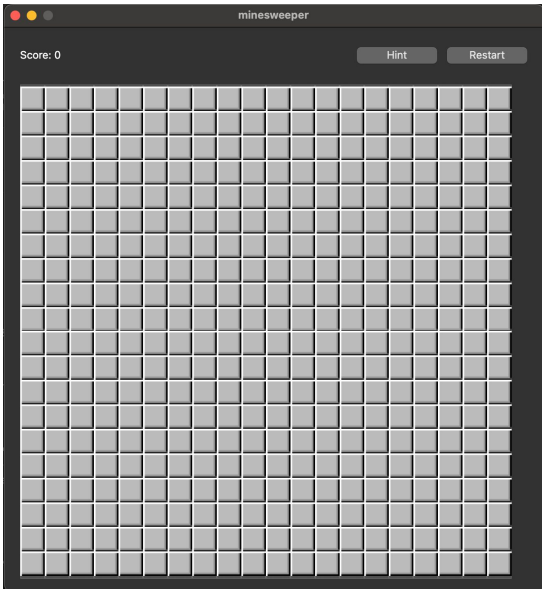
Documentation

1- Introduction

Minesweeper is an iconic logic puzzle video game developed by Microsoft in early 90s. The aim of the game is to uncover all squares in a grid that do not contain a mine. Players must carefully deduce which squares are safe and which squares are mine based on the numbers on the cells. This project is an exact replica of this classic developed using C++ and QT framework.

2- Game Mechanics and Usage

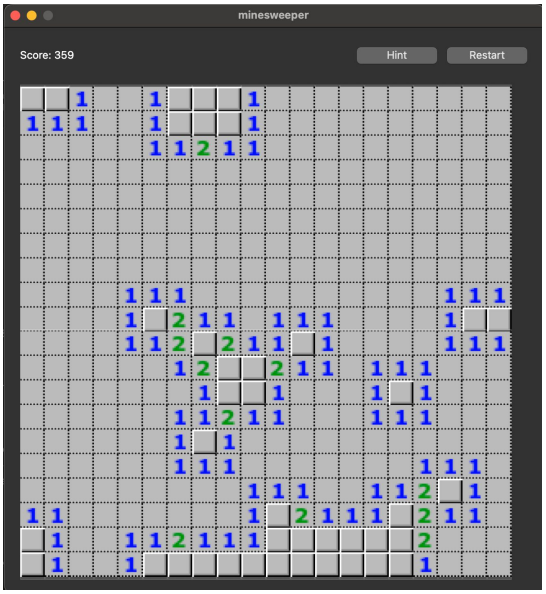
When the game is started, a grid with unrevealed cells is displayed.



Then, the player may choose one of these actions:

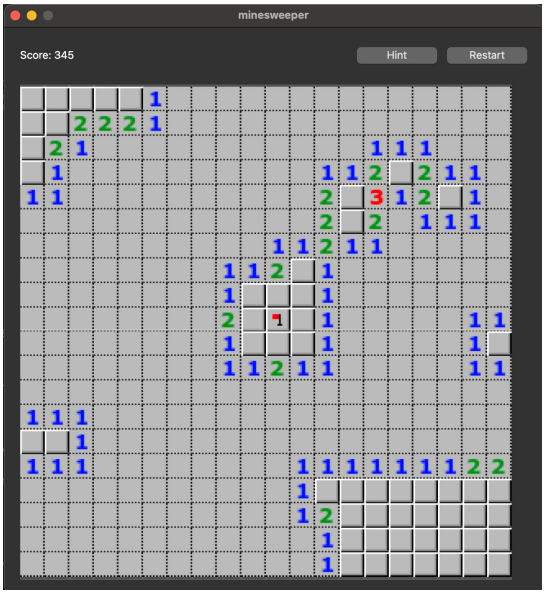
A. Left click a cell

This action simply reveals a cell and its suitable neighbors. If the cell is already revealed, it does nothing.



