

目录 content

1

初步分析

2

数据结构设计

3

功能结构设计

俄罗斯方块：网格图

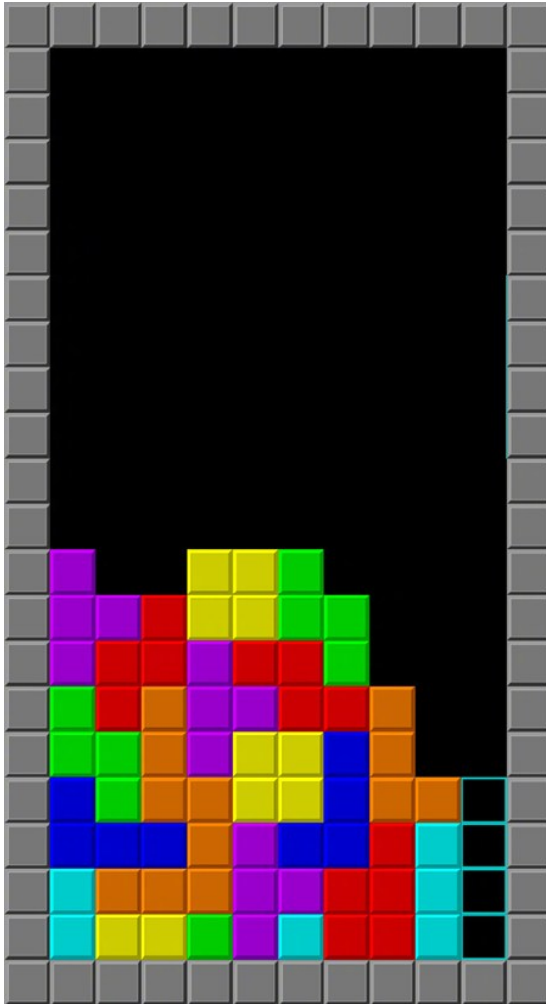
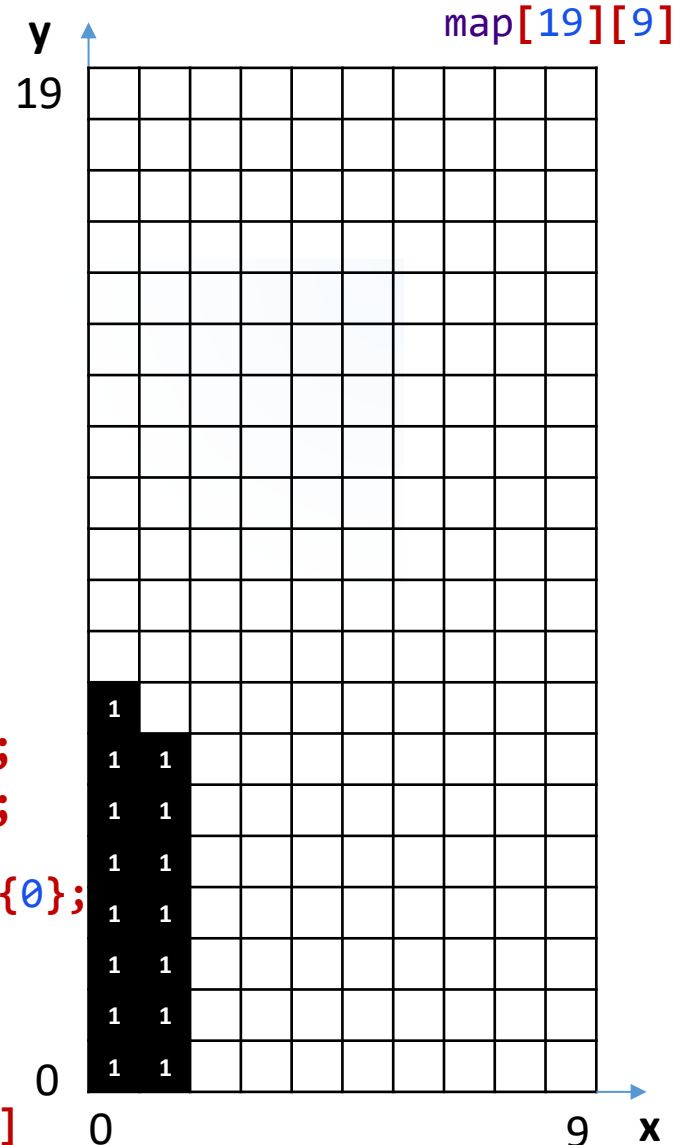
游戏界面由20行×10列的可视区域组成

- 横向范围：X轴区间
[0, 9] (共10列)
- 纵向范围：Y轴区间
[0, 19] (共20行，
Y=0为底部，Y=19为顶部)
- 被方块填充位置：1
- 未被方块填充位置：0

```
const int WIDTH  = 10;
```

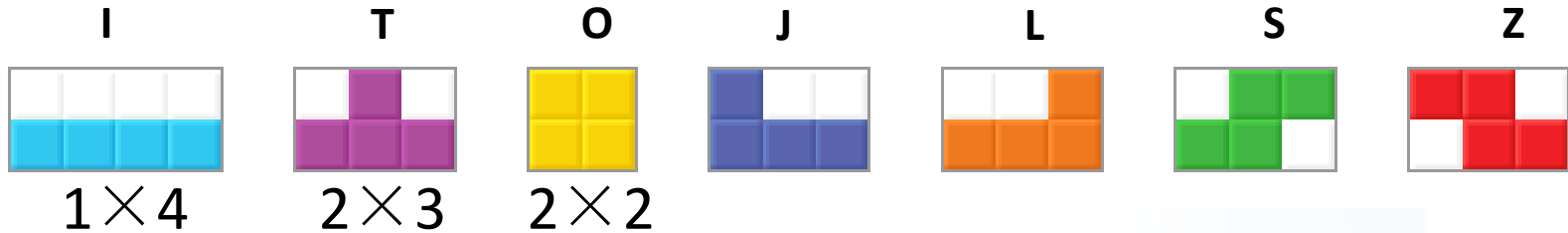
```
const int HEIGHT = 20;
```

```
char map[HEIGHT][WIDTH]={0};
```

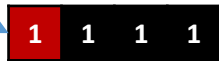


俄罗斯方块：方块图

7种不同形状的四格方块 (I/T/O/J/L/S/Z型)

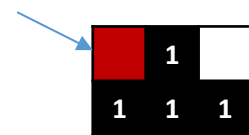


cube[0][0]



char cubeI[1][4]={1,1,1,1};

cube[0][0]



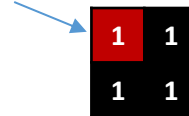
char cubeT[2][3]={{0,1,0},{1,1,1}};

