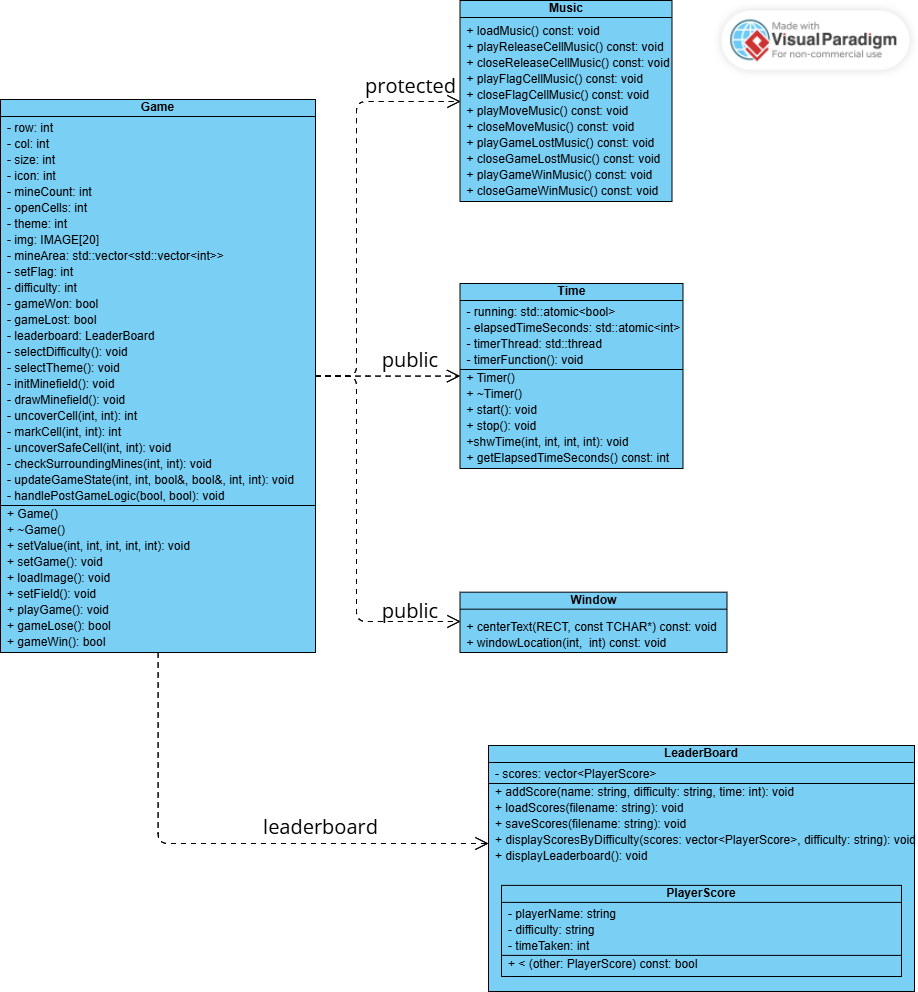
1. 类设计

1.类图

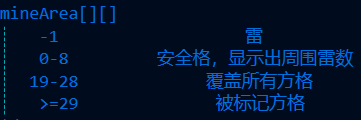


2.数据成员说明

**（1）Game类**

|  |  |  |
| --- | --- | --- |
| 数据成员名称 | 数据类型 | 含义与用途 |
| row | int | 存储扫雷盘面的行数 |
| col | int | 存储扫雷盘面的列数 |
| size | int | 存储每个方格的边长 |
| icon | int | 存储游戏其他图标的大小 |
| mineCount | int | 存储游戏中雷的总数，用于跟踪和设置游戏中的雷的数量 |
| openCells | int | 存储已打开的方格数量，用于跟踪玩家已经揭开的方格数 |
| theme | int | 存储当前主题的标识，可能用于切换游戏的视觉主题 |
| setFlag | int | 储存以标记方格的总数 |
| img | IMAGE[20] | 储存图像资源 |
| mineArea | std::vector<std::vector<int>> | 存储雷区布局的二维向量数组，用于记录游戏区域内每个方格的状态 |
| difficulty | int | 储存游戏难度级别 |
| gameWon | bool | 存储游戏是否胜利的状态 |
| gameLost | bool | 存储游戏是否失败的状态 |
| leaderboard | LeaderBoard | LeaderBoard类，用于记录玩家用时和排名 |

