

Final Project: Minesweeper

«Your time is limited, so don't waste it living someone else's life. Don't be trapped by dogma - which is living with the results of other people's thinking. Don't let the noise of others' opinions drown out your own inner voice. And most important, have the courage to follow your heart and intuition.» — Steve Jobs

😬 “Yes! Let us implement the best Minesweeper game ever!” 😎

1 The rules of the game

Since the game appeared in the 1960s, it has many variations and offshoots. Let us start having a look to the basic game and, then, we can have a look to which alternatives exist, even though it is almost impossible to make a complete list. And, with a bit of imagination, you can always invent your own version of the game!

The basic game

The Minesweeper is a single-player puzzle video game. At the beginning of the game, a 2D grid of identically looking tiles (or squares) is presented to the player. Some of them, unknown to the player, hide a mine (*armed tile*), some others not (*safe tile*). The number of armed tiles is known to the player and it determines the difficulty of the game. The goal of the game is to uncover all the tiles which do not contain a mine. If a tile hiding a mine is revealed, the player loses the game. If all the tiles not containing a mine are revealed, the player wins the game. Safe tiles contain a number signalling how many of the 8 adjacent tiles are armed¹. Since revealing tiles next to a 0-tile cannot make the player lose the game, clicking on a 0-tile will make all the adjacent tile to be revealed as well. If in this process another 0-tile is hit, the process repeats over and over again. To avoid mistakes, the player has the possibility to mark tiles as armed, which implies they cannot be revealed (unless the label is removed). When the last safe tile is revealed, the game automatically marks all the remaining tiles as armed, while all the armed tiles are revealed when one of them is hit (usually indicating in some way the one hit by the player). Video games often give players the possibility to compete and so does the Minesweeper. The parameter to determine who did the better job in two won games is time. This means that a timer starts when the first tile is revealed and stops when the game is ended. Typically it is shown and advances in real-time, just to add some excitement/stress to the game.

Variations and offshoots

The game itself opens to many possibilities and, indeed, many variations of it have been already implemented. Here below a list of different versions of the game.

- (i) A first alternative is to play not on a rectangular field. Any shape can be used, but tiles are always square.
- (ii) A similar idea is to change the shape of the tile. Triangles or hexagons are the most common alternative shapes. This reduces the maximum number of adjacent tiles.
- (iii) A more challenging option is to allow more mines to be under the same tile. Even if in principle not needed, usually a maximum number of mines allowed to stay together is fixed. Moreover, the player is commonly told about how many tiles contain one mine, how many contain two, and so on.
- (iv) Another rule that can be changed is the number of dimensions of the grid. It exists a 3D version of Minesweeper, which implies the possibility to have up to 26 adjacent mines per tile. Of course, the idea to have to safely walk on a mined field is a bit lost, but you could have to swim underwater in a dangerous sea with invisible poison localised here and there. . .







¹Tiles touching in only *one* point are *also* considered adjacent!

- (v) Staying closer to the basic game, it is possible to impose periodic boundary conditions. This implies that all tiles have the same number of adjacent tiles and it forces the player to pay more attention close to the borders.
- (vi) If you think about (or if you played the game), you will probably realise that it can happen to have to risk. In principle the first choice always contains a non-zero probability to loose the game. A first variation of the original game is to rule out this possibility.
- (vii) A much more trickier alternative is to rule out luck from the game. In this case, at no stage in the game, the player could loose because of bad luck. Said differently, the game will ensure that the entire grid can be fully deduced starting from the initial open space.

Clearly, some of the variants can be combined to make the game more interesting, or new ones could be introduced². Another aspect, which is not really a variant of the game but rather a help for the user, consists in allowing the player to *reveal again* an already revealed tile. This will make the game uncover all the adjacent unrevealed and not-marked-as-armed tiles, but, potentially, it can as well make the player loose. A safer variation of this idea is to reveal all the adjacent tiles *only if* as many adjacent tiles as the number of the chosen tile are marked as armed. It could be done iteratively by the game also on the tiles that are revealed after a tile has been chosen (as a cascade effect).

