：

操作系统把程序的工作目录设定为**可执行程序（exe 文件）所在目录**。

3.3.2 相对路径与绝对路径

相对路径：

由这个文件所在的路径引起的跟其它文件（或文件夹）的路径关系。

相对于当前工作目录。

如何书写：

“../” 或 “..\\”：向上走一级目录。

“<子目录名>/” 或 ”<子目录名>\\”：向下走一级目录。

例：设当前工作目录为：D:\aaa\bbb\，则：

程序中相对路径表示	实际文件位置
111.txt	D:\aaa\bbb\ 111.txt
ccc\\222.txt	D:\aaa\bbb\ ccc\222.txt
ccc/ddd/333.txt	D:\aaa\bbb\ ccc\ddd\333.txt
../444.txt	D:\aaa\444.txt
..\\555.txt	D:\aaa\555.txt
../../666.txt	D:\666.txt
../..\\777.txt	D:\777.txt

绝对路径：

直接从根目录开始的完整路径信息。绝对路径不依赖工作目录。

3.4 动态内存申请结构体数组

3.4.1 动态内存申请

malloc()函数：

原型：void *malloc(size_t n); /*size_t 是无符号整型*/

分配一块不小于 n 的存储，返回其地址。无法满足时返回空指针值。

```
number = (DATAITEM*)malloc(data->number * sizeof(DATAITEM));
```

动态分配必须检查成功与否

动态存储释放函数 free。

原型：void free(void *p);

free 释放 p 指的存储块。

当内存不再使用时，务必使用 free()函数将内存块释放。

3.4.2 结构体

无效结构定义：

结构成员不能是被描述的结构本身。

非法结构描述的例子：

```
struct invalid
{
    int n;
    struct invalid iv;
};
```

结构变量的使用：

设有 POINT pt1, pt2;　　CIRCLE circ1, circl2;

整体赋值

同类型结构变量可整体赋值，效果是各成员分别赋值：pt2 = pt1;

结构不能做相等/不等比较不等比较

成员访问

访问结构成员用圆点运算符（访问结构成员用圆点运算符（.），具有最高优先级，自左向右结合）

例：

```
pt2.y = pt1.y + 2.4;
circ1.center.x = 2.07;
[circ1.center.y = pt1.y;
```

结构指针要点：

1.必须将指针指向一个确定的结构变量，如：p=&a; 或动态内存分配实现。

2.通过结构变量的指针访问结构的方法为: a.num、(*p). num、p -> num

3.区分下面的两种用法：

p ->n++ 和 ++p -> n　等价于　(p -> n)++ 和 ++(p -> n)

4.p++或++p 是指针 p 指向数组的下个元素,而这个元素是由结而这个元素是由结构体类型数据组成的，它不是一个简单的变量。

3.4.3 动态内存申请结构体数组

范例如下：

```
typedef struct data_type
{
    int age;
    char name[20];
} data;

data *bob = NULL;
bob = (data *)malloc(sizeof(data));
if (bob != NULL)
{
    bob->age = 22;
    strcpy(bob->name, "Robert");
    printf("%sis%dyearsold\n", bob->name, bob->age);
}
else
{
    printf("mallocerror!\n");
    exit(-1);
}
free(bob);
bob = NULL;
```

4 实验 4 部分

4.1 外部程序调用函数——system 函数

system 是一个 C/C++ 语言的函数。windows 操作系统下 system( )函数详解主要是在 C 语言中的应用，system 函数需加**头文件<stdlib.h>**后方可调用。

功能：发出一个 DOS 命令

用法：int system(char *command);

system("pause") 可以实现冻结屏幕，便于观察程序的执行结果；

system("CLS") 可以实现清屏操作。

例：调用外部程序

```
//    Hello.c -- Hello.exe
#include <stdio.h>
int main( )
{
    printf("Hello world!/n");
    return 0;
}
// ------------------------------------
#include <stdlib.h>
#include <stdio.h>
int main( )
{
    system("Hello.exe");
    return 0;
}
```

4.2 程序计时技术——clock( )函数

C 语言中求程序执行的时间可以使用 clock()函数，这个函数返回从“开启这个程序进程”到“程序中调用 clock()函数”时之间的 CPU 时钟计时单元（clock tick）数，其类型为 clock_t，头文件为<**time.h**>

使用范例如下：

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
int main()
{
    long i = 100000000;
    clock_t start, finish;
    double TheTimes;
    printf("做%ld 次空循环需要的时间为", i);
    start = clock();
    while (i--);
    finish = clock();
    TheTimes = (double)(finish - start) / CLOCKS_PER_SEC;
    printf("%fs\n", TheTimes);
    return 0;
}
```

4.3 链表（重点!!!!!）

4.3.1 基本知识

采用链式数据结构：

一环扣一环，通过一个元素保存的其它元素的地址找到其它元素。

链表：

物理上：一系列非连续、非顺序的存储单元；

逻辑上：一个有序的元素序列；

特点：**逻辑上有序，物理上无序！**

动态：链表是一个结点一个结点，一步步构造出来的，无法像数组一样在定义时一次性初始化。

举例：

