東南大學 SCHOOL OF INTEGRATED CIRCUITS, SEU

集成电路学院

计算机科学基础I

—— 数组II

东南大学 集成电路学院 朱彬武

E-mail: bwzhu@seu.edu.cn

回顾——数组定义


```
int main() {
    int n;
    cin >> n;

    int arr[100];
    cout << "请输入要逆序的数: \n";

    for (int i = 0; i < n; i++) {
        cin >> arr[i];
    }

    cout << "逆序结果: \n";
    for (int i = 0; i < n; i++) {
        cout << reverseNumber(arr[i]);
        if (i != n - 1)
            cout << "\n";
    }

    return 0;
}
```

数组定义

数组赋值

数组遍历

数组访问

- 数组中的每个元素通过下标（索引）来访问，要注意有效范围（0～数组的大小-1）
- 数组的大小在定义完之后就固定不变，且必须是整型常量

数组初始化和遍历

数组的初始化

- 列表初始化：`int array[5] = {2,3,5,7,11};`
- 部分初始化：`int array[5] = {2,3}; //等价于{2,3,0,0,0}`
- 全部元素初始化为0：`int array[5] = {0}; //全为0`

  `int array[5] = {}; //全为0`
- 自动推断大小：`int array[] = {2,3,5,7,11,13,15}; //编译器替你数数`

遍历数组

- 数组遍历通常都是使用for循环，让循环变量 i 从 “0 到 < 数组的长度”，这样循环体内最大的i正好是数组最大的有效下标。

```
int array[] = {2, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23};
for(int i = 0; i < 12; i++){
    cout << array[i] << endl;
}
```

是不是很呆？
还要自己数！

- 注意：不要写成 “<= 数组的长度”

数组的大小

- sizeof(数组名) 给出整个数组所占据的字节

```
int array[] = {2, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23};
cout << sizeof(array) << endl; //48
```

- 数组的大小 = sizeof(数组名)/sizeof(数组名[0])

  = 整个数组占的字节数/单个元素占的字节数

即使修改了数组大小，也不需要修改长度

```
int array[] = {2, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23};
for(int i = 0; i < sizeof(array) / sizeof(array[0]); i++){
    cout << array[i] << endl;
}
```

数组的赋值


```
int array[12];
array = {2, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23};  // ❌
```

```
int a[] = {2, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23};
int b[] = a; //❌
```

- 数组本身不能被赋值

```
int a[] = {2, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23};

int b[sizeof(a)/sizeof(a[0])];

for(int i = 0; i < sizeof(a)/sizeof(a[0]); i++){
    b[i] = a[i]; //✅
}
```

- 要把一个数组的所有元素交给另一个数组，必须要借助数组遍历

例题——输出素数


```
void printPrime(const int maxNumber){
    for(int num = 2; num < maxNumber; num++){
        isPrime = 1;

        for(int i = 2; i < num; i++){
            if(num % i == 0){
                isPrime = 0;
                break;
            }
        }

        if(isPrime == 1){
            cout << num << endl;
        }
    }
}
```

当前代码效率不够高，为什么？

如果知道了k是素数，2k，3k，4k，…，等k的倍数一定是合数，不需要再用循环判断了

Initial: assume 2..30 are prime

2	3	4	5	6	7	8	9	10	11
12	13	14	15	16	17	18	19	20	21
22	23	24	25	26	27	28	29	30	

构造素数表（利用数组优化）

要构造maxNumber以内的素数表（埃氏筛法）

（1）令k为2，

（2）将2k，3k，4k直至n*k<maxNumber的数标记为非素数

（3）令k为下一个没有被标记为非素数的数，重复（2）；直至所有的数都已经尝试完毕

提示：开辟prime[maxNumber]，初始化其所有元素为1，prime[k]为1，表示k是素数

埃氏筛法实现


```cpp
void buildPrimeTable(const int maxNumber) {
    int isPrime[maxNumber] = {0};
    // 0和1不是素数
    for (int i = 2; i < maxNumber; ++i) {
        isPrime[i] = 1;
    }

    // 构造素数表
    for (int k = 2; k < maxNumber; k++) {
        if (isPrime[k] == 1) {
            // 筛选出k的倍数，标记为合数
            for (int n = 2; n * k < maxNumber; n++) {
                isPrime[n * k] = 0;
            }
        }
    }

    int count = 0;  // 计数器
    for (int i = 2; i < maxNumber; i++) {
        if (isPrime[i]) {
            cout << i << "\t";
            count++;
            if (count % 5 == 0) {
                cout << endl;
            }
        }
    }
    cout << endl;
}
```

