**CCPROG2 Machine Project (Tiam-Lee)**
**Minesweeper**

**Release Date:** January 29, 2024 (H)
**Due Date:** April 1, 2024, 12:00 noon

# 1 Project Description

In this project, students will apply the skills they have learned in the course to design and implement a fully-functional text-based MINESWEEPER game. Through this project, students should be able to demonstrate the following learning outcomes:

- **LO1.** Analyze problem requirements by describing input specifications, processes, and target output.
- **LO2.** Design and implement algorithmic solutions from defined problems and requirements by applying knowledge of computing fundamentals using appropriate data types and constructs including recursion.
- **LO3.** Design, execute, and document various classes of test cases and their corresponding results.
- **LO4.** Determine and apply proper debugging techniques using programming constructs and/or computing tools.
- **LO5.** Apply simple coding techniques such as inline comments, version documentation, and following coding standards for program readability.
- **LO6.** Exhibit intellectual honesty, responsibility, and punctuality, conforming to Christian principles.

# 2 Project Specifications

This section contains the specifications for the program to be implemented in this project.

## 2.1 Overview

In this project, students must implement a C program for the game MINESWEEPER. The program should allow a user to play the MINESWEEPER game multiple times, following the rules of the game which are outlined below. Additionally, the program must allow the user to create their own MINESWEEPER levels which can be saved in text files and retrieved later.

## 2.2 Minesweeper

MINESWEEPER is a logic puzzle video game generally played on personal computers. In MINESWEEPER, the goal is to clear the board by inspecting on tiles in a grid. In doing so, the player must avoid detonating the mines which are randomly assigned to specific cells in the grid. The game is won when all non-mine tiles have been revealed. On the other hand, the game is lost when the player inspects a mine.

The game was greatly popularized by Microsoft in the 1990s, having been bundled with every Microsoft Windows operating system until Windows 8. The game was also bundled with other operating systems such as KDE, GNOME, and Palm OS.

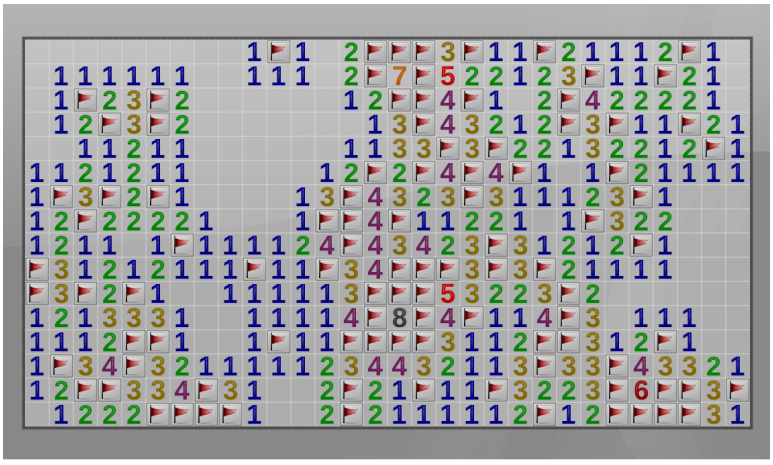


Figure 1: A finished game of KMines, a free and open-source variant of MINESWEEPER (Photo by Brandenads/ CC BY-SA 4.0)

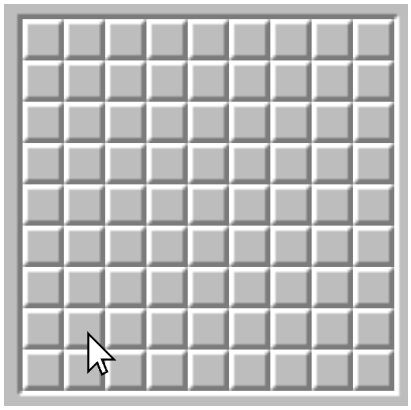
MINESWEEPER is a single player game. The game is played on a grid of cells. Some of the cells contain mines, which are randomly assigned before the game starts. These mines are not visible to the player. The rules of the game are as follows.