```
struct UserAccount
{
    char UserNO[15];
    char Name[20];
    char ID[19];
    char Gender;
    double Balance;
    struct UserAccount *pNextUser;
};
```

数据域

指针域：用来保存下一个结点的地址

采用**结构体的自引用方式**实现。

链表：由一系列结点组成；

结点：以结构体方式实现；

每个结点包括两个部分：

存储数据元素的**数据域**

存储下一个结点地址的**指针域**；

所有结点（结构）由**动态分配创建**。从指向表首结点的指针出发，沿链接可顺序访问表中各结点。该指针代表整个表。通常把**最后结点的指针置空**表示结束。

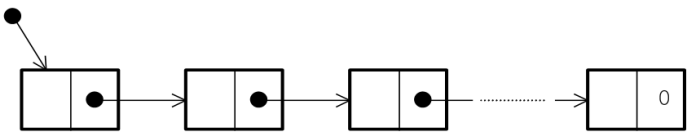


图 7　单向链表示意图

更好的方法：数据域单独声明（不必强求掌握）

```
typedef struct UserInfo
{
    char szID[11];    // ID
    char szName[11]; //  姓名
} USERINFO;

typedef struct UserLinkNode
{
    USERINFO Data; //  数据
    //  下一结点指针
    USERLINKNODE *pNextUser;
} USERLINKNODE;
```

无效结构定义：

结构成员不能是被描述的结构本身。

非法结构描述的例子：

```
struct invalid
{
    int n;
    struct invalid iv;
};
```

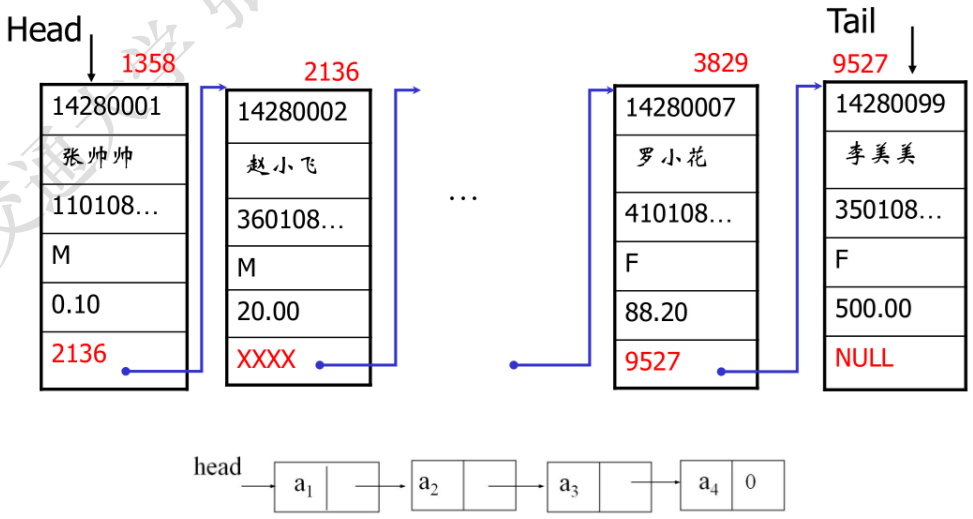


图 8　单向链表举例示意图

单向链表初始化：

①**声明链表结点**数据结构；

②声明链表头**指针变量**；

③声明**第一个链表结点变量**，填充结点数据并将结点变量的地址赋值给链表头指针；

④依次声明链表**后续结点**，并将它们**串联起来**；

⑤使用初始化好的链表开展业务操作，完成业务功能

```
typedef struct UserAccount
{
    char UserNO[15];
    char Name[20];
    char ID[19];
    char Gender;
    double Balance;
    struct UserAccount *pNextUser;
} Node;

int main()
{
    int n = 10;
    Node *head, *tail;
    head = NULL;
    tail = NULL;
    initLink(head, tail, n);
    return 0;
}
```

```
void initLink(Node *h, Node *t, int n)
{
    for (int i = 0; i < n; i++)
    {
        Node *new_node;
        new_node =
            (Node*)malloc(sizeof(Node));
      getInfo(new_node); //填写数据域
        if (h == NULL)
        {
            h = new_node;
            t = h;
         new_node->pNextUser =NULL;
        }
        else
        {
            t->pNextUser = new_node;
            t = t->pNextUser;
            t->pNextUser = NULL;
        }
    }
}
```

4.3.2 链表的插入和删除

将新结点作为最后一个元素增加到链表的尾部：

如果链表为空，将新的结点当成头结点和尾结点；

如果链表不为空，则将该结点作为尾结点的下一个结点，修改尾结点指针；

如果没有尾结点指针，则需要从头结点开始找到最后一个结点。

图 9　链表插入尾部示意图

