

day07

Ep01 复习指针和数组

• 数组

- **定义**：实际分配了内存空间
- **声明**：通知编译器名字可用
- 在数组名做加减运算时：会以元素类型为基类型的指针（加/减一个元素）【对于二维数组来说是直接加一行】
- 具有相同数据类型的变量（元素）的集合（数组）
- 储存在栈上，在内存中连续存储。
- 存储特征
 - 不能太大
 - 只需知道数组名（首地址）就可以求出整个数组的地址
 - 数组名+n（下标）* sizeof 数据类型
- 数组的使用

```
int arr[10] = {1,2,3,4,5,6,7,8,9,0};
```

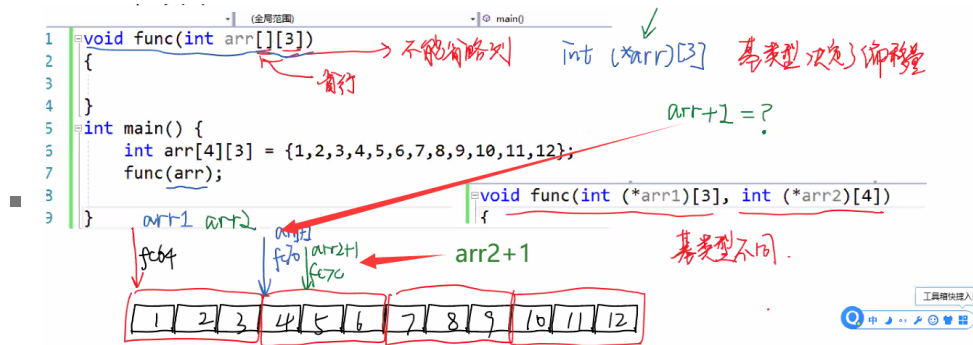
= 初始化运算符

③ arr[3];

[] 访问数组的元素

- 数组的传递
 - 对于函数而言，数组的传递是用指针实现的，传入的是数组的首地址（数组名）
- 指针数组VS数组指针
 - 指针数组=全为指针的 `int *arr[10]`，每个元素都是指针
 - 数组指针=指向数组的指针 `int(*p)[10]`，一个指针变量，指向数组。
- 二维数组

表达式	类型	数值
arr[0]	int[3]	addr
arr[0][0]	int	1
arr	int[4][3]	addr.
arr+1	int(*)[3]	addr+12



- 二维数组本质为一个一维数组，传递时指针的偏移根据基类型决定

• 字符串

- char类型的字符数组 以\0结尾（内存中的0）
- 字符串字面值常量存放在代码段而不是数据段
- `scanf("%s", str) =>` 读取以一个单词，遇到空格结束。

`char str[10]`

`scanf("%s", str);` 单词

- `gets(str);` 读取一行

`fgets(str, sizeof(str), stdin);`
读取一行，读取

字符串操作

- | | | | |
|---------------------|----------------------|----------------------|---------------------|
| <code>strlen</code> | <code>strcpy</code> | <code>strcpy</code> | <code>memset</code> |
| <code>strcat</code> | <code>strncat</code> | <code>memcpy</code> | |
| <code>strcmp</code> | <code>strncmp</code> | <code>memmove</code> | |

这些函数有风险。

2 指针操作:

1. 访问越界

`char str[3];`
`strcpy(str, "hello world");` +



Ep02指针

• 指针滥用

- 指针指向的位置不可写（比如存在字符串常量里）

-
- 指针类型

- 基于指针指向的变量类型
- 指针在使用之前必须初始化 即
 - 指针必须指向已经分配的空间

- 指针的传递

- 函数调用原理

-



- 主调函数传递地址给被调函数
- 被调函数用解引用来修改结束值
- 如果函数传递出来的数据需要修改主函数的值，则应用指针传递数据（传递地址）

Ep03

- 指针II

- 在被调函数内要改变原函数的值，则要用指针传递变量
- 指针的偏移

- 服务于数组
- 可以通过指针的偏移来访问数组中的元素
- 指针的基类型决定指针偏移的程度
- 将堆指针的加减称之为指针的偏移
- 加就是向后偏移，减就是向前偏移

- ```
1 int main()
2 {
3 const int N=5;
4 int a[N]={1,2,3,4,5};
5 //数组名储存的是数组的起始地址
6 int *p;
7 p=a; //初始化指针
8 //此处不需要取地址符
9 //因为数组名=数组首地址=第一个元素的地址
10 for(int i=0;i<5;i++)
11 {
12 printf("%3d",*(p+i));
13 //p = &a[4]; //逆序输出
14 //printf("%3d",*(p-i));
15 }
16 printf("\n");
17
18
19
```

```
20 }
21
```

```
1 void print(int *p)
2 {
3 for(int i=N;i<5;i++)
4 {
5 printf("%3d",*(p+i));
6 }
7
8 }
9 int main()
10 {
11 const int N=5;
12 int a[N]={1,2,3,4,5};
13 int *p;
14 p=a; //初始化指针
15 }
```

