UNIVERSIDAD
COMPLUTENSE
MADRID

Fundamentals of Programming II

Presentation of the Practice (Part I)

Index

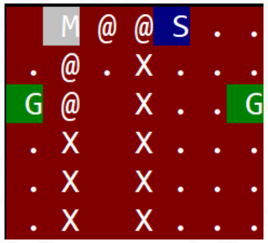
1. Introduction

- ✓ This practice aims to develop, incrementally, different variants of the **master miner** game
- ✓ The practice covers lessons 1 to 5 in **two parts**.
- ✓ Some specific **objectives** are: multidimensional arrays, modular programming, pointers and dynamic memory.
- ✓ The last submission date of part I is: **13th April** at 23:55.
- ✓ The practice will be submitted to the Virtual Campus through the task **Submission of the Practice (Part I)**, which will allow you to upload .cpp and .h files with the source code. One of the two members of the group, and only one, will be responsible for uploading it. Remember to put the name of the group members in a comment at the beginning of each source code file.

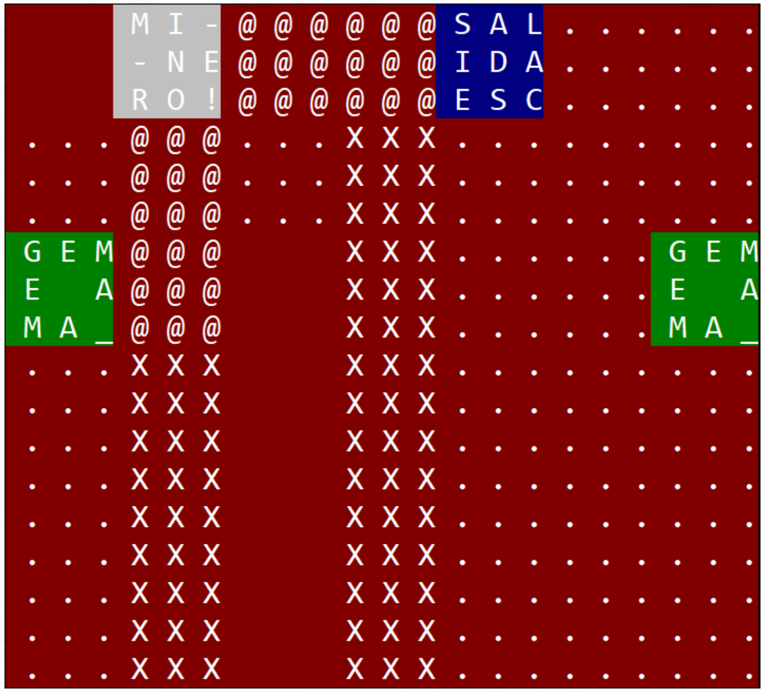
2. Game Description

- ✓ The practice is to develop a C ++ program that allows you to play a version of the **Master Miner** (https://en.wikipedia.org/wiki/Master_Miner).
- ✓ This game, created by Dan Illowsky, is about a miner whose goal is to collect gems.
- ✓ In our version, the more gems you collect, the more points you accumulate, not being necessary to collect all the gems to pass to the next level.
- ✓ The **objective** of the game is to collect the maximum number of gems and then go to the exit cell.

- The game takes place on a **plane of the mine** that represents a mine cut vertically. Below is a possible level of the game at 1:1 scale and 1:3 scale.



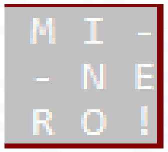
```
  M @ @ S . .
. @ . X . . .
G @   X . . G
. X   X . . .
. X   X . . .
. X   X . . .
```



```
      M I - @ @ @ @ @ @ S A L . . . . . .
      - N E @ @ @ @ @ @ I D A . . . . . .
      R O ! @ @ @ @ @ @ E S C . . . . . .
. . . @ @ @ . . . X X X . . . . . . . . .
. . . @ @ @ . . . X X X . . . . . . . . .
. . . @ @ @ . . . X X X . . . . . . . . .
G E M @ @ @       X X X . . . . . . G E M
E   A @ @ @       X X X . . . . . . E   A
M A _ @ @ @       X X X . . . . . . M A _
. . . X X X       X X X . . . . . . . . .
. . . X X X       X X X . . . . . . . . .
. . . X X X       X X X . . . . . . . . .
. . . X X X       X X X . . . . . . . . .
. . . X X X       X X X . . . . . . . . .
. . . X X X       X X X . . . . . . . . .
. . . X X X       X X X . . . . . . . . .
. . . X X X       X X X . . . . . . . . .
. . . X X X       X X X . . . . . . . . .
```