```
int AddUserToTail( Node *pHead, //链表头指针
Node *pTail, //链表尾指针
Node *pNewUser//新结点指针
)
{
    if (pNewUser == NULL)
        return -1; //  插入失败
    if (pHead == NULL)
    { //  如果还没有元素
        pHead = pNewUser;
        pTail = pHead; //  头尾指针指向同一个结点
    }
    else
    {                                   //  把结点信息附到链表尾部
        pTail->pNextUser = pNewUser; //  加到尾部
        pTail = pTail->pNextUser;     //  修改尾结点指针
    }
    pTail->pNextUser = NULL; //  最后一个结点的后继置空
    return 1;                   //  插入成功
}
```

将新结点作为第一个元素增加到链表的头部：

如果链表为空，将新的结点当成头结点和尾结点；

如果链表不为空，需要将**新结点的下一个结点置为原来的头结点**。

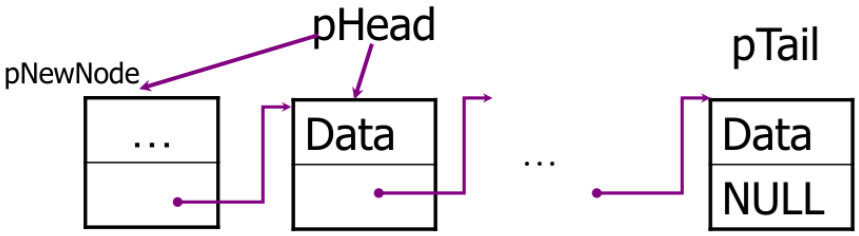


图 10　链表插入头部示意图

```
int AddUserToHead(struct UserAccount *pHead, //链表头指针
struct UserAccount *pTail, //链表尾指针
struct UserAccount *pNewUser//新结点指针
)
{
    if (pNewUser == NULL)
        return -1; //  插入失败
    if (pHead == NULL)
    { //  如果还没有元素
        pHead = pNewUser;
        pTail = pHead; //  头尾指针指向同一个结点
        pTail->pNextUser = NULL;
    }
    else
    {                                   //  把结点信息附到链表头部
        pNewUser->pNextUser = pHead; //  加到头部
        pHead = pNewUser;               //  修改头结点指针
    }
    return 1; //  插入成功
}
```

将新结点按照某种次序要求插入到指定位置：

使用两个工作指针 s 和 t；

s 指针：指向链表上待插入位置前面的结点；

t 指针：指向待插入结点；

插入操作：**t->pNextUser = s->pNextUser;**

s->pNextUser = t;

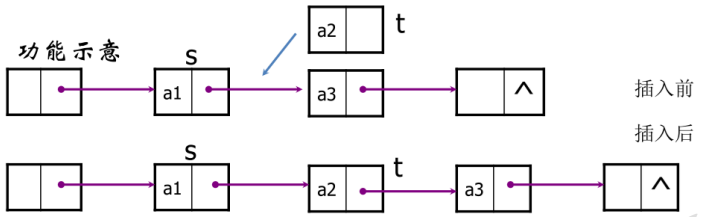


图 11　链表插入头指定位置示意图

删除链表中的结点：

使用两个工作指针 s 和 t；

s 指针：指向待删除结点的前序结点；

t 指针：指向待删除结点；

删除操作：**s->pNextUser = t->pNextUser;**

注意处理删除的结点！

```
s->pNextUser = t->pNextUser;
free(t);
t = NULL;
```

遍历链表：

```
int ShowData(struct UserAccount *pHead) // 给出头指针
{
    struct UserAccount *pNode; // 用于遍历的指针变量
    if (pHead == NULL)
        return -1;
    for (pNode = pHead;         // 从第一个元素开始
         pNode != NULL;         // 判断当前结点是否为空
         pNode = pNode->pNext // 准备处理下一个结点
    )
    {
        // 对每个结点出输出信息
        printf("%11s\t%11s\t\n", pNode->Data.szID,
               pNode->Data.szName);
    }
    return 1;
}
```

4.3.3 链表的优缺点

链表优点：

可以克服数组需要预先知道数据大小的缺点；

可以充分利用计算机内存空间，实现灵活的内存动态管理；

允许插入和移除表上任意位置上的结点，非常灵活

链表缺点：

增加了结点的指针域，会有额外空间开销 ；

查找或访问一个结点较为麻烦和低效；

使用上的灵活性带来的代码阅读和理解上的困难

链表使用技巧：

设计合适的链表结点结构体；

谨慎并规范化地使用链表中的指针，避免各类指针“飞掉”，务必保证链表的各个结点是能“被找到”的；

一定要注意链表结点空间的释放问题。

4.4 排序（重点!!!!!）

4.4.1 冒泡排序

基本排序：

