递归

- 递归算法最外层一定是一种选择结构，即

if(问题最简单时的条件){
对应操作
}
else{
对应操作（拆分成规模较小的问题并调用递归函数）
}

字符串（字符数组）

①初始化

初始化三种方式

char a[]={'f','i','r','s','t','\0'};
char a[]={"first"};（系统自动补上'\0'）
char a[]="first";（系统自动补上'\0'）

若不进行初始化则需要保证字符串长度足够容纳所有字符以及'\0'

②scanf()和 gets()

- scanf()

函数读取到**空格**、**制表符**、回车、或文件结束符（EOF）为止

- gets()

从标准输入设备（如键盘）读取字符到 s 所指向的数组中，直到读到文件末尾或者新行符'\n'。新行符从键盘缓冲区中丢弃，最后一个字符读入后写入一个 '\0'。若成功则返回 s，若无字符读入数组或者读取失败返回空指针 NULL。

③printf()和 puts()

- printf()从第一个字符开始输出，直到遇到'\0'
- puts()输出字符串并在输出后添加换行符'\n'

数组操作

①插入与删除元素

②排序

- 冒泡排序（两两比较→确定是否互相交换）

```
void bubbleSortDown1(int a[],int size)
{
    int loc,i,temp;/*总共需要比较 size-1 趟。每一趟确定 a[loc]的值*/
    for(loc = size-1;loc >= 1;loc--){/*从下标为 0 ~ loc 数组元素中依次进行比较交换*/
        for(i = 0;i <= loc-1;i++)
          if(a[i] > a[i+1]){ /*相邻两个元素交换*/
              temp = a[i];
```

```
                    a[i] = a[i+1];
                    a[i+1] = temp;
                }
            }
        }
```

- 选择排序（找最大→放到末尾→反复）

```
    void selectSortDown(int data[],int elementCount)
    {
        int i,maxLoc,temp;

        for (i = 0; i <= elementCount-2; i++){
            maxLoc = findMax(data,i,elementCount - 1);
            if (i != maxLoc){
                temp = data[i];
                data[i] = data[maxLoc];
                data[maxLoc] = temp;
            }
        }
    }
```

③查找

- 线性查找（依次查找）
- 折半查找（前提是已经排序）(可用递归实现)

```
    int binarySearch(int a[],int searchKey,int left,int right)
    {
        int middle;
        while(left <= right){
            middle = (left + right)/2;
            if (searchKey == a[middle])
                return middle;
            else if(searchKey < a[middle])
                    right = middle - 1;
            else
                    left = middle + 1;
        }
        return -1;
    }
```

二维数组

- 二维数组相当于每个元素都是一个数组的数组
- 要访问一个二维数组元素,除了要知道数组首地址、行下标、列下标，还需要知道**数组的列数**
- 二维数组名做形参，第 2 个下标必须给出,如 int array[]**[10]**

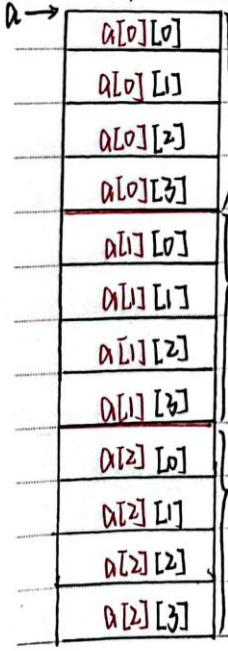
by 周梓媛

内存区：

1. 静态存储区：内存在编译时就已分配好了，这块内存在程序的整个运行期间都存在，存放静态数据，全局数据和常量。
2. 栈区：函数（包括main函数）内局部变量的存储单元都可以在栈上创建，函数执行结束时这些存储单元自动被释放。栈内存分配运算内置于处理器的指令集中，效率很高，但分配的内存容量有限（存所有变量）
3. 堆区：也称动态内存分配，用malloc或new申请任意大小的内存，在适当的时候用free或delete释放内存，动态内存的生命周期可以由我们决定，如果我们不释放内存，程序将在最后才释放掉内存。
4. 常量区：存放常量，如字符串常量。
5. 程序代码区（上面四个区完和数据区）：存放运行或准备运行的程序代码。