1. In each turn, the player chooses a specific cell to inspect. This tile is then "revealed".
2. If the cell contains a mine, it explodes and the player loses.
3. Otherwise, if there is at least one mine adjacent to the cell, a number is displayed showing the total number of mines adjacent to it. Note that this number can never be more than 8, as there are only a maximum of eight adjacent tiles to any cell.
4. If there are no mines adjacent to the cell, all the adjacent cells that are not yet revealed are automatically revealed as well. This effect can cascade (e.g., if some of the revealed tiles also don't have mines around them).
5. The player can place and remove markers (usually represented with a flag) to help him remember where the mines are.
6. If the player has revealed all non-mine tiles, the player has won the game.

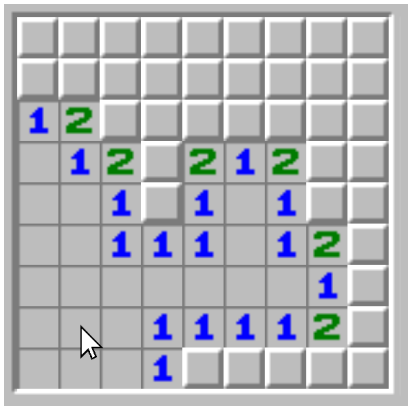
When the game ends (regardless of whether the player has won or not), the locations of all the mines are revealed to the player, and total time taken is displayed.

There are many online versions of MINESWEEPER, one of which can be found here. Please take the time to play the game yourself so you can be familiarized with the rules and nuances of the game.

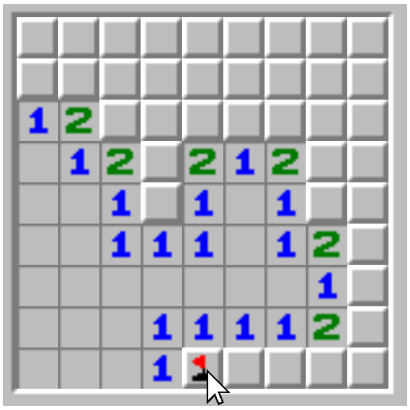
The following shows a simple step-by-step example of a MINESWEEPER game.



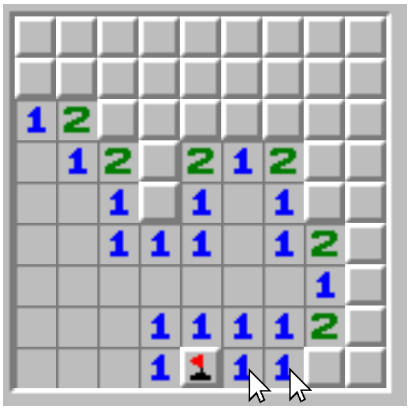
(a) Game begins. Player inspects a tile.



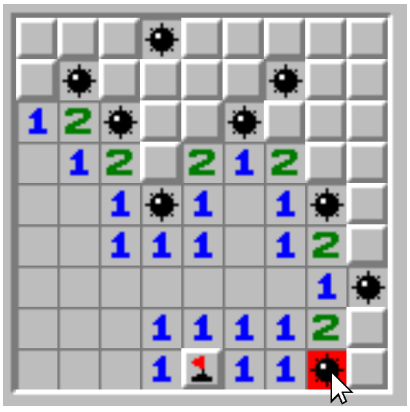
(b) There are no mines around the selected tile. The surrounding tiles are also revealed in a cascading manner.



(c) The player puts a flag on this cell, since it definitely contains a mine!



(d) Given that knowledge, player can now safely inspect these two tiles.



(e) Player makes an unwise choice to inspect this tile. The mine explodes, and the game is lost. Locations of all mines are revealed.

## 2.3 Program Flow

When the program starts, the program displays the profile menu, which lets the user pick a profile See section 2.3.1 for more details on profiles. When a profile is picked, the program enters the main menu.

The main menu should contain **at least** five options:

- **Start Game:** If the user selects this option, the program should proceed to the game proper. See section 2.3.2 for more details.
- **Create Level:** If the user selects this option, the program should go to the level editor. See section 2.3.3 for more details.
- **Change Profile:** If the user selects this option, the program should go to the profile selection menu to allow the user to change profile, create a new one, or delete an existing profile. See section 2.3.1 for more details on profiles.
- **View Statistics:** If the user selects this option, the program should display the statistics screen. See section 2.3.5 for more details.
- **Quit:** If the user selects this option, the program should terminate without crashing (i.e., runtime error).