以下具体说明mineArea的含义和用途，它是用来记录每个方格的状态，其不同的值代表方格不同状态（如下图所示）。游戏初始化时，雷被随机放置，并且 mineArea 中的每个元素被赋予一个介于 -1 到 8 之间的值，-1表示雷，0~8表示方格周围的雷数。为了隐藏方格的初始状态，所有的 mineArea 值增加了 20，这样方格的值就会在 19 到 28 之间，从而实现一种简单的加密。玩家在游戏中的操作（如打开方格或标记方格）会改变 mineArea 中对应方格的值，通过加减 20 来反映方格的状态变化



**（2）Time类**

|  |  |  |
| --- | --- | --- |
| 数据成员名称 | 数据类型 | 含义与用途 |
| running | std::atomic<bool> | 用于跟踪计时器是否正在运行的状态标志。线程安全地检查和更新计时器的运行状态，以确保在多线程环境下正确同步 |
| elapsedTimeSeconds | std::atomic<int> | 存储计时器从启动到当前的总运行时间，单位为秒。用于记录时间流逝，可以被外部读取以显示或计算时间差 |
| timerThread | std::thread | 存储计时器功能运行的线程。 允许计时器在后台独立运行，不阻塞主线程，同时可以执行其他任务 |

**（3）LeaderBoard类**

|  |  |  |
| --- | --- | --- |
| 数据成员名称 | 数据类型 | 含义与用途 |
| scores | std::vector<PlayerScore> | 存储玩家分数的列表。用于维护一个玩家分数的集合，其中每个分数都包含玩家的姓名、难度级别和所用时间 |

其中嵌套结构体 PlayerScore 数据成员

|  |  |  |
| --- | --- | --- |
| 数据成员名称 | 数据类型 | 含义与用途 |
| playerName | string | 用户的姓名 |
| difficulty | string | 用户选择的难度级别，用于 表示用户在哪个难度级别下取得分数，用于分类和过滤分数 |
| timeTaken | int | 用户游戏胜利所耗时间，以秒为单位。根据所用时间对玩家的分数进行排名，时间越短，排名越前 |

3. 函数成员和全局函数

(1) Game类

1. **void setValue(int \_row, int \_col, int \_mineCount, int \_size, int \_icon)**

初始化游戏的基本参数，设置游戏盘面的行数、列数、雷的总数、单个方格的大小和其他图标的大小，重新初始化 mineArea 数组的大小。

1. **void selectDifficulty()**

利用EasyX绘制难度选择界面，界面上绘有不同的难度按钮。当玩家使用鼠标左键点击其中一个按钮时，GetMouseMsg() 函数会捕获鼠标点击事件。根据点击的位置，difficulty 值被设置为简单（1）、中等（2）或困难（3）。随后，游戏会根据所选的难度级别，使用 switch 语句和setValue（…）函数来配置游戏的行数、列数、雷的总数、方格大小和图标大小等参数，确保游戏难度与玩家的选择相匹配

1. **void selectTheme()**

绘制主题选择界面，界面上绘有不同的主题选项。当玩家使用鼠标左键点击其中一个按钮时，GetMouseMsg() 函数会捕获鼠标点击事件，一旦某个主题被选中，theme 值会根据玩家的选择进行更新。随后，加载与新主题相关的图像资源，确保游戏界面的元素。

**d）void loadImage()**

根据selectTheme()更新的theme值，加载玩家所选主题的图像资源，主要利用了EasyX中的loadimage（…）函数

1. **void initMinefield()**

在游戏开始时随机在雷区内布置雷，并更新周围方格以显示邻近雷的数量。实现这一功能时，首先将整个 mineArea 二维数组初始化为0，然后，使用洗牌算法在雷区中随机选择位置放置雷，并将这些位置在 mineArea 中标记为 -1。对于每个雷周围的8个相邻方格（如果存在），增加它们的值以反映邻近的雷数。

1. **void drawMinefield()**

负责生成并显示扫雷的盘面。首先初始化图形模式，设置窗口大小和背景颜色，然后清除屏幕以准备绘制。该函数通过循环遍历雷区的每个方格，根据 mineArea 数组中的数据，使用加载好的图像资源来表示不同的方格状态：未翻开的方格、数字方格、被标记的方格以及雷。同时它还绘制了游戏界面的其他元素，比如计时器、计数器（剩余雷数）和游戏控制按钮