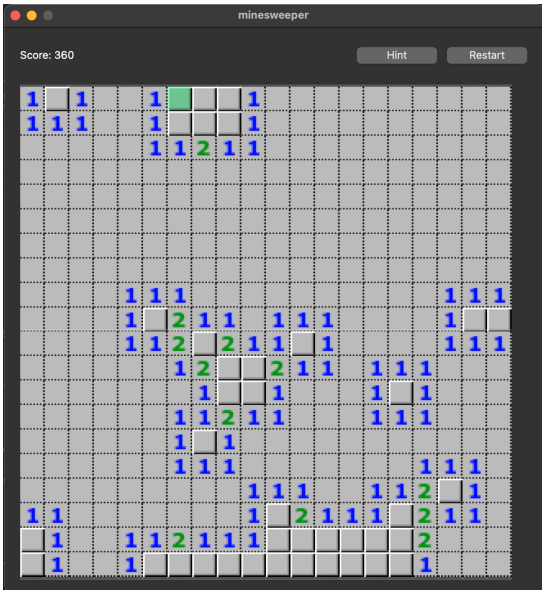
B. Right click a cell

This action simply flags a cell. When a cell is already flagged, right clicking it doesn't change anything. However, right clicking a flagged cell simply unflags it.



C. Hint

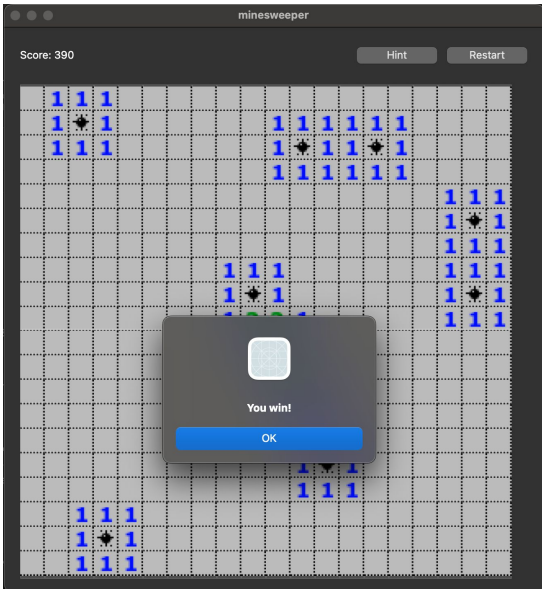
This button gives a hint to the player based on the current state of the game. If hint button is clicked twice, given hint is clicked automatically. If there are no hints available, nothing happens.



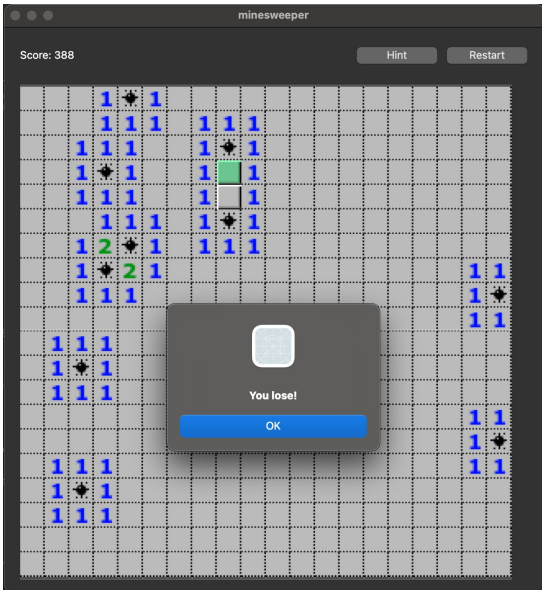
D. Restart

This function simply restarts the board.

When the player opens all non-mine cells, he or she wins the game.



However, if the player clicks a mine cell, the game is over and “You lose” pops up.



3- Design and Implementation

The implementation process of the program can simply be examined in two parts, which are UI design with state management and backend design.

3.a - UI Design, State Control and Implementation

The UI design aims to create the original and well-known design of “Mine Sweeper” game. To achieve this we have implemented our design mainly in 2 files, namely ***cellbutton.cpp*** and ***main_layout.cpp.*** In this section we are going to analyze the structure and implementation details of these files. The state updates are achieved thanks to the use of QT native feature signal and slot connections, which will be explained in detail in section ***3.b.***

Our main UI design is divided into two parts where the first part is top layout and bottom layout. The top layout contains score label, hint and restart buttons while the bottom layout contains the grid for cells.