### 2.3.1 Profiles

Profiles are part of the game as a way to let multiple players (i.e., accounts) play while keeping track of their own respective statistics. Each profile is uniquely identified by a name. There can only be a **maximum of 10 profiles** within the game.

Profile information persists even when the program is terminated, which means that you must store them in a file. It is up to you to design the format of this file (or files), as long as you are able to retrieve all the information needed for the functionalities stated in this document.

The profile menu allows the user to create a new profile, select an existing profile, or delete an existing profile.

When creating a new profile, the program should ask the user to input the name of the profile. Profile names are strings containing uppercase letters only, with a minimum of 3 and a maximum number of 20 characters. Additionally, each profile name must be **unique**. Profile creation should only proceed if the name is valid. Once a profile is successfully created, it is also automatically selected, and the program should proceed to the main menu.

When selecting an existing profile, the program should display a list of existing profiles, in **alphabetical order**. The program should then ask the user which profile they want to load.

When deleting an existing profile, the program should ask the user to input the name of the profile to be deleted. If it exists, the program should ask for a confirmation to proceed with the operation. Once confirmed, the program should delete all information about that profile. The program should then return to the profile menu.

Each profile keeps track of a set of statistics for each player (i.e., account). For more information on which statistics need to be kept track of, refer to section 2.3.5.

### 2.3.2 Game Proper

When the user starts a game, the following flow should be observed:

1. The program should ask the user if he wants to play a Classic Game or a Custom Game.
2. If the user selects Classic Game, there are two options: Easy and Difficult. The only difference is the size of the board and the number of mines. For Easy, the board size is $8 \times 8$ (8 rows by 8 columns) with exactly 10 mines. For Difficulty, the board size if $10 \times 15$ (10 rows by 15 columns) with exactly 35 mines. The program should generate a random level of the desired difficulty. A different level should be generated every time this option is selected.
3. If the user chooses Custom Game, the program should ask the user to input the level file he wants to play. The program should then load the level from this file. When entering the file name in the program, it should **not** include the `.txt` extension (the program automatically appends this). Assume the level files are all inside a folder called **levels**, located in the **same directory** relative to the executable program.

4. Once the selected level has been loaded, the game shoud start. In each turn, the player should be allowed to either (1) inspect a specific tile, (2) put a flag on a specific tile, or (3) remove a flag that was previously placed.
5. When the game ends (either by winning or losing), the program should reveal the locations of all the mines. The program should then return to the main menu.
6. At any point during the game, there should be an option to quit (i.e., forfeit the game). Doing this counts as a loss. In this case, the program should still display the locations of all the mines before returning to the main menu.

#### 2.3.3 Level Editor

When the user chooses "Create a Level", the following flow should be observed.

1. The program should first ask the user to provide a file name where the level will be saved. Assume the levels are saved as text files inside a folder called **levels**, located in the **same directory** relative to the executable file (you may assume this folder already exists).When entering the file name in the program, it should **not** include the `.txt` extension (the program automatically appends this).
2. If the name provided already exists in the directory, the program should inform the user that the level cannot be created, and then go back to the main menu.
3. If there are no file name conflicts, the program should then ask the user to input the number of rows and the number of columns of the level. The number of rows can be any integer between 5 and 10 (inclusive), while the number of columns can be any integer between 5 and 15 (inclusive). Any values outside of this range are invalid. The number of rows can be different from the number of columns (i.e., non-square grids).
4. The program should then provide an interface to the user to place mines on specific locations on the grid. There should also be an option to delete a mine that has already been placed. Once the user is satisfied with the location of the mines, the program should provide an option to finish the level and save it.
5. The program should only allow the level to be saved if it is valid. There are only two kinds of invalid levels: (1) a level where every cell contains a mine, or (2) a level without any mines. If the player attempts to save an invalid level, the program should display an error message and go back to the editor interface to allow the user to fix the problems.
6. If the level is valid, the program should proceed saving it to the file using the file name that was previously specified. After this, the program should return to the main menu.

#### 2.3.4 Level File Format

Level sets should be saved as text files (with a `.txt` extension) in the following format. Note that your program will be tested with custom level files from your instructor, so **it is important that your program complies with this format**.