2 Implementing the basic game

Playing a game can be challenging, but often implementing a game is even more challenging. The first part of this project is to implement a *basic* version of the basic game, which is played within the terminal. The idea is to get the grid displayed and to enter the coordinate of the tile we want to reveal. If we neither win nor loose, we get the grid displayed again and we are asked again for new coordinates. To play the game in this way, some notation is needed and it is summarised in the following table.

Displayed tile		Description
PROJECT START	FINAL	
@		An unrevealed tile.
A		An unrevealed tile marked as armed.
*		A revealed mine.
L		The tile which made the player loose.
W		Revealed mine when the game was won.
0		A revealed tile with no adjacent mine.
1 2 3 4 5 6 7 8	1 2 3 4 5 6 7 8	A revealed tile with as many adjacent mine(s) as the number indicates.

As the left two columns explicitly show, in the first phase of the development, we will not care about colours. Also the possibility to mark a tile as armed (A) or to highlight mines at the end of the game can be added later. This means that at the beginning a tile can either contain a digit, a * or a @.

Designing the code

The design phase is crucial to have a good code, which is then easy to extend. This is the main reason why you are guided quite in detail. Everything here is only a suggestion that you are highly encouraged to follow, but you may do it differently *if you have a reason to do so*. A possible approach to the problem of implementing the Minesweeper game is having two grid variables, one **board** and one **mask**. The former contains the grid as it has to appear if all the tiles were revealed, while the latter keeps track of what is shown to the player. Generally, only the **mask** is printed to the screen, while the **board** is used internally to reveal tiles and implement the logic of the game. Some blocks of functionality have to somehow exist in the code. Let us briefly discuss what they should do.

²There are, e.g., also multi-player versions of the game, but this goes beyond the scope of the project.

- At the beginning of the game, the dimensions of the grid as well as the difficulty (i.e. the number of mines) should be set. You can either propose levels of difficulty to the player and ask him to select one, or you could ask the player to choose both the size of the grid and the number of mines. In the second case, some checks are mandatory. Even if you could be tempted to do it with `printf` and `scanf`, you should provide command line options to allow the player setting the parameters when the game is launched. For example, to play on a 10 times 12 grid with 24 mines, it should be enough to execute the game as

```
./minesweeper -r 10 -c 12 -m 24
```

where the order of the options should not matter. Have a look to the `getopt` function³ contained in the `unistd.h` library and to [some examples](#)⁴. Which other options could be useful?

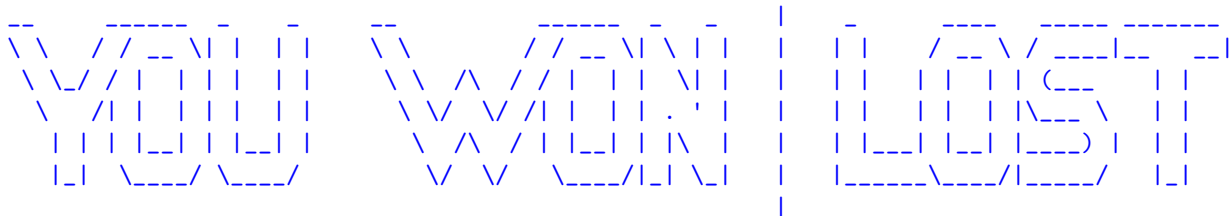
- Some tools to print the grid to the standard output are needed. You must consider how to structure your output, but it should probably look like the following one (at the beginning of the game)

```

      0  1  2  3  4  5  6  7  8  9
=====
0 |  |  |  |  |  |  |  |  |  |  | 0
  |  |  |  |  |  |  |  |  |  |  |
=====
1 |  |  |  |  |  |  |  |  |  |  | 1
  |  |  |  |  |  |  |  |  |  |  |
=====
2 |  |  |  |  |  |  |  |  |  |  | 2
  |  |  |  |  |  |  |  |  |  |  |
=====
3 |  |  |  |  |  |  |  |  |  |  | 3
  |  |  |  |  |  |  |  |  |  |  |
=====
4 |  |  |  |  |  |  |  |  |  |  | 4
  |  |  |  |  |  |  |  |  |  |  |
=====
      0  1  2  3  4  5  6  7  8  9

```

or the same with `@` instead of `■`. You could also set up some fancy way of saying to the user that the game was won or lost. What about [this](#)?