Firstly we will start the analysis of the cellbutton interface with function signatures;
signals:

- Signals:
  1. void leftClicked()
  2. void rightClicked()
- Protected functions:
  1. void mousePressEvent(QMouseEvent *event)

The mousePressEvent function takes a single event and determines with a selection whether the given event is a right click or a left click. Depending on the selection results function emits signals which will then be used for manipulating grid state.

```
if (event->button() == Qt::LeftButton) {
        emit leftClicked();
    } else if (event->button() == Qt::RightButton) {
```

```
        emit rightClicked();
    }
```

Now let us consider the **private data fields** used in main_layou.cpp:

```
int numRows;
int numCols;
int score;
bool waitUntilRestart;
int numMines;
int hintPressedCount;
int hintRow;
int hintCol;
int *giveHint(std::vector<std::vector<Cell>> *grid);
std::vector<QPixmap> icons;
enum CellType {
    ZeroCell = 0,
    OneCell = 1,
    TwoCell = 2,
    ThreeCell = 3,
    FourCell = 4,
    FiveCell = 5,
    SixCell = 6,
    SevenCell = 7,
    EightCell = 8,
    EmptyCell = 9,
    FlagCell = 10,
    HintCell = 11,
    MineCell = 12,
};
QLabel *scoreLabel;
QPushButton *hintButton;
QPushButton *restartButton;
QGridLayout *gridLayout;
QHBoxLayout* setupTopLayout();
QGridLayout* setupBottomLayout(QSize cellSize);
std::vector<std::vector<Cell>> *grid;
```

As name suggests ***numRows*** and ***numCols*** hold the number of rows and columns. ***Score*** holds the current score, ***numMines*** holds the number of mines in current game, ***hintRow*** and ***hintCol*** holds the position of the hinted cell at the map after hint button is pressed. ***hintPressedCount*** and ***waitUntilRestart*** is used for altering and limiting the user actions; namely hint button press and right and left click actions on cells. ***CellType*** is a numeration for increasing readability in cell types, ***grid*** is a multidimensional array for holding Cells (data storage class for cell properties which is

explained in detail in ***3.c.1***). Other properties are related to the ui components that are part of QT component library.

Now we can consider the signals:

```
void lost();
void won();
void revealedCell(int row, int col);
void scoreUpdated(int newScore);
void restart();
void hint(int row, int col);
```

As name suggests ***lost*** and ***won*** signals are emitted in game over conditions, ***scoreUpdated*** is used when new cells are revealed and score should be updated. ***restart*** is for resetting the map and recreating the map state. ***hint*** is used for signaling the hinted cell found by the algorithm which will be described in detail in 3.c.3 ***revealedCell*** is for signaling in case user wants to reveal a new cell any algorithm decides to reveal a cell according to the game rules.

Here is the slots whose implementations will be discussed later in state management part.

```
void updateScoreLabel(int newScore);
void restartGame();
```

updateScoreLabel updates the score label and ***restartGame*** restarts the game by altering the cell state hence the frontend.

Now we will consider the functions in main_layout in detail with all the implementation details.

1. Constructor

   In this function we initialize the variables, create the icons vector for cells, call the functions to create the bottom and top layouts of our UI.

2. ***setTopLayout***

   The ***setupTopLayout*** function is responsible for configuring and returning the top layout of the game . Top layout includes the score label, hint button, and restart button, organized horizontally.

   1. **QHBoxLayout Initialization**:
      - A new horizontal layout (***QHBoxLayout***) named ***topLayout*** is instantiated.
   2. **Score Label**:
      - A ***QLabel*** named ***scoreLabel*** is created with the initial text "Score: 0".
      - The score label's alignment is set to left alignment horizontally and center alignment vertically.
      - The score label is added to the ***topLayout***.
   3. **Hint Button**:

- A ***QPushButton*** named ***hintButton*** is created with the text "Hint".
- A lambda function is connected to the hint button's clicked signal, which handles hint functionality:
  - If the game is not waiting for a restart and the hint button has not been pressed previously, the ***giveHint*** function is called to provide a hint.
  - If a hint is provided, the hint row and column are updated, the hint press count is set to 1, and the ***hint*** signal is emitted.
  - If the hint button has been pressed previously, the cell at the hinted location is revealed, and the hint press count and hint coordinates are reset.