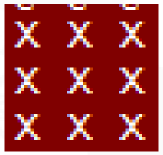
- ✓ The miner can move inside the plane to the **right** or **left** while maintaining his level with respect to the depth of the mine, or **up** or **down** by changing the level of depth inside the mine.
- ✓ If the miner moves horizontally, even if the cell below is excavated he maintains his level, he does not fall.
- ✓ The miner cannot cross walls or stones.
- ✓ When making a move, the miner moves to the next cell, according to the move. The movement can be carried out if the adjacent cell is empty (previously excavated), or with soil, or if it has a gem. In fact, if it has a gem he collects it and occupies the cell.

- ✓ If the cell he wants to go to has a stone, then he must move it to occupy that site. A stone can be moved if the cell to which it is to be moved is empty.
- ✓ The cells may contain:
  - **The miner**



```
M I -
- N E
R O !
```

  - **Wall**. The wall cannot be crossed or pushed to occupy that cell. The only way to destroy a wall is with dynamite.

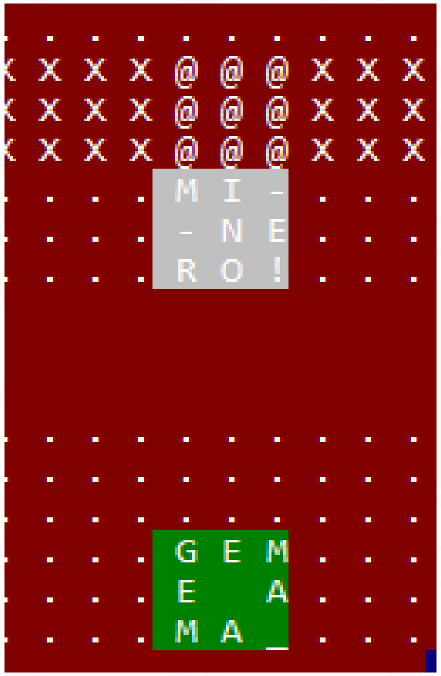


```
X X X
X X X
X X X
```

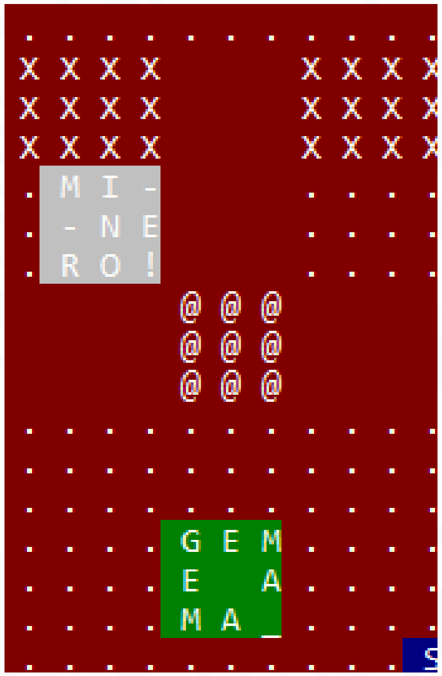
- **Stone**. The stones prevent the miner from passing. He can move them horizontally if the following cell is empty. If it is not empty (soil, wall, gem or other stone) they cannot be displaced. To move them the miner must move towards them

  Stones may fall down due to gravity when the cell underneath is empty. In order for the cell below to be empty, the miner must excavate the ground, leaving him under the stone, but it does not crush him (he is a very strong miner), when he moves from under the stone, the cell is free and the stone falls as many squares as there are free.