Initial: assume 2..30 are prime

2	3	4	5	6	7	8	9	10	11
12	13	14	15	16	17	18	19	20	21
22	23	24	25	26	27	28	29	30	

埃氏筛法的复杂度：O(nloglogn)

这段代码还有优化空间，有兴趣的去搜埃氏筛法的最终实现

数组传参

数组作为函数参数

```
void traverse(int b[],              ) {
    for(int i = 0; i <        ; i++){
        cout << b[i] << endl;
    }
}

int main() {
    int a[3] = {1, 2, 3};
    traverse(a,                        );
}
```

但是传参的时候好像是做了这样一件事：`int b[] = a`

- 参数表里的`int b[]`，实际上等价于`int *b`，这是一个指针变量，b存储的值是a[0]的地址

传入数组长度的必要性

```
void traverse(int b[], int length) {
    for(int i = 0; i < length; i++){
        cout << b[i] << endl;
    }
}

int main() {
    int a[3] = {1, 2, 3};
    traverse(a, sizeof(a)/sizeof(a[0]));
}
```

- 对于64位操作系统，指针变量的大小固定是8个字节
- 不能通过sizeof(b)/sizeof(b[0])推出数组长度，因此必须传入一个参数来表示数组的大小

数组作为函数参数的等价写法

1
```
void traverse(int arr[], int length) {
    //...
}
```

2
```
void traverse(int *arr, int length) {
    // ...
}
```

3
```
void traverse(int arr[3], int length) {
    // ...
}
```

写arr[3]只是为了好看，不包含数组大小的信息，实际上传的仍然是指针

例题——查找数组特定元素

编写一个函数 findIndex，用于在整数数组中查找指定元素 x，若找到则返回该元素**第一次出现的下标**，若未找到则返回 -1。

```
int findIndex(int arr[], int n, int x) {
    int ret = -1;
    for (int i = 0; i < n; i++) {
        if (arr[i] == x) {
            ret = i;
            break;
        }
    }
    return ret;
}
```

学习任何容器，我们都会首先学习它的遍历、增、删、改、查等基本操作，更多基础练习请参考Lab8。

二维数组

二维数组定义

- 二维数组可以理解为每一个数组元素都是一个一维数组
- `int mat[3][5]`，可以理解为一个3行5列的矩阵，在内存里按行连续存放

a[0][0]	a[0][1]	a[0][2]	a[0][3]	a[0][4]
a[1][0]	a[1][1]	a[1][2]	a[1][3]	a[1][4]
a[2][0]	a[2][1]	a[2][2]	a[2][3]	a[2][4]

- 数组定义：<类型> 数组名称[行数][列数]

二维数组初始化

- 列表初始化：
```
int mat[2][3] = {{2,3,5},      // {2,3,5,7,11,13}
                 {7,11,13}};
```
- 部分初始化：
```
int mat[2][3] = {{2},          // {{2,0,0}
                 {7,11}};          {7,11,0}}
```
- 全部元素初始化为0：
```
int mat[2][3] = {};
```
- 自动推断大小：
```
int mat[][3] = {{2,3,5},
                {7,11,13}};
```

注意：只能省略行数，不能省略列数

```
int mat[2][] = {{1,2,3},{4,5,6}};
```
❌ 非法

```
int mat[][] = {{1,2,3},{4,5,6}};
```
❌ 非法

对于数组mat[M][N]，
mat[i][j]的地址 =
mat[0][0]的地址 + (i*N+j)

二维数组遍历


```
int mat[2][3] = {{2,3,5},
                 {7,11,13}};

for(int i = 0; i < 2; i++){
    for(int j = 0; j < 3; j++){
        cout << mat[i][j] << "\t";
    }
    cout << "\n";
}
```

```
int mat[2][3] = {{2,3,5},
                 {7,11,13}};

for(int i = 0; i < sizeof(mat)/sizeof(mat[0]); i++){
    for(int j = 0; j < sizeof(mat[0])/sizeof(mat[0][0]); j++){
        cout << mat[i][j] << "\t";
    }
    cout << "\n";
}
```

