

Monster Sweeper

COLLABORATORS

	<i>TITLE :</i> Monster Sweeper		
<i>ACTION</i>	<i>NAME</i>	<i>DATE</i>	<i>SIGNATURE</i>
WRITTEN BY	Jordi Fita	July 10, 2020	

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME
38cfb1596353	2011-07-31	Monster Sweeper's downloads are now a pair of images instead.	jfita
8f352f6d4202	2011-07-31	Monster's Sweeper's screenshot is now floating on the right.	jfita
8fce3c97147c	2011-07-23	Corrected some grammatical and spelling mistakes in monstersweeper.	jfita
e7f21a559d06	2011-07-23	Added the list of keys for 'Mamono Puzzle'.	jfita
b0d4cee36fa2	2011-07-23	Added an screenshot of monstersweeper.	jfita
4ddb2f6af764	2011-07-23	I've made the abstract section a proper named section (?Introduction?) and made the ?Download? section a subsection of this.	jfita
82f157100563	2011-07-23	I no longer use %zu for printf, as MingW32 doesn't support it.	jfita
542ed88ab005	2011-07-22	Now I am forcing to reveal all cells except for ?blind? modes in monstersweeper.	jfita
58f08527d533	2011-07-22	Changed the color pairs to be less confusing.	jfita
ffde96227b3b	2011-07-22	Added a handmade flash function because the builtin function doesn't work with colors?	jfita

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME
529710421e76	2011-07-22	Added the tab key to move between invisible cells to monstersweeper.	jfita
d020e959e124	2011-07-22	The cell's height in monstersweeper is now of 1 character instead of 2.	jfita
c2f61993c8d0	2011-07-22	When the cell value in monstersweeper is 0, I now show an space instead.	jfita
05ab8cfd13cc	2011-07-22	monstersweeper now shows all the monster if the game is over regardless of whether is visible or not.	jfita
77e93bb343e2	2011-07-22	Added the Makefile section to monstersweeper and now I also build its makefile and the rest of the program from itself. Added the resulting file to ignore.	jfita
5c9d8290233f	2011-07-22	Added the download section to monstersweeper.	jfita
fa6df6453f97	2011-07-22	The player experience now only increments if she survived the attack.	jfita
9733073c88b2	2011-07-22	Corrected a bug printing invalid characters in monstersweeper.	jfita
af9b85c774d6	2011-07-22	Corrected a problem with attacking the player when the monster has less level that the player?	jfita
05f7a22f9ab9	2011-07-22	Added the main function to monstersweeper.	jfita
f908bff08fd4	2011-07-22	Changed the way the monster attacks the player.	jfita
fe343ddc08d9	2011-07-22	Added the difficulty section and also added the coloration of the monster characters.	jfita

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME
e3c4db832e39	2011-07-22	`StatusView` and `GameView` now use a new `Window` class to manage their curses window pointer as RAI instead of having a destructor and declaring their undefined copy constructors and assignment operators. They are still uncopyable, because the default copy constructor and assignment operator can't be synthesized.	jfita
8a4887960804	2011-07-22	`getch()` return type is `int` not `chtype`, and thus the if that made sure that the returned wasn't `ERR` did nothing at all because `chtype` is unsigned and `ERR` is -1.	jfita
1ece7afd06f4	2011-07-21	Added the section with the curses library initialization and cleanup to monster sweeper.	jfita
f5e0dc7ea777	2011-07-21	I now initialize `remaining_cell_` in monstersweeper.	jfita
bb8b4e3f6e5d	2011-07-21	Added a flash when the player levels up in monstersweeper.	jfita
9df5960ddd88	2011-07-21	Added padding and left-adjustment to the calls to mvwprintw in monstersweeper.	jfita
ba3c407871bb	2011-07-21	I now initialize the `Game` `next_level_experience_` membre attribute with the first value.	jfita
684d5a5390f9	2011-07-21	GameView can scroll the whole board.	jfita
b2ed214397e1	2011-07-21	Inverted the left and right arrows when scrolling the game view in monstersweeper.	jfita
dd99d559be0b	2011-07-21	Corrected an invalid use of std::max instead of std::min in monstersweeper.	jfita
b30da25d53e9	2011-07-21	Removed a debugging leftover from monstersweeper.	jfita

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME
b7894873d3ce	2011-07-21	Fixed some mistakes in getting the GameView size and the reveal login in monstersweeper.	jfita
ea872018141a	2011-07-21	Added the GameView section to monstersweeper.	jfita
4315180d4c91	2011-07-21	Added the width() getter and the runtime_exception when the constructor fails to StatusView.	jfita
fab00cfaa58c	2011-07-21	Added the section about the `StatusView` to monstersweeper.	jfita
e2d71717448c	2011-07-21	The game's elapsed time in monstersweeper is not limited to 9999 seconds.	jfita
642eccc9dc03	2011-07-21	Added the ?End of Game? subsection to ?Model? in monstersweeper.	jfita
5975dc2692eb	2011-07-21	Fixed some incorrect C++ syntax from monstersweeper's fragments.	jfita
778e9cb90287	2011-07-20	Finished the section about the model.	jfita
288a5f01e6af	2011-07-19	Added half the ?revealing cells? subsection of ?Model?.	jfita
99535ba6ab34	2011-07-19	Added the section of the model along with its ?board? and ?monsters? subsections.	jfita
e41fe180ce07	2011-07-18	Removed an unused Timer class fragment link from Monster Sweeper.	jfita
a772376eeae	2011-07-18	Fixed some invalid fragments labels in Monster Sweeper.	jfita
09fa20768146	2011-07-18	Added the section about the main loop to Monster Sweeper.	jfita