**g） void playGame()**

核心函数，它首先初始化游戏环境，包括设置游戏参数、清空计时器、重置已翻开和已标记雷的方格计数。进入主循环后，该函数监听鼠标事件，通过 GetMouseMsg() 捕获玩家的操作。对于每一次鼠标左键点击，它调用 uncoverCell() 或checkSurroundingMines()来翻开方格；如果是右键点击，则调用 markCell() 来标记或取消标记雷。随着游戏进行，playGame() 调用updateGameState()来持续更新游戏界面，展示最新状态的雷区，包括翻开的方格、标记的雷和剩余雷数的实时计数，以及调用showTime()更新游戏所耗时间( s )。playGame() 还负责监控游戏的胜利条件——即所有安全方格被打开，或者失败条件——即玩家点击雷。在游戏胜利或失败时，它调用 handlePostGameLogic()来处理游戏结束的逻辑。

1. **void iupdateGameState(int total, int openCells, bool& gameWon, bool& gameLost, int x, int y)**

负责根据玩家的每一步动作来更新扫雷游戏的当前状态。它首先调用 drawRemainingMinesCount() 函数在界面上绘制或更新剩余雷数的显示，让玩家知道还有多少雷未被标记。接着，该函数检查游戏是否达到胜利条件，即所有非雷区方格都已被翻开，若是，则触发胜利逻辑并更新胜利状态为 true。如果玩家翻开了一个雷区导致游戏失败，函数会播放失败的音乐，执行失败逻辑，并将失败状态更新为 true

1. **void handlePostGameLogic(bool gameWon, bool gameLost)**

在扫雷游戏结束时被调用，无论是胜利还是失败。它首先将计时器重置为0，清除已设置的标记数，然后进入一个循环，等待玩家做出下一步操作。在这个循环中，函数通过 GetMouseMsg() 获取鼠标消息，检测鼠标左键的点击事件。如果玩家点击了界面上的特定区域，函数将执行相应的操作：如果点击了重新开始按钮，游戏将重置并重新开始；如果点击了退出按钮，将退出当前循环;入果点击了返回按钮，将重新选择难度和主题。循环将持续直到玩家选择退出，此时 exit 标志被设置为 true，循环随后终止，意味着退出游戏。

1. **int uncoverCell(int x, int y)**

处理翻开方格的逻辑。当玩家翻开一个方格，并且该方格在 mineArea 数组中的值介于19到28之间时，表示该方格之前是被加密的，现在需要解密并更新状态。代码首先将 mineArea[x][y] 的值减去20，将其转换到-1到8的范围内，这代表方格的实际状态：雷（-1）、没有雷（0）或者周围雷的数量（1-8）。每当一个方格被成功翻开，openCells 计数器增加，跟踪已翻开的方格数量。如果翻开的方格周围有雷（即 mineArea[x][y] 的值在1到8之间），则在界面上显示相应的雷数图像，并播放相应的BGM。如果翻开的方格周围没有雷（即值为0），则调用 uncoverSafeCell() 函数来递归地翻开周围所有安全的方格，并播放相应的BGM。如果方格下是雷（即 mineArea[x][y] 的值为-1），则播放失败的音乐，并调用 gameLose() 函数来处理游戏失败的逻辑

1. **int markCell(int x, int y)**

实现了扫雷游戏中的插旗和取消插旗功能。如果方格的值大于等于39，表示该方格已被标记（插旗），函数会减少20，更新方格状态以取消标记，同时更新界面显示和播放相应的BGM。相反，如果方格的值在19到28之间，表示方格未被标记，函数会增加20来标记方格，并更新界面显示和播放相应的音乐。无论插旗还是取消插旗，都会更新 setFlag 计数器，以跟踪玩家的标记数量

1. **void uncoverSafeCell(int x, int y)**

起始方格为安全方格（mineArea[x][y] = 0），并更新界面显示。然后，函数使用一个 std::vector 来存储待翻开的方格队列。对于队列中的每个方格，它检查周围相邻的方格，如果相邻方格的值是20（表示一个未翻开的安全方格），则将其翻开（减去20，并更新为0），更新界面，并将其添加到队列中以继续检查其周围方格。如果相邻方格的值在21到28之间（表示一个未翻开的方格，周围有雷），则仅更新界面并增加 openCells 计数。这个过程一直持续到队列中的所有方格都被处理完毕，从而有效地一次性翻开所有与起始方格相邻的安全区域

1. **void checkSurroundingMines(int x, int y)**

实现快速展开的特殊玩法，首先计算给定坐标(x, y)周围的八个格子范围，确保不超出棋盘边界。然后，遍历这八个格子，统计其中标记为地雷的数目。若统计的地雷数等于(x, y)位置上的数字（即该位置正确指示了周围地雷数量），函数将