1. The first line of the file contains two integers $r$ and $c$ separated by a single space. $r$ will be between 5 and 10 inclusive, while $c$ will be between 5 and 15 inclusive. These represent the number of rows and the number of columns respectively.
2. This is followed by $r$ lines with $c$ characters each, showing the locations of the mines. Cells with no mines are represented with a dot (`.`), while cells with mines are represented with an `X`.

Here are three examples of level files in the specified format.

```
6 8
..X.....
.......X
...XX...
........
X....X..
..X..X..
```

```
7 5
.....
.XXX.
.X.X.
.X.X.
.X.X.
.XXX.
.....
```

```
7 8
..X...X.
.XX.....
X....X..
........
..X...X.
..X...X.
.XX....X
```

#### 2.3.5 View Statistics

The statistics screen should show **at least** the following information for the **currently logged in** profile:

1. Name of the profile
2. Number of Games Won (for each game mode)
3. Number of Games Lost (for each game mode)
4. 3 Most Recent Games

For the **3 Most Recent Games**, the following information must be viewable for each game:

1. The type of the game (Classic or Custom)
2. The outcome of the game (won, lost, or quit)
3. The final snapshot of the state of the board.
    - If the player won, should show all the mine locations and all tiles should be revealed.
    - If the player lost, all the mine locations should be shown, and the mine that exploded should be highlighted. Additionally, the non-mine tiles successfully revealed by the player and the flags prior to the explosion should be shown.
    - If the player quit, do not show the locations of the mines. Show only the flags and the tiles revealed by the player so far.

Note that profile statistics for all profiles should not reset even when the program is terminated.

### 2.4 Interface

This project is text-based. The program is expected to be run from the terminal / command-line. Students are given the freedom to decide the type of user interface they want to implement for the program, as long as it is intuitive to use and complies with the specifications stated in this document. Students are encouraged to express their creatity in presenting the game, while observing compliance with the specifications.

Although MINESWEEPER games are usually presented as a board that gets updated as the player reveals more tiles, for this project, you can choose not to update the board but instead append the updated state of the board on the succeeding lines. For example, a viable implementation would be something like this:

```
Current board:
  0 1 2 3 4 5 6 7
0 . . . . . . . .
1 . . . . . . . .
2 . . . . . . . .
3 . . . . . . . .
4 . . . . . . . .
Inspect [I] or Flag [F]: I
Enter row to inspect: 2
Enter column to inspect: 0

Current board:
  0 1 2 3 4 5 6 7
0 . . . . . . . .
1 . . . . . . . .
2 1 . . . . . . .
3 . . . . . . . .
4 . . . . . . . .
Inspect [I] or Flag [F]:
```

Here, we can see that the new state of the board is simply appended below the previous state. This kind of interface, though not the most ideal, is enough to get full credit for this project.

Throughout the entire program, if the user enters an invalid input, the program should not crash. Whenever an invalid input is entered, the program should display an error messaage and ask the user to try again. However, it may be assumed that the input of the user always conforms to the expected data type (i.e., the user will never enter a string if he is expected to enter an integer).

For students who want to implement a persistent board that gets updated as the game is played, we provide a custom user interface library that can help implement these features. The library also allows you to add simple colors to the program interface. This library is discussed in more detail in section 3. Use of this library is optional, but can be useful for creative and adventurous students who want to deliver more visually appealing projects.

# 3 Custom User Interface Library (optional)

In the project zip file, we provide the following files for those who want to implement more advanced user interface features.

- `interface.c`: a C file containing some helper functions for advanced user interface operations
- `interfacesample.c`: a source file containing an example usage of the different features of the `interface.c` library, which can also be used to test if the features are working properly on your machine

The `interface.c` file contains the following useful functions that can help you design the interface of your game:

| **Function Header** | **Description** |
|---|---|
| `void iMoveCursor(int x, int y)` | Moves the cursor to the given location on the command line. `x` and `y` are the coordinates of the desired location. You can use this before calling `printf` or `scanf` to change where you display or scan characters on the screen. Note that any existing text that is already displayed on that section of the screen may be overwritten. |
| `void iSetColor(int color)` | This function sets the color of succeeding output statements like `printf`. The color parameter is an integer representing the desired color (see color list below). All succeeding display will be of the specified color until this function is called again with a different parameter. |
| `void iClear(int x, int y, int w, int h)` | Clears a rectangular portion of the screen, starting from the position at (`x`, `y`) and having a width and height of `w` and `h` respectively. For example, the call `iClear(0,0,5,5)` deletes all text in a $5 \times 5$ rectangle from the top-left corner of the terminal. Note that this function also moves the cursor to (`x`, `y`) after the clearing operation. |
| `void iHideCursor` | Hides the text cursor (visually). |
| `void iShowCursor` | Shows the text cursor (visually). |