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME
7b84de505d18	2011-07-18	Added the abstract and license of Monster Sweeper.	jfita

Contents

1	Introduction	1
1.1	Controls	1
2	The Game Loop	2
3	The Model	6
3.1	The Board	6
3.2	Monsters	7
3.3	Revealing cells	10
3.4	End of Game	13
4	Views	14
4.1	Window	14
4.2	Status View	15
4.3	Game View	18
5	Difficulty Modes	23
6	Initialization and Cleanup of ncurses	25
7	Main	27
8	monstersweeper.cpp	27
9	Makefile	28
10	License	29

List of Figures

1 Screenshot of *Monster Sweeper* 1

1 Introduction

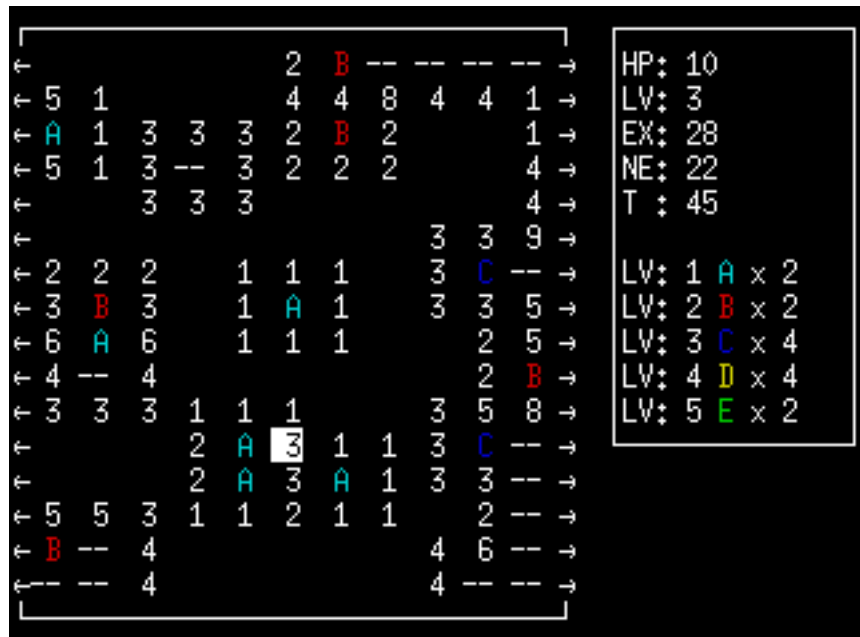
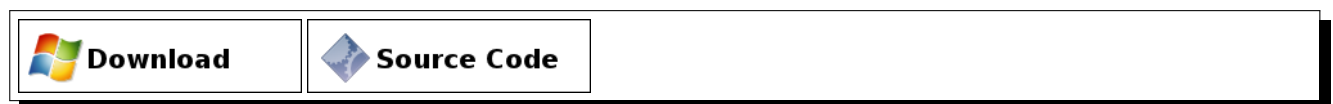


Figure 1: Screenshot of *Monster Sweeper*

Monster Sweeper is an ASCII demake of Hojamaka Games's popular *Mamono Sweeper*, a clever and unique take on the classic *Minesweeper*, where mines are replaced with monsters and the player now has RPG-like character stats to keep track of.

Similar to *Minesweeper*, the numbers of each cell indicate the sum of every monster's level in the surrounding squares. Using logic, the player can guess the location of the low-level monsters that can be fought without losing any hit points and gain experience in order to level up.

This version of *Monster Sweeper* is made with C++ and uses the *curses library* to draw the characters on the terminal and to receive the player input.



1.1 Controls

Monster Sweeper is controlled only with the keyboard. Here are listed the keys that *Monster Sweeper* accepts:

- Arrow keys or h, j, k, and l: Move the cursor around the screen.
- Space: reveals the cell under the cursor, if the cell is not already visible.
- 0: Moves the cursor to the first column.
- \$: Moves the cursor to the last column.
- g: Moves the cursor to the first line.
- G: Moves the cursor to the last line.
- Tab: Moves the cursor to the next unrevealed cell.

- q: Exits the game.
- Escape: Aborts the current play and allows the player to select a new difficulty mode.

2 The Game Loop

The application spends most of its time looping inside the function where the game is played; where the main loop is. This function expects an already set up board as a parameter and then creates the necessary curses windows — one window to show the actual game play while the other shows statistics such as the remaining hit points, experience points, etc. — and starts the game.