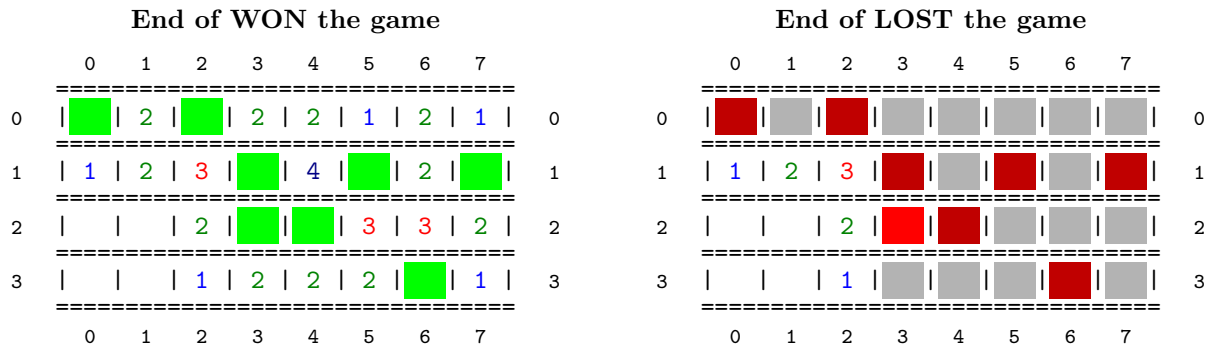
- You should set up a timer to tell the player about her/his performance. It is enough to update it together with the update of the grid. Where and how is it better to print the timer? It would be nice for the player to keep under control the total number of mines and how many tiles were set as armed.
- One important part of your code will deal with the `mask` and `board` initialisation. After memory is correctly allocated⁵, you have to fill them properly before letting the game start. The `mask` setup is straightforward, just fill it with `■` (ot `@` at the beginning). For the `board`, some more work is required. There you have to distribute the mines randomly and you need to put in each square which does not contain a mine the number of adjacent mines. Remember that, fixed the size of the field and the number of mines, not all the games should be the same!
- The probably most challenging part is the implementation of the logic needed to reveal tiles. Here, you will need some functionality to modify the `mask` according to the user input and the content of the `board` as well as some handling of the state of the game. In particular, you need to consider that, after having revealed a tile, the game could be ended because the player either won or lost. The cascade mechanisms to reveal tiles next to any without adjacent mines or next to an already revealed tile also belong to this block of functionality.

³This is only a possibility and you are free to implement the command line option parser yourself.

⁴In the whole document, all the text coloured in Plum hides clickable links to useful web pages.

⁵And do not forget to free it when the program terminates!

- At each step of the game, you need to parse the user input, i.e. the coordinates to reveal a tile and, later in the project, an action to be performed. Revealing a tile, is the only possibility for the player in the first implemented version of the game, but then she/he should be able to mark a tile as armed and to disarm it. A nice way to handle these different actions is to ask the user to insert an action (`r`, `a`, `d`, `h`, `q` respectively to *reveal*, *arm* or *disarm* a tile, to get *help* or to *quit*), optionally followed by coordinates (e.g. `r 0 0` to reveal the tile in the upper left corner).
- Remaining tools you will probably need are some mechanism to check if the game is finished and some aesthetics. For example, you should think about how to allow the user to mark a tile as armed and also implement the colours as showed before. Do not forget that when the game is won or lost you need to modify once more the `mask` and display it to the user, giving the information about the outcome of the game, too.



- Not necessary to say it, some functionality to debug and test your code are also mandatory!