– 方块是一个二维数组

– 实心格子: 1

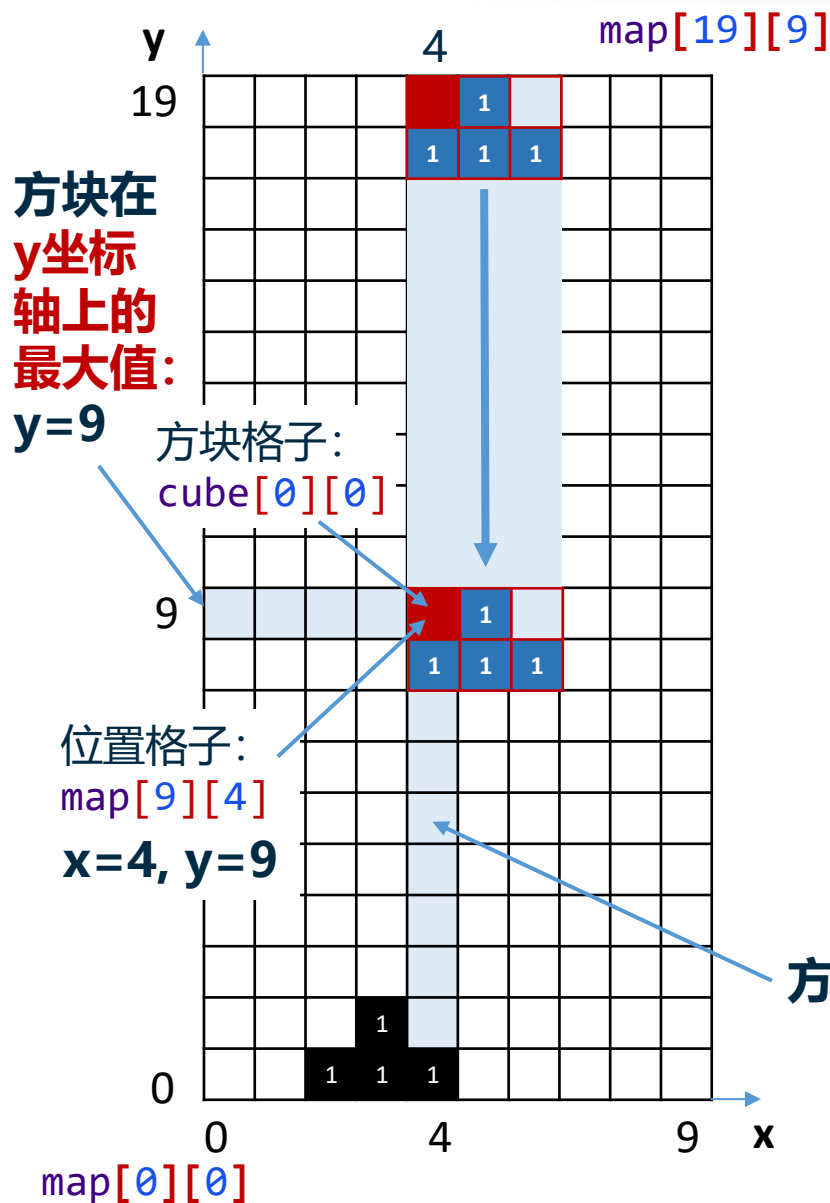
– 空心格子: 0

cube[0][0]



char cubeO[2][2]={{1,1},{1,1}};

俄罗斯方块：方块在网格图中的位置



• 方块在网格中的坐标

- ✓ 方块格子左上角在网格图中的位置 x, y (x 轴上的最小值, y 轴上的最大值)

• 方块落下:

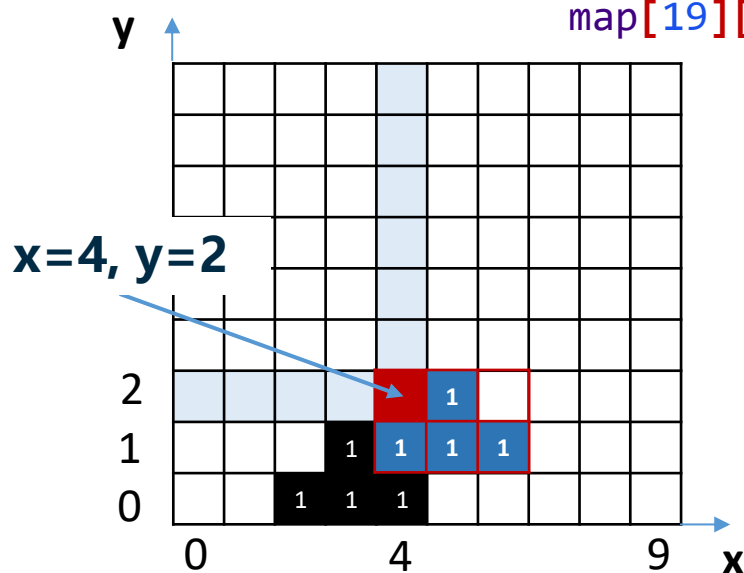
- ✓ 方块从 $x=4, y=19$ 位置开始下落, 方块在整个下落期间无冲突

• 方块落下的条件: 无冲突

- ✓ 冲突: 方块实心格不能落在位置图上已经被填充过的位置
- ✓ 如果 方块格子取值1, 方块格子所在位置格子取值1, 冲突

俄罗斯方块：方块在网格图中的填充

`map[19][9]`

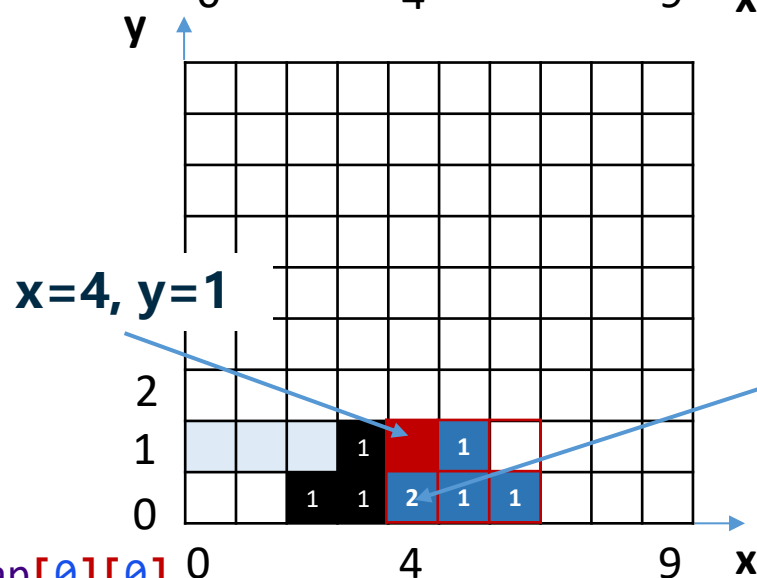


• 方块落下的最终位置

- ✓ 出现冲突位置: $x=4, y=0$
- ✓ 冲突时方块在网格中的坐标:
 $x=4, y=1$
- ✓ 最后一个无冲突位置坐标:
 $x=4, y=2$
- ✓ 方块落下的最终位置: $x=4, y=2$