指针与二维数组

1. 以二维方式理解二维数组，它存储时却是一维的



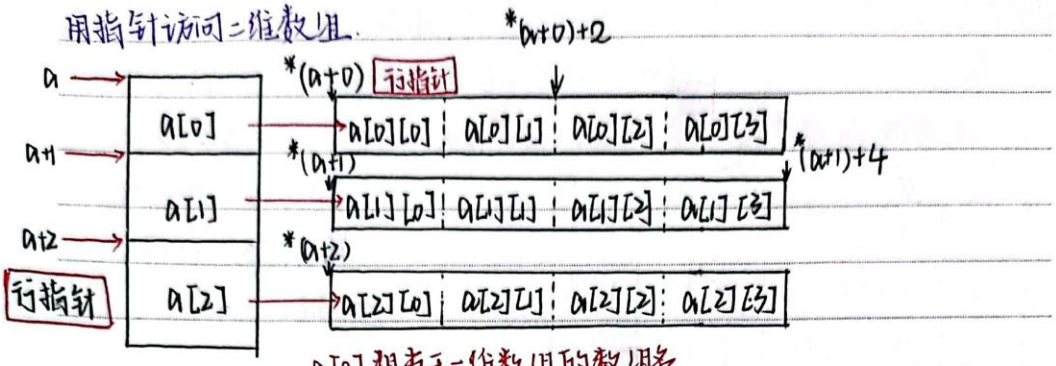
这是一个以a[0]为数组名的一维数组，
共有4个int元素。

名为a[1]的一维数组

m行n列的二维数组，
可以看成m个一维数组
每个一维数组有n列

名为a[2]的一维数组

用指针访问二维数组.



a[0]相当于一维数组的数组名

判断:

① int a[3][4];

int *p=a; X 编译错误,无法实现类型转换.

int(*p)[3]=a ✓ 正确,a是地址,且为行指针,即行指针指向有3个int类型元素的数组.

② int a[3][4];

int *p = a[0] ✓ a[0]为列指针,指向每个一维数组中的每个元素
↳也是一维数组的数组名.

③ int a[3][4];

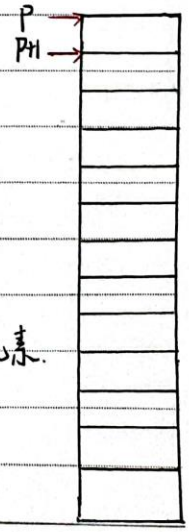
int *p = &a[0][0] ✓ a[0]=*(a+0)=*a=&a[0][0]. ☆

两种看待方式:

① 以一维数组的角度看待二维数组:

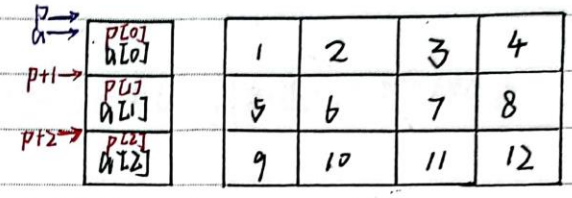
```
int a[3][4];   int *p=&a[0][0];
for(int i=0, i<3; i++)
    for(int j=0, j<4; j++)
        *p++ = i+j; //相当于 *p=i+j; p++;
```

P被定义为指向int的指针,它指向的是个一维数组,共有12个元素.



② 以二维数组的方式看待数组

```
int a[3][4] = {1,2,3,4,5,6,7,8,9,10,11,12}
int (*p)[3] = a;   P此时为行指针，指向了一个包含4个整数的一维数组.
for(int i=0; i<3; i++)
    for(int j=0; j<4; j++)
        *(*(p+i)+j) = i+j;
```

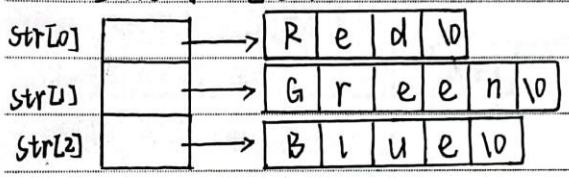


指针p将以和数组a完全相同的视角去看待二维数组，p和a的使用方法相同，即 *(*(p+i)+j) = p[i][j] = a[i][j] = *(*(a+i)+j)

指向字符串的二维数组

```
char* str[3] = {"Red", "Green", "Blue"};
```

str是个数组，数组内每个元素的类型都是指针，指向一个字符串常量首字符的地址.



