第二小节，二

1.指针和一维数组间的关系

1.数组名的特殊意义及其在访问数组元素的作用

数组名代表着数组中首个元素的首地址

数组在存储的时候是进行连续存储的

2.指针运算的特殊性及其在访问数组元素中的作用

p+1与p++是两个不同的操作

p+1中p没有改变

p++中p的值发生了改变和移动

p++并非是将指针变量p的值简单的加1，而是加上1*sizeof（基类型）个字节。

3.数组和指针作为函数参数进行模拟按值调用中的相似性

数组名和用一维数组的指针变量作函数实参。向被函数传递的是

数组的起始的地址，都是模拟按引用调用

因此，数组作函数的参数和指针做函数的参数都是在调用变量的函数的地址。

2.指针和二维数组间的关系

1.二维数组的行地址和列地址

首先我们有

数组a【】

和指向a数组的指针p

那么有a的首地址==p的首地址

那么就有a【1】 == *（a+1）；//注意，类型必须相同，否则1，报错2，将会移动位数发生变化

a+2实际上是个地址，*地址==地址上存储的值

看一个大的结论

a【i】【j】 ==(*(a+i))【j】 ==*(a【i】+j) == *(*(a+i)+j)

这是个啥？

等价，地址的等价。

a【i】【j】 ==*(a【i】+j)二维数组的读法是：a[i][j] == a[i]的地址加上j个sizeof个地址，求数值

a【i】【j】 ==(*(a+i))【j】a【i】 == a的地址加上i个sizeof地址，再求解j个地址从而求解

a【i】【j】== *(*(a+i)+j) a【i】【j】 == a的地址加上i个sizeof地址，再求解j个sizeof地址，求解

牢记公式：：

a【i】== *(a+i)

一个i，一个*，一个j，一个*，前面的放在内层，后面的放在外层

2.通过二维数组的行指针和列指针来引用二维数组元素

对行进行初始化处理的指针称为行指针

对列进行初始化处理的指针称为列指针

3.指针数组及其应用

1.指针数组用于表示多个字符串

由若干基类型相同的指针所构成的数组称为指针数组。

数组中的每一个元素都是一个指针，并且这些指针指向相同的数据类型。

来看一个数组的问题：

二维数组的的地址问题：二维数组a【i】【j】中a【i】代表着什么？

实际上a【i】代表着二维数组的指向第i行的首地址的

当然我们也可以用*（a+i）来进行表示

所以就会有这样的问题出现了？

神奇的排序：指针排序，字典排序，索引排序---------特征，在不改变存储的情况下，完成排序

适用对象：多维数组进行排序，有内存空间的要求，有位置移动和复杂度的要求

其中字符串指针充当了字符索引，而字符串作为其中的内容。构成了一个字典或者索引

这种通过移动字符串索引地址实现的排序称为索引排序。

相比于二维数组实现排序而言，这种方法的执行的效率相对更高一些

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define max_len 4
#define N 150
```

对字符串指针进行排序，使得字符串指针是拍好序列的，读的时候，

读取字符串指针的头即可读到字符串的第二维的数组内容

使用交换排序方法

实现字符串按字典顺序排序：字典排序

```
void sort(char *ptr[],int n)
{
int i;
int j;
char *temp = NULL;
for(i = 0;i<n-1;i++)
```

```
{
        for(j = i+1;j<n;j++)
    {
                if(strcmp(ptr[j],ptr[i])<0)使用字符对比函数进行对比操作
                {
                    temp = ptr[i];
                    ptr[i] = ptr[j];
                    ptr[j] = temp;
        }
    }
}
}

int main()
{
int i,n;
char name[N][max_len];二维数组
char *pStr[N];字符串指针，指向每一个二维数组中的一维数组每一行的头地址
printf("How many countries? \n");
scanf("%d",&n);
getchar();
printf("Input their names: \n");
for(i = 0;i<n;i++)
{
pStr[i] = name[i];
gets(pStr[i]);
}
使用gets进行累计的输入处理

sort(pStr,n);
printf("Sorted results: \n");
for(i = 0;i<n;i++)
{
puts(pStr[i]);  使用puts进行连续的输出处理
```