If the miner moves down with a stone or several stones on top of him, they fall on top of the miner, but they do not crush him. If they fall on any other element, such as soil or a wall, they remain on top of that element.

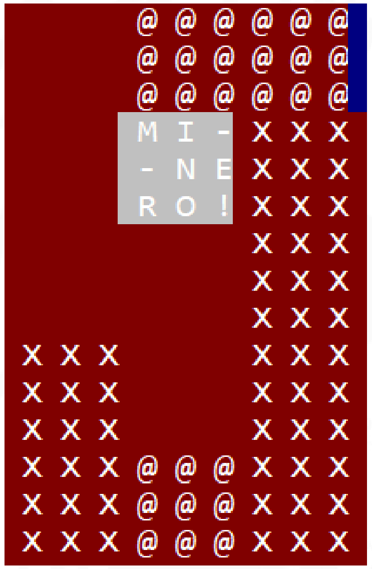


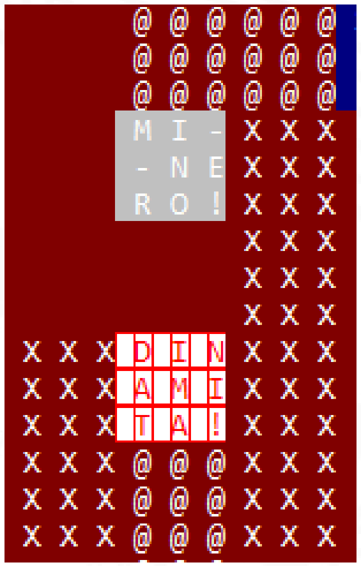
```
. . . . . . . . . .
x x x x @ @ @ x x x
x x x x @ @ @ x x x
x x x x @ @ @ x x x
. . . . M I - . . .
. . . . - N E . . .
. . . . R O ! . . .



. . . . . . . . . .
. . . . . . . . . .
. . . . . . . . . .
. . . . G E M . . .
. . . . E   A . . .
. . . . M A _ . . .
```

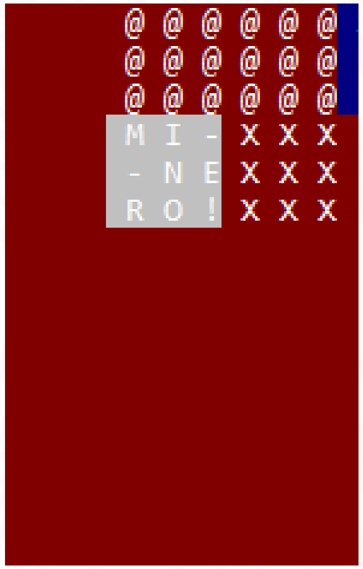


```
. . . . . . . . . . .
X X X X       X X X X
X X X X       X X X X
X X X X       X X X X
. M I -       . . . .
. - N E       . . . .
. R O !       . . . .
    @ @ @
    @ @ @
    @ @ @
. . . . . . . . . . .
. . . . . . . . . . .
. . . . . . . . . . .
. . . . G E M . . . .
. . . . E   A . . . .
. . . . M A _ . . . .
. . . . . . . . . . S
```

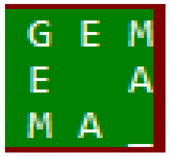
- **Dynamite.** The dynamite is carried by the miner and when he decides to release it, it falls vertically due to gravity, when it reaches the bottom it explodes. The explosion causes the 8 squares around to be empty. It is the way to remove walls and stones. If it explodes affecting a gem this is eliminated, and if it explodes affecting the miner he dies.



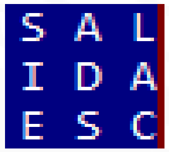




- **Gem**. When the miner arrives at a cell that has a gem he collects it and the cell is empty. Gems block the movement of stones, that is, it is not possible moving a stone if there is a gem behind.