再次遍历这8个相邻格子，若遇到标记为地雷或特殊状态的格子，触发游戏失败流程：播放失败音乐，设置游戏状态为失败。若遇到20至28范围内的数字格子（表示已标记但非地雷），减去20以还原其实际未标记时的值，并更新图像表示。同时增加已打开的单元格计数，如果该格子值为0，递归调用uncoverSafeCell()函数继续揭开周围的安全格子

1. **void gameLose()**

处理游戏失败的情况，对游戏区域中的每一个方格进行遍历并更新方格状态，如果方格标记为未被标记的雷，则调整其状态以显示为地雷，并更新图像；若方格被标记为旗帜（代码中40到48范围表示玩家标记的雷），同样调整状态并更新为对应图像，显示为已确认的地雷。同时显示失败画面，调用stop()函数终止计时

1. **void gameWin()**

处理游戏胜利的情况，对游戏区域中的每一个方格进行遍历并更新方格状态，打开所有地雷的位置给玩家看。在棋盘下方的固定位置放置游戏胜利的图片，并播放胜利音乐，调用stop()函数来结束计时。记录得分，根据难度级别（从数组difficultyLevel中选择对应字符串，基于当前游戏难度），将玩家名、难度等级以及游戏用时（以秒为单位）添加到排行榜（leaderboard）中。保存排行榜到文件leaderboard.txt中，确保成绩持久化。显示排行榜，让玩家能看到自己和历史成绩的排名

(2) Music类

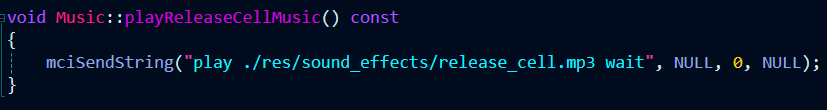
1. **void loadMusic() const**

加载游戏中使用到的各种音效文件。以其中一个为例，



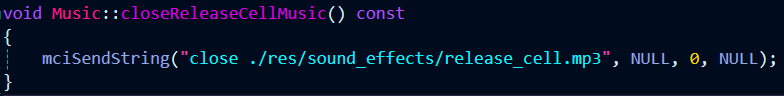
这行代码的作用是使用MCI命令打开位于./res/sound\_effects/路径下的release\_cell.mp3音频文件，用于释放单元格时的音效。NULL参数表示不接收返回的信息，0表示不等待操作完成，直接发送下一个命令

1. **void playReleaseCellMusic() const**



播放位于指定路径的release\_cell.mp3音频文件，并且程序会阻塞直到音频播放结束。其中wait表示继续执行下一条命令前等待当前音频播放完毕

1. **void closeReleaseCellMusic() const**



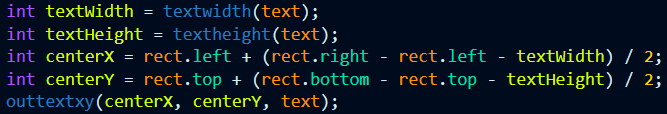
关闭之前打开的音频资源

剩余函数与后两个之间无明显差别，仅仅是不同音乐的区别，原理相同

(3) Window类

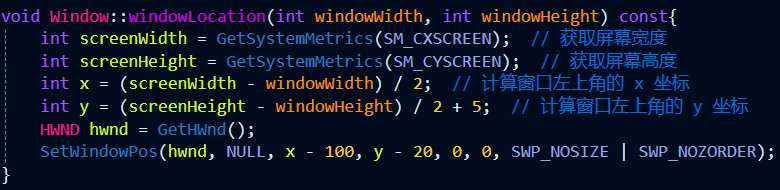
1. **void centerText(RECT rect, const TCHAR\* text) const**

它的功能是在一个矩形区域内居中绘制文本，首先使用textwidth(text)计算文本的宽度和使用textheight(text)计算文本的高度；然后计算水平居中位置centerX（矩形区域左右边距之差一半加上左边距）计算垂直居中位置centerY（矩形区域上下边距之差一半加上上边距）；最后使用outtextxy(centerX, centerY, text)在计算出的居中位置绘制文本