Thinking about functions signature

So far we spoke in quite general terms about which functionality needs to be implemented, but you could still wonder about how to organise your code in a reasonable way. To implement the basic game, you will not need some particularly involved structure and a set of functions together with a new data type to encapsulate the grids will probably be enough. Here in the following you can find some suggestions. However, consider that this is neither a complete list, nor a unique way to proceed to get a solution. If you have a reason to do something different, feel free to do so, but be ready to explain why you decided in that way.

```
typedef struct Minesweeper {
    unsigned short int nRows;
    unsigned short int nCols;
    unsigned short int nMines;
    /*grid_type*/ **mask;
    /*grid_type*/ **board;
} Minesweeper;
```

There must exist one and only one `Minesweeper` variable in your program. Ideally, global variables *should be avoided* and your `Minesweeper` variable should be passed around to functions. Since the grid can be potentially kind of large, it is better to copy around to function a pointer to a `Minesweeper` variable. If you feel particularly unconformable in doing so, you can have a `Minesweeper global` variable, but be ready to explain how would have you done to pass it around.

```
//Printing tools
void PrintHeader(/*...*/);
void PrintLine(/*...*/);
void SetStringToBePrint(/*...*/); // (*)
void PrintGrid(/*...*/);
void YouLost(void);
void YouWon(void);
//Setup Minesweeper
Minesweeper ResetMinesweeper(void);
```

```

void AllocateMemory(/*...*/);
void FreeMemory(/*...*/);
unsigned short int DrawRandomNumberBetweenZeroAnd(unsigned short int);
bool IsSiteInsideTheGrid(/*...*/);
void PutMinesOnBoard(/*...*/);
void FillBoardWhithMineAdjacentNumbers(/*...*/);
void InitBoard(/*...*/);
void InitMask(/*...*/);
void InitMinesweeper(/*...*/);
//Revealing tools
void RevealTilesFromEmptyOneOpeningFullSpace(/*...*/); // (*)
bool RevealAdjacentTilesOfRevealedTile(/*...*/); // (*)
bool RevealTile(/*...*/);
void RevealMinesInMask(/*...*/);
//Additional player actions
bool ArmTile(/*...*/); // (*)
bool DisarmTile(/*...*/); // (*)
void PrintInGameHelp(/*...*/); // (*)
void QuitGame(/*...*/); // (*)
bool MakeAction(/*...*/); // (*)
//Handling user input
void GetCoordinatesAndAction(/*...*/);
bool CheckActionAndCoordinates(/*...*/);
unsigned short int CountTypeOfTileInMask(/*...*/);
//Dealing with command line options
void PrintHelper(void);
void ParseCommandLineOptions(/*...*/);
void CheckParsedCommandLineOptions(/*...*/);

```

In the list of functions above, it is up to you to decide which arguments should be passed. In some cases, there is a natural proposal, but in general you have to take decisions. Functions marked with **(*)** can be implemented in a second phase, when a basic version of the game is working. Of course, this will imply some other functions as well as the `main` to be adjusted.

Observe that the functions `PrintInGameHelp` and `PrintHelper` have different responsibilities. The first should give suggestions about which commands to give in order to play *during the game*, while the second will be called if the user gives a particular command line option starting the program.

General remarks

Before having a look in detail to what you are required to do, let us put together some remarks you will probably need at some point in you project. Even though some of them could sound cryptic at a first reading, look at them carefully now and spend some time trying to answer the corresponding questions, so that you will remember about while implementing your code. At that time, everything will be clear.

- Since the flow of the game is tightly connected to the standard output of the program, you should probably use the standard error or a log file to deal with debug information. How do you print to the standard error? How do you redirect the standard error to a file? If you are not familiar with the redirection operators of your shell, you can read about them on the web⁶. Printing the debug information to the standard error is not so bad, since you can anytime suppress it redirecting it to `/dev/null`. There are also **debug approaches** based on pre-processor directives. Feel free to use what makes you more conformable, but you *must* avoid temporary `printf` debug-statements.
- Which data type should be used for `mask` and `board`? You should display numbers as well as different symbols. Do not forget what you learnt about casting variables in C. Is there a smart way to print or store numbers as literal characters?