- 指针的自增自减运算符

```
1 int main(){
2 int a[3]={2,7,8};
3 int *p;
4 int j;
5 p = a;
6 j=*p++; //j=*p;*p++;
7 printf("a[0]=%d,j=%d,*p=%d\n",a[0],j,*p);
8 // 2, 2, 7
9 j=(*p)++; //先进行j=*p运算 在对(*p)++
10 printf("a[0]=%d,j=%d,*p=%d\n",a[0],j,*p);
11 // 2, 7, 7
12
13 }
```

○

•

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main()
5 {
6 int a[3]={2,7,8};
7 int *p;
8 int j;
9 p=a;
10 j=*p++; //j=*p;p++;
11 printf("a[0]=%d, j=%d, *p=%d\n", a[0], j, *p); //2, 2, 7
12 j=p++; //j=*p; (*p)++;
13 printf("a[1]=%d, j=%d, *p=%d\n", a[1], j, *p);
14 system("pause");
15 }
```

- void型指针

- 传入时需要强制类型转换成char型

- 动态数组

- ```
1  int main()
2  {
3      int needSize;
4      char *pStart;
5      scanf("%d",&needSize);
6      pStart=(char*)malloc(needSize);
7      strcpy(pStart,"Hello");
8      puts(pStart);
9      free(pStart);//要把free后的指针置为NULL
10     pStart=NULL;
11
12 }
```

- 既然都是内存空间，为什么还要分栈和堆呢？栈是机器系统提供的数据结构，计算机会在底层对栈提供支持：分配专门的寄存器存放栈的地址，压栈出栈都有专门的指令执行，这就决定了栈的效率比较高。堆则是C/C++函数库提供的，它的机制是很复杂的，例如为了分配一块内存，库函数会按照一定的算法（具体的算法可以参考数据结构/操作系统）在堆内存中搜索可用的足够大小的空间，如果没有足够大小的空间（可能是由于内存碎片太多），就有可能调用系统功能去增加程序数据段的内存空间，这样就有机会分到足够大小的内存，然后进行返回。显然，堆的效率比栈要低得多。

- 栈空间和堆空间的差异实例

- ```
1 char *printStack()
2 {
3 char c[] = "I am printStack";
4 puts(c);
5 return c;
6 }
7 char *printMalloc
8 {
9 char *p=(char*)malloc(20);
10 strcpy(p,"I am printMalloc");
11 puts(p);
12 return p;
13 }
14 int main()
15 {
16 char *p;
17 p=printStack;
18 put(p);//打印会乱码，
19 //函数调用完毕之后会自动释放栈空间
20 p=printMalloc;
21 put(p); //打印不会乱码，对于堆来说只要不free都是属于堆的空间。
22 free(p);
23 p = NULL;
24 }
```

- 用C实现vector

## realloc

语法:

```
#include <stdlib.h>
void *realloc(void *ptr, size_t size);
```

功能: 函数将`ptr` 对象的储存空间改变为给定的大小`size`。参数`size`可以是任意大小, 大于或小于原尺寸都可以。返回值是指向新空间的指针, 如果错误发生返回NULL。

```
1 #include<stdio>
2 #include<string>
3 #define CAPACITY 20
4 int main()
5 {
6 char *p=(char*)malloc(CAPACITY);
7 char c;
8 int i=0, cap=CAPACITY;
9 while(scanf("%c",&c)!=EOF)
10 {
11 if(i==CAPACITY-1)
12 {
13 cap=2*cap;
14 p=(char*)realloc(p, cap);
15 }
16 p[i]=c;
17 i++;
18 }
19 p[i]=0;
20 puts(p);
21 free(p);
22 }
```

○

- 字符指针&&字符数组初始化

```
1 int main()
2 {
3 char *p='hello';
4 char c[20]="hello";
5 c[0]='H';
6 //p[0]='H';//字符串常量区不可写
7 p="how are u";
8 //c="how are u";//因为c是常量
9
10 }
```

○ strc

# day08

## Ep01 写在前面

- 代码问题: 拿出纸和笔, 分析问题, 提出解决方案并检查和调整。
- 锻炼自己的调试能力
- 找到问题解决, 提问。

- 复习指针和数组
- 二级指针和函数指针
- 讲解习题（103个数找三个有差异的数）

## Ep02 数组指针和二维数组

- 数组指针和二维数组

- 定义