4. **Restart Button**:
   - A ***QPushButton*** named ***restartButton*** is created with the text "Restart".
   - The restart button's clicked signal is connected to the ***restartGame*** slot.
5. **Adding Components to Layout**:
   - The hint button and restart button are added to the ***topLayout***.
   - The alignment for these buttons within the ***topLayout*** is set to right alignment.
6. **Score Update Connection**:
   - The ***scoreUpdated*** signal is connected to the ***updateScoreLabel*** slot to update the score label when the score changes.

3. ***setBottomLayout***

   The ***setupBottomLayout*** function configures and returns the bottom layout of the Minesweeper game interface. This layout contains the grid of cells, each represented by a ***CellButton***. It also sets up the necessary signal and slot connections to handle user interactions and game state updates.

   Detailed Description:

   1. **QGridLayout Initialization**:
      - A new grid layout (***QGridLayout***) named ***gridLayout*** is instantiated.
      - The horizontal and vertical spacing of the grid layout is set to 0.
   2. **Creating and Adding Cell Buttons**:
      - The function iterates through the grid using nested loops, creating a ***CellButton*** for each cell.
      - For each cell:
        - A ***CellButton*** is instantiated.
        - The default icon for an empty cell is assigned to the button.
        - The cell button's margins are set to 0.
        - The button's size is fixed based on the provided cell size.
        - The button is added to the grid layout at the corresponding row and column.

3. **Connecting Signals and Slots**:
    - Now we will consider the details of signals and slots that were mentioned before
        - **Left Click**:
            - When a cell button is left-clicked, the lambda function checks if the cell is not flagged, not revealed, and the game is not waiting for a restart.
            - If the cell contains a mine, the game enters the lose state, all mines are revealed, and a lose popup is displayed.
            - If the cell does not contain a mine, the cell and its neighbors are revealed using the BFS algorithm, and the hint state is reset if necessary.
        - **Reveal Cell**:
            - The ***revealedCell*** signal updates the icon of the cell button based on the number of neighboring mines and marks the cell as revealed.
        - **Right Click**:
            - When a cell button is right-clicked, the lambda function toggles the flag status of the cell if it is not revealed, the game is not waiting for a restart, and the cell is not the current hint cell.
            - The icon of the cell button is updated accordingly.
        - **Lose Condition**:
            - The ***lost*** signal updates the icon of the cell button to a mine if the cell contains a mine.
        - **Win Condition**:
            - The ***won*** signal updates the icon of the cell button to a mine if the cell contains a mine.
        - **Restart**:
            - The ***restart*** signal resets the icon of the cell button to an empty cell.
        - **Hint**:
            - The ***hint*** signal updates the icon of the cell button to indicate a hint cell.

4. ***didWin***

    The ***didWin*** function checks if the player has won the game by verifying that all non-mine cells are revealed. It iterates through the grid and ensures that every cell which is not a mine has been revealed.

```
bool won = true;
    for (int i = 0; i < getRows(); i++) {
        for (int j = 0; j < getCols(); j ++) {
            if (!isRevealed(i, j) && !isMineAtCell(i, j)) {
                won = false;
                break;
            }
```

```
        }
    }
    return won;
```

Other functions are mainly getter and setters for data encapsulation which won't be explained in detail.

One challenge we have encountered was the sizing of the grid. We first took an approach of letting the height and width of the grid linearly proportional with number of rows and cols time icon size. However there were some problems about the cell spacings as we increased or decreased the number of cols and rows. Hence we decided to set the spacing of cells inversely proportional to the number of rows and columns hence solved the bug in UI.

3.b - Backend Design and Implementation

The back-end of the program can be examined in 3 main parts: Storage of cell data, openNeighbors algorithm, and hint algorithm.