• 方块落下后的方块融合

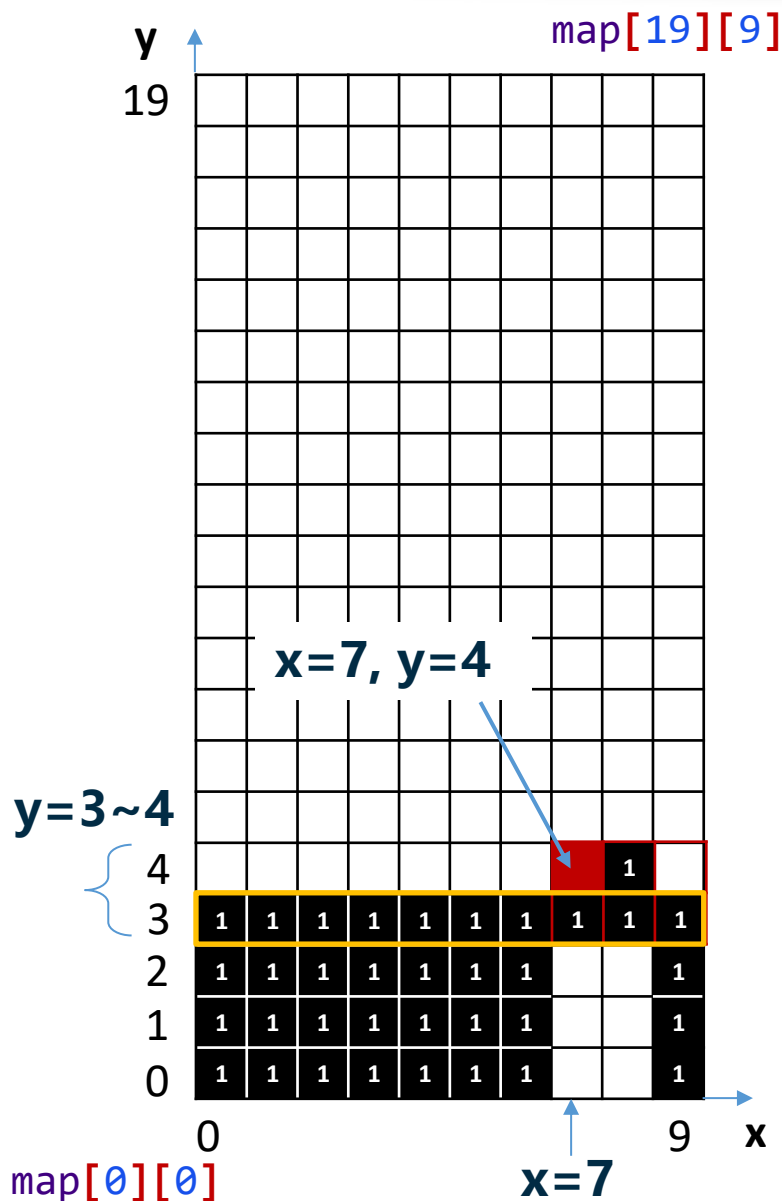
- ✓ 将方块中的每个实心格子在网格图中的位置置 1



冲突位置
 $x=4, y=0$

`map[0][0]`

俄罗斯方块：方块的消除



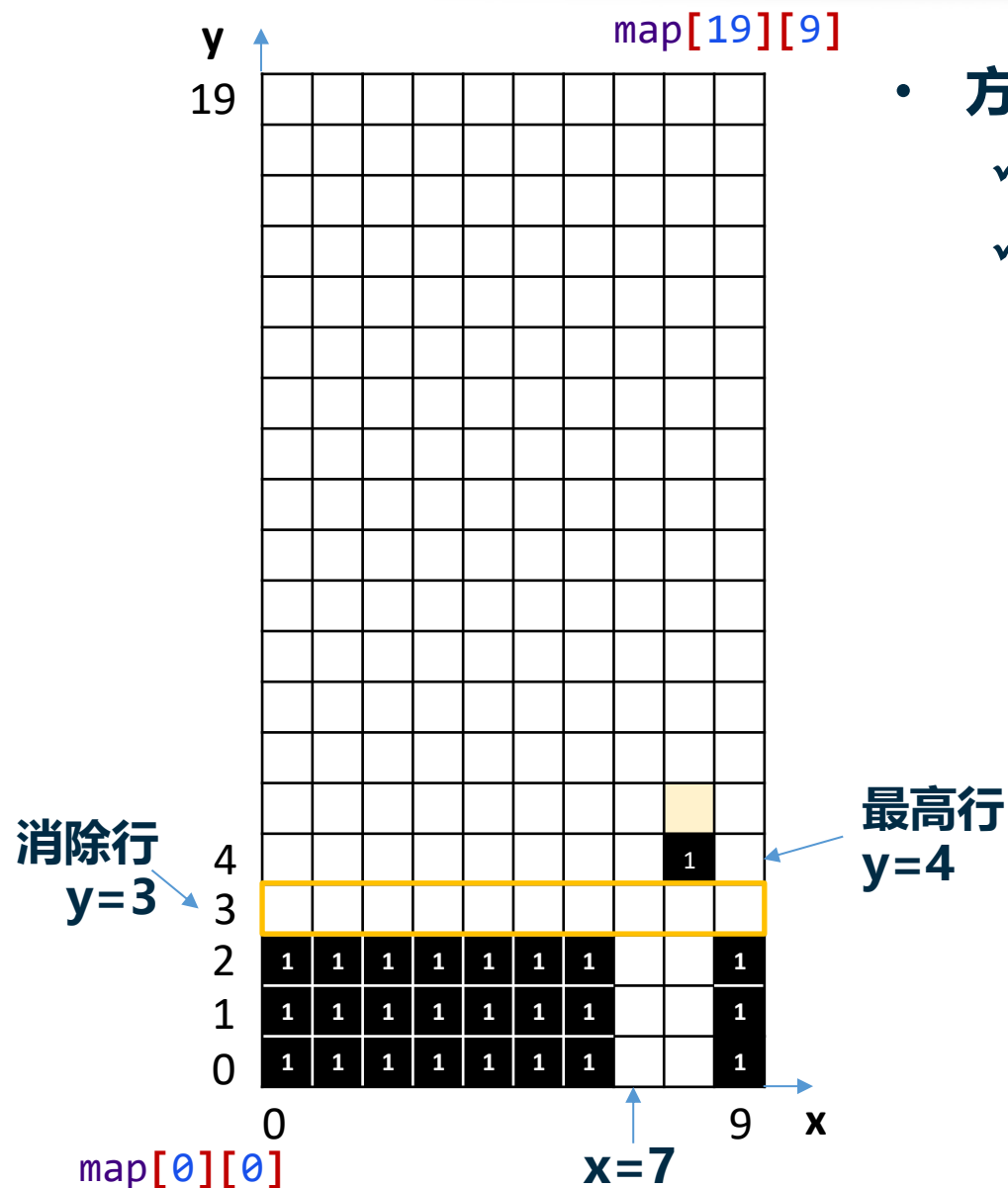
- **方块消除：**

- ✓ x, y = 方块落下的最终坐标
- ✓ 从 y 行开始，遍历方块的高度行数
- ✓ 从 $x=0$ 列开始遍历 x ，坐标位置全部为 1，可消除
- ✓ 消除：清零