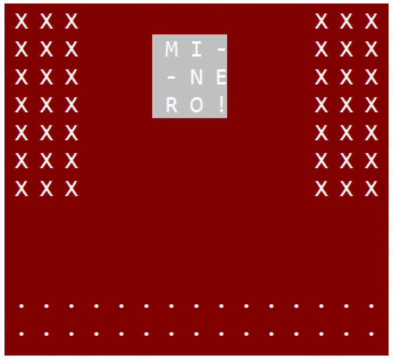


- **Exit**. There is only one exit that you have to reach to pass to the next level.

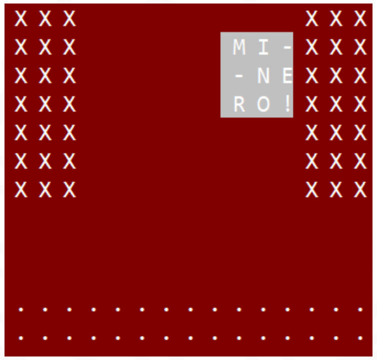


- **Free**. It is a square that belongs to an already excavated tunnel or a hole that is there from the beginning of the game.

- ✓ The miner has all the dynamites he needs.
- ✓ The miner can move in all directions, without falling into the empty space.



```
X X X                   X X X
X X X       M I -       X X X
X X X       - N E       X X X
X X X       R O !       X X X
X X X                   X X X
X X X                   X X X
X X X                   X X X



. . . . . . . . . . . . . . .
. . . . . . . . . . . . . . .
```



```
X X X                   X X X
X X X             M I - X X X
X X X             - N E X X X
X X X             R O ! X X X
X X X                   X X X
X X X                   X X X
X X X                   X X X



. . . . . . . . . . . . . . .
. . . . . . . . . . . . . . .
```

- ✓ The game ends successfully when the miner has reached the exit cell. The game ends with failure if the explosion of a dynamite reaches the miner or because he has been locked between stones or walls, he cannot reach the exit and it is necessary pressing the “Esc” key to finish.

3. The Program

- ✓ The program will simulate the dynamics of the game in **console mode**. Initially, it will show a **menu** with three options:

```
1. Play 1:1 scale game.
2. Play 1:3 scale game.
0. Exit.
```

- ✓ Option 1 shows the game in 1:1 scale, option 2 in 1:3 scale.
- ✓ The game starts by loading the first level from the "*1.txt*" file, if you manage to reach the exit of this level, you will have access to the next level defined in the "*2.txt*" file, and so on until you reach the last level. When this level is passed, a "Game Over" message will appear. The last level is defined as a program constant.

- The program will give options for entering the **movements from the keyboard** or using a **file**. In this last case, the user will be asked about the name of this file. To do this, a file containing one letter per movement will be given. The letter ‘D’ is dynamite, ‘R’ is right, ‘L’ left, ‘U’ up and ‘W’ down. This option is useful for testing.
- Games may be played until option 0 is chosen.

3.1. Program Data

- ✓ Define an enum type *tCell* that allows to represent at least the different elements that may be in the mine: *FREE*, *SOIL*, *GEM*, *STONE*, *WALL*, *EXIT*, *MINER*, *DYNAMITE*.
- ✓ To maintain the state of the mine, use a two-dimensional array of type *tCell*. Declare the corresponding constant *MAX* = 50 and type *tPlane* to represent the two-dimensional array.
- ✓ Define the struct type *tMine* to describe the complete state of the mine, containing at least:
  - The two-dimensional array of type *tPlane*.
  - The number of rows *nrows* and columns *ncolumns* in the two-dimensional array (both <= *MAX*)
  - The row and column where the miner is located.

- ✓ Define also the struct type *tGame* that describes the state of the game containing at least:
  - The state of the mine (plane and position of the miner) of type *tMine*.
  - The number of gems collected since the game started.
  - The number of movements.
  - The number of dynamites that the miner has used.
- ✓ The program also uses an enum type *tKey* to represent the movements that the miner can perform: *UP*, *DOWN*, *RIGHT*, *LEFT*, *EXIT*, *NOTHING* and *TNT*.