```
}

        system("pause");
        return 0;
}
```

1. 数组指针主要作用于对多个字符串进行处理操作，因此在实际应用中字符指针数组更加常用，
2. 并且字符指针比字符二维数组更加有效，可以加快字符串的排序速度。
3. 必须在使用字符指针之前必须进行初始化操作

讲讲物理排序和索引排序

通过移动字符串在实际物理存储空间中的存储位置而实现的排序，称为物理排序。

这种通过移动字符串索引地址实现的排序称为索引排序。

移动指针要比移动数组快的多。移动的时候只需要改变指针的指向即可，而无需修改字符串在内存中的地址。

相比于二维数组实现排序而言，这种方法的执行的效率相对更高一些

2.指针数组用于表示命令行参数这个笔试的时候极为有用

即为带命令行参数的main函数和参数处理

也就是等待命令行输入的参数

一般的格式如下：

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

第一个argc被声明为整形变量，用于存放命令行中参数的个数，因为至少需要存放程序的名称，所以至少为1，

第二个argv被声明为字符串指针数组，用于接收命令行参数

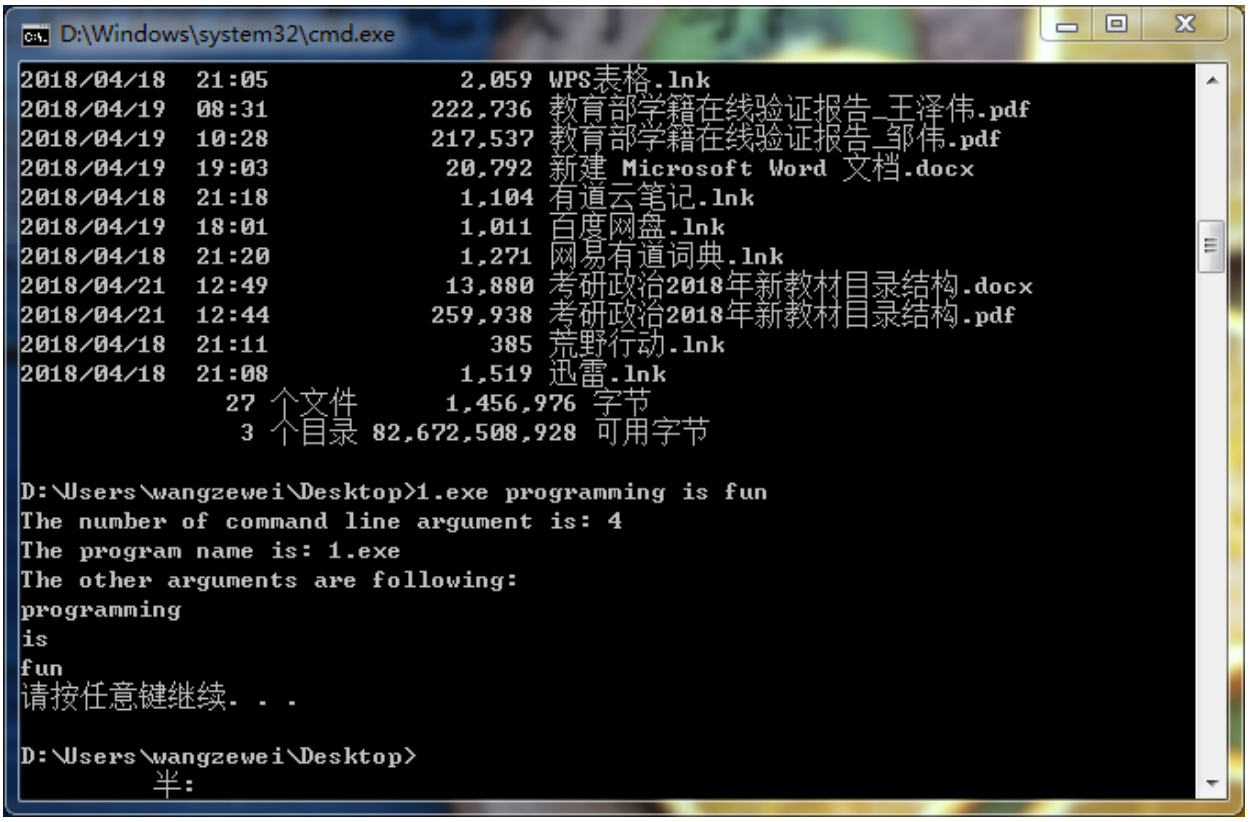
所有的命令行参数都当做字符串来处理，所以这里的指针数组argv依次指向命令行中的参数，且以\0为结尾

命令行参数很有用，尤其是在处理批处理命令中使用较为广泛，比如可以通过命令行参数向程序传递这个程序所要处理的文件的名字，还可以用来指定命令的选项等。当不需要参数的时候，一般会使用void main（）来表表明没有参数

看例子：