Inside the game loop I have to draw the board with its revealed and hidden squares as well as the current game's status. Then the game patiently waits for the player to press a key and acts according to it.

This game loop must keep running until either the board is completely revealed or the player lost too many hit points. At this time, the function returns whether the player won or lost.

```
<<function that plays the game>>=
bool Play(Game game)
{
    <<clear screen>>
    <<create the curses windows>>
    while (!game.IsDone())
    {
        <<draw the board>>
        <<draw the game status>>
        <<wait for the player to press a key>>
        <<act according to the pressed key>>
    }
    <<draw the board>>
    <<draw the game status>>

    return game.DidPlayerWin();
}
```

In this function, the central object is the `game` passed as a parameter that not only contains the actual square's contents, but also keeps the whole game's status such as the player's hit points, experience points, current level, etc.

However, before I am able to show to the player any of this information, I need to clear the screen and then create the two windows, here called views, that the curses library will use to draw to the screen.

To clear the screen, I **erase** any previously drawn character from the whole screen and then **refresh** to tell curses to update the terminal screen.

```
<<clear screen>>=
erase();
refresh();
```

Both windows used to draw on the screen, are actually a *view* as defined in the **Model-view-controller (MVC) patter**, while the game objects acts as the *model* and this function is the *controller*. Being views, the two windows require the reference to the game object that they need to get the information from. Of course, each view will retrieve and show show different information of the same model.

The view for the game play, besides the actual game to draw, also requires the *maximum number of columns* that can use. This, and the game board's width, is used to know whether the view needs to scroll or not to be able to show the whole board. Given that this maximum width depends on the status view's width and the columns available from the screen, plus a separation between the two views, I need to create the status view first.

```
<<create the curses windows>>=
StatusView status_view(game);
GameView game_view(game, COLS - status_view.width() - 1);
```

These views are defined below and each is tailored to show the relevant information about the game as contained in the `game` object. Since both views know how to draw themselves — actually is almost the only thing they know to do — I call their `draw` function at the appropriate time.

```
<<draw the game status>>=
status_view.Draw();
```

```
<<draw the board>>=
game_view.Draw();
```

And with this, the only remaining work to do for this function is to wait for the player to press a key and either update the views — such as move the game view's cursor — or update the game by exposing new squares.

I can sit and wait for the player to hit a key by calling the `getch()` function implemented by `curses`.

```
<<wait for the player to press a key>>=
int key = getch();
```

This function is declared in the `curses` header file that I need to include.

```
<<headers>>=
#include <curses.h>
```

If `getch` returns an error (i.e., the constant `ERR`) then I can't continue playing the game and must exit. I signal this error throwing an exception of type `runtime_error`. This exception class is defined in the standard `stdexcept` header file.

```
<<headers>>=
#include <stdexcept>

<<wait for the player to press a key>>=
if (ERR == key)
{
    throw std::runtime_error("there was an error reading input");
}
```

If the exception is not thrown, then I must have read a character from the terminal and thus I can act according to the pressed key. The simplest way to do handle this is by using a `switch` statement based on the `key` variable.

```
<<act according to the pressed key>>=
switch(key)
{
    <<input key cases>>
    default:
        // ignored.
        break;
}
```

Here, I need to have the different key cases for each of the accepted key commands. Any key that is not handled here is an unknown command and gets ignored in the `default` case. Another options would be to warn the user about the invalid key, but I find this annoying.

To begin, I am going to consider the `q` key, which means that the player wants to quit the game. As I have can't inform the function's caller that it needs to end the application this using the normal return, which is used to tell whether the player won or lost, I am going to think quitting the game as an “exceptional situation” and thus throw an exception.

```
<<input key cases>>=
case 'q':
    throw quit_exception();
    break;
```

This exception class is simply a `runtime_error` derived exception that I use to be able to tell apart an actual error condition from simply wanting to quit the game. Perhaps this is stretching a little too far the definition of “exceptional case”, but I believe to be a decent solution in this case.

```
<<quit_exception class>>=
class quit_exception: public std::runtime_error
{
    public:
        quit_exception():
            std::runtime_error("quit")
        {
        }
};
```

The escape key, in contrast to `q`, ends the current game without actually quitting the application. To tell that I want to abort the game, I use the same method that quitting, but using a different exception class — `abort_game_exception` — to handle the two cases differently from the outside.

```
<<input key cases>>=
case 27: // Escape.
    throw abort_game_exception();
    break;
```

```
<<abort_game_exception class>>=
class abort_game_exception: public std::runtime_error
{
    public:
        abort_game_exception():
            std::runtime_error("stop game")
        {
        }
};
```