```
#include <stdio.h>
#define n 5
int main( )
{
    int a[n], i, j, med;
    for (i = 0; i < n; i++)
        scanf("%d", &a[i]);
    for (i = 0; i < n; i++)
        for (j = 0; j < n - 1; j++)
            if (a[j] > a[j + 1])
            {
                med = a[j];
                a[j] = a[j + 1];
                a[j + 1] = med;
            }
    for (i = 0; i < n; i++)
        printf("%6d", a[i]);
    return 0;}
```

链表冒泡排序：

```
/*
 *函数名称：sortLink1
 *函数功能：链表排序——冒泡排序
 *输入参数：int n  生成数据条数
 *          confvod* conf  配置文件参数
 *返回值：void
 *版本信息：create by Lifeng Zhang, 2023-05-16
 *          repair by Lifeng Zhang, 2023-05-22
 */
void sortLink1(int n, confvod* conf)
{
    int i, j, cnt = 0;
    LINKNODE* A, * B = NULL, * t = NULL;
    A = (LINKNODE*)malloc(sizeof(LINKNODE));
    start = clock();
    for (i = 0; i < n; ++i)
    {
        A = conf->LinkHead->next; // A指向第一个结点
        B = A->next;
        t = conf->LinkHead;
        for (j = 0; j < n - i - 1; ++j)
        {
            if (A->record.item3 < B->record.item3)
            {
                A->next = B->next;
                B->next = A;
                t->next = B;
            }
            t = t->next;
            A = t->next;
            B = A->next;
        }
    }
    stop = clock();
    duration = (double)(stop - start) / CLOCKS_PER_SEC;
    printf("\n\n链表排序——冒泡排序用时：%f s \n\n", duration);
    system("pause");   //暂停显示结果
}
```

4.4.2 快排

排序函数 qsort 在<stdlib.h>里定义。

void qsort(void *base, size_t n, size_t size, int (*cmp)(const void *, const void *));

base: void 指针参数是待排序数据组首地址

（qsort 不知道数组元素类型）

size_t：无符号整数类型 unsigned int

n: 数组中待排序元素个数

size : 数组中待排序元素的占用空间大小

cmp: 指向函数的指针（实际运用中函数名就是函数指针）

用 qsort 时必须通知所需的排列方式（比较元素大小的准则），它把“较小”元素排在前面

cmp 是比较准则，实际参数必须符合 qsort 对比较函数的类型要求和功能要求

要对数组 students 里的 sum 个学生记录排序，调用形式（scrcmp 是自定义的比较函数，下面讨论）：

qsort(students, snum, sizeof(StuRec), scrcmp);

将比较学生分数函数取名 scrcmp，其原型应是：

int scrcmp(const void *vp1, const void *vp2);

返回值必须是 int,两个参数类型必须都是 const void *

如果升序，比较函数在 vp1 所指对象的值“大于” vp2 所指对象的值时返回正值，两元素相等时返回 0， vp1 所指对象的值“小于” vp2 所指对象的值时返回负值。

“大小”关系应根据排序需要确定

在函数体内要对 vp1 ， vp2 进行强制类型转换后才能得到正确的返回值，不同的类型有不同的处理方法。

qsort 对 int 数组排序

```c
#include <stdio.h>
#include <stdlib.h>
int cmp(const void *a, const void *b)
{
    return (*(int *)a - *(int *)b); //  升序
    // return(*(int *)b-*(int *)a);   降序
}
int main()
{
    int s[10000], n, i;
    scanf("%d", &n);
    for (i = 0; i < n; i++)
        scanf("%d", &s[i]);
    qsort(s, n, sizeof(s[0]), cmp);
    for (i = 0; i < n; i++)
        printf("%d ", s[i]);
    printf("\n");
    system("pause");
    return 0;
}
```

qsort 对 double 数组排序

```
#include <stdio.h>
#include <stdlib.h>
int cmp(const void *a, const void *b)
{
    return ((*(double *)a - *(double *)b > 0) ? 1 : -1);
}
int main()
{
    double s[1000];
    int i, n;
    scanf("%d", &n);
    for (i = 0; i < n; i++)
        scanf("%lf", &s[i]);
    qsort(s, n, sizeof(s[0]), cmp);
    for (i = 0; i < n; i++)
        printf("%.2lf ", s[i]);
    printf("\n");
    system("pause");
    return (0);
}
```

/*本来是因为要判断如果 a==b 返回 0 的，但是严格来说，两个 double 数是不可能相等的，只能说 fabs(a-b)<1e-20 之类的这样来判断，所以这里只返回了 1 和-1 */

qsort 对一个字符数组排序

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
int cmp(const void *a, const void *b)
{
    return (*(char *)a - *(char *)b);
}
int main()
{
    char s[10000], i, n;
    scanf("%s", s);
    n = strlen(s);
    qsort(s, n, sizeof(s[0]), cmp);
    printf("%s", s);
    printf("\n");
    system("pause");
    return 0;
}
```

