Life Game Project 设计文档

一、文件结构

源代码编写在三个文件中，分别是：`main.c`、`conway.h`、`conway.c`.

其中：

1. `main.c` 主要包含用户交互的部分；
2. `conway.h` 主要包含 `Conway` 类和相关函数的声明；
3. `conway.c` 主要包含 `Conway` 相关的函数实现。

二、程序实现过程

2.1 `main.c` 的实现细节

在 `main.c` 中，由于脚手架代码已经完成了大部分内容，因此最需要关注的仅有主函数中判断输入指令的一部分内容。

使用 `switch` 能简单地判断指令类型，之后使用 `scanf` 读入参数即可。对于非法指令的判定，由于细节繁多，仅仅使用了简单的 `default` 忽略。

在该处容易忽略的是使用 `delete_grids()` 函数及时清空内存。当初始化或读入数据时，由于原来的状态完全失效，甚至宽高都可能发生改变，因此需要及时清空内存。

我实现了额外功能：食物规则开关。这是因为注意到，食物规则会让细胞有“不规则”的移动，从而使原规则下的生命游戏中一些稳定的结构失效，无法达到观察效果，所以加入了此开关。

另一个实现的额外功能是关于随机化密度 `density` 的指令 `'p'` 。由于期望密度实际上就是随机生死时，每个细胞生存的概率，因此我决定让用户自己决定网格密度。

此外，我还额外实现了对于稳定状态和空白状态的特殊判断。在自动演化 `automatic_evolve()` 函数中，如果检测到下一个状态与目前完全相同，且食物规则被关闭，那么很容易证明之后的状态不再变化。由于无需考虑时间消耗，同时便于用户观察，因此我并没有选择停止演化计算，而是仅仅加入了提示，告知用户已经可以使用 `'b'` 指令停止演化；而如果检测到目前的网格已经完全空白，那么就永远无法诞生新的细胞，故同样加入提示。

2.2 `conway.h` 的实现细节

首先是状态设计。为了实现方便，使用了 `#define` 而非 `enum` 来实现存储状态的可读化。我使用了 `-1,0,1,2` 四个整数值来分别代表越界、死亡、存活、强化状态。我并没有存储食物状态，这是因为在我的实现方法中，食物在生成之初就直接被分配和消耗，这也符合作业要求中“回合开始”的描述。

之后是 `Conway` 的结构体设计。在脚手架代码给出的三个变量之外，我设置了 `maxFoodNum`、`density`、`food` 三个变量。其中，`maxFoodNum` 代表随宽高而自动生成的食物个数。这是为了可读性的考虑，而且将该数值单独储存而非每次计算有助于修改规则（只需要在初始化时修改）。`density` 和 `food` 分别代表了初始化时的期望密度（即生成存活细胞的概率）和食物规则的开关，其目的已在2.1部分说明。

然后说明函数功能。大部分函数由脚手架给出，在此不再赘述其功能，仅描述自定义的几个函数：

1. `void set_density(Conway *c, float p)`：该函数用于设置指定网格的 `density` 参数。封装为函数是安全性考虑，以及符合类设计的风格。
2. `void reverse_food_rule(Conway *c)`：该函数用于开关指定网格的食物规则。由于仅有开关两个状态，因此为了用户使用便捷，采用的是反转开关，而非设置状态。
3. `int get_neighbor(const Conway *c, int x, int y)`：该函数用于获取一个网格附近四个网格的活细胞数量。该函数在生成食物和演化中都有所使用。
4. `void init_food(Conway *c)`：该函数给指定的网格生成并消耗食物。
5. `bool Conway_cmp(const Conway *a, const Conway *b)`：该函数用于比较两个网格是否完全相同，设计目的是判断当前网格是否已经到达稳定状态（无食物规则时）。
6. `bool Conway_empty(const Conway *c)`：该函数用于判断一个网格是否已经完全空白。

2.3 `conway.c` 的实现细节

首先，我在该文件中导入了 `<math.h> <time.h> <stdbool.h>` 三个库，便于编写程序。由于此部分并不会介绍编程过程，因此仅选择值得注意的实现方法进行分析，而其他的繁琐步骤已经写在了注释里。