```

1 #include<iostream>
2 #include<string>
3 using namespace std;
4 void print(int p[][4],int row);
5 int main()
6 {
7 int a[3][4]={1,3,5,7,2,4,6,8,9,11,13,15};
8 int (*p)[4];
9 int b[4]={1,2,3,4};
10 //&b+1=>偏移16个字节的下一个数组 基类型是大小为16个字节的数组
11 /*(*(a+1)+1) 得到4的办法 偏移两次
12 //第一次偏移成一个一维数组 第二次再次偏移成一个一维数组中的一个元素
13 p=a;
14 print(a,3);
15 /*
16 p=(int (*)[4]malloc(16*100));
17 p[99][3]=1000; //动态二维数组
18 */
19 }
20 void print(int p[][4],int row)
21 {
22 printf("sizeof(p)=%d\n",sizeof(p));
23 //sizeof只看类型，所以这里p的类型是指针 占4个字节
24 for(int i=0;i<row;i++)
25 {
26 for(int j=0;j<sizeof(*p)/sizeof(int);j++)
27 {
28 printf("%3d",*(p[i]+j)); //比较容易看懂
29 //等价于 *(&p[i])+j
30 }
31 cout<<endl;
32 }
33 }
```

- 

## Ep03 指针和数组复习

- 数组

- 定义：数据类型+数组名+[数组大小]

- 初始化 = {0} //自动推导大小：忽略数组大小，根据初始化列表自动推断

- 调用数组：数组名+[数组下标]

- 数组下标范围0~数组大小-1

- []的原理：数组指针的偏移 `p[i]` 等价于 `*(&p+i)`

- 数组和指针

- 数组名的数据类型？数值？
  - 数组名 = 首个元素的地址 = 数组的地址 = 首地址的指针
  - 类型以**元素类型为基类型的指针**
  - 不能修改指向但是可以修改指向位置的值：const pointer //指针常量
- e

#### ○ 二维数组

- 定义：数据类型+数组名+[行] [列]
- 内存分布：同一行连续存储，行与行之间也连续存储，地址连续。
  - 按行存储
  - 行与行之间也连续存储
- 本质：数组的数组 可以看作是大小为**[行]**的**[列]**个一维数组
- 二维数组数组名的含义：int arr[M] [N]
  - arr：基类型是int[N]的指针。
  - arr+1 地址值增加了 1 \* sizeof (int) \*N
  - 在函数内：`*arr+1` 1 \* sizeof (int) \*N

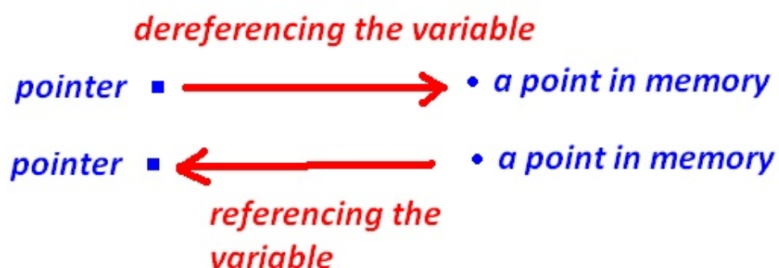
#### ○ 字符数组

- 字符串的特点：以\0结尾的字符数组
  - 不会检查数组的大小
- 字符串的使用问题：
  - char str[20] ="hello world"
  - 使用函数读取的时候：请务必用memset将空间中所有的位置都设置为0；
  - 使用memset清空 `char str[30]`

```
1 char str[30];
2 memset(str,0,sizeof(str));
```
  - 一些函数
    - strlen.. 表格见笔记
  - 大小：什么时候可以用sizeof

### • 指针

- 概念：存放了已分配空间的地址的一个变量
- 定义：基类型 \*指针名
  - 指针的数据类型由基类型决定
  - 解引用、间接访问的空间大小和解释方式
    - 解引用：dereferencing the variable 的过程



- 间接访问：用指针访问变量类型
- 指针的偏移：



- 根据指针的基类型的大小进行地址的增加/减少
- 注意指针的偏移和数组之间的联系。
- 动态数组
  - 分配在堆上
  - (基类型\*)malloc(数组大小 \* sizeof(基类型))
  - 申请: p=(基类型\*)malloc(数组大小 \* sizeof(基类型))
    - 不要越界: 0~ (数组大小-1)
  - 释放 free (p) , 注意此时P的指向不能偏移
  - 注意此时还需将p=NULL 避免野指针。
  - 、
- 指针的传递
  - 在被调函数里修改主函数的值?
    - 1. 主调函数将变量的地址传入被调函数 (实参= 地址)
    - 2. 被调函数用接引用 (间接访问) 的方式修改变量的值
      - 间接访问的方式: \* [] ->

## Ep04 二级指针与函数指针

- 二级指针与函数指针

| 表示形式                                  | 含义                            | 地址值     |
|---------------------------------------|-------------------------------|---------|
| a                                     | 二维数组名, 指向一维数组 a[0] , 即 0 行首地址 | 0x2000  |
| a[0] ,<br>*(a+0),<br>*a               | 0 行 0 列元素地址                   | 0x2000  |
| a+1, &a [1]                           | 1 行首地址                        | 0x2010  |
| a [1] , *(a+1)                        | 1 行 0 列元素 a[1][0] 的地址         | 0x2010  |
| a[1]+2,<br>*(a+1)+2,<br>&a[1][2]      | 1 行 2 列元素 a[1][2] 的地址         | 0x2018  |
| *(a[1]+2),<br>*(*(a+1)+2),<br>a[1][2] | 1 行 2 列元素 a [1] [2] 的值        | 元素值为 13 |

| 表示形式                                                                                     | 含义                                         | 地址值    |
|------------------------------------------------------------------------------------------|--------------------------------------------|--------|
| <code>a</code>                                                                           | 二维数组名，指向一维数组<br><code>a[0]</code> ，即0行的首地址 | 0x2000 |
| <code>a[0]</code> ， <code>*(a+0)</code> ， <code>*a</code>                                | 0行0列的元素地址                                  | 0x2000 |
| <code>a+1</code> ， <code>&amp;a[1]</code>                                                | 1行首地址                                      | 0x2010 |
| <code>a[1]</code> ， <code>*(a+1)</code>                                                  | 1行0列元素 <code>a [1] [0]</code> 的地址          | 0x2010 |
| <code>a[1]+2</code> ， <code>*</code><br><code>(a+1)+2</code> ， <code>&amp;a[1][2]</code> | 1行2列的元素 <code>a[1][2]</code>               | 0x2018 |
| <code>*(a[1]+2)</code> ， <code>*</code><br><code>(a(a+2)+2)</code>                       | 1行2列元素 <code>a[1][2]</code> 的值             | 元素值为13 |

- 每次偏移+4
- 数组指针
  - 二维数组可以看成元素为一维数组的数组
  - 一维数组的数组名在进行偏移和解引用之后，可以看作是以元素类型为基类型的指针
- 如何排序二维数组
  - 数组指针
    - strcpy/ memcpy
  - 二级指针
    - 交换指针排序
  - 函数指针
- 心中要由一幅指针指向的内存图
- 常见问题