- **方块消除得分：**

- ✓ 1行：100分
- ✓ 2行：300分
- ✓ 3行：500分
- ✓ 4行：800分

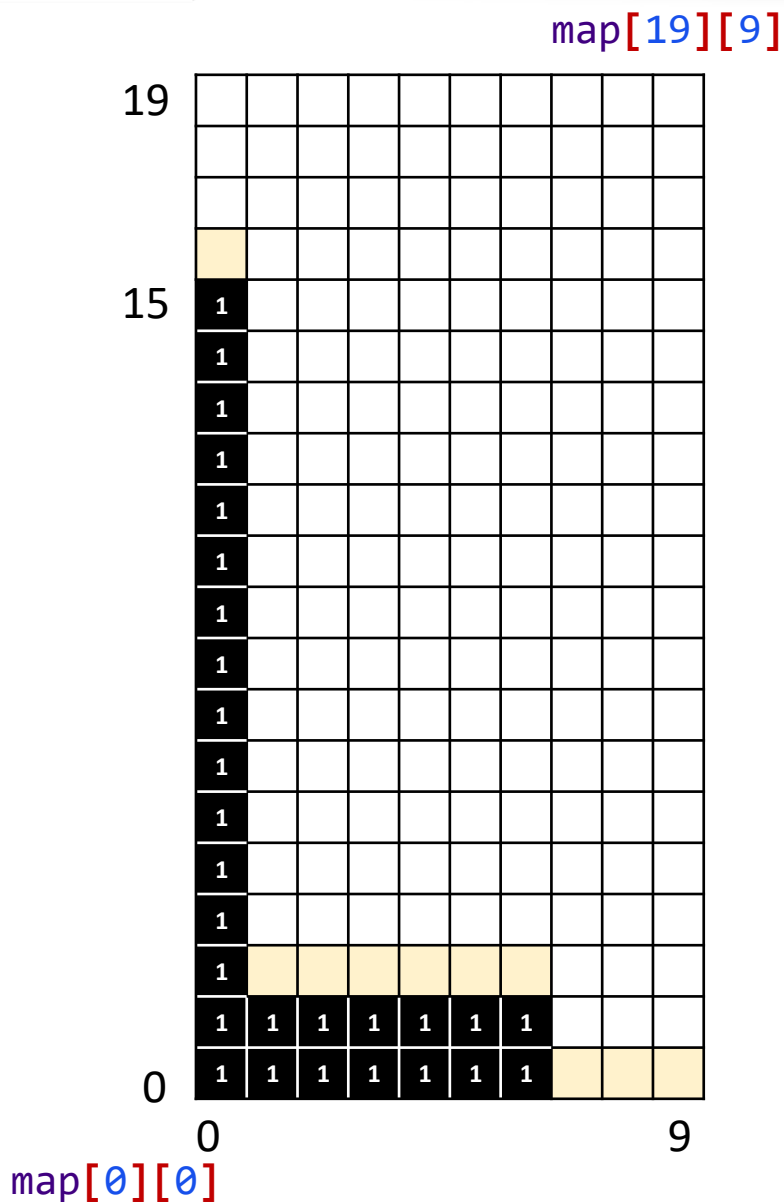
俄罗斯方块：方块的消除



方块消除后

- ✓ 被消除的行之上所有行下移
- ✓ 如：当前消除行 $y=3$
累积的最高行 $y=4$
 - ✓ $y=4$ 行 移动到 $y=3$ 行
 - ✓ $y=3$ 行的所有格子清0

俄罗斯方块：游戏结束



- 游戏结束：
 - ✓ 计算最高行， $Y \geq 15$
 - ✓ 下落、消除后，还高于16行，结束
 - ✓ 消除完成后要更新每一列的最高位置，判断是否超高

目录 content

1

初步分析

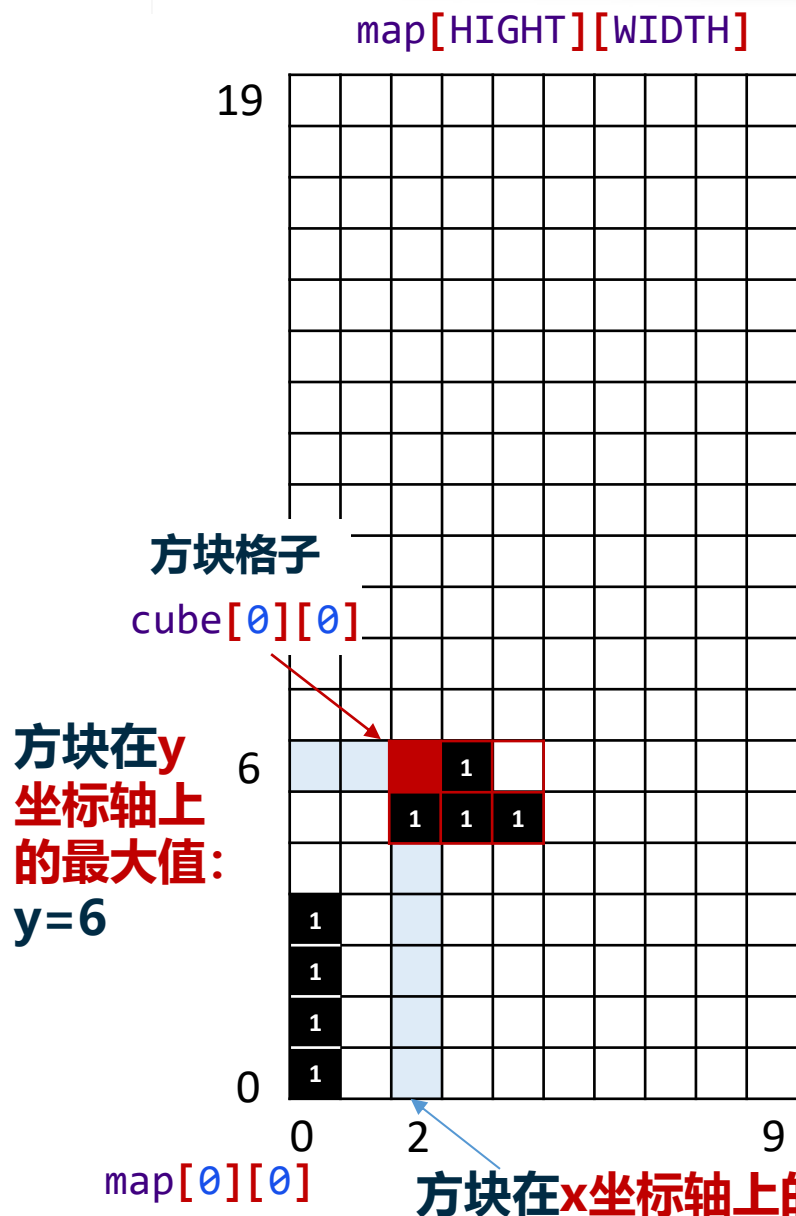
2

数据结构设计

3

功能结构设计

俄罗斯方块：方块与网格图的位置关系



- 网格图用二维数组表示

```
const int WIDTH  = 10;
const int HEIGHT = 20;
char map[HEIGHT][WIDTH]={0};
char map[0][0]=1;  网格位置填充
```