qsort 对结构体排序

```
#include <stdio.h>
#include <stdlib.h>
struct node
{
    double data;
    int no;
} s[100];
int cmp(const void *a, const void *b)
{
    struct node *aa = (node *)a;
    struct node *bb = (node *)b;
    return (((aa->data) > (bb->data)) ? 1 : -1);
}
int main()
{
    int i, n;
    scanf("%d", &n);
    for (i = 0; i < n; i++)
    {
        s[i].no = i + 1;
        scanf("%lf", &s[i].data);
    }
    qsort(s, n, sizeof(s[0]), cmp);

    for (i = 0; i < n; i++)
        printf("%d %lf\n", s[i].no, s[i].data);
    system("pause");
    return 0;
}
```

qsort 原理对链表排序

```
/*
 *函数名称：LINKNODE* getpar
 *函数功能：链表排序——快排比较程序1
 *输入参数：LINKNODE* begin, LINKNODE* end
 *返回值：LINKNODE*
 *版本信息：create by Lifeng Zhang, 2023-05-16
 */
LINKNODE* getpar(LINKNODE* begin, LINKNODE* end)
{
    int key = begin->record.item3;
    int t;
    LINKNODE* p = begin;
    LINKNODE* q = p->next;


    while (q != end)
    {
        if (q->record.item3 > key)
        {
            p = p->next;
            // 进行值交换
            t = p->record.item1;
            p->record.item1 = q->record.item1;
            q->record.item1 = t;
            t = p->record.item2;
            p->record.item2 = q->record.item2;
            q->record.item2 = t;
            t = p->record.item3;
            p->record.item3 = q->record.item3;
            q->record.item3 = t;
        }
        q = q->next;
    }
```

```
    t = p->record.item1;
    p->record.item1 = begin->record.item1;
    begin->record.item1 = t;
    t = p->record.item2;
    p->record.item2 = begin->record.item2;
    begin->record.item2 = t;
    t = p->record.item3;
    p->record.item3 = begin->record.item3;
    begin->record.item3 = t;
    return p;
}


/*
 *函数名称：void sortLink2
 *函数功能：链表排序——快排比较程序2
 *输入参数：LINKNODE* begin, LINKNODE* end
 *返回值：void
 *版本信息：create by Lifeng Zhang, 2023-05-16
 *          repair by Lifeng Zhang, 2023-05-22
 */
void sortLink2(LINKNODE* begin, LINKNODE* end) //  进行值交换
{

    if (begin != end) //  递归
    {
        LINKNODE* par = getpar(begin, end);
        sortLink2(begin, par);
        sortLink2(par->next, end);
    }
}
```

5 其他部分

5.1 文件读取

5.1.1 基本知识

文件结构－FILE

FILE：结构类型

用 typedef 定义，stdio.h

文件类型指针：FILE * fp

fopen 打开文件

返回 FILE 指针

若文件打开出错，fopen 返回空指针值

通过这种指针可进行各种文件操作。

原型：FILE *fopen(const char *filename, const char *mode)

filename 是文件名；

mode 指明文件打开方式。

	+(读/写)	b(二进制)	b +r
只读：r	r+	rb	rb+w
只写：w	w+	wb	wb+a
追加：a	a+	ab	ab+

文件打开以后必须要检查指针是否为空，以确保正确地打开了一个文件。

```
if ((fp = fopen("filename", "r")) == NULL)
{
    printf("Can't open this file! \n");
    exit(1);
}
```

关闭文件

能同时打开的文件数有限，文件用完后应关闭流数关闭文件用函数 fclose

原型：int fclose(FILE * stream)

正常时返回 0 ，否则返回 EOF。 。

对于输出流，即执行写操作的文件，关闭前将输出流缓冲区的数据送入文件；而后释放缓冲区。

对于输入流，即打开只用来读的文件，关闭时丢弃文件缓冲区中的内容。

程序退出时所有文件将自动关闭

5.1.2 常用函数

字符读写函数：

int fgetc(FILE *fp)

int fputc (char ch,FILE *fp)

```
#include <stdio.h>
#include <stdlib.h>
void main()
{
    char c;
    FILE *fp;
    if ((fp = fopen("f1.txt", "r")) == NULL)
    {
        printf( “Can’t open this file ! \n ”);
        exit(1);
    }
    while ((c = fgetc(fp)) != EOF)
        printf("%c", c);
    printf("\n");
    fclose(fp);
}
```

行式读写的函数：

char *fgets(char *buff, int n, FILE*fp)

int fputs(char *str, FILE *fp)

```
#include <stdio.h>
#include <stdlib.h>
void main()
{
    FILE *fp;
    char str[10];
    if ((fp = fopen("c3.txt", "r")) == NULL)
    {
        printf("Cannot open file!");
        exit(1);
    }
    fgets(str, 11, fp);
    puts(str);
    fclose(fp);
}
```

```
#include <stdio.h>
#include <stdlib.h>
void main()
{
    FILE *fp;
    char st[20];
    if ((fp = fopen("c3.txt", "w+")) == NULL)
    {
        printf("Cannot open file!");
        exit(1);
    }
    printf("input a string:\n");
    gets(st);
    fputs(st, fp);
    fclose(fp);
}
```