```

1 int a[4][4];
2 int *p=a;
3 int *q=a[0];
4 //p&&q的数据类型一样能够说明a个a[0]一样吗?
5 //不一样 p++和q++就可以看出区别
6 int (*p)[4]=a;
7

```

- 二级指针
  - 定义：基类型为指针类型的指针
  - 改变指针的指针
  - 二级指针的传递 //晚上重听

```

1 #include <stdio.h>
2 void change(int *pi) {
3 *pi = 5;
4 }
5 int main() {
6 int i = 10;
7 int j = 5;
8 int *pi = &i;
9 int *pj = &j;
10 printf("i = %d, j = %d, *pi = %d, *pj = %d\n", i, j, *pi, *pj);
11 change(pi);
12 printf("i = %d, j = %d, *pi = %d, *pj = %d\n", i, j, *pi, *pj);
13 }

```

- 改变i的值需要传入i的地址

## • 函数指针（概念复杂，使用简单）

- 定义：存储函数入口地址的指针
- 使用场景：间接引用函数
- 功能：传递

```

1 //定义函数指针
2 //void (*p)(形参);//这是一个指针，返回值为void。
3 void b(){
4 printf("i am ironman \n");
5 }//返回值是void 无形参
6 void a(void(*p)(int))
7 {
8 p(3);
9 }
10 int main(){ //定义的时候要形参
11 void(*p); //调用的时候要传递实参
12 p=b;
13 //p (2);//使用函数指针间接访问b函数
14 a(p);
15 }//函数名单独使用，就是函数指针

```

- 
- 函数指针的实际使用场景
  - 集中某个地址
  - 元素为指针的数组
  - 对某商品进行排序的时候 交换指针比交换结构体简单
  -

## • 使用指针数组排序字符串

```

1 char b[5][10]={"lele","lili","lilei","hanmeimei","zhousi"};
2 char* p[5];
3 for (int i=0;i<5;i++)
4 {
5 p[i]=b[i];//对于二维数组的列下标来说 一个列下标表示一行，每行的开头
6 //元素就是基类型为一整行的指针。
7 }
8 //排序p用冒泡排序
9 char **p2 = p;

```

```

9 for(int i=4;i>=1;--i)
10 {
11 for(int j=0;j<i;++j)
12 {
13 if(strcmp(p2[j],p2[j+1]))
14 {
15 char *tmp=p2[j];
16 p2[j]=p2[j+1];
17 p2[j+1]=tmp; //交换
18 }
19 }
20 print(p2);
21 puts("-----");
22 for(int i=0;i<=4;i++)
23 {
24 puts(b[i]);
25 }
26 }

```

- 二维数组VS二级指针

- 在默认的情况下二者没有联系

- ```

1   int a[4][3];
2   /*
3   a => 数组名 /指向数组的指针
4   a[0] => int[3]/int*
5   a[0][1] => int
6   //二维数组的解引用流程不涉及二级指针
7   */
8

```

-

day09

Ep01 写在前面

- 函数
- 下午复习 完成作业和笔记的修订

Ep02 函数

- 函数定义and函数声明

- 在定义函数之前需要在主程序前声明函数
- 在头文件里声明，可省略声明但是需要引入头文件
- 函数不能嵌套定义但是可以嵌套调用

- 形参实参

- 形参：形式参数，函数定义时的参数
- 实参：实际参数，函数使用时传入函数的参数
- \

- 全局变量

- ```
1 #include<stdio>
2 #include<stdlib>
3 int i = 10;
4 void print()
5 {
6 printf("i am print i =%d\n",i)
7 }
8 int main()
9 {
10 int i = 5;
11 printf("i am main %d\n",i);
12 print();
13 }
```

- 会提高阅读难度

- setjmp

- ```
1 #include<stdio>
2 #include<stdlib>
3 #include<setjmp>
4 // jmp_buf envbuf; //设置中断点
5 void b(jmp_buf envbuf;)
6 {
7     printf("i am func b\n");
8     longjmp(envbuf,5);
9 }
10 void a(jmp_buf envbuf;)
11 {
12     printf("before b,i am");
13     b(envbuf); //通过setjmp跳过下面的语句
14     printf("finish b,i am");
15 }
16 }
17 int main()
18 {
19     jmp_buf envbuf;
20     int ret;
21     ret = setjmp(envbuf);
22     if(0==ret)
23     {
24         a(envbuf);
25     }
26 }
```

- 递归调用