```
#include <stdio.h>
#include <stdlib.h>
int main(int argc,char *argv[])用于存放命令行中参数的个数，用于接收命令行参数
{
     int i;
    printf("The number of command line argument is: %d\n",argc);输出存放命令行中参数的个数
    printf("The program name is: %s\n",argv[0]);
    if(argv>0)
        printf("The other arguments are following:\n");
    {
                for(i = 1;i<argc;i++)
            {
                                printf("%s\n",argv[i]);输出接收命令行参数
            }
    }


        system("pause");
        return 0;
}
```



野指针的第一次出现

```
for(i = 0;i<n;i++)
{
pStr[i] = name[i];
gets(pStr[i]);
}
```

假如注销pStr[i] = name[i];程序将会异常停止

因为这一行是在对元素进行初始化操作，而当没有进行初始化时，就会导致pStr【i】指向一个不确定的存储单元，从而产生非法访问内存的错误。

所以，使用指针数组之前必须对指针数组进行初始化。

4. 动态数组------动态数组长度能变且仅能改变一次

1.C程序的内存映像

一个编译后的C程序获得并使用4块在逻辑上不同且不同于目的的内存储区

第一块　只读存储区　　机器码和字符串常量等只读数据-------与程序同生死的-只可读

第二块　静态存储区　　全局变量和静态变量-------与程序同生死的-可读可修改

第三块　堆-动态存储区　　自由存储区，程序可以利用C的动态分配内存函数来使用它

栈-动态存储区 保存函数调用时的返回地址，函数的形参，局部变量，和CPU的使用状态<函的参数的参数和相关>

-------与程序块同生死的-可读，可改，可申请，可释放

第一块　命令行参数　　命令行中读取的命令和内容

深度理解堆和栈

个人感觉这里的堆 应该指的是heap而非数据结构中的堆。

栈区（stack）— 由编译器自动分配释放 ，存放函数的参数值，局部变量的值等。其操作方式类似于数据结构中的栈。

堆区（heap） — 一般由程序员分配释放， 若程序员不释放，程序结束时可能由OS回收 。

区别和联系：

1.申请方式

堆是由程序员自己申请并指明大小，在c中malloc函数 如p1 = (char *)malloc(10);

栈由系统或编译器自动分配，如声明在函数中一个局部变量 int b; 系统自动在栈中为b开辟空间

2.申请后系统的响应

栈：只要栈的剩余空间大于所申请空间，系统将为程序提供内存，否则将报异常提示栈溢出。

堆：首先应该知道操作系统有一个记录空闲内存地址的链表，当系统收到程序的申请时，会遍历该链表，寻找第一个空间大于所申请空间的堆结点，然后将该结点从空闲结点链表中删除，并将该结点的空间分配给程序，另外，对于大多数系统，会在这块内 存空间中的首地址处记录本次分配的大小，这样，代码中的delete语句才能正确的释放本内存空间。另外，由于找到的堆结点的大小不一定正好等于申请的大 小，系统会自动的将多余的那部分重新放入空闲链表中。

3.申请大小的限制

栈：在Windows下,栈是向低地址扩展的数据结 构，是一块连续的内存的区域。这句话的意思是栈顶的地址和栈的最大容量是系统预先规定好的，在WINDOWS下，栈的大小是2M（也有的说是1M，总之是 一个编译时就确定的常数），如果申请的空间超过栈的剩余空间时，将提示overflow。因此，能从栈获得的空间较小。

堆：堆是向高地址扩展的数据结构，是不连续的内存区域。这是由于系统是用链表来存储的空闲内存地址的，自然是不连续的，而链表的遍历方向是由低地址向高地址。堆的大小受限于计算机系统中有效的虚拟内存。由此可见，堆获得的空间比较灵活，也比较大。

4.申请效率的比较：

栈由系统自动分配，速度较快。但程序员是无法控制的。

堆是由new分配的内存，一般速度比较慢，而且容易产生内存碎片,不过用起来最方便.

体会：

使用栈就象我们去饭馆里吃饭，只管点菜（发出申请）、付钱、和吃（使用），吃饱了就走，不必理会切菜、洗菜等准备工作和洗碗、刷锅等扫尾工作，他的好处是快捷，但是自由度小。

使用堆就象是自己动手做喜欢吃的菜肴，比较麻烦，但是比较符合自己的口味，而且自由度大，想吃啥可以点啥。

2.动态内存分配函数使得动态数组成为可能

(1)函数malloc

用于分配若干字节的内存空间返回一个指向该内存首地址的指针

函数原型为void* malloc（unsigned int size）

void* 通常称为通用指针或者无类型的指针，用来说明基类型未知的指针，而在使用中使用强转得到类型的方式

例如：

（返回值类型*）malloc（sizeof（数据的类型））

（返回值类型*）malloc（申请的内存空间的数量，申请的单个内存空间）