1. **void windowLocation(int windowWidth, int windowHeight) const**

使用GetSystemMetrics函数获取屏幕的宽度(SM\_CXSCREEN)和高度(SM\_CYSCREEN)；然后计算窗口左上角的理想x坐标，使得窗口在水平方向上居中，计算窗口左上角的理想y坐标，同样确保垂直居中，并且向上偏移了5个单位（避免太靠下，通过测试而来的经验结果）；最后调用SetWindowPos函数来改变窗口的位置（保持窗口的尺寸不变，并且不改变它与其他窗口的堆叠顺序，实现窗口在屏幕上的精确定位）



(4)Time类

1. **void showTime(int ROW, int COL, int SIZE, int icon)**

在界面上显示经过的秒数。首先设置文本颜色为红色，并加载自定义字体资源，设置字体样式和大小。将 elapsedTimeSeconds 转换为字符串，并计算文本宽度。

然后在指定位置输出时间字符串。最后，移除字体资源以释放资源

1. **void timerFunction()**

作为计时器线程的核心函数，负责更新经过的时间。使用 std::chrono::steady\_clock 来获取高精度的时间.在循环中，不断检查当前时间与上次记录时间的差值，如果至少过去了1秒，则 elapsedTimeSeconds 加一，并更新 lastTime 为当前时间。

1. **void start()**

如果计时器未在运行（!running），则将 running 设置为 true，表示计时器现在处于运行状态，然后创建一个 std::thread 对象 timerThread，它将执行 Timer 类的 timerFunction()。this 指针被传递给线程，以便 timerFunction 可以访问 Timer 实例的成员变量

1. **void stop()**

停止计时器，设置running为false，检查timerThread是否为可加入（joinable）状态来判断后台线程是否已经启动且尚未结束。如果线程是可加入的，调用timerThread.join()。它会阻塞当前线程（调用stop函数的线程），直到timerThread所代表的后台线程执行完毕自然结束，或者响应了停止信号并完成清理工作后结束

(5) LeaderBoard类

1. **void addScore(const std::string& name, std::string& difficulty, int time)**

向排行榜中添加一个新的玩家分数，创建一个 PlayerScore 结构体实例，包含玩家名称、难度和用时，然后将其添加到 scores 向量中。为了保持排行榜的有序性，使用 std::sort() 对分数列表进行排序。接着，计算并输出玩家在所选难度中的排名

1. void loadScores(const std::string& filename)

从文件中加载分数到排行榜。使用 std::ifstream 从指定的文件名中读取分数数据。对于文件中的每一行，解析出玩家名称、难度和用时，创建 PlayerScore 实例，并将其添加到 scores 向量中。在加载完成后，关闭文件流并对分数列表进行排序

1. void saveScores(const std::string& filename)

将排行榜中的分数保存到文件。使用 std::ofstream 创建或覆盖指定文件名的内容。遍历 scores 向量，将每个 PlayerScore 实例的数据写入文件中。写入完成后，关闭文件流

1. void displayScoresByDifficulty(std::vector<PlayerScore> scores, const std::string& difficulty)

显示指定难度下的分数排行榜。输出指定难度的排行榜标题，并遍历 scores 向量，对于与指定难度匹配的 PlayerScore 实例，将其数据格式化后输出到控制台

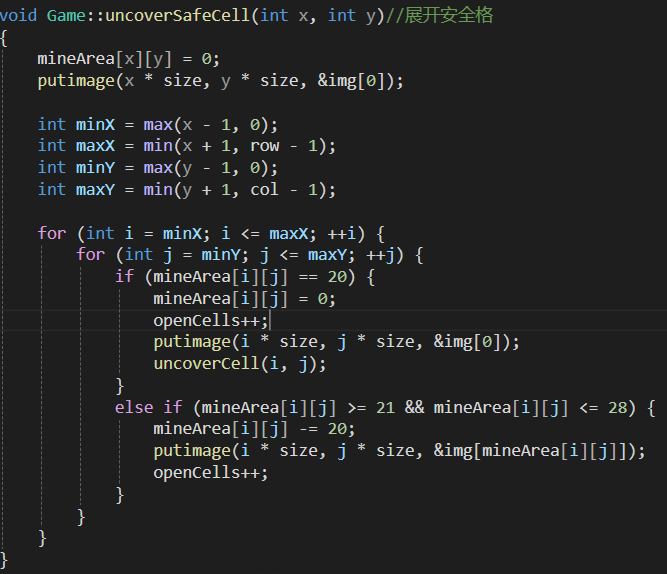
1. void displayLeaderboard()