Other keys that I want to manage are the movement keys that move the game view’s cursor around and that scroll the view when the game board is too big to fit within the screen. In *Monster Sweeper* I use the arrow keys as well as the *vi*-like movement keys — `h`, `j`, `k`, and `l` — as they are frequently used in *roguelike* games. I also accept the diagonal keys defined by curses.

```
<<input key cases>>=
case KEY_DOWN:
case 'j':
    game_view.MoveDown();
    break;

case KEY_LEFT:
case 'h':
    game_view.MoveLeft();
    break;

case KEY_RIGHT:
case 'l':
    game_view.MoveRight();
    break;

case KEY_UP:
case 'k':
    game_view.MoveUp();
    break;

case KEY_A1: // Upper left of keypad
    game_view.MoveUp();
    game_view.MoveLeft();
```

```

        break;

case KEY_A3: // Upper right of keypad
    game_view.MoveUp();
    game_view.MoveRight();
    break;

case KEY_C1: // Lower left of keypad
    game_view.MoveDown();
    game_view.MoveLeft();
    break;

case KEY_C3: // Lower right of keypad
    game_view.MoveDown();
    game_view.MoveRight();
    break;

```

As shortcuts, the game view also allows to move to the first and last column as well as the first and last row. Here I am again following, albeit somewhat loosely, the vi conventions and use 0 (zero) to move to the first column, \$ to jump and the last column, g to go to the first row, and G to move to the last row.

```

<<input key cases>>=
case '0':
    game_view.GoToFirstCol();
    break;

case '$':
    game_view.GoToLastCol();
    break;

case 'g':
    game_view.GoToFirstRow();
    break;

case 'G':
    game_view.GoToLastRow();
    break;

```

Also as a shortcut, the tab key moves to the next cell that has yet to be revealed, if there is any.

```

<<input key cases>>=
case '\t':
    game_view.GoToNextInvisibleCell();
    break;

```

The last remaining key I need to handle is the key that reveals a board's cell. I only need to ask to `game_view` where the cursor is currently located at and pass this position to `game`. The `game` object has all the logic that governs what entails when revealing a cells, hence in this function I don't need to worry about anything else.

The key to discover a square is space or the key at the center of the keypad, defined as `KEY_B2` by curses.

```

<<input key cases>>=
case ' ': // space
case KEY_B2: // Center of keypad
{
    GameView::Cursor cursor(game_view.BoardCursor());
    game.Reveal(cursor.x, cursor.y);
}
break;

```

This is all what the main loop, or the *controller* if you wish, needs to do to play *Monster Sweeper*.

3 The Model

As mentioned earlier, the game's *model* is stored in the `Game` class, which is the central hub of information about the game's current status.

```
<<Game class>>=
class Game
{
    public:
        <<Game board cell structure definition>>

        <<Monster types definition>>

        <<Game constructor>>

        <<Game public functions>>

    private:
        <<Board type definition>>

        <<Position structure definition>>

        <<Experience types definition>>

        <<Flash screen function>>

        <<Game private attributes>>
};
```

3.1 The Board

The most relevant piece of information in the `Game` class is the board where the monsters are located and that the player must reveal according to her and the monster's.

As not every cell in the board has a monster, I need a way to differentiate between the two kind of cells: these with monsters and these without. I also need to know whether the cell has already been revealed or not. The easiest way to put all this is to have a structure with the information about the cell. Each cell has a boolean member that tells if the cell is revealed, whether has monster or not, and either the monster type or the sum of every monster in the surrounding cells.

The last piece of information—either the monster or the sum of levels—is stored in a `chtype` variable. This is because I want to make easier to show the monster on the screen by storing the actual monster character in the cell. Give that `chtype` is defined as an unsigned integer, I can use the same data type to store the sum of level. To know how I need to interpret that value (i.e., monster or sum of levels) I need to look at the structure's `is_monster` boolean member. This variable's default value is `false`.

```
<<Game board cell structure definition>>=
struct Cell
{
    bool is_monster;
    bool is_visible;
    chtype value;

    Cell():
        is_monster(false),
        is_visible(false),
        value(0)
    {
    }
};
```

The board, thus, can be defined as a array of `Cell` but, since I need to have an arbitrary and dynamic board size, the easiest is to store the board as a vector of `Row` instead, being `Row` defined as another vector of `Cell`, to form a 2D array.

```
<<headers>>=
#include <vector>

<<Board type definition>>=
typedef std::vector<Cell> Row;
typedef std::vector<Row> Board;

<<Game private attributes>>=
Board board_;
```

3.2 Monsters

Another key piece of information that I need to store in the `Game` class is the definition of the different kind of monsters. Each monster has the same attributes: level, the attack points, the experience points, and the number of monster of that kind that remain on the board. Again, all this information is better defined in a structure.

```
<<Monster types definition>>=
struct Monster
{
    size_t level;
    size_t attack_points;
    size_t experience_points;
    size_t remaining;

    Monster(size_t level, size_t attack_points, size_t experience_points,
            size_t remaining):
        level(level),
        attack_points(attack_points),
        experience_points(experience_points),
        remaining(remaining)
    {
    }
};
```

The monster identifier—that is, the character that I put in the ‘board’ to represent the monster—isn’t stored in the `Monster` structure itself. Instead, as I have to lookup the monsters’ data when revealing squares later, I have `map` structure to assign the identifiers to their corresponding structure.

```
<<headers>>=
#include <map>

<<Monster types definition>>=
typedef std::map<ctype, Monster> MonsterMap;

<<Game private attributes>>=
MonsterMap monsters_;
```

This map isn’t initialized in the constructor as the others member attributes are. This is because when the board is created, I don’t want to pass all the monsters as parameters to the constructor. Instead, `Game` provides a member function to *add* new monster definitions. This function not only adds a new definition to the map, but also places the monsters randomly on the board, so after I added all the monsters definitions, the board game is also ready. As a caveat of having the initialization of the board and the monsters definition outside the constructor, I could end up with a board without any monster whatsoever. However, I see this as a valid board setup, albeit one not very exciting.