格式化读写函数：

fscanf(文件指针，格式字符串，输入地址表列)

fscanf(fp,“%d%s”,&i,s);

fprintf(文件指针，格式字符串，输出表列)

fprintf(fp, "%d%s", i, s);

```
#include <stdio.h>
#include <stdlib.h>
#define n 6
void main()
{
    FILE *fp;
    int i, x;
if((fp=fopen("f1.dat", "w+") )==NULL)
{
        printf("Can’t open this file! \n");
        exit(1); }
for (i = 0; i < n; i++)
{
        scanf("%d", &x);
        fprintf(fp, "%6d", x);
}
fclose(fp);

if((fp=fopen("f1.dat", "r+") )==NULL)
{
        printf("Can’t open this file! \n");
        exit(2); }
for(i=0;i<n;i++)
{
        fscanf(fp, "%d", &x);
        printf("%6d", x); }
printf("\n");
fclose(fp);
}
```

读写数据块的函数：

int fread(char *buffer, int size, int count, FILE *fp)

buffer：读 存放读/ 写数据的起始地址。

size：数据块的字节数。

count：读 读/ 写的数据块块数。

fp：文件指针。

函数调用成功：返回 count 的值

int fwrite(char *buffer, int size, int count, FILE *fp)

例：fwrite(fa, 4, 5, fp);

从 fa 所指空间中，每次输出 4 个字节到（一个实数）到 fp 所指的文件中，写 连续写 5 次，即写 5 个实数到 fp 所指文件中。

5.2 常用函数

ferror( )函数——测试文件操作是否出错

ferror(文件指针)

功能:判断被操作文件的当前状态，返回

0 值——未出错

非 0 值——出错

char *strrchr(const char *str, int c) <string.h>

功能：在参数 str 所指向的字符串中搜索最后一次出现字符 c（一个无符号字符）的位置。

返回值：该函数返回 str 中最后一次出现字符 c 的位置。如果未找到该值，则函数返回一个空指针。

举例：strrchr(path,‘\\’);

strrchr(path,'/');

char *strstr(const char *haystack, const char *needle)　　<string.h>

功能：在字符串 haystack 中查找第一次出现字符串 needle 的位置，不包含终止符'\0’。

返回值：该函数返回在 haystack 中第一次出现 needle 字符串的位置，如果未找到则返回 null。

举例：strstr(path,":\\");

strstr(path,":/");

char *strncpy(char *dest, const char *src, size_t n)　　<string.h>

功能：把 src 所指向的字符串复制到 dest，最多复制 n 个字符。当 src 的长度小于 n 时，dest 的剩余部分将用空字节填充。

返回值：该函数返回最终复制的字符串。

int access(const char *filenpath, int mode);

功能：mode 为 0 时表示检查文件的存在性，如果文件存在，返回 0，不存在，返回-1。

Filenpath 文件或文件夹的路径，当前目录直接使用文件或文件夹名

头文件：io.h

举例：

if (access("conf.ini", 0) == 0) //检查配置文件是否存在

if(access(conf->filesavepath,0)==0) //检查文件存放路径是否存在

char *strcat(char *dest, const char *src)

把 src 所指向的字符串追加到 dest 所指向的字符串的结尾。

char *_getcwd(char *buffer, int maxlen)

可用来获取程序的当前工作目录。

_mkdir()

创建文件路径（建文件夹）

_chdir()

改变工作路径

void _splitpath(const char *path, char *drive, char *dir, char *fname, char *ext)

头文件是#include <direct.h>。

第一个是待处理的完整的文件名路径，四个参数分别代表四个需要从原始文件路径中截取的字符串，有驱动器盘符(drive)，中间的路径（dir），文件名(fname)，和后缀名（ext）

void _makepath(const char *path, char *drive, char *dir, char *fname, char *ext)

头文件是#include <direct.h>。

第一个是要储存的的完整的文件名路径，四个参数分别代表四个需要从结合字符串，有驱动器盘符(drive)，中间的路径（dir），文件名(fname)，和后缀名（ext）

atoi(str*);

将字符串数字转化为 int 型数字

5.3 .cpp 文件与.h 文件、代码编写规范

5.3.1 cpp 文件与.h 文件

在 C 语言编程中，将要实现的应用写成.c 文件。

系统级的应用，我们会写成**含有 main 函数的.c 文件**，来实现系统级的函数调用，已达成我们所要的功能；**具体的各个功能模块，我们习惯写成单独的.c 文件**，然后在主程序 main 函数之前，会 include 到所需模块的.h 头文件中。

这样会使软件组织结构清晰明了，便于各个模块的调试工作，提高了工作效率。

.h 文件中一般是声明，**包括：变量声明、宏定义、枚举声明、结构体声明、函数声明等。不要在.h 文件中定义变量，但可以声明变量。**如果其他模块想要调用该模块的变量和函数，直接包含该模块的头文件即可。[3]