⁶Beyond the shell manuals, there are good explanations on the web. A starting point could be [this](#) and the link in it for a more detailed analysis.

- Once you have a *working* function to reveal only the tile at the coordinate given by the user, you can implement the more advanced feature to reveal all the surrounding ones if such a tile has no adjacent mines. This is a well known algorithm which is called **flood fill**, which is naturally recursive. The basic idea is to make some checks to stop the recursion, act on one tile and then let the recursion start.

```
void RevealTilesFromEmptyOneOpeningFullSpace(/*...*/) {
    /* 1) Stop the recursion if outside the board, if the board contains a
     *    mine or if the mask was opened.
     * 2) Reveal tile
     * 3) If the tile has no adjacent mine, start recursion
     */
}
```

- When the code to reveal a tile is ready (including the flood fill algorithm), you can add the code to dig under an already revealed tile. This means that the player can ask to reveal again a tile and in this case the code should reveal all the adjacent tiles which have not been set as armed. Be careful to check that there are enough armed tiles before revealing them!
- Colours in the terminal are not universal and you could run in portability issues. Being aware of this aspect, we will ignore it in this project. Read about **formatting in the terminal** and try to figure out how to achieve it in your C program. It could be useful to know that `\e[` in `bash` is mapped to `\033[` in C. If you really cannot use colours in your game (but what a pity!), use a different notation for each type of tile. Actually, the responsibility of printing colours to the screen should be of the printing functions and the variables `mask` or `board` should not know anything about colours! A tile set as armed, for example, should be set to `A` in the `mask` and then the function which prints to the screen the grid should convert `A` to `■`. In any case, your notation must be explained in the documentation of the game.
- It would be nice to clean the terminal at each player choice of coordinates to avoid the board to vertically move. Have a look to the `int system (const char* command)` function of the `stdlib.h` library.
- Write a readable code. You do not have to write a professional and releasable code, but this is not a good excuse to write something that nobody (even yourself in few weeks) will be able to understand. There are even books about clean code⁷. To get a reasonable result, it is enough to give reasonable names to variables and functions. Good names often do not require explanations as the functions given above demonstrate. Just think before giving a name, it usually suffices. Too short names are highly discouraged, with few exceptions. For example using `m` and `b` instead of `mask` and `board` is not wise and renaming them in a second moment could be quite painful.
- Test your code. Test it again. And again. It could sound crazy, annoying, pedantic, whatever, but it will come the day you will regret not having tested your code a bit more you did. Guaranteed. About **software testing** there are not only books, but even code development models based on it. We will not go so far, but we definitely need to test our game. Would you be happy to play a not working game? Or, even worse, to loose a game because of a bug? A basic approach could be to test each function you developed alone. Given a function type `FunctionExample(/*...*/)` you could write a `bool FunctionExample_test(/*...*/)` which gets the same parameters and returns `true` if the test passed and `false` otherwise. You could then have some code like

```
void RunTests(/*...*/) {
    unsigned short int testsPassed=0, testsFailed=0;
    if(FunctionExampleOne_test(/*...*/)) testsPassed++; else testsFailed++;
    if(FunctionExampleTwo_test(/*...*/)) testsPassed++; else testsFailed++;
    //...
    printf("\n %u tests run, %u passed, %u failed.\n\n",
           testsPassed+testsFailed, testsPassed, testsFailed);
}
```

⁷Robert C. Martin – *Clean Code: A Handbook of Agile Software Craftsmanship*.

to run your tests (maybe in a different executable). You could even set up a `struct` per function to collect its arguments and, maybe, a `struct` containing several `structs` to be passed to `RunTests`⁸.