示例① score数组中存放了3个学生4门功课的成绩，通过调用一维数组方式看待ave函数计算所有学生所有成绩的平均值，通过调用二维数组方式看待search函数计算第n个学生4门功课的成绩

```
int score[3][4];
```

所有：ave(&score[0][0], 12) → ave(int *p, int n)

实参 形参

```
{sum += *p;
 p++;}
```

第九个同学

一个学生：search(score, 2) → search(int(*p)[4], int n)

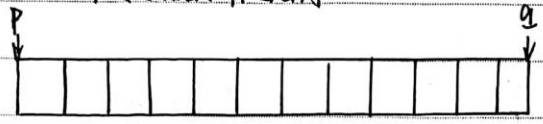
实参　　形参

{sum += *(*(p+n)+j)

j++; }

示例② 写一个inverse函数，将一个3×4的二维数组中的值按逆序重新存放。

角度1：以一维数组的角度看待



for(int *p=&a[0][0], int *q=p+n; p<q; p++, q--)

{ int temp=*p;　　inverse(&a[0][0], n)

*p=*q;　　inverse(int *p, int n)

*q=temp; }

角度2：以二维数组的角度看待

a[0][0]	a[0][1]	a[0][2]	a[0][3]
a[1][0]	a[1][1]	a[1][2]	a[1][3]
a[2][0]	a[2][1]	a[2][2]	a[2][3]

b[0][0]			
		b[2][2]	b[2][3]

b[2][3] = a[0][0]

b[2][2] = a[0][1]　找出规律.

b[2][1] = a[0][2]　　inverse(a, b)

即 b[M-1-i][N-1-j] = a[i][j];　　inverse(int a[][N], int b[][N])

for(int i=0; i<M; i++)

for(int j=0; j<N; j++)

b[M-1-i][N-1-j] = a[i][j];

结构体变量的传参.

① 使用结构体变量作为函数的返回值。(√)

```
int main()
{ struct point point;
  point = Setpoint(1,2);
  Display(point)
  return 0;
}
```

函数① (函数返回值的类型为结构体)

```
struct point Setpoint(int x, int y)
{ struct point M;
  M.x = x;
  M.y = y;
  return M; }
```

[传值方式]

函数② `void Display(struct point M)`

```
{ printf("%d", M.x);
  printf("%d", M.y); }
```

运行结果：2 3.

主程序：point　　子程序：M

point.x	2 ←	M.x	2
point.y	3 ←	M.y	3

返回时赋值

不用担心内存释放.

因为是先返回再释放，故返回的时候已经赋值.

(结构体与结构体之间可以等号赋值)

② `int main()` [错误示例]

```
{ struct point point;
  point = Setpoint;
  return 0; }
```

```
void Setpoint(struct point M)
{ M.x = 2;
  M.y = 3; }
```

↑ 在函数内部为结构体赋值，但实质并没有修改，本函数没有返回值，所以结束后内存就被释放了

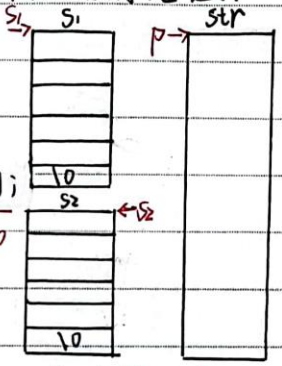
③传址调用

```
Setpoint(&point);        void Setpoint(struct point *p)
                         { p->x=2;  或 (*p).x=2;
                           p->y=3;     (*p).y=2;
                                  ↑ 别忘了括号 }
```

指针函数与malloc

指针函数⇒该函数的返回类型是指针类型.