- ```
1 #include<stdio>
2 #include<stdlib>
3
4 int main()
5 {
6 int f(int n);
7 int n;
8 while(scanf("%d",&n)!=EOF)
```

```

9 {
10 //printf("%d!=%d",n,f(n));
11 //非递归实现
12 int first = 0;
13 int second = 1;
14 int third;
15
16 for(int i=0;i<n;i++)
17 {
18 third=first+second;
19 first=second;
20 second=third;
21 }
22 cout<<third<<endl;
23 }
24 }
25 int f(int n) //斐波那契数列
26 {
27 if(n==1) return 1;
28 if(n==2) return 2;
29 return f(n-1)+f(n-2);
30 }
31 /*
32 int f(int n)
33 {
34 if(0==n)
35 {
36 return 1;
37 }
38 return n*f(n-1);
39 }
40 */

```

#### ○ 汉诺塔

```

1 #include<stdio>
2 #include<stdlib>
3 int f(int n);
4 int main()
5 {

```

## day10

### Ep01

#### • 局部变量和全局变量

```

1 //int k=10; //需要放到函数之上 全局变量
2 void print()
3 {
4 printf("k=%d",k)
5 } //此时无法打印k 未声明的标识符

```

```

6 | int k=10;
7 | int main ()
8 | {
9 | int i=10
10 | {
11 | int j=5; //变量在离自己最近的大括号内有效
12 | }
13 | printf("j=%d",j); // 未声明的标识符
14 | }

```

- 全局变量从开始到结束都占用储存空间
- 太多的全局变量容易出错
- 全局变量放在数据段
- 全局变量只能初始化一次
- 借用全局变量的需要加 `extern`
- 局部变量有效范围在最近的 `{ }` 之间

## • 静态和动态存储方式

- 常用变量就是动态变量
- 静态存储在静态区
- 仅初始化一次
- 静态局部变量是在编译时赋初值的，即只赋初值一次
- 用 `static` 修饰全局变量，那么该全局变量将不能被其他文件引用
- 用 `static` 修饰函数，那么该函数将不能被其他文件引用。

```

1 | void print()
2 | {
3 | static int i=0;
4 | i++;
5 | printf("i am print%d",i)
6 | }

```

## Ep02

## • 结构体

- ```

1 | //为了避免浪费空间，在定义结构体的时候要将小字节的放到一起
2 | struct student_t{
3 |     int num;
4 |     char name[20];
5 |     char sex;
6 |     int age;
7 |     float score;
8 |     char addr[30];
9 | };
10 |
11 | int main()
12 | { //构造数据类型
13 |     struct student_t s= {1001,"aki",'M',20,98.5,"shanghai"};
14 |     struct student_t sArr[3];
15 |     printf("%d %s %c %d % 5.2f %s\n",s.num,s.sex,s.age,s.score,s.addr);
16 |     for(int i=0;i<3;i++)

```

```

17     {
18         scanf_s("%d %s %c %d %f
%s"&sArr[i].num,sArr[i].name,&sArr[i].age,&sArr[i].score,sArr[
i].addr)
19     } //输入
20     for(int i=0;i<3;i++)
21     {
22         printf ("%d %10s %c %d %f
%s"sArr[i].num,sArr[i].name,sArr[i].age,sArr[i].score,sArr[i].
addr);
23         //输出
24     }
25
26
27
28 }

```

○ 结构体指针

- 一个结构体变量的指针就是该变量所占据的内存段的起始地址。可以设一个指针变量，用来指向一个结构体变量，此时该指针变量的值是结构体变量的起始地址。

```

1  typedef struct student{
2      int num;
3      char name[20];
4      float score;
5  }student_s, *pstudent_s;
6  //等价于 pstudent_s struct student*
7  int main()
8  {
9      int num;
10     student_s s={1001,"wwe",95.5};
11     student_s sArr[3]={1001,"wwe",95.5,1002,"apex",85.5,1003,"ooc",75.5}
12     pstudent_s p;
13     p=&s;
14     printf("%d %s %f\n",(*p).num,(*p).name,(*p).score);
15     //间接访问
16     num=p->num++; //num=p->num p->num+=1
17     cout<<num<<" "<<p->num;
18     // 1001 1002
19     num=p++->num; //提出++ 先num=p->num 再++p
20     cout<<num<<" "<<p->num;
21     // 1001 1005
22 }

```

● 链表

○ 顺序存储

- 物理连续&&逻辑连续

○ 链式存储

- 逻辑连续不一定物理连续

```

1  struct student{
2
3  }

```


- 增删查改：//晚上重新听 头插法

```
list.h* x list.c main.c* main.c* main.c func.h func.c
(全局范围)
1 #include <stdio.h>
2 #include <stdlib.h>
3
4
5 typedef struct student{
6     int num;
7     struct student *pNext;
8 } Student_t, *pStudent_t;
9
10
```

```
list.c* x main.c* main.c* main.c func.h func.c main.c main.c main.c main.c func.h
1 #include "list.h"
2
3 void listHeadInsert(pStudent_t *ppHead, Student_t **ppTail, int val)
4 {
5     pStudent_t pNew=(pStudent_t) calloc(1, sizeof(Student_t));
6     pNew->num=val;
7     if(NULL==*ppHead)//判断链表是否为空
8     {
9         *ppHead=pNew;
10        *ppTail=pNew;
11    }else{
12        pNew->pNext=*ppHead;
13        *ppHead=pNew;
14    }
15 }
```