1. 在 `new_conway()` 函数中：

`res.maxFoodNum = ceil((m + n) / 4.0);` 一句是由于作业要求文档中对食物数量的描述为"地图的边长/2(向上取整)"。由于没能清楚知道是宽高中的哪一项，因此我取了平均数，同时用 `ceil()` 函数向上取整。另一种向上取整的方法是 `((m+n+3)/4)`，利用强制转化取整，此处为了可读性选择了库中的函数。

之后初始化二重指针的部分，首先初始化指向数组的指针，之后依次申请内存。

2. 在 `delete_grids()` 函数中：

需要注意释放内存的顺序和申请内存的顺序相反。这是因为如果先释放指向数组的指针，则无法释放第二维的指针。

3. 在 `init_random()` 函数中：

使用 `srand(time(NULL))` 来保证随机，之后通过随机浮点数依次决定每个位置的生死。

4. 在 `get_neighbor()` 函数中：

预定义了常量类型的 `dx[],dy[]` 数组，用于遍历周围四个格子。

5. 在 `init_food()` 函数中：

在这里使用了 `cnt` 记录已经生成的食物数量，和 `while (cnt < c->maxFoodNum)` 来判断食物何时生成完成。严格来说，这样的运行时间是不稳定的，这是因为可能多次生成到强化细胞的格子，而这种生成在程序中被视为无效生成，不计入生成次数。一种更好的（也是我尝试过的）做法是使用随机打乱数组算法，或使用类似于多重采样的稳定方式来获取随机数，但是考虑到食物数量较少，同时并不会面对太大量的数据，因此更简单的实现方法是性价比更高的。

另一个需要注意的点是：细胞移动过来消耗食物，等价于设置原位置为死亡，新位置为强化细胞。

6. 在 `next_generation()` 函数中：

将原来的无返回值改为了 `int` 类型返回值，以便获取前文提到的稳定/空白状态信息。先生成食物再演化，按照规则调用 `get_next_state()` 函数即可。

7. 在 `get_state()` 函数中：

判断越界的逻辑为 `if (x >= c->m || y >= c->n)` 这是因为编译时有 `warning` 提示：`x,y` 都是无符号型变量，不可能小于0，因此无需判断。

8. 在 `get_next_state()` 函数中：

按照规则实现即可。需要注意的是在回合开头被强化过的细胞不受限制，所以需要在一开始就特判，直接存活并恢复为普通细胞。

三、问题及经验总结

1. 在初始化 `Conway` 类型时，由于对数组和指针理解不足，没理解二维指针的本质，从而导致一开始申请内存方法有误，直接使用了 `malloc(m * n * sizeof(int))`，然而实际上，正确的方式应该是先申请指向每一行数组的指针数组，之后再依次申请每一行。
2. 在实现 `init_random()` 时，一开始简单地使用了 `rand()%2 ? STATE_ALIVE : STATE_DEAD` 来初始化。但这样的代码可维护性是很差的，仅能使用于两种状态且各占 50% 概率的时候。之后，为了调整存活概率，改为使用随机浮点数来判断。
3. 一开始，我并没有使用 `get_neighbor()` 函数来对周围格子计数，而是直接在 `init_food()` 和 `get_next_state()` 函数中单独写一遍。为了代码的良好封装，我定义了这个函数，同时也保留了更新的可能，例如将周围4个格子改为周围8个格子只需改变 `dx[],dy[]` 两个数组和 `for` 循环范围。
4. 在实现过程中，犯了类似将 `(x+dx[i],y+dy[i])` 错写为 `(x+dx[i],**x**+dy[i])` 的错误。之后，通过打印每次演化和食物生成的信息，手动查错，找到了该错误并成功修正。保留了调试代码，仅注释删除。
5. 在文件输入输出流中，需要使用 `fprintf()` 和 `fscanf()` 函数，而非常用的 `printf()` 和 `scanf()` 函数。这点是由于我对文件操作不熟悉导致的。