3.2. Loading the Mine Plane

- ✓ The game planes of the different levels will be read from a text file.
- ✓ For example, the plane in the previous example corresponds to the file *1.txt*:

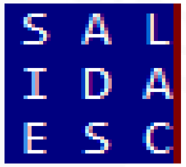
```
7 10
GW M WWWWW
GW   WWSGS
SW G WWSSS
GSSG  WSSS
SSSSSSSSSS
SGSSWGSWES
SSSSWSSWWW
```

- ✓ The first line defines the number of rows and columns of the mine.
- ✓ An array of characters follows, where ’ ’ (blank) represents a‘ *FREE* ’space, that is, a hole in the mine.
- ✓ ‘S’ represents the soil, ‘G’ a gem, ‘T’ a stone, ‘W’ a wall, ‘E’ the exit and ‘M’ to the player or miner.
- ✓ Different files with different levels will be left on the Virtual Campus.

3.3. Mine Plane Display

- ✓ The plane of the mine can be visualized on two scales: 1:1 and 1:3.
- ✓ At the beginning of a game the user chooses if he/she wants to use one scale or another.
- ✓ In the 1:1 scale each cell corresponds to a character in the console, while in the 1:3 scale each cell corresponds to a matrix of 3*3 characters in the console.
- ✓ Each time you go to visualize the state of the plane, delete the contents of the console window first, so that the plane is always shown in the same position and the sensation is more visual. To delete the console use: *system ("cls");*

- ✓ In the 1:1 projection the following characters are used to represent the data of the mine: '@' stone, 'X' wall, '.' soil, 'M' miner, 'D' dynamite, 'G' gem and ' E 'exit.
- ✓ The 3:3 scale projection enlarges the original plane by 3 units and is only used when viewing the game.
- ✓ The characters used can be the same as in the 1:1 projection repeated 9 times or some original design can be used. For example:



```
S A L
I D A
E S C
```

- ✓ To carry out the projection, the following matrixes must be created and will be used together to visualize the mine. We will use the definitions of types *tPlaneCharacters* and *tPlaneColours* as defined types of
  - *char tPlaneCharacters [3*MAX][3*MAX]:* this matrix saves the characters for each cell.
  - *int tPlaneColours [3*MAX][3*MAX]:* this matrix saves the colour of each cell.
- ✓ This projection leaves the mine a bit "narrow", if we want to expand the width we can apply *setw* as follows:

```
cout << setw(2) << tPlaneCharacters[_][_];
```

3.4. Colour the Mine Plane

- ✓ By default, the **foreground** colour, the one with which the character strokes are shown, is white, while the **background** colour is black. We can change those colours, of course, although we must do it using routines that are specific to Visual Studio, so we must be aware that the program will not be portable to other compilers.
- ✓ We have 16 different colours to choose from, with values from 0 to 15, both for the foreground and for the background. 0 is black and 15 is white. The others are blue, green, cyan, red, magenta, yellow and gray, in two versions, dark and light.

- ✓ Visual Studio includes a library, *Windows.h*, which has, among others, routines for the console. One of them is *SetConsoleTextAttribute()*, which allows you to adjust the background and foreground colours. Include this library and this routine in the program:

```
void backgroundColour(int colour) {
   HANDLE handle = GetStdHandle(STD_OUTPUT_HANDLE);
   SetConsoleTextAttribute(handle, 15 | (colour << 4));

}
```

- ✓ Simply provide a background colour (1 to 14) and that routine will establish it, with the foreground colour white (15). You must change the background colour every time you have to draw a cell and then return it to black (0).

3.5. Reading of the Miner´s Movement

- ✓ The miner moves horizontally and vertically using the **arrows**, the “D” key is used to launch the dynamite and the “Esc” key is used to exit the game at any time.
- ✓ To facilitate the tests you can save the movements in a text file whose name will be given at the beginning of the program.
- ✓ When using this option the movement keys will be: “U” up, “W” down, “L” left, “R” right and “D“ dynamite.