```
1 int main()
2 {
3     pStudent_t phead=NULL,ptail=NULL;
4     while(scanf("%d", &num)!=EOF){
5         listHead
6     }
7 }
```

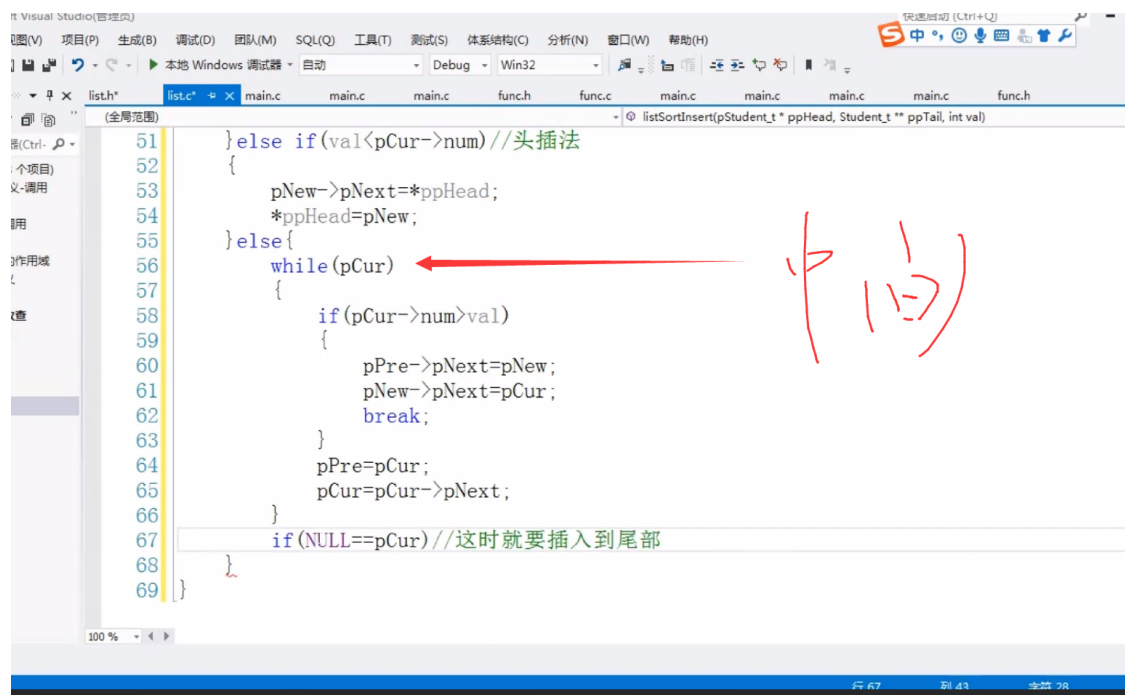
```
2 {
3     *ppHead=pNew;
4     *ppTail=pNew;
5 }else{
6     (*ppTail)->pNext=pNew;
7     *ppTail=pNew;
8 }
9 }
```

```

54         *ppHead=pNew;
55     }else{
56         while(pCur)
57         {
58             if(pCur->num>val)
59             {
60                 pPre->pNext=pNew;
61                 pNew->pNext=pCur;
62                 break;
63             }
64             pPre=pCur;
65             pCur=pCur->pNext;
66         }
67     }

```

• 顺序插入



Ep03 复习

- 函数
- 结构体

- 声明：struct+结构体名{ 成员列表};
- 定义结构体变量
 - struct+结构体名{ 成员列表};
 - 初始化列表
 - sizeof：不是简单相加，而要考虑对齐类型（一般为4字节对齐）
- 访问成员：. 运算符比 * 高
 - 结构体名字可以和变量名相同
 - 支持取地址 & 和赋值 =
- 结构体数组
 - 没啥差别
- 结构体指针

- 访问成员时使用箭头->
- 解引用需要加*()
- typedef: 给类型取别名
- 链表
 - 结点定义

```
1 struct node{
2     int data;
3     struct node* pNext;
4 };
```

- 头插法

```
1 if(){
2
3 }//判空 为空则插入第一个节点
4 else{
5
6     p.tail->p.next;
7
8 }//若非空先链接新节点之后头指针前移
```

- 尾插法

```
1 if(){
2
3 }//判空 为空则插入第一个节点
4 else{
5
6     p.tail->p.next;
7
8 }//若非空先链接新节点之后尾指针后移
```

- 有序插入