The `next_level_experience` parameter to this function is explained in the next section.

```
<<headers>>=
#include <cassert>

<<Game public functions>>=
void AddMonster(chtype monster_type, size_t level, size_t attack_points,
    size_t experience_points, size_t monsters_to_add,
    size_t next_level_experience)
{
    assert(monsters_to_add > 0 && "At least add one monster");
    assert(level > 0 && "Invalid monster level");
    assert(free_cells_.size() > monsters_to_add && "Too many monsters");
    assert(monsters_.find(monster_type) == monsters_.end() &&
        "Monster already defined");

    monsters_.insert(MonsterMap::value_type(monster_type,
        Monster(level, attack_points, experience_points, monsters_to_add)));
    <<store level experience>>
    for (; monsters_to_add > 0 ; monsters_to_add--)
    {
        Position pos(free_cells_.back());
        SetMonsterCell(pos.x, pos.y, monster_type);
        free_cells_.pop_back();
    }
}
```

The function `SetMonsterCell` simply sets the monster type to the specified cell.

```
<<Game public functions>>=
void SetMonsterCell(size_t x, size_t y, chtype monster_type)
{
    Cell &cell(Get(x, y));
    cell.is_monster = true;
    cell.value = monster_type;
}
```

The `Get` function using by `SetMonsterCell` returns the reference to the cell at the position passed as parameter. This function also checks that the coordinates are within the board limits, although it only does an `assert` instead throwing an exception on error; a precondition of this function.

```
<<Game public functions>>=
size_t width() const
{
    return board_[0].size();
}

size_t height() const
{
    return board_.size();
}

Cell &Get(size_t x, size_t y)
{
    assert(y < height());
    assert(x < width());

    return board_[y][x];
}

const Cell &Get(size_t x, size_t y) const
{
    return static_cast<const Cell &>(const_cast<Game &>(*this).Get(x, y));
}
```



```
}
```

Notice how I am using a `free_cell_member` vector to get the position where I have to put the monster to. This vector is initialized in the `Game` constructor with all the board's possible positions, which are simple structures with an `x` and `y` coordinate, meaning that all the cells are free to put monsters at this point. Then, using the standard `random_shuffle` function, I randomly shuffle this positions to use later to put the monsters in a random position.

By using `random_shuffle` the iterator of the structure to shuffle must be *random* and as a consequence I can't use a `list`, that a priori seems to be a better fit, because its iterators are sequential. However, even with the data structure drawback, by shuffling the positions beforehand I don't have to hunt for a free cell for each monster I need to add, avoiding a possibly long initialization time if there are a lot of monsters.

```
<<Position structure definition>>=
struct Position
{
    size_t x;
    size_t y;

    Position(size_t x, size_t y):
        x(x),
        y(y)
    {
    }
};
```

```
<<Game private attributes>>=
std::vector<Position> free_cells_;
bool reveal_everything_;
```

```
<<headers>>=
#include <algorithm>
```

```
<<Game constructor>>=
Game(size_t width, size_t height, size_t hit_points, bool reveal_everything = true):
    board_(height, Row(width)),
    monsters_(),
    free_cells_(),
    reveal_everything_(reveal_everything),
    <<other Game constructor initializations>>
{
    assert(width > 0 && "Invalid board width");
    assert(height > 0 && "Invalid board height");
    assert(hit_points > 0 && "Invalid initial hit points");

    for(size_t y = 0 ; y < height ; ++y)
    {
        for(size_t x = 0 ; x < width ; ++x)
        {
            free_cells_.push_back(Position(x, y));
        }
    }
    std::random_shuffle(free_cells_.begin(), free_cells_.end());
}
```

Notice also that there is a `reveal_everything` parameter to `Game` constructor. This flag tells whether all the cells must be revealed in order to win the game or if it only is necessary to reveal the cells containing no monsters. This is used later when verifying the winning condition.

3.3 Revealing cells

Revealing the cells is the meat of the game and, in consequence, where most of the game's logic is. To reveal a cell, I check whether the specified cell is already revealed or not, because for already visible cells I need to do nothing at all. If the cell isn't yet revealed, I make this cell visible and decrease the number of total remaining cells on the board, which is initialized in the constructor.

```
<<Game private attributes>>=
size_t remaining_cells_;

<<other Game constructor initializations>>=
remaining_cells_(width * height),

<<Game public functions>>=
void Reveal(size_t x, size_t y)
{
    Cell &cell(Get(x, y));
    if (!cell.is_visible)
    {
        cell.is_visible = true;
        <<reveal the cell>>
        <<initialize the game timer>>
        --remaining_cells_;
    }
}
```

When revealing the cell, I have two possible cases: cells with monsters and empty cells. When I encounter a monster, the monster attacks the player as many times as the difference between the its level and the player's. If the player can sustain the attack, the monster is defeated and I have to increase the player's experience points according to the monster type and check if I must increase the player's level.