- 方块用二维数组表示

```
char cube[2][3] =
{
    {0, 1, 0},
    {1, 1, 1}
};
```

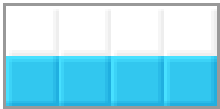
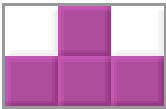
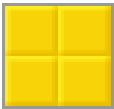


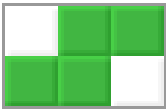

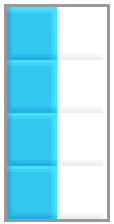
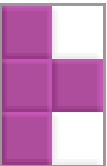
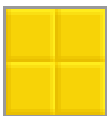
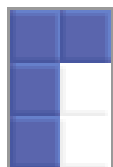
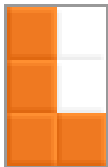
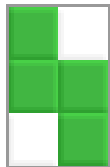
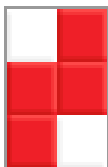

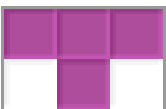





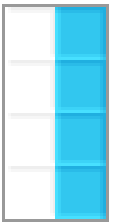
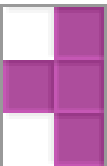
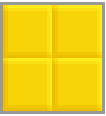
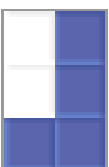

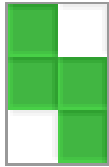
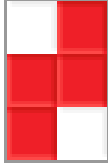
方块格子填充

- 方块在网格图中的位置用 cube[0][0] 的位置坐标表示

x=2, y=6

俄罗斯方块：方块旋转与存储

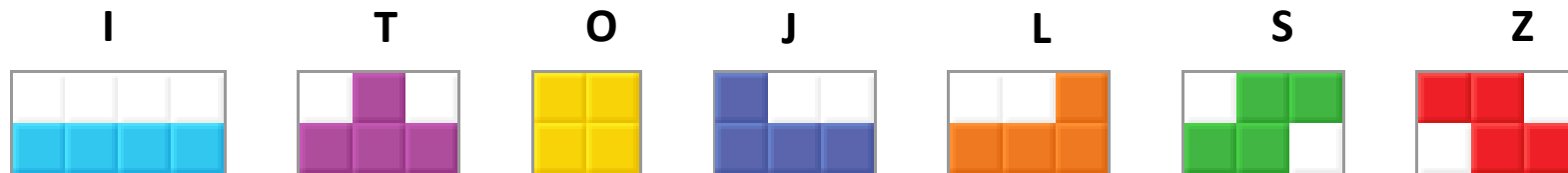
– 旋转操作：支持0°、90°、180°、270°四向旋转（最大不超过4×4）

	I	T	O	J	L	S	Z
0	 1×4	 2×3	 2×2	 2×3	 2×3	 2×3	 2×3
90	 4×1	 3×2	 2×2	 3×2	 3×2	 3×2	 3×2
180	 1×4	 2×3	 2×2	 2×3	 2×3	 2×3	 2×3
270	 4×1	 3×2	 2×2	 3×2	 3×2	 3×2	 3×2

问题：如何管理方块及其旋转？

俄罗斯方块：方块旋转与存储

– 方块存储结构（用二维数组初始化方块结构）



- 用不同的方块数组初始化不同的方块形状
- 方块的不同旋转角度外形可能一样，仍可用不同的数组进行管理
- 用空间换复杂度，能用预处理解决的不在运行时解决

– 可以用同一个存储结构吗？

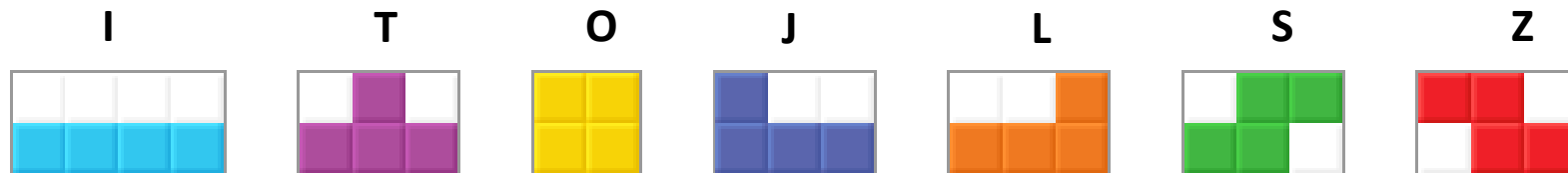


270

```
char cubeT180[4][4] =  
{ {1, 1, 1, 0},  
  {0, 1, 0, 0} };  
char cubeT90[4][4] =  
{ {1}, {1, 1}, {1} };
```

俄罗斯方块：方块旋转与存储

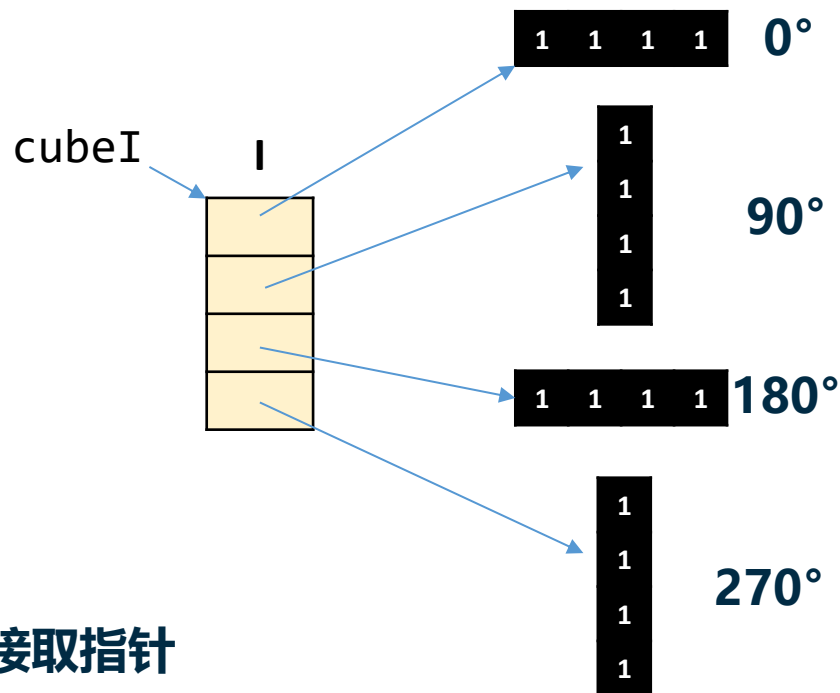
– 方块存储结构（用指针数组管理四个旋转方向）



– 旋转操作：0°、90°、180°、270°

可以映射为 0、1、2、3

```
char* cubeI[4] =  
{  
    (char*)cubeI0,  
    (char*)cubeI90,  
    (char*)cubeI180,  
    (char*)cubeI270  
};
```