```
1 //使用两个指针，找到合适位置和之前位置
2 if(){
3
4 }//判空 为空则插入第一个节点
5 if else{
6     while(){} //找到合适位置和之前位置
7     p.tail->p.next;
8 }
9 else{}
10 //若非空先链接新节点之后尾指针后移
```

Ep01 写在前面

- 函数

- 递归函数：调用自身的函数
- 变量以及作用域
 - static 变量：仅初始化一次，只在本文件内使用。

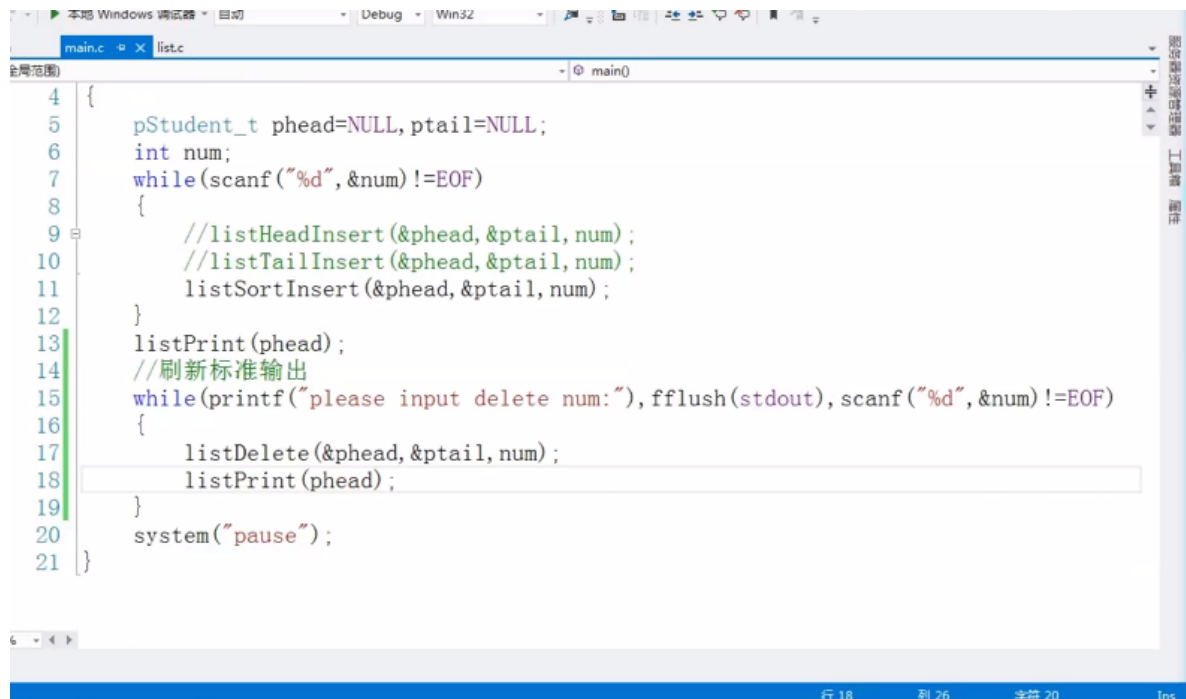
- 链表

- 头插法，
 - 新建节点，插入的值进行初始化
 - 判空，如果为空，则新节点赋值给头尾指针
 - 若不为空，新节点的Next值指向原有头节点
 - 新节点成为新的头节点（头指针）
- 尾插法
 - 新建节点，插入的值进行初始化
 - 判空，如果为空，则新节点赋值给头尾指针
 - 若不为空，新节点的Next值指向原有尾节点
 - 新节点成为新的尾节点（尾指针）
- 有序插入
 - 新建节点，插入的值进行初始化
 - 判空，如果为空，则新节点赋值给头尾指针
 - 若不为空
 - 如果新节点的值小于头指针，则直接头插法
 - 如果大于头指针，则遍历链表
 - 找到比插入值大的节点位置，进行头插法
 - 如果遍历链表未找到
 - 则直接尾插法

Ep02 链表的删除

-

```
    if(NULL==pCur)//没有找到对应结点
    {
        printf("Don't find deleteNum\n");
        return;
    }
    if(pCur==*ppTail)
    {
        *ppTail=pPre;
    }
    free(pCur);
```



```
4 {
5     pStudent_t phead=NULL, ptail=NULL;
6     int num;
7     while(scanf("%d", &num) != EOF)
8     {
9         //listHeadInsert(&phead, &ptail, num);
10        //listTailInsert(&phead, &ptail, num);
11        listSortInsert(&phead, &ptail, num);
12    }
13    listPrint(phead);
14    //刷新标准输出
15    while(printf("please input delete num: "), fflush(stdout), scanf("%d", &num) != EOF)
16    {
17        listDelete(&phead, &ptail, num);
18        listPrint(phead);
19    }
20    system("pause");
21 }
```

Ep03 链表的修改

见代码

Ep04 联合体和枚举

- 联合体

- ```
1 union data{
2 int i;
3 char c;
4 float f;
5 }
6 typedef struct student {
7 int num; //学号
8 float cham;
9 }Student_t, * pStudent_t;
10 int main()
11 {
12 union data a; //
13 student_t s;
14 a.i=10;
15 a.c='A';
16 a.f=98.5;
17 //此时 a.i/a.c/a.f都用一个地址
18 //一个时间内只有一个成员有效
19 s.num=1001;
20 s.cham=1024;
21 //此时二者独立，占不同地址
22 }
```

- 枚举

```

1 enum weekday{sun,mon,tue,wed,thu,fri,sat};
2 int main()
3 {
4 enum weekday workday;
5 workday=1;
6 printf("wed=%d\n",workda);
7
8 }

```

## Ep05 常用数据结构和算法

- 算法的目的是为了提高存取效率
- 常用栈，队列，二叉树，堆，红黑树，哈希表等数据结构的增删改查
- 通过排序算法掌握时间复杂度与空间复杂度
- 掌握常用查找算法
- 算法学习方法讲解
- 在一开始设计好函数的目的是很重要的

## Ep06 栈 stack

- 栈 数据结构的笔记见vs //之后贴出