```
int *pi;
pi = （int*）malloc（2）//申请两个字节内存的空间
```

(2)calloc函数

用于给同一类型的数据项分配连续的存储空间并赋值为0

函数原型为void* calloc（unsigned int num，unsigned int size）

第一个参数确定了申请内存空间的数量，第二个参数确定了每个空间的字节数

void* 通常称为通用指针或者无类型的指针，用来说明基类型未知的指针，而在使用中使用强转得到类型的方式

例如：

（返回值类型*）malloc（申请的内存空间的数量，sizeof（数据的类型））

（返回值类型*）malloc（申请的内存空间的数量，申请的单个内存空间）

int *pi;

pi = （int*）malloc（10，sizeof（int））//申请10*2int个字节内存的空间

(3)free函数

释放向系统动态申请的由指针指向的存储空间。

函数原型为void* free（void *p）

其所可以释放的只有malloc和clloc申请的内存的指针

例如：

free（p）//释放p所指的内存的空间，释放空间。

(3)realloc函数

用于改变原来分配的存储空间的大小

函数原型为void* realloc（void *p，unsigned int size）

函数的功能是将指针p所指的地址的存储空间的大小进行修改改为size个字节

函数的返回值是新分配的储存空间的首地址，与原来分配的首地址不一定相同

一旦改变了指针变量的指向，原来分配的内存与数据也就随之丢失了，

所以，不要轻易改变指针变量的值，一旦改变无法挽回

void* 通常称为通用指针或者无类型的指针，用来说明基类型未知的指针，而在使用中使用强转得到类型的方式

例如：

int *pi;

pi = （int*）malloc（10，sizeof（int））//申请10*2int个字节内存的空间

pi = （int*）realloc（pi，sizeof（float））//用于改变原来分配的存储空间的大小，将int的大小改为float的大小

3.长度可变的一维动态数组

思想分析的角度：

啥，指针可以代替数组的方式来使用了？，

p地址=数组的0元素的地址=数组的名称代表的地址

数组是连续存储的=指针在申请地址也是连续的

基本相同，差别不大

使用内存申请，申请需要的内存的空间，并返回首地址，然后使用首地址来代替数组名，来实现对动态数组的产生过程。相当于建立了一个动态的数组，可以通过*（p+i）的方式或者p[i]的方式来访问数组的元素的值。

并且需要对其进行内存的释放free函数的使用。

看一个例子吧：