5.3.2 代码编写规范

每个用户的文件级代码模块必须编写一个说明；模块说明要求以注释的形式出现在模块的首部；为了减少每次的工作，每个人应先编写好不变部分，每新建一个模块将不变部分复制到文件首部，再行更改可变部分。

代码注释要恰到好处。代码注释作用如下：

对重要变量说明其作用；对每一段函数代码加注释说明功能；

对重要的语句加注释说明其功能；

对重要的函数调用的参数加实际参数说明；

在自己认为必要的地方加上注释；

可以采用/* */和//进行注释。

根据语句间的层次关系采用缩进格式书写程序，每进一层，往后缩进一层；函数内的变量声明与执行语句要缩进一层；缩进长函数调用语句；

在逗号后面和语句中间的分号后面加空格，如：

```
int i, j, k;
for (i = 0; i < n; i++)
Result = func(a, b, c);
```

在二目运算符的两边各留一个空格，如：

```
a > b          a <= b          i = 0
```

关键字两侧，如 if () …，不要写成 if() …；

类型与指针说明符之间一定要加空格：char *szName;

在结构成员引用符号.和->左右两加不加空格：pStud->szName, Student.nID

函数的变量说明与执行语句之间加上空行;

每个函数内的主要功能块之间加空行表示区隔;

一行语句不要写的太长，将长语句分成多行写;

不要在一行中写多条语句。

标识符命名：

标识符包括**符号常量、变量、函数名、类型名、成员名、C++中的类名**等需要编程者命名的各种文字符号；

标识符命名必须符合语法规则，命名最好能有一定的含义，并且尽量采用英文。

符号常量的命名用大写字母表示，如果符号常量由多个单词构成，两个不同的单词之间可以用下划线连接：#define MAX_LEN 50；

变量名一般需要反映变量的用途，如果变量名由多个单词构成，每个单词的首字符要大写（驼峰命名法）：int AllData;

C 常见变量前后缀规范（应该不用实记）：

类型	前缀	范例
int	n 或 i	int nSum, iSum;
char	ch	char chTemp;
double	d	double dSum;
float	fl	float flSum;
char *	sz	char *szBuffer;
char []	sz	char szBuffer[100];
point	p	int *pnBuffer;
pointer to pointer	pp	int *ppnBuffer;
array	arr 或 rg	int narr[10], nrg[10];

函数一般情况下应该少于 100 行；

函数定义一定要包含返回类型，没有返回类型要加 void；

函数调用如果过长，则每个实参分别占一行；

写比较表达式时，将常量放在左边；

指针变量总是要初始或重置为 NULL；

使用{}包含复合语句，即使是只有一行。

5.3 程序健壮性与程序可读性

程序的健壮性是程序遇到相关特殊情况处理问题的一种能力。这种能力是指，**程序在遇到无效输入、错误输入、某些外在压力的情况下**，程序能够正确解决问题的程度。

程序的可读性则涉及到**代码的清晰度和易读性**。可读性好的程序容易理解和护，有助于团队协作和后期代码的改进和扩展。编写可读性高的代码可以采用一些良好的编码风格和规范，：

（1）使用有意义的命名：选择描述性的量、函数和类名，避免使用含糊不清或过于简单的名称。

（2）编写注释：在关键部分添加注释，解释代码的意图、算法或实现细节。注释应该清晰明了，帮助其他开发人员理解代码的作用和特殊考虑。

（3）结构化代码：使用适当的缩进、空格和换行来组织代码，使其具有层次结构和可读性。避免冗余的代码和复杂的逻辑。删除不必要的重复、无效或过度复杂的代码，使得代码更加简明扼要。减少嵌套，将长函数拆分为多个小函数以提高可读性。

（4）遵循一致的编码风格：在整个项目中使用一致的编码风格，包括缩进、命名规范、函数等。

（5）预防性维护：在编写代码的同时，考虑后续维护工作的需要。采用模块化设计和良好的代码结构，以降低维护成本并提升可读性。（可以思考一下实验 2 到实验 3 代码的转变）

可读性好的代码通常更易于调试和修复错误，从而提高程序的健壮性。反过来，健壮的程序通常使用一致的编码习惯和结构，使得代码更易于阅读和理解。

5.4 程序设计文档撰写方法

以实验 4 程序设计文档为例：

1.概述

1.1 标识（文档名称与文档编号）

1.2 范围（使用或适用范围）

2.程序设计需求（功能需求）

3.程序详细设计

3.1 功能详细设计（文件功能阐述）

3.2 程序与外部程序协同设计（与外部文件调用关系）

3.3 配置文件设计（参见本文档 3.2 节）

3.4 程序工程文件组织设计（实验的功能、.cpp 文件与.h 文件列举）

3.5 内存数据数据模型设计（如何记录文件中的数据）

3.6 函数接口设计（文件所有函数的名称、作用、输入参数、返回值）

3.7 函数详细设计

（主要函数设计、函数功能、输入参数、返回值、流程图）

程序入口函数 main

程序主函数 run

……

3.8 程序交互设计（如何实现界面交互，提示语是什么等等）

系统主菜单设计

修改配置文件子菜单设计

菜单循环展示设计

用户交互提示信息设计

数据记录显示输出交互设计

……

5.5 多源头工程组织文件的方法与意义

程序工程化组织方式是指将程序开发过程中的各个环节进行**规范化、标准化和自动化**，以提高开发效率、降低维护成本和提升软件质量。常见的程序工程化组织方式包括版本管理、代码审查、持续集成和持续交付等。