```

 后进先出
 可以用数组实现，也可以用链表实现
 void init_stack(pStack stack);
 void pop(pStack stack) //出栈
 void push(pStack stack, int val) //入栈
 int top(pStack stack) //返回栈顶元素
 int empty(pStack stack) //判断栈是否为空
 int size(pStack stack) //返回栈中数据的元素个数

```

```

13 }
14 pCur=stack->phead;
15 stack->phead=pCur->pNext;
16 free(pCur);
17 pCur=NULL;
18 }

```

```

2 #include <stdlib.h>
3 #include <string.h>
4
5 #define MaxSize 5
6 typedef int ElemType;
7 typedef struct{
8 ElemType data[MaxSize];
9 int front,rear;
10 }SqQueue_t;
11
12

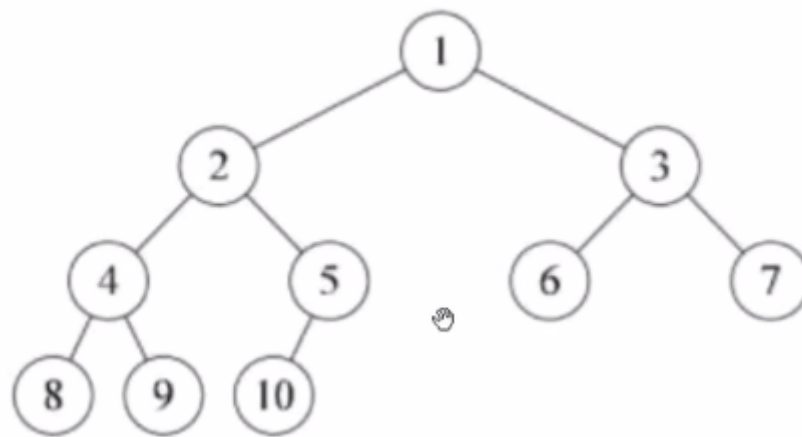
```

## Ep06队列

-

## Ep07二叉树

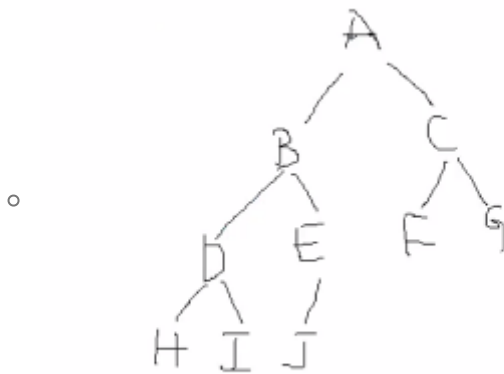
- 



- 二叉树的第  $i$  层至多有  $2^{i-1}$  个结点；深度为  $k$  的二叉树至多有  $2^k - 1$  个结点；对任何一棵二叉树  $T$ ，如果其终端结点数为  $n_0$ ，度为 2 的结点数为  $n_2$ ，则  $n_0 = n_2 + 1$ 。
- BFS
- 堆排
- 度：有几层即度为几
- $k$  层的二叉树最多有  $2^k - 1$  个节点
- 深度为  $k$ ，有  $2^k - 1$  个节点的二叉树叫满二叉树
- 完全二叉树
  - 仅允许最后一层有空节点且空缺在右边
  - 对任意结点，如果其右子树的深度为  $j$ ，则其左子树的深度必为  $j$  或  $j+1$ 。度为 1 的点只有 1 个或 0 个

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 typedef char ElemType;
5
6 typedef struct node_t {
7 ElemType c;
8 struct node_t *pleft;
9 struct node_t *pright;
10 } Node_t, *pNode_t;
```

- 如何正确的分析前序遍历
  - 根+左+右的顺序输出



- ABD HIE JCFG

- 中序遍历

- HDIBJEAFCG 将上图一脚踩扁 按照顺序输出
- 

```

1 //
2
3 void buildBinaryTree(pNode_t*, pQueue_t*, pQueue_t*, ElemType);|

33 void buildBinaryTree(pNode_t* treeRoot, pQueue_t* queHead, pQueue_t* queTail, Elem
34 {
35 pNode_t treeNew=(pNode_t) calloc(1, sizeof(Node_t));
36 pQueue_t queNew=(pQueue_t) calloc(1, sizeof(Queue_t));
37 pQueue_t queCur=*queHead;
38 treeNew->c=val;
39 queNew->insertPos=treeNew;
40 if(NULL==*treeRoot)
41 {
42 *treeRoot=treeNew;
43 *queHead=queNew;
44 *queTail=queNew;
45 }else{
46 (*queTail)->pNext=
47 if(NULL==queCur->insertPos->pleft)
48 {
49 queCur->insertPos->pleft=treeNew;
50 }else if(NULL==queCur->insertPos->pright)

```

## day12

### Ep01

- 最坏时间复杂度T(n)
- 时间复杂度：执行次数了，忽略常数。
- 为什么要学算法：处理较多数据时效率高

### Ep02

-