```
#include <stdio.h>
#include <stdlib.h>
///这个函数用作与输入
void input(int *p,int n)//此处传入的是一个指针的值哦
{
int i;
for(i = 0;i<n;i++)
{
scanf("%d",&p[i]);//这里却将他当做数组进行使用了
}
}

double Average(int *p,int n)//此处传入的是一个指针的值哦
{
int i;
double sum = 0;
for(i = 0;i<n;i++)
{
sum = sum + p[i];;//这里却将他当做数组进行使用了
}
return sum/n;
}

int main()
{
        int *p = NULL;
    int n;
    double aver;
    printf("now,pleas cat\n");
    scanf("%d",&n);

p = (int *)malloc(n*sizeof(int));
 printf("int地址下的地址是%p",p);
```

```
    p = (int *)realloc(p,n*sizeof(int));//recollo下的结果
    printf("double地址下的地址是%p",p);


申请了内存之后，需要进行判断是否内存申请成功了
if(p == NULL)
    {
                printf("no enough memory !\n");
        exit(1);
    }
    printf("input %d score\n",n);
    input(p,n);
    aver = Average(p,n);
    printf("aver is %.2f\n",aver);
   free(p);//别忘了释放内存
        system("pause");
        return 0;
}8
```

4.长度可变的二维动态数组

```
p = (int *)malloc(p, n*m*sizeof(int));
```

或者是

```
 p = (int *)calloc(n*m, sizeof(int));
//申请内存的时候采用m*n的形式进行申请m*n大小的内存
```

使用访问的时候使用 按列或者按行来进行处理：

m行n列的存储单元

```
for（i = 0; i<m;i++）
{
    for（j = 0; j<m;j++）
    {
    sum = sum + p[i*n+j];
    第i行第j个就是i*列数，加上j 的个数的过程
    第几行X列的个数，加上第几个
    }
}
```

5. 本章拓展内容

关于内存的一些错误和一些问题：

1. 内存分配未成功就使用--------->内存的分配未空，或者申请内存未成功
2. 内存分配成功了，但是未进行初始化就使用--------->包括使用未进行初始化的指针和数组变量
3. 内存分配成功了，也进行了初始化，但是发生了越界使用--------->就是使用了越界使用
4. 忘记了释放内存，造成了内存的泄露--------->比如写文件或使用数据库时，忘记了关闭文件或者数据库，或者申请了内存和线程，
5. 最后没有进行释放，导致了内存的泄露《相当于借了东西不还一样》，程序在每一次循环的时候都在申请内存，多次循环会导致内

存耗尽的结果

{

方法如下：

仅仅在需要时才使用的malloc

使用malloc的时候就必须要使用free

重复利用malloc的内存为佳

例如：

goto :Exit(用于进行释放内存，解决废料)

}

1. 释放内存之后仍然继续使用--------->访问已经不存在的或是已经被释放了的内存地址，释放了内存但却仍然使用它，将导致野指针。
2. 例如：char *ptr = null; purs(ptr);**//指针初始化了但是找不到指向，或是内存被释放却还在使用，称为野指针**
3. **释放内存的结果只是改变了内存中存储的数据，使该内存存储的内容变成了垃圾，指向垃圾内存的指针，称为野指针。**
4. **内存被释放后，指向它的指针不会自动变成空指针，野指针不是空指针，而是不知道指到什么值得指针**
5. **对呀空指针我们使用if（str==null）就可以检查其是否是空指针。而野指针不是空，而是一个不知道什么东西的东西**

定义指针需要初始化，要么null，要么指向合法内存
尽量把malloc放到入口处，free放到函数的出口处

1. 缓冲区溢出攻击
2. 缓冲区的溢出的攻击处理
3. gets（），puts（），

等函数对数组的长度没有进行控制，很容易被何况攻击和溢出处理。internet蠕虫
输入限制字符串长度的函数
fgets（str，N*sizeof（char），stdin）
而使用strncpy(),strcat（）等n族字符处理函数，通过增加一个参数来限制字符串处理的最大长度，可以防止发生缓冲区的溢出