显示所有难度级别的分数排行榜。调用 displayScoresByDifficulty 方法三次，分别为 "初级"、"中级" 和 "高级" 难度级别，显示所有级别的分数排行

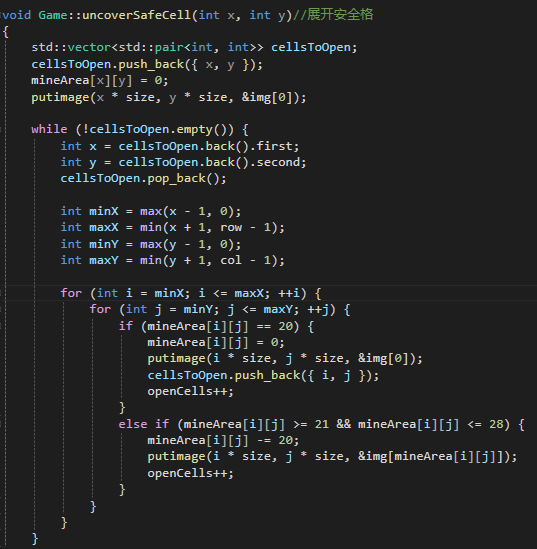
二、主要技术难点和及实现方案/算法设计

1.连锁反应算法

连锁反应，点击空白方格（周围无地雷）会引发连锁反应，自动揭开其周围所有相连的安全格。

首先，想到的是使用递归算法，当用户点击空白方格时，对每个邻近格子进行检查：如果发现空白方格，将其打开，更新游戏画面，并递归调用uncoverCell(i, j)函数继续展开该格子周围的空间，更新数据。如果发现数字方格，将其打开，更新游戏画面和数据。起初在初级难度测试时，由于相邻空白格数量不多，所以运行速度迅速，没有出现延迟和卡顿的现象。但是，在中级和高级难度测试时，由于相邻空白格数量多，出现了深度递归，导致运行速度缓慢，甚至游戏动画出现卡顿，对于扫雷这个追求速度的游戏造成了极大困扰。

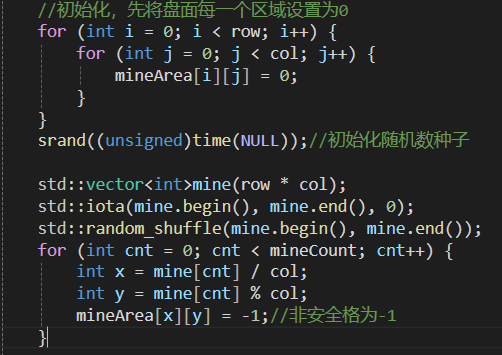
经过查阅资料，递归算法其实属于算法的基本结构，与其类似的还有迭代算法。首先，定义了一个cellsToOpen的vector容器（下面以容器指代cellToOpen），它的目的时为了储存待处理的方格的坐标（通过pair的操作已经将x和y捆绑在一起），并将已知的安全方格通过cellsToOpen.push\_back的操作放在容器的末尾。随后进入循环，结束循环的条件是容器中所有待处理的方格处理完（即容器变为空），在每次循环迭代开始时，通过cellsToOpen.back()获取容器末尾的坐标对，并立即使用pop\_back()方法将其移除，接着探索周围方格，若为安全方格则放入容器末尾，若为数字方格则打开，每次操作后均更新游戏画面和数据。随着所有可达且应展开的格子被处理完毕，容器最终会变为空，循环结束。此时，与初始点击安全方格相邻的所有可安全展开的区域都将被成功打开。

通过实际游戏对比两种算法和查阅相关资料，迭代算法运行速度的确比递归算法快，原因可能是如果相邻的空白格子很多，可能会导致深度递归，从而增加栈溢出的风险，而迭代方法使用循环和队列替代递归，有效避免了这一问题。并且迭代避免了额外的函数开销，内存使用更加稳定。