- 因为是二维数组，所以遍历需要两层for循环
- 二维数组的元素访问：mat[i][j]，不要写成mat[i, j]
- 同样，可以用sizeof推导出行数和列数

  sizeof（二维数组名）= 所有元素所占的字节数

例题——tic-tac-toe游戏

- 读入一个3×3的矩阵，矩阵中的数字为1表示该位置上有一个“X”，为0表示为“O”
- 程序判断这个矩阵中是否有获胜的一方，输出0表示获胜的是“O”，输出1表示获胜的是“X”，或输出无人获胜

- 基本操作：对行、列、对角线做遍历

二维数组赋值


```
const int size = 3;
int board[size][size];
int result = -1; // -1:没人赢，1: X赢，0: O赢

//读入棋盘
for(int i = 0; i < size; i++){
    for(int j = 0; j < size; j++){
        cin >> board[i][j];
    }
}
```

函数调用


```
result = checkRow(board, size);
if (result == -1) result = checkCol(board, size);
if (result == -1) result = checkMainDiag(board, size);
if (result == -1) result = checkAntiDiag(board, size);

if (result == 1) {
    cout << "X wins\n";
} else if (result == 0) {
    cout << "O wins\n";
} else {
    cout << "No one win\n";
}
```

遍历正对角线和反对角线


```c
//检查正对角线
int checkMainDiag(int board[][3], int size) {
    int numOfX = 0, numOfO = 0;

    for (int i = 0; i < size; i++) {
        if (board[i][i] == 1) {
            numOfX++;
        } else if (board[i][i] == 0) {
            numOfO++;
        }
    }

    if (numOfX == size) return 1; // X 赢
    if (numOfO == size) return 0; // O 赢

    return -1;
}
```

```c
//检查反对角线
int checkAntiDiag(int board[][3], int size) {
    int numOfX = 0, numOfO = 0;

    for (int i = 0; i < size; i++) {
        if (board[i][size - 1 - i] == 1) {
            numOfX++;
        } else if (board[i][size - 1 - i] == 0) {
            numOfO++;
        }
    }

    if (numOfX == size) return 1; // X 赢
    if (numOfO == size) return 0; // O 赢

    return -1;
}
```

- 二（多）维数组传参时，只有第一维可以省略，后面的维度必须明确
- 正对角线元素：`board[i][i]`，反对角线元素：`board[i][size-1-i]`

遍历行和列


```c
//检查行
int checkRow(int board[][3], int size){
    for (int i = 0; i < size; i++) {
        int numOfX = 0, numOfO = 0;

        for (int j = 0; j < size; j++) {
            if (board[i][j] == 1) {
                numOfX++;
            } else if (board[i][j] == 0) {
                numOfO++;
            }
        }

        if (numOfX == size) {
            return 1;   // X 赢
        }
        if (numOfO == size) {
            return 0;   // O 赢
        }
    }

    return -1;  // 所有行都没人赢
}
```

```c
// 检查列
int checkCol(int board[][3], int size){
    for (int j = 0; j < size; j++) {
        int numOfX = 0, numOfO = 0;

        for (int i = 0; i < size; i++) {
            if (board[i][j] == 1) {
                numOfX++;
            } else if (board[i][j] == 0) {
                numOfO++;
            }
        }

        if (numOfX == size) {
            return 1;   // X 赢
        }
        if (numOfO == size) {
            return 0;   // O 赢
        }
    }
    return -1;
}
```

Q：两个函数能否合并起来，或者说能否用一个check函数同时完成行和列的检查?

二维数组转置


```
int checkLines(int board[][3], int size, bool checkRow) {
    for (int i = 0; i < size; i++) {
        int numOfX = 0, numOfO = 0;
        for (int j = 0; j < size; j++) {
            int v = checkRow ? board[i][j] : board[j][i];
            if (v == 1) numOfX++;
            else if (v == 0) numOfO++;
        }
        if (numOfX == size) return 1;
        if (numOfO == size) return 0;
    }
    return -1;
}
```