- Organise your code in different source code files. If you are wondering about why you should do something like that, *facilitate code reuse* and *share code between projects* are the immediate answers⁹. Debugging and reading the code gets also quicker when you have to move in files with few dozens of lines instead of few hundreds. Clearly, the code splitting has to be done with a pinch of salt and putting one function per file would not be a wise way of doing it. In our project the code can be naturally grouped according to its functionality and, hence, this could suggest how to organise it in files. But also a minimal division like putting all the functions together in a file would be fine¹⁰. Having the code implemented in at least one separate file allows testing it in a completely separate program, in which the own code is included as `#include "myCode.h"`. You should have learnt how to compile code which is separated in several files, but if you still have some doubts you can read it again [around in the web](#).

3 Your tasks

And now, it is time to code! Well, not so fast. Your first task is to have clear in mind what you have to do in the project and, hence, read this section very carefully and come back to the previous ones whenever you need it. Basically, your project is divided in two parts, which are going to be described here below.

Part I: Implement the basic game

You should start implementing the basic game, whose rules have to match those described in §1. In order to do so, follow §2 and provide your code with all the functionality needed to handle the game. If something is not clear to you, ask before doing (but read the previous sections before asking). In the beginning, focus on the basic feature of the game (e.g. print grid without colours to screen, reveal one tile at once). You will improve your game later. There are some features, which you are required to respect. Just to stress them again, you can find in the following list the aspects of the game to be implemented. Again, for more details, refer to the previous sections (for example, you are *highly* encouraged to use the suggested structure of functions and data type).

- Your executable must accept a set of command line options (e.g. `-r`, `-c`, `-m`). A mandatory one is `-h`, which does not run the game, but explains how to play to the user and gives information about the existing command lines options. It should be possible to run the program without options as well. In this case, a default game should start.
- Your program has to be able to print the grid in a sufficient nice way to the output. At the beginning, use the notation in the first column of the table in §2 and only at the end turn it into the coloured version.
- You need a timer in your game. Use it to tell the player at each step how much time elapsed from the game start. Give to her/him also the information about how many tiles were armed and how many mines are in the game.
- The possible actions the player can do at each step are reveal, arm or disarm a tile as well as get help or quit.
- Revealing a tile without adjacent mines must make all the adjacent tiles be revealed as well (as cascade effect).
- A revealed tile containing a number n could be *digged*, i.e. revealed again. This should make all the unarmed adjacent tiles be revealed, if there are at least n adjacent marked-as-armed tiles.
- The end of the game should be handled printing the grid a last time. If the player won, all the tiles with a mine have to be replaced by . If the player lost, all the tiles with a mine have to be showed as , emphasising the one  that made the player loose.

⁸A more elegant, but rather advanced approach, makes use of [variadic functions](#)

⁹[Here](#), you can find a nice overview about why but also about how to achieve the splitting.

¹⁰This file can be split later on if it grows too much.

- Implement a debug mode in your code. It does not matter if you do it using the standard error, a log file or a pre-processor macro. You must just avoid to add and later remove `printf` statement to understand what was wrong.
- Write readable code. It is important as such, but it is a must if you get in trouble and you want someone to help you. . .
- Split your code into several files. It is up to you how, but implementing everything in the same file is not an option.
- Each function you implement needs to be tested. Do it as you think it is better, but do it in a way that you can run your tests over and over again. Having a separate executable which runs the tests and produces a report like

```

-----
Running 3 tests...
 1/3 Function_One      OK
 2/3 Function_Two     FAILED
 3/3 Function_Three   OK
...done!
-----
2 test(s) passed, 1 test(s) failed!
-----

```

sounds like a good idea. Please, avoid to implement all the functions at first and all the tests afterwards. Having some tests for a function allows you to check if you broke something changing or adding some code¹¹. If you print some output in the tests, redirect it to a tests log file, so that the report does not get messed up. Despite the fact you could be thinking this is a boring task, it is how it works in real life. Testing your code is the only way to constantly guarantee it works.

Part II: Extend the basic game

Once you have a working game, fantasy (together with some motivation) can bring you anywhere. To consider your project finished you need to either

A: implement variation (vi) together with one other variation of the game described in §1,

or, if you feel very (but very) motivated,

B: implement a Graphic User Interface (GUI) for the game.