多文件、多模块程序开发技术是一种**将程序划分为多个文件或模块**来完成开发任务的方法。它可以使***程序结构更加清晰，便于团队合作和维护***。通常，这种开发方式会涉及到模块化设计、模块间接口定义、编译链接等技术。

在多文件、多模块程序开发中，常用的技术包括：

（1）模块化设计：将程序按照功能或逻辑划分为多个模块，每个模块负责实现一部分功能，并提供必要的接口供其他模块使用。

（2）头文件和源文件分离：将代码的声明与实现分离，通常将声明放在头文件（.h 等）中，将实现放在源文件（.cpp 等）中。**（整个课程所做的事情）**

（3）模块间依赖管理：处理不同模块之间的依赖关系，确保模块之间的正确调用和编译链接。**（注意各模块逻辑关系，尤其是调用关系）**

（4）编译和链接：编译器将源文件编译成目标文件，链接器将多个目标文件组合成可执行程序或库文件。

（5）模块测试和集成测试：对每个模块进行单元测试，确保其功能正确。然后进行集成测试，验证模块之间的协作和整体功能。（在实验 5 中尤其体现）

提高开发效率：程序工程化组织方式可以各种工具，以提高开发团队的协作效率和开发速度。而多文件、多模块程序开发技术能够将**程序拆分为可管理的模块，每个模块专注于特定功能**从而加快整体项目的开发进度。

降低维护成本：通过两种方式方式，管理和维护代码变得更加规范化和系统化，减少了错误和 bug 出现的可能性，并且便于进行版本控制和追踪变更记录。**当需要进行修改或添加新功能时，只需关注相关的模块，不会对整个程序产生影响，降低了维护的难度和风险。**

提升软件质量：两种方式提供了更好的**结构化设计和模块化复用能力**，减少了代码冗余和重复性工作，提高了软件的稳定性、可靠性和可扩展性。

通过采多文件、多模块程序开发技术，可以提高代码的可维护性、可重用性和可测试性，同时降低开发过程中的风险和复杂度。

5.6 小组协作模式开发程序的想法

小组协作是开发程序时非常重要的环节，能够激发团队成员之间的合作和创造力。在小组协作模式下，每个成员承担一定责任，共同努力实现项目目标。

小组协作开发程序时需要**设定明确的项目目标**。小组协作的成功与否很大程度上取决于项目目标的明确性。在开始开发前，需要确保所有成员都知道项目的目标，并且认可这些目标。这将帮助团队更好地规划开发流程、分配任务并分解目标。

小组协作开发程序时需要**进行有效的沟通**。沟通对于小组协作至关重要。在开发过程中，需要定期召开会议，讨论进展、遇到的问题以及下一步行动计划。每个成员都需要提供清晰、详细的反馈和建议，以确保整个团队可以有条理进行后续工作。

小组协作开发程序时应当**分配任务和角色**。每个成员都应该能够承担适合自己水平的任务，如果一个成员遇到了问题，其他成员应该积极协助，并提供支持和指导。在小组协作开发程序时，需要确定每个人的角色和职责也是至关重要的，例如实验 5 中的**项目负责人、过程控制员、文档编制员**等。

小组协作开发程序时应当**组织有效的工作流程**。小组协作需要一定的组织和计划。在项目开始前需要**确定开发进度、任务分配及截止时间**，然后根据这些信息制定相关的计划和工作流程。并且在开发过程中需要及时跟踪项目的进度与质量。

小组协作开发程序时也应当**及时解决冲突**。在小组协作中，也难免会出现意见不一致等问题，此时需要及时处理以及沟通。如果无法解决，则需要借助其他外力如第三者（如学长、导师等）进行指导。

通过小组协作，能够高效地完成程序开发，提升每个成员的团队协作能力。每个成员作为一个团队一部分，都有能力从中学到知识和技能，不断提升自身水平。

参考文献

[1] 杨武杰. C 语言的指针数组. https://yangwujie.github.io/slides/c/01.html. 2023-02.

[2] 蔡泽基✔ᵀᴱᴸ. 指针数组和数组指针（非常易懂）. https://blog.csdn.net/itszok/article/details/121198169. 2021-11-08.

[3] TGrit. C 语言之.c .h 文件的规范说明. https://blog.csdn.net/weixin_44643510/article/details/113823852. 2021-02-16