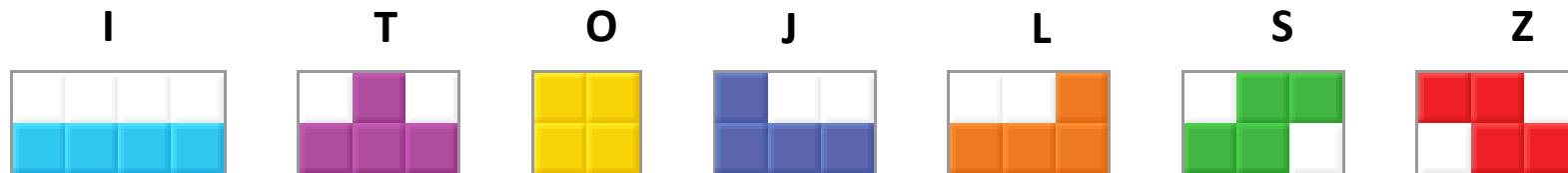


– 获取指定类型和角度方块的时候，可以直接取指针

```
char (*cube)[4] = (char(*)[])cubeI[0];
```

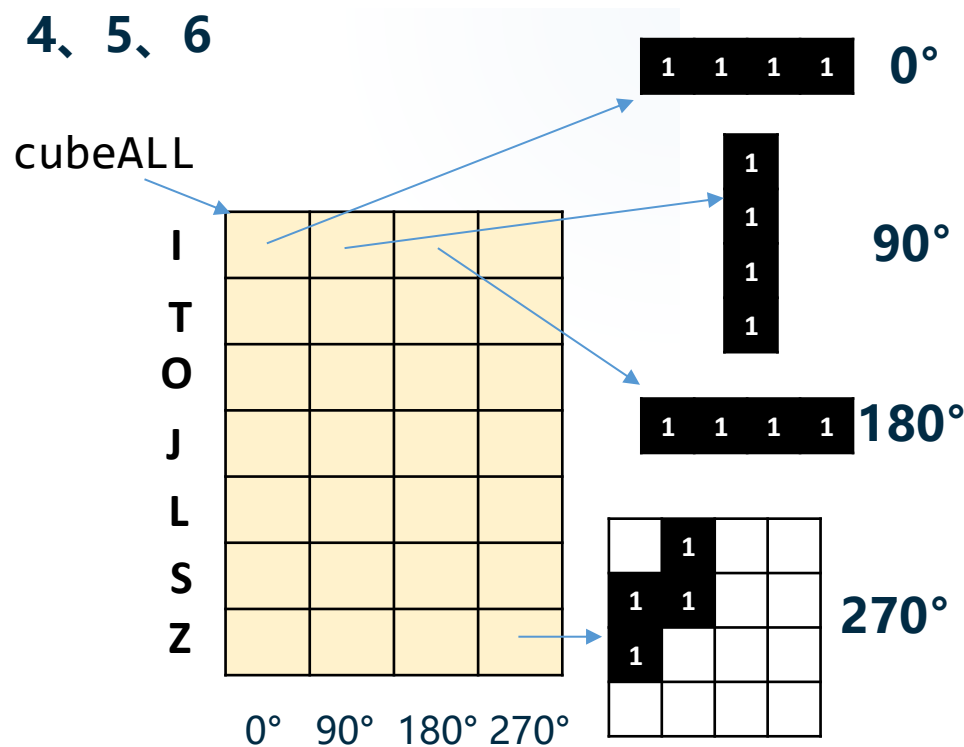
俄罗斯方块：方块旋转与存储

- 方块存储结构（用二维指针数组管理7个方块的4个旋转方向）



- 7种方块可以映射为 0、1、2、3、4、5、6

```
char* cubeALL[7][4] =  
{  
    {  
        (char*)cubeI0,  
        (char*)cubeI90,  
        (char*)cubeI180,  
        (char*)cubeI270  
    },  
    { }  
};
```

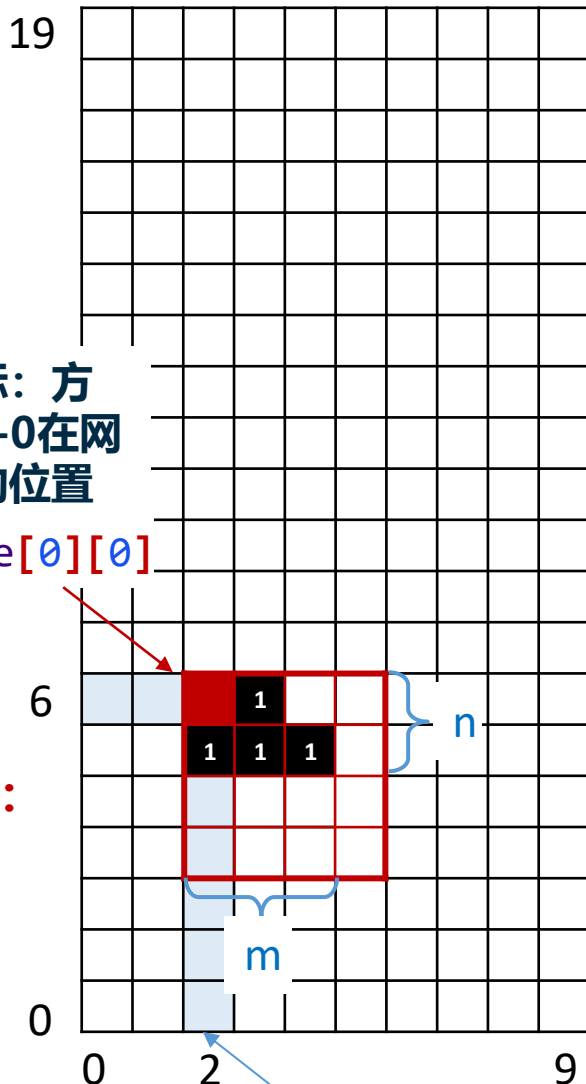


- 获取指定类型和角度方块的时候，可以直接取指针

```
char (*cube)[4] =(char(*)[])cubeALL[0][0];
```

俄罗斯方块：方块的存储结构

map[HEIGHT][WIDTH]



方块坐标：方块格子0-0在网格图中的位置

cube[0][0]

方块在y坐标轴上的最大值：y=6

map[0][0]

方块在x坐标轴上的最小值：x=2

// 方块类型

```
enum CubeType
```

```
{
```

```
    TI=0, TT, TO, TJ, TL, TS, TZ
```

```
};
```

// 方块旋转角度

```
enum CubeAngle
```

```
{
```

```
    A0=0, A90, A180, A270
```

```
};
```

```
typedef struct {
```

```
    int    x, y;
```

```
    char*  cube;
```

```
    enum CubeType  type;
```

```
    enum CubeAngle angle;
```

```
    int  m, n;
```

```
} Cube;
```