For the `iSetColor` function, the parameter is one of these color codes. You may pass either the variable names (which hold integer constants), or the actual integer value corresponding to the color that you want. Passing any other value aside from these may cause unpredictable results. For this project, you are limited to these colors only.

| **Color** | **Variable** | **Integer Code** |
|---|---|---|
| White / reset | `I_COLOR_WHITE` | 0 |
| Red | `I_COLOR_RED` | 1 |
| Blue | `I_COLOR_BLUE` | 2 |
| Green | `I_COLOR_GREEN` | 3 |
| Yellow | `I_COLOR_YELLOW` | 4 |
| Cyan | `I_COLOR_CYAN` | 5 |
| Purple | `I_COLOR_PURPLE` | 6 |

To use the functions in `interface.c` in your own program, you must first reference using `#include "interface.c"`. Please refer to the `interfacesample.c` source file to see an example of how to use the custom library. This source file may also be compiled and run to check if the library is working properly. Try to compile and run this program, and you should see a short working program demonstrating the various interface functionalities.

# 4 Generating Random Numbers

Part of this project requires the generation of a random locations for mines. To achieve this, you will need to generate random integers in a given range. The following code snippet shows how to do this:

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main()
{
    int random;

    srand(time(0)); // Initialize the seed for the random number generator

    random = rand() % 10; // Generate a random number from 0 to 9
    printf("%d\n", random);

    return 0;
}
```

First, you need to import `stdlib.h` and `time.h`. Then, at the beginning of the main function, you need to call `srand(time(0));`. This initializes the seed for the random number generator. You only need to call this once, at the beginning of your main function. From then onwards, you can use `rand() % n` whenever you want to generate a random integer from 0 to $n - 1$.

# 5 Documentation

You are required to add the following documentation (i.e., comments) in your source code.

At the beginning of your program, add a comment containing the project summary, replacing the text between `<` and `>` with the appropriate information. At the end of the program, add the academic honesty declaration as a comment.

The structure of your program should look like this:

```
/*
    Description: <describe what this program does briefly>
    Author: <student's full name>
    Section: <section>
    Last Modified: <date when last revision was made>
    Acknowledgments: <list of references used in the making of this project>
*/
<Preprocessor Directives>

<Function Implementations>

int main()
{
    <Program Codes>

    return 0;
}

/*
    This is to certify that this project is my own work, based on my personal
    efforts in studying and applying the concepts learned. I have constructed
    the functions and their respective algorithms and corresponding code by
    myself. The program was run, tested, and debugged by my own efforts.
    I further certify that I have not copied in part or whole or otherwise
    plagiarized the work of other students and/or persons.

    <your full name>, DLSU ID# <number>

*/
```

For each function that you define, add the following comment, replacing the text between < and > with the appropriate information:

```
/*  <Description of what the function does>
    @param <name> - <description of parameter 1>
    @param <name> - <description of parameter 2>
    @param <name> - <description of parameter 3>
    ...
    @return <description of return value>
    Pre-condition: <preconditions> */
<Function Header>
```

For example:

```
/*  This function capitalizes all the small letters in a string.
    @param str: the string to capitalize
    @return the number of strings capitalized
    Pre-condition: str only contains alphabetic characters */
double capitalize(char str[])
{
    int ctr = 0;
    int count = 0;
    while (str[ctr] != '\0')
    {
        if (str[ctr] >= 'a' && str[ctr] <= 'A')
        {
            str[ctr] += 32;
            count++;
        }
        ctr++;
    }
    return count;
}
```

## 5.1 Test Script

As part of the requirements of the project, students are required to submit a test script document. The test script document shows the tests done to ensure the correctness of the program. The test script should be presented in a table and should follow the format below:

| **Function Name** | **#** | **Test Description** | **Test Input** | **Expected Result** | **Actual Result** | **P/F** |
|---|---|---|---|---|---|---|
| `capitalize` | 1 | `str` contains only capital letters. | `str="HELLO"` | `str="HELLO"`, return: 0 | `str="HELLO"`, return: 0 | P |
| `capitalize` | 2 | `str` contains only small letters. | `str="hello"` | `str="HELLO"`, return: 5 | `str="HELLO"`, return: 5 | P |
| ... | ... | ... | ... | ... | ... | ... |

There should be **at least three test classes** (as indicated in the Test Description) per function. There is no need to test functions that are intended for interface display / design.

Test descriptions are supposed to be unique and should indicate classes/groups of test cases on what is being tested.

Given the sample code above, the following are four distinct classes of tests:

1. testing with `str` containing all capital letters
2. testing with `str` containing all small letters
3. testing with `str` containing a mix of capital and small letters
4. testing with `str` is an empty string (edge case)

The following test descriptions are **incorrectly formed**:

- testing with `str = "HeLLo"` (too specific)
- testing if the function can correctly count the number of capitalized letters (too general)
- testing if `str` contains special characters (not necessary - already defined in precondition)

### 5.2 Implementation Requirements

The project must abide by the following implementation guidelines:

- The program should be fully implemented in the C programming language (C99, **NOT** C++). It should compile properly with the command: `gcc -Wall -std=C99 <yourMP.c> -o <yourExe.exe>`.
- Do not use brute force. Observe appropriate use of conditional statements, loops, and functions.
- Only standard C libraries are allowed. The use of any external library apart from the custom helper library described in section 3, is **NOT** allowed. Modifying the custom helper library is also **NOT** allowed.
- Codes must be modular (i.e., split into several logical functions). Codes that are not modularized properly will not be accepted, even if the program works properly.
- `break` (outside switch statements), `continue`, `label` and `goto` statements are **NOT** allowed in CCPROG2. You can use them later when you have much more experience and wisdom in programming. Calling the `main()` function is not allowed.
- Proper documentation is required, and must follow the format specified in section 5. Also include internal documentation (comments) in the program.
- Students must test their program, and a corresponding test script should be submitted as part of the requirements of this project. The format of the test script document is described in section 5.1.

## 6 Bonus Points

Students can earn up to 10 bonus points (exceeding the maximum score) by implementing additional features that will enrich the quality of the project. The amount of bonus points is up to the discretion of the instructor. Bonus points will only be given for meaningful and non-trivial additions to the base project. Ideas for bonus features include, but are not limited to the following:

1. Displaying a timer and keeping track the score (i.e., time taken to solve the board)
2. Maintaining a leaderboard of the best times
3. Intuitive user interfaces (e.g., instead of asking the user to input the row and column number to inspect, allow them to choose visually using the arrow keys)

Bonus points will only be given to projects that have met **all** the minimum requirements. Please also make sure that additional features **do not conflict** with the minimum requirements stated in the specifications; otherwise, these may result in some of those requirements not being met. When in doubt, please consult the instructor.

## 7 Deliverables / Checklist

These are the list of deliverables for this machine project:

| Delivarable | Description | Filename | File Type |
|---|---|---|---|
| Source file | The source code for the program. There should only be one source file. | `<surname>_<section>_minesweeper.c` | `.c` file |
| Test Script | The test script document. | `<surname>_<section>_testscript.pdf` | `.pdf` file |

The deliverables above are to be submitted through the course page on or before April 1, 2024, 12:00 noon.

In addition, all students are required to demo their projects to the instructor. During the demo, the instructor may ask questions about the implementation of the project, or ask the students to make changes to the code on the spot. The schedule for the demo will be announced by the instructor at a later date.

**Please note that more than just delivering a working program, the most important outcome of this project is for the students to learn and have a good understanding of how their program works. Therefore, failure to answer the questions in a convincing manner during the demo will be interpreted as a failure of this goal, and will result in a grade of 0.**

# 8 Academic Honesty

Honesty policy applies. Please take note that you are **NOT** allowed to borrow and/or copy-and-paste in full or in part any existing related program code from the internet or other sources (such as printed materials like books, or source codes by other people that are not online). You should develop your own codes from scratch by yourself. **Violating this policy is a serious offense in De La Salle University and will result in a grade of 0.0 for the whole course**.