I also make the screen flash a brief moment to warn the player that she has leveled up. However, I found out that the curses **flash** function doesn't seem to work when the colors output is enables. Thus, I had to make my own `FlashScreen` function that shows the whole screen in reverse mode for 50 milliseconds and then restores the window as it were. This function makes the characters to drop any attributes — i.e., colors — but this actually doesn't matter because once the cell is revealed, everything is redrawn again anyway.

```
<<Flash screen function>>=
void FlashScreen()
{
    struct StripAttributes
    {
        StripAttributes()
        {
            for(int y = 0 ; y < LINES ; ++y)
            {
                for(int x = 0 ; x < COLS ; ++x)
                {
                    mvaddch(y, x, mvinch(y, x) & A_CHARTEXT);
                }
            }
        };

        attron(A_REVERSE);
        StripAttributes reverse;
        refresh();

        napms(50);

        attroff(A_REVERSE);
    };
}
```

```
StripAttributes restore;
refresh();
}
```

```
<<reveal the cell>>=
if (cell.is_monster)
{
    MonsterMap::iterator monster_iter = monsters_.find(cell.value);
    assert(monster_iter != monsters_.end() && "Invalid monster");
    Monster &monster(monster_iter->second);
    --monster.remaining;
    if (monster.level > player_level_)
    {
        player_hit_points_ -=
            std::min(monster.attack_points * (monster.level - player_level_),
                    player_hit_points_);
    }
    if (player_hit_points_ > 0)
    {
        player_experience_ += monster.experience_points;
        if (player_experience_ >= next_level_experience_)
        {
            ++player_level_;
            next_level_experience_ = GetLevelExperience(player_level_ + 1);
            FlashScreen();
        }
    }
}
```

The `player_hit_points_`, `player_experience_`, `player_level_`, and `next_level_experience_` are private member attributes of `Game` that are initialized to default values in the constructor, except for `player_hit_points_` that the caller passes its value as a parameter to the constructor.

```
<<Game private attributes>>=
size_t next_level_experience_;
size_t player_hit_points_;
size_t player_experience_;
size_t player_level_;
```

```
<<other Game constructor initializations>>=
next_level_experience_(9999),
player_hit_points_(hit_points),
player_experience_(0),
player_level_(1),
```

To get the experience points required to advance to the next level, I use the until now unexplained `next_level_experience` parameter to the `AddMonster` function. This parameter is actually the value that I need to know in order to allow the player to level up. I store this value in a map, like I do for the monsters themselves, except that instead of using a structure, I now only save the experience points value.

```
<<Experience types definition>>=
typedef std::map<size_t, size_t> ExperienceMap;
```

```
<<Game private attributes>>=
ExperienceMap levels_experience_;
```

```
<<other Game constructor initializations>>=
levels_experience_(),
```

When the caller passes the required experience points to level up as parameter to the `AddMonster` function, then I store this parameter mapped to the monster's **next level**, not the actual monster's level. If the monster's level is the first, then I initialize the `next_level_experience_` with this value, as I need to have this variable initialized before playing the game.

```
<<store level experience>>=
levels_experience_[level + 1] = next_level_experience;
if (1 == level)
{
    next_level_experience_ = next_level_experience;
}
```

With this setup, retrieving the next level experience with `GetLevelExperience` is straightforward.

```
<<Game public functions>>=
size_t GetLevelExperience(size_t level)
{
    assert(levels_experience_.find(level) != levels_experience_.end() &&
           "Level experience not found");
    return levels_experience_[level];
}
```

If the cell that is to be revealed is empty and doesn't contain a monster, then, instead of juggling with the player's statistics, I need to compute the cell's value as the sum of the surrounding monsters' level.

If the resulting value is zero, for convenience, the game also reveals all the neighbour cells, by calling the function recursively and passing the position of the neighbour cells to reveal. Remember that if a cell has been already revealed this function does nothing, that is why is so important to set the `is_visible` variable to `true` as soon as possible: to avoid infinite recursion when a neighbour's cell is also empty and calls this function with the initial cell's position.

```
<<reveal the cell>>=
else
{
    size_t start_x = x > 0 ? x - 1 : 0;
    size_t end_x = x < width() - 1 ? x + 2 : x + 1;
    size_t start_y = y > 0 ? y - 1 : 0;
    size_t end_y = y < height() - 1 ? y + 2 : y + 1;

    cell.value = 0;
    for (size_t cell_y = start_y ; cell_y < end_y ; ++cell_y)
    {
        for(size_t cell_x = start_x ; cell_x < end_x ; ++cell_x)
        {
            cell.value += GetMonsterLevelAtCell(cell_x, cell_y);
        }
    }

    if (0 == cell.value)
    {
        for(size_t cell_y = start_y ; cell_y < end_y ; ++cell_y)
        {
            for (size_t cell_x = start_x ; cell_x < end_x ; ++cell_x )
            {
                Reveal(cell_x, cell_y);
            }
        }
    }
}
```