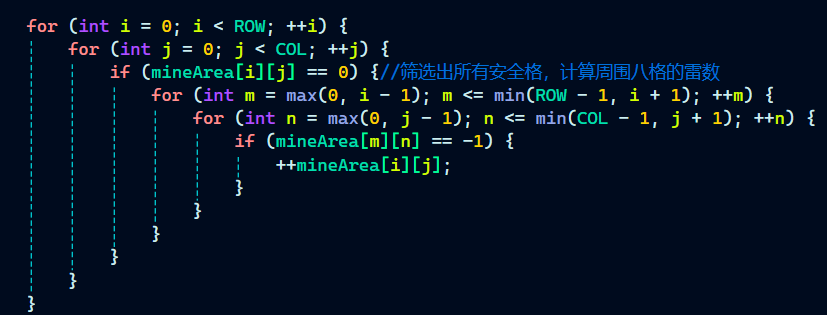
2.随机雷分布算法

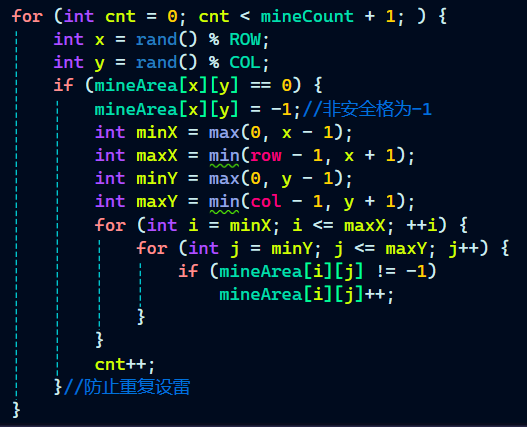
 一开始的基本思路是使用生成随机数，随机数除以行数得到的余数记为x坐标，除以列数得到的坐标设为y坐标，并且通过一个if条件判断避免重复设雷。但是这个算法的劣势非常明显，最直接的问题是可能会生成重复的位置，如果有重复，则需要重新选择位置。这种方式在雷数量较多时，算法效率低下，因为随着雷的增加，找到一个未被占用的位置越来越困难，可能需要多次尝试，浪费时间和计算资源。

后来决定基于Fisher-Yates shuffle洗牌算法解决随机雷分布问题。首先通过srand设置随机种子确保随机性。接着，构建一个代表游戏盘面所有方格的vector容器，并用iota函数填充连续整数作为初始索引。利用random\_shuffle对容器内部的元素进行洗牌，实现方格索引的随机排列，以此决定地雷的随机且均匀分布位置。随后，取洗牌后的前mineCount个索引设置对应的方格设为地雷。改进后的算法优势主要体现在，一，实现了每个方格设置为雷的可能性相同，进而达到彻底的随机分布，避免某些区域的雷过于密集或过于稀疏的情况出现（第一种算法从定性的角度来看，没有实现概率相同）；二，不会出现雷重复的问题，也不需要再调用资源进行重复筛选，这样提高了运行效率，并节省时间和资源；三，在必要的时候，该算法可以通过相同的初始种子值，实现相同的雷区布局，有利于调试程序，或者说方便用户多做“错题”提高扫雷水平。



3. 统计安全格周围雷的数量

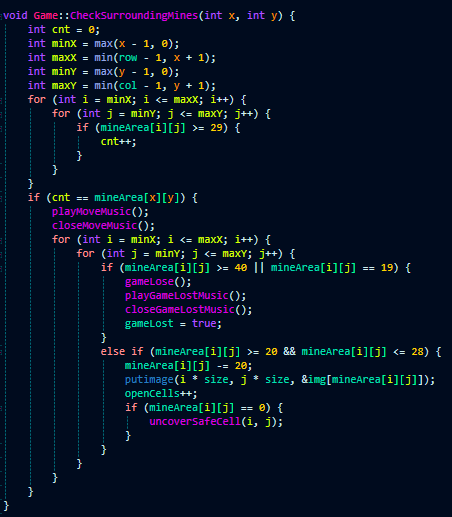
 一开始想法简单粗暴，直接筛选出所有的安全格，然后统计安全格周围雷的数量，套了4层for循环和2次if条件判断，问题还是因为效率。首先，它要遍历整个二维数组，检查是否是安全格，如果是之后还要进一步遍历其周围方格统计雷数量，这一部分已经很占用资源和时间了，主要问题还是对于每个非雷格，都会执行一次额外的嵌套循环，这导致效率极低。

 后来就摒弃了这种方式，转为以雷为中心，在随机生成雷的同时，同时计算周围安全格的雷数，避免了单独的遍历计算过程，提高了效率。将随机生成雷和计算安全格周围雷数和为一体，仅对实际布下的雷进行周围格子的雷数累加，减少了不必要的检查和计算，更加高效。