3.b.1 Storage of Cell Data

For each cell of the grid, a cell class instance is created. A typical cell object has 3 integer data fields which are row, column, and neighborMineCount and 3 boolean properties: isMine, isFlagged, and isRevealed. Depending on the necessity, these data fields are manipulated using getters and setters.

```
// Constructor of the cell class
Cell::Cell(int neighborMineCount, bool isMine, bool isFlagged,
                    bool isRevealed, int row, int col)
: neighborMineCount(neighborMineCount), isMine(isMine),
    isFlagged(isFlagged), isRevealed(isRevealed), row(row), col(col) {}
```

When the program starts, a 2D vector named grid is created. This vector represents the game board and is initially empty.

```
std::vector<std::vector<Cell>> grid;
```

Then, createMap function is called to fill this vector with Cell objects. During this process, createMap places mines at random locations within the grid, ensuring each game starts with a unique mine layout.

```
std::vector<std::vector<Cell>> createMap( int numRows, int numCols,
        int numOfMines );
```

As the game continues, the cells are accessed through grid vector and modified accordingly using getters and setters of the Cell class.

3.b.2 openNeighbors Algorithm

When an unrevealed cell is clicked and the cell is not a mine, openNeighbors function is called. This function is responsible for making adjacent cells visible if they are also unrevealed and not mines. Also, if the revealed cell is adjacent to a mine, rather than just showing an empty cell, its neighborMine count is showed. This process is recursive, meaning cells are revealed until a mine is encountered or the boundary of the map is reached.

```
void MainLayout::openNeighbors(
    std::vector<std::vector<Cell>> *grid, int row, int col );
```

In order to achieve this recursive behavior, a **Breath First Search** algorithm is used.

```
void MainLayout::bfsHelper( std::vector<std::vector<Cell>> *grid,
    int row, int col, std::queue<Cell> *myQueue, std::vector<Cell> *visite
```

BFS Algorithm:

```
BFS(row, col):
    VISIT(row, col)
    REVEAL(row, col)

    IF(NEIGHBOR_MINE_COUNT(row, col) > 0):
            POP_QUEUE()
            BFS( poppedRow, poppedCol )
            RETURN;

    FOR (neighborRow, neighborCol) in NEIGHBORS(row, col):
        IF not(VISITED(neighborRow, neighborCol)):
            ADD_QUEUE(neighborRow, neighborCol)

    POP_QUEUE()
    BFS( poppedRow, poppedCol )
```

The BFS algorithm first visits the current cell and reveals it. After revealing the cell, it checks if it reached to a mine neighbor. If so, it doesn't add its neighbors to the queue, it just pops from the queue and runs the bfs again. Otherwise, it adds all available neighbors to the queue. Then, it pops and runs the bfs again.

After helper bfs function reveals the corresponding cells accordingly, score is properly calculated and win conditions are checked.

3.b.3 Hint Algorithm

The hint algorithm returns the coordinate of a cell which is guaranteed not to be a mine. However, while determining the cell, it only uses the information available to the player. If it is not possible to deduce a guaranteed safe cell, the algorithm simply returns null. However, this hint algorithm is not perfect. Even though, it doesn't return a mine as a hint, in some cases, it may not find a hint although it is possible to do so. The reason why it is not implemented perfectly is

because a perfect hint algorithm is very complex implement and it is not expected in the homework.

Hint Algorithm:

There are 3 states for a cell

```
0  --> not determined
-1 --> mine guarenteed
1  --> empty guaranteed
```

In the beginning, mark all cells as not determined. Then, mark all revealed cells as empty-guaranteed.

Then, while a new info can be deduced, all cells are traversed and mine_guaranteed and empty_guaranteed cells are marked. In this process these rules are applied:

> If a cell's mineNumber is equal to the sum of the number of guaranteed mine neighbors and number of not determined neighbors, then all not determined neighbors are definitely mines.
>
> If a cell's mineNumber is equal to number of guaranteed mine neighbors, then all not determined cells are definitely empty.

After determining mine_guaranteed and empty_guaranteed cells, coordinates of the first encounter of empty_guaranteed cell is returned.