例:设计一个指针函数, char* func(char*s1, char*s2),
完成2个字符串的相加,返回相加后的地址.




```
char* func(char*s1, char*s2)
{ char *str, *p;
  int len=strlen(s1)+strlen(s2)+1;
str=(char*)malloc(sizeof(char)*len)
void* 强转 char*
memset(str, 0, len*sizeof(char)); 初始化为0
  p=str
  while(*s1)[当s1指向的不是\0时]
    *p++=*s1++;
  while(*s2)
    *p++=*s2++;
  *p='\0'; 最后一个赋为'\0'
  return str;}
```

```
int main()
{ char s1[]="123456"
  char s2[]="78910"
  char *str=NULL;
  str=func(s1,s2);
  printf("%s", str);
  free(str) //释放func中申请并传递过来的内存
  return 0;}
```

有几点区分：

sizeof()：测量括号内部的量所占的内存，空间大小。

strlen()：只能测量字符串的长度，遇到'\0'停止，结果不算'\0'。

① char arr[] = {'a','b','c','d','e'}

- sizeof(arr) 计算的是arr数组的大小，为5×1字节
- strlen(arr) 后面'\0'位置未知，随机值。

② char arr[] = "abcde"; 会在。

- sizeof(arr) 字符串初始化arr尾部加上'\0'，即 |a|b|c|d|e|\0| 则其所占内存为6×1=6字节
- strlen(arr) '\0'位置确定，5。

③ char *p = "abcde";

- sizeof(p) 4/8 地址的大小为4/8(64位/32位)个字节
- strlen(p) strlen会顺着p所指向的地址依次往后，直到找到'\0'，而"abcde"后面有'\0'。故长度为5

char *s = "abcde";

printf("%s", str) 打印的时候是找到str所指向的内容，依次向后，直到遇到'\0'才停止。

字符串的两种形式：

1. char s[14] = "Hello World"; 声明一个字符数组
2. char* s = "Hello world"; 声明一个指针，指向字符串

如果采用第一种写法，由于字符数组的长度可以让编译器自动计算，所以声明时可以省略字符数组的长度。如：char s[] = "Hello world";
字符数组的长度可以大于字符串实际长度，如 char s[50] = "Hello"; 但不能小于

char *与 char a[]的本质区别

1. 大小：
   - 当定义 char s[10] 时，编译器会给数组分配10个单元，则其大小为10个字节
   - 当定义 char* s 时，这是个指针变量，只占四个字节，用来保存一个地址。
     printf("%p", s); 打印s的单元所保存的地址。
     printf("%p", &s); 打印s这个变量本身所在内存单元地址。

2. "读""写"能力
   - char* a = "abcd"; 此时"abcd"存放在常量区，通过指针只可以访问字符串常量，而不可以改变它
     如 s[0] = 'z'; //错误
   - char a[10] = "abcd"，此时"abcd"存放在栈，可以通过指针去访问和修改数组内容。
     如 s[0] = 'z'; //正确
   - 为什么字符串声明为指针时不能修改，而声明为数组便可修改？
     因为系统会将字符串的字面量保存在内存的常量区，这个区是不允许用户修改的。声明为指针时，指针变量存储的只是一个指向常量区的内存地址，因此用户不能通过地址去修改常量区。声明为数组时，编译器会给数组单独分配一段内存，字符串字面量会被编译器解释成字符数组，逐个字符写入这段新分配的内存中，在栈区的内存是可修改的。

- 为提醒用户，声明为指针后不得修改，常用const说明符，保证该字符串是只读的。 const char *s = "Hello World";

3.赋值时刻：

- char *a = "abcd"; 在编译时刻确定
- char a[20] = "abcd"; 在运行时确定

4.存取效率：

- char *a = "abcd", 存于静态存储区，慢
- char a[10] = "abcd", 存于栈上，快

5.能否赋值：

- 字符指针可以指向另一个字符串 char *s = "Hello";
  s = "world"; //正确
- 字符数组变量不能指向另一个字符串，char s[] = "Hello";
  数组的地址不可修改，即它头指不可变 s = "World"; //错误

相同点：

char *a 和 char a[10] 中，a都是指向第一个字符的指针