4.快速开启

快速开启，如果玩家确定某个数字周围的所有地雷都已被正确标记，可以按下左键来快速揭开其周围剩余的安全格。

尝试了两种解决方案，方案一就是直接遍历数字方格周围的方格，统计标记方格数量，在对比周围雷数量进行判断。

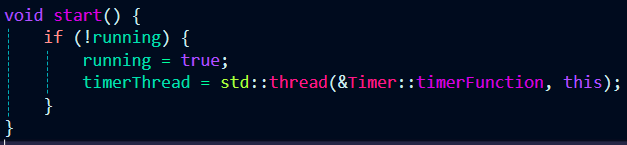
方案二是定义一个和mineArea[][]一样的二维数组flagArea[][]，每标记一个方格，其周围方格的flagArea[][]++,最后对比周围雷的数量和flagArea[][]即可，这样可以有效避免循环遍历的操作，不利之处在于标记方格时需要循环遍历。

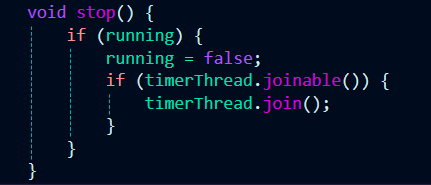
通过实际操作对比两种解决方案，意外发现方案一其实效果更好，但是它从认知角度来看非常简单粗暴，目前我也没有想到更好的解决方案，最终还是选择使用方案一。

5.游戏时间记录

一开始想法很单纯，写一个Timer类，进入游戏start()启动计时器，游戏结束stop()结束计时器，但是这样做问题特别多，当计时器在游戏的主线程中运行时，游戏主线程的执行经常会被阻塞

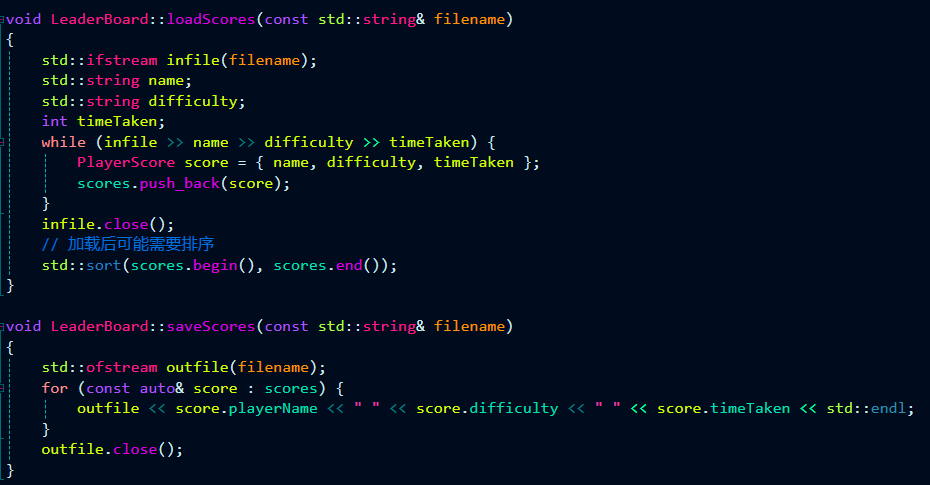
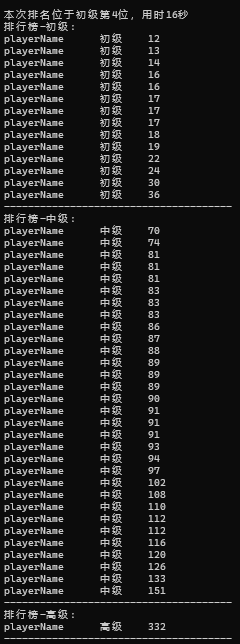
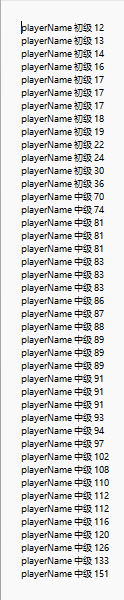
通过将计时器的逻辑放在一个独立的线程中运行，可以避免主线程的阻塞影响到计时器的准确性。这样，即使游戏主线程因为某些原因暂时无法继续执行，计时器线程仍然可以独立运行，保持时间的连续性。

在游戏开始时，通过调用线程的 start() 方法来启动计时器线程

在游戏结束或需要停止计时器时，通过调用线程的 join() 方法来等待计时器线程安全结束

6.文件读写

loadScores()负责从文件中读取分数数据，更新排行榜状态，并保持分数的有序性。saveScores ()方法则负责将当前排行榜的状态持久化到文件中，这样即使游戏关闭，分数数据也不会丢失



右下为控制台输出内容，左下为记事本记录