// 方块位置坐标

// 方块形状

// 方块类型

// 方块旋转角度

// 方块实际宽高

目录 content

1

初步分析

2

数据结构设计

3

功能结构设计

俄罗斯方块：方块的初始化

存储所有的方块

```
char* cubeALL[7][4] =
{
    {
        (char*)cubeI0,
        (char*)cubeI90,
        (char*)cubeI180,
        (char*)cubeI270
    },
    { }
};
```

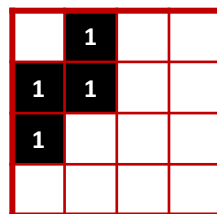
存储所有方块的宽

```
int cubeALLwidth[7][4] =
{
    {4, 1, 4, 1},
    { }
};
```

存储所有方块的高

```
int cubeALLheight[7][4] =
{
    {1, 4, 1, 4},
    { }
};
```

输入方块类型、旋转角度、初始坐标x



```
// 方块类型
enum CubeType
{
    TI=0, TT, TO, TJ, TL, TS, TZ
};

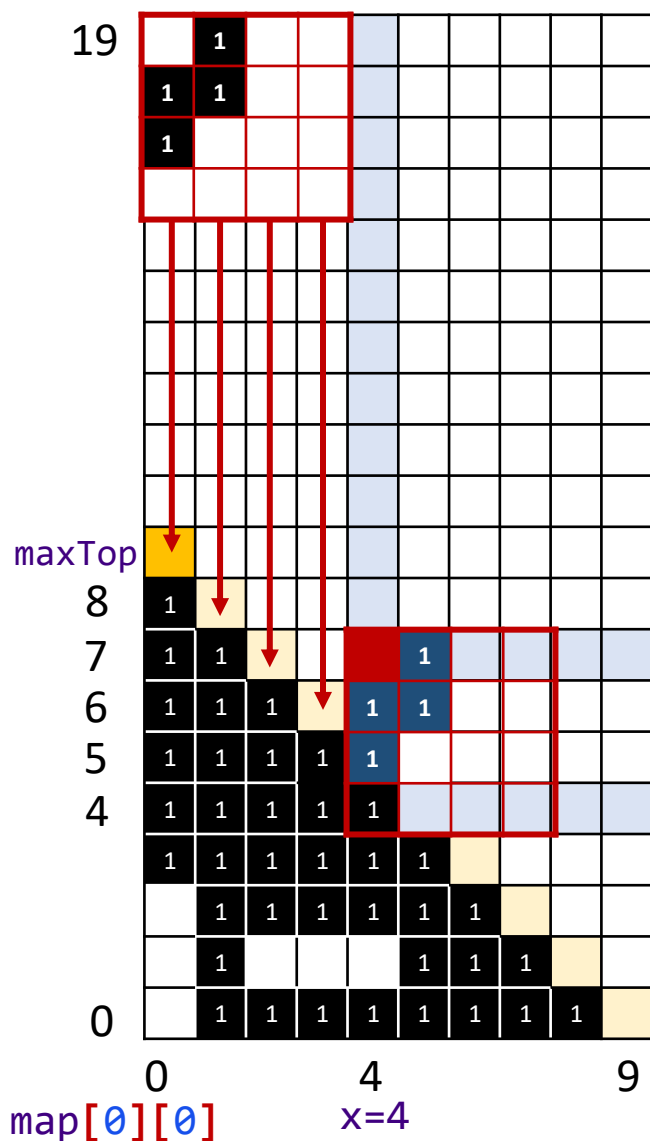
// 方块旋转角度
enum CubeAngle
{
    A0=0, A90, A180, A270
};
```

```
struct Cube cube;
cube.type=TZ;
cube.angle=A90;
cube.cube=cubeAll[TZ][A90];
cube.m=cubeAllwidth[TZ][A90];
cube.n=cubeAllheight[TZ][A90];
cube.x=current;
```

初始坐标 y 根据冲突检测位置设置

俄罗斯方块：网格图的冲突检测

map[HEIGHT][WIDTH]



传统俄罗斯方块是一层一层下落的，但本作业不需要逐层下落，可以加速一下

- 记录每一列的最高点行位置
- 所有列的最高点 `maxTop` 用来判断是否游戏结束
- 方块下落覆盖列的最高点 `currentTop` 用来做冲突检测的开始位置，跳过上方的非必要行，降低冲突检测次数

```
char mapTop[WIDTH] = {0};
```

```
char maxTop, currentTop;
```

m 方块宽

n 方块高

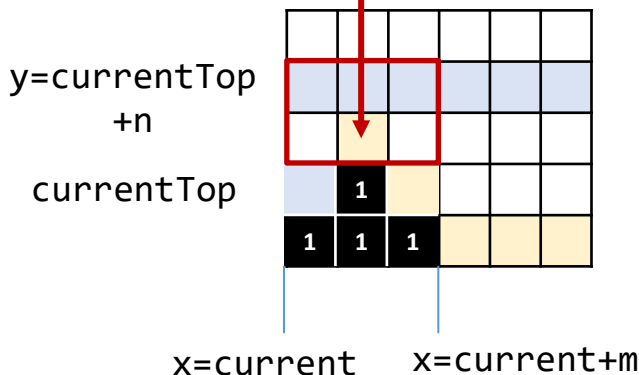
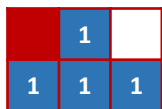
```
x = current;
```

```
y = currentTop + n;
```

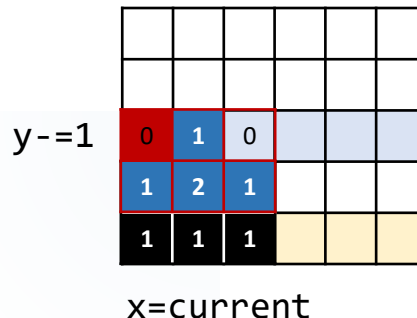
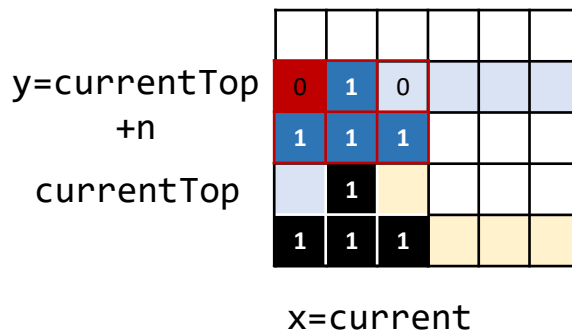
冲突检测
起始坐标

俄罗斯方块：网格图的冲突检测

m 方块宽
n 方块高



当前下落冲突检测的位置 $x=\text{current}$



初始化冲突检测开始位置

- 取 $x=\text{current} \sim x=\text{current}+m$ 之间最大的mapTop
- $y=\text{currentTop}+n$
(从方块即将接触原来堆积边界开始检测)