Alternative A

It is the one you are encouraged to approach. You are free to choose any variation of the game, but consider that some of them are really challenging. Here some subjective comments on each of them.

- (i) The logic of the game does not change, but handling the field in the memory requires some modification. Of course any shape can be put in a rectangle. . .
- (ii) Having not square tiles makes the printing part of the code much trickier. The flood fill algorithm should be adapted as well.
- (iii) This variation requires more work, since more functionality has to be added. How to signal more than one mine? How to have numbers higher than 9 under a tile? How to give to the player the information about the armed tiles? Nothing impossible, but longer than others.

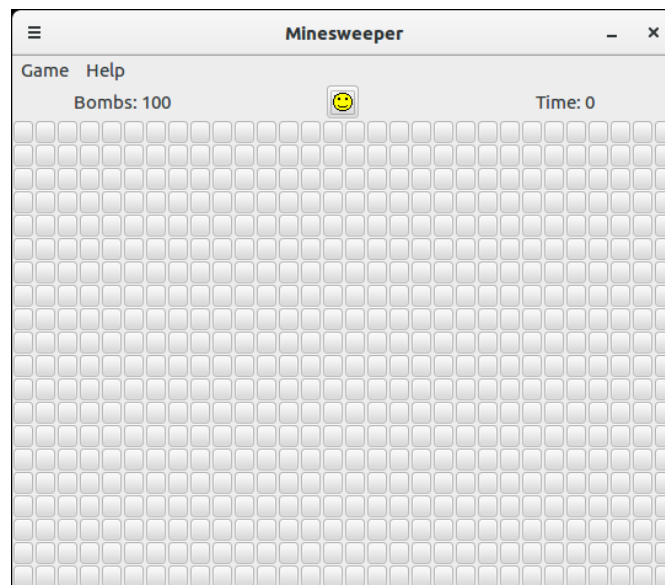
¹¹Unless you drastically change the responsibility of a function, all the previously written tests should continue to pass when you implement a new feature.

- (iv) Implementing the Minesweeper in 3D is not so hard, but not easy as well. The main points to address are about how to print the grid and how to put larger numbers under a tile. Probably, to play it in a terminal, it should be limited to few layers, printing them one next to each other and telling their order to the player.
- (v) This is probably the easiest and quickest one. Go for it if you do not feel too strong.
- (vi) *This is mandatory, you must do it!*
- (vii) Do not do it, but if you decide to rule out luck from the game, well... good luck!

Alternative B

It is definitely a task of another, much higher level. It requires lots of work and it is proposed here more for completeness and just to make you discover a new world, maybe for the future. Only if you feel very motivated and if you feel you well understood everything was explained in the lecture, you can go for it now. Still, consider to discuss the way to proceed with me, before doing anything.

The starting point is to choose a library to design our *application*¹². We will use **GTK** and, at start, we must learn how to use it. In the [official documentation webpage](#) there are nice tutorials which should be a good starting point to discover the needed topics. Copy the examples contained there in new files and try to compile and run them. Here you could encounter some issues due to the version of **GTK** you are using. Be sure that the version you need is installed on your machine and if not you need to install it. Usually on Linux operative systems **GTK** is already installed, but you could need a more recent version. On OSX, instead, you could use **brew** to [install it](#). Once you get familiar with the library, you can start experimenting with it in the Minesweeper direction. In this step you will need to read deeper in the documentation or have a look to some book on the topic. Work in a different file with respect to those of your original code. See how far you get, do not hesitate to ask and never despair. If you choose to go in this direction, a *semi-working GUI* is more than enough to consider the project concluded! For example, if you get to display the closed mask to the user with one single button to completely open it, you should be very happy and this fulfils already the requirements¹³.



¹²Well, yes, if we build a GUI for our game we can call it application.

¹³Clearly, you need to understand what it is going on in your code, so blindly copying examples from the web is not enough, even if your code does what you want.