- ✓ To read the keys pressed by the user, implement the following subprogram:
  - *tKey readKey():* returns a value of type *tKey*, which can be one of the four addresses if the corresponding direction arrows are pressed; the *Exit* value, if the *Esc* key is pressed; o Nothing if any other key is pressed.
- ✓ The function *readKey()* will detect the user's pressing of special keys, specifically the arrow keys (directions) and the *Esc* key (exit).
- ✓ The *Esc* key does generate an ASCII code (27), but the arrow keys do not have associated ASCII codes. When they are actually pressed, two codes are generated, one that indicates that it is a special key and a second that indicates which one it is.

- ✓ The special keys cannot be read with *get()*, as this function only returns one code. We can read them with the *_getch ()* function, which returns an integer and can be called a second time to get the second code, if it is a special key. This function requires that the *conio.h* library be included.

```
cin.sync();
dir =_getch(); // dir: tipo int
if (dir ==  0xe0) {
   dir = _getch();
   // ...
}
// If here dir is 27, it is the Esc key
// If dir is 68, it is the D key
```

- ✓ If the first code is *0xe0*, it is a special key. We will know which one with the second result of *_getch ().*

```
↑   72   ↓   80  →   77  ←   75
```

4. Miner´s Movement

- ✓ Once the user indicates the direction, we have to make the player move on the plane. For this, you must implement the following function:
  - `bool makeMovement(tGame &game, tKey key)`: performs the movement of the miner in the indicated direction. This function defines this new state of the mine taking into account the movement of the miner. For example, if he tries to move to a cell that contains a wall, as he cannot cut it up, he stays where he is without performing any action. On the other hand, if he wants to move to a cell that contains a stone, the first thing to do is check that behind the stone there is nothing to move the stone. Once the stone is moved, you must check if it falls into the empty space due to gravity.

5. Modules to Implement

5.1. Mine Module

✓ This module defines the type *tCell*, the type *tPlane*, and the type *tMine*. To implement the functionality of the mine you must use at least the following functions:

- `void loadMine(ifstream& file, tMine& mine): reads data from a file, count the number of gems it contains and saves the miner's position.`
- `void draw1_1(const tMine& mine): draws the mine at 1:1 scale.`
- `void draw1_3(const tMine& mine): draws the mine at 1:3 scale and uses the following function draw3x3.`
- `void draw3x3(tCell cell, tPlaneCharacters characters, tPlaneColours colours, int i, int j): draws the squares enlarged three times. Specifically, the cell serves to update the plane of characters and colours in the coordinates i, j.`

5.2. Game Module

- ✓ In this module the type *tGame* is defined and the operations necessary to manage the game are implemented. You must implement at least the following functions:
  - `bool loadGame(tGame& game, int level): loads the game of the indicated level, that is, initializes the game and loads the mine. You must also manage if the upload file is not available. The level of the game matches the name of the file. On the Virtual Campus, load files will be left called: "1.txt", "2.txt", "3.txt" and "4.txt".`
  - `bool makeMovement(tGame& game, tKey key): when the miner moves he causes the state of the mine and the movements counter to change. Possibly the gem counter and the number of dynamites also change. Any action of the miner counts as a movement, including dropping the dynamite and not being able to move forward by having a wall or stone in front of him.`

- void draw(const tGame& game): it shows the state of the mine taking into account the scale. In addition, it must show the number of gems collected so far, the number of movements made and the dynamites that the miner has used.

6. Navigation

- ✓ The main program will display the menu

```
1. Play 1:1 scale game.
2. Play 1:3 scale game.
0. Exit
Introduce an option:
```

and will read the option selected by the user. Next, the program will ask the user if he/she wants using the keys or if he/she prefers the computer using the movements from a file:

```
1. Introducing movements from keyboard.
2. Introducing movements from file.
0. Exit
Introduce an option:
```

✓ When a level is completed, the program offers the user is the possibility to go to the next level or leave

```
1. Play next level.
0. Exit.
Introduce an option:
```

7. Main Program

- ✓ The main program will display the menu and read the option selected by the user.

8. Implementation Aspects

- ✓ Do not use global variables: each subprogram, in addition to the parameters to exchange data, must declare its own local variables, those that need to be used in the code.
- ✓ Do not use jump instructions such as *exit*, *break* (beyond the clauses of a *switch*) and *return* (at places other than the last instruction of a function).