冲突检测

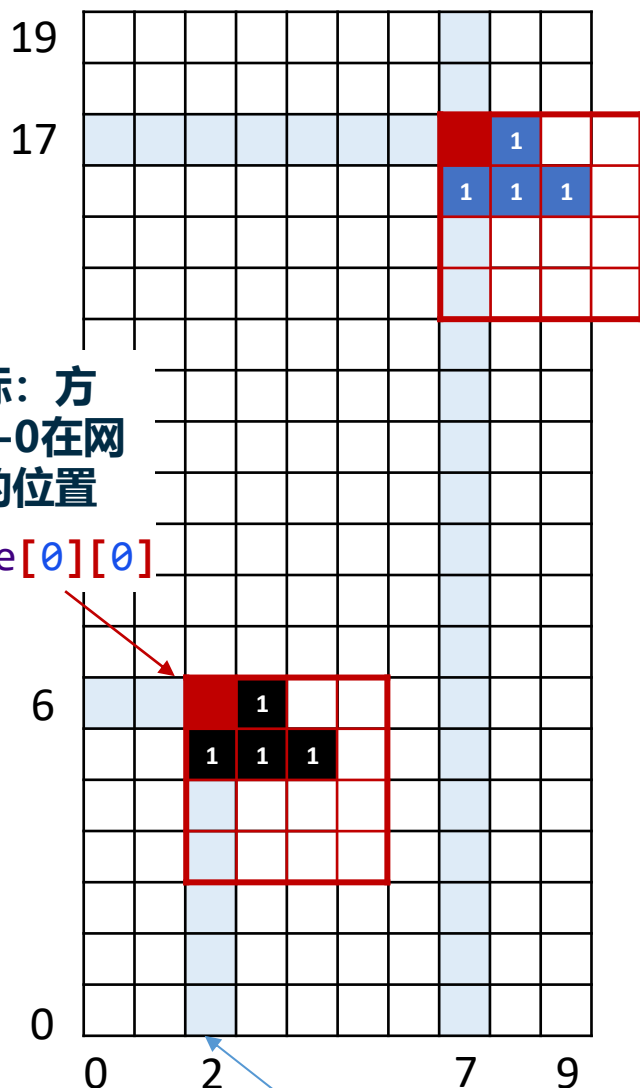
- $x=\text{current}$
 $y=\text{currentTop}+n$
- map格子与cube格子叠加，任何一个格子 >1 ，即有冲突；否则无冲突
- 无冲突：更新 y

继续下落并冲突检测

- $y-=1$ ，下落
- 如果有冲突，退回到上一步的坐标值作为下落后的位置坐标
- 有冲突，不更新方块的 y 坐标

俄罗斯方块：网格图的边缘检测

当前下落冲突检测的位置 $x = \text{current}$

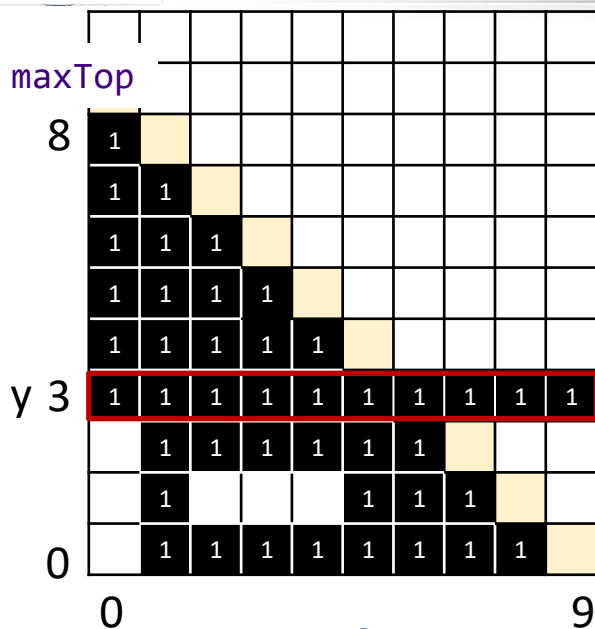


传统俄罗斯方块是左右逐步移动的，碰到边缘的时候可以判断边缘冲突，不允许再左右移动

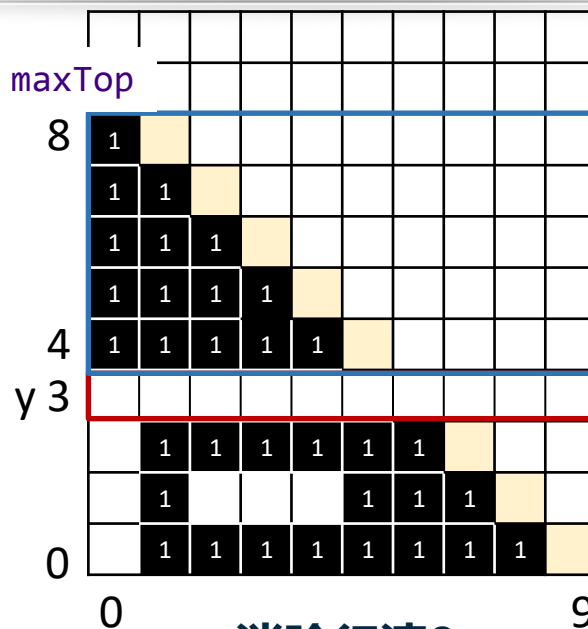
- 输入的x坐标是否会导致方块右侧触碰边界？
- $\text{WIDTH} - m$ 是方块位置坐标中 y 的最大值
- 数据结构保存方块实际宽度与高度

```
typedef struct
{
    int x, y;                // 方块位置坐标
    char* cube;              // 方块形状
    enum CubeType type;      // 方块类型
    enum CubeAngle angle;    // 方块旋转角度
    int m, n;                // 方块实际宽高
} Cube;
```

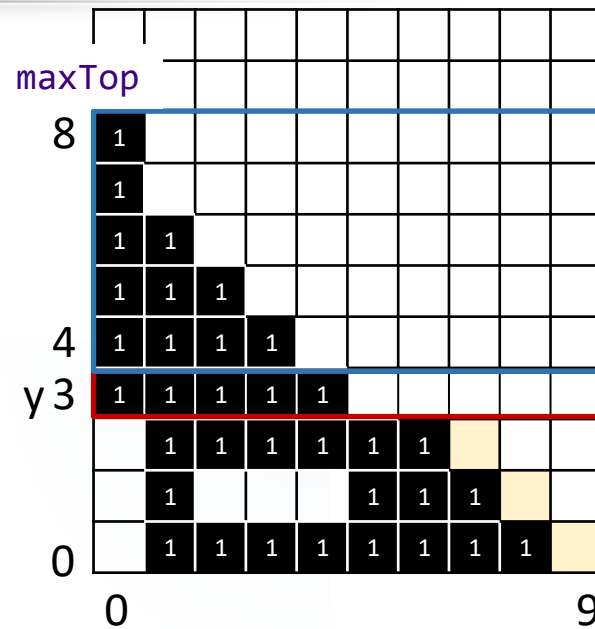
俄罗斯方块：删除行



— 判断消除行



— 消除行清0



将 $y+1 \sim \text{maxTop}$ 行

拷贝到 $y \sim \text{maxTop}-1$ 行

可以逐行拷贝

也可以内存块拷贝



将 maxTop 行清0