The function that I use here to get a monster level in a cell, `GetMonsterLevelAtCell`, is as simple as it sounds: it gets a cell, checks whether there is a monster in there, and returns its level. If there is no monster, then it always returns 0.

```
<<Game public functions>>=
```

```
size_t GetMonsterLevelAtCell(size_t x, size_t y) const
{
    const Cell &cell(Get(x, y));
    if (cell.is_monster)
    {
        MonsterMap::const_iterator monster(monsters_.find(cell.value));
        assert(monster != monsters_.end());
        return monster->second.level;
    }
    return 0;
}
```

The only thing that remains to do when revealing a cell is to start the game timer. The game counts the number of seconds that have elapsed since the first time a cell was revealed. For me, the easiest way to get the number of seconds elapsed is to store the time the first cell was revealed and then, each time I need to know the elapsed time, get the current time and subtract it from this initial time, but not allowing this time to be more than 9999 seconds, mostly to be able to tell beforehand how many digits I need to display the time on the screen. And also, quite blunt, a player that spends 9999 seconds (almost 3 hours) isn't very good at this game.

```
<<headers>>=
#include <ctime>
```

```
<<Game public functions>>=
time_t GetElapsedTime() const
{
    if (start_time_ > 0)
    {
        return std::min(time_t(9999), std::time(NULL) - start_time_);
    }
    return 0;
}
```

The `start_time_` attribute is initialized to zero in the constructor for me to know that the game hasn't started yet. Thus, when the user reveals a cell, I check this attribute and set the current time if I find that its value is the zero.

```
<<Game private attributes>>=
time_t start_time_;
```

```
<<other Game constructor initializations>>=
start_time_(0)
```

```
<<initialize the game timer>>=
if (0 == start_time_)
{
    start_time_ = std::time(NULL);
}
```

3.4 End of Game

The game ends when the player has revealed, without loosing all her hit points, all the cells that didn't have monsters or all the cells without losing all her hit points, depending on the `reveal_everything_` member attribute.

To know whether the player still has hit points, I simply check whether the variable is zero. For the cells is a little more involved, but not that much. First, knowing that I decrease the number of remaining monsters each time a monster is revealed, I find how many monsters remain to be revealed on the board. Then, I check if the remaining cells and the remaining monsters are the same, but I may require the number of remaining cells to be zero or not, depending on the `reveal_everything_` flag. If this flag is set, then there can be no remaining cell, and thus no remaining monsters. Otherwise, there can remain any number of cells as long as they all are cells with monsters inside.

```
<<Game public functions>>=
bool IsDone() const
{
    if (player_hit_points_ == 0)
    {
        return true;
    }

    size_t remainingMonsters = 0;
    for(MonsterMap::const_iterator monster = monsters_.begin() ;
        monster != monsters_.end() ; ++monster)
    {
        remainingMonsters += monster->second.remaining;
    }
    return (!reveal_everything_ || remaining_cells_ == 0) &&
        remaining_cells_ == remainingMonsters;
}
```

Later, to check whether the player won or not, I only check whether the game is done but the player still has some hit points left.

```
<<Game public functions>>=
bool DidPlayerWin() const
{
    return IsDone() && player_hit_points_ > 0;
}
```

4 Views

Monster Sweeper has two views that show information about the game. One of these views simply shows the game's statistics, such as the remaining hit points, the level, the experience, the next level experience, the game time and the number and level of every kind of monster. This is called the *StatusView*.

The other view is where the player will spend most of the time and is the view that shows the board on the screen. This view must show revealed and hidden cells as well as the cursor the player controls. This view is called *GameView*.

Although the two views are independent of each other, both views are still based on curses and thus require some common structures. Namely, the window to draw on.

4.1 Window

Both views show their content inside a curses *window* that they must manage. However, managing a window means that I have to take care to initialize the proper data structures and release them when are no longer needed.

A common C++ idiom to encapsulate this data initialization and posterior release is to wrap the raw resources in a **Resource Acquisition Is Initialization (RAII)** class. This class sole responsibility is to hold the window's pointer, in this case, created in its constructor and release the pointer's data on its destructor.

```
<<Window class>>=
class Window
{
public:
    Window(int nlines, int ncols, int begin_y, int begin_x):
        window_(newwin(nlines, ncols, begin_y, begin_x))
    {
        if (0 == window_)
        {
            throw std::runtime_error("couldn't create window");
        }
    }
}
```

```

    }

    ~Window()
    {
        delwin(window_);
    }

    <<Window public functions>>

    <<make Window class uncopyable>>

private:
    WINDOW *window_;
};

```

Beside the initialization and release of the window's pointer, I also need to access to this class' pointer to draw on. The cleanest way is to add a new member function that just returns the pointer.

This function can't be `const` because the caller could modify the window using the pointer. Although in the eyes of the C++ compiler this is still `const` in a bitwise sense (i.e., the bits inside the class don't change), this is not my idea of a `const` function, which means that the class status doesn't change. That is why I don't make this function `const`.

```

<<Window public functions>>=
WINDOW *Get()
{
    return window_;
}

```

Lastly, to avoid nasty and hard to find bugs, I don't allow to make copies of the `Window` class. This way, I won't call `delwin` on the same pointer twice just because the pointer got copied from object to object. To disallow copies, I define the copy constructor and the assignment operator as protected, but I leave them **undefined**. If someone tries, by mistake, to make a copy, the compiler won't allow them.

```

<<make Window class uncopyable>>=
protected:
    Window(const Window &);
    Window &operator=(const Window &);

```

4.2 Status View

As I said, the `StatusView` class is the responsible to show to the player the game's non-essential, although highly useful, data such as:

- The player's remaining hit points.
- The player's current level.
- The player's current experience points.
- The experience points required in order to advance to the next level.
- The game's elapsed time.
- The level and remaining number of each kind of monster.

```

<<StatusView class>>=
class StatusView
{
public:
    <<StatusView constructor>>

```

```

    <<StatusView public functions>>

    private:
        <<StatusView constants>>

        <<StatusView private attributes>>

};

```

Besides creating the required data structures to allow me to show this data, I have to modify the definition of the model in order to get all the information. Mostly, I have to add getters to the Game class.

The first data I need from ‘Game’ is the number of monsters it has. I need this information to know how many lines the status window needs to have. As I will also need to get the monsters’ data structure later when showing their level, the best way I can think of counting the different kind of monsters is by writing the getter that returns the monster map and then just call its `size()` member function.

```

<<Game public functions>>=
const MonsterMap &monsters() const
{
    return monsters_;
}

```

Based on this data and knowing that I need to show the level, the hit points, the experience, the next level experience, and the game’s time, I can compute the lines required to show all the information. The width is always fixed to 7 characters, to allow enough space to show everything, plus the two characters I need to show the box around the window. The height has a minimum of 8 lines, and then I must add the number of monsters in the game.

```

<<StatusView private attributes>>=
const Game &game_;
Window window_;

```

```

<<StatusView constants>>=
static const int kWindowWidth = 14 + 2;
static const int kWindowMinHeight = 8;

```

```

<<StatusView constructor>>=
StatusView(const Game &game):
    game_(game),
    window_(kWindowMinHeight + game.monsters().size(), kWindowWidth,
            0, COLS - kWindowWidth)
{
}

```

With the window created, showing the information is just a matter of calling the required Game getters and the curses `mvwprintw` function with the relevant bits. I use the uncommon “left adjustment” (i.e., `-width` flags) in order to remove any leftover characters when the values shrink by a digit.

However, I have to use `waddch` when outputting the monster’s characters because the characters could contain color information that `mvprintw`—which in turn calls `printf`—would throw away. Therefore, I have to split the line in multiple calls to `mvwprintw`, `waddch`, and `wprintw`.

```

<<StatusView public functions>>=
void Draw()
{
    box(window_.Get(), ACS_VLINE, ACS_HLINE);

    int line = 1;
    mvwprintw(window_.Get(), line++, 1, "HP: %-4llu",
        static_cast<unsigned long long>(game_.player_hit_points()));
    mvwprintw(window_.Get(), line++, 1, "LV: %-4llu",

```



```

        static_cast<unsigned long long>(game_.player_level()));
mvwprintw(window_.Get(), line++, 1, "EX: %-4llu",
        static_cast<unsigned long long>(game_.player_experience()));
mvwprintw(window_.Get(), line++, 1, "NE: %-4llu",
        static_cast<unsigned long long>(game_.next_level_experience() -
        game_.player_experience()));
mvwprintw(window_.Get(), line++, 1, "T : %-4lu",
        static_cast<unsigned long>(game_.GetElapsedTime()));
line++;

const Game::MonsterMap &monsters(game_.monsters());
for(Game::MonsterMap::const_iterator monster = monsters.begin() ;
    monster != monsters.end() ; ++monster)
{
    mvwprintw(window_.Get(), line, 1, "LV: %llu ",
        static_cast<unsigned long long>(monster->second.level));
    waddch(window_.Get(), monster->first);
    wprintw(window_.Get(), " x %-3llu",
        static_cast<unsigned long long>(monster->second.remaining));
    ++line;
}

wrefresh(window_.Get());
}

```

The getters for Game are trivial.

```

<<Game public functions>>=
size_t player_hit_points() const
{
    return player_hit_points_;
}

size_t player_level() const
{
    return player_level_;
}

size_t player_experience() const
{
    return player_experience_;
}

size_t next_level_experience() const
{
    return next_level_experience_;
}

```

The only other function I need for this view is its window's width, used in the function with the main loop to create the other view, as already mentioned. Here, I can cast the `const` of `window_` away because I know that `getmaxxx` doesn't modify the window at all. Without this cast I couldn't declare the getter `const`.

```

<<StatusView public functions>>=
int width() const
{
    return getmaxxx(const_cast<Window &>(window_).Get());
}

```

4.3 Game View

The `GameView` class is conceptually the same as the `StatusView`: a window where to draw some data from the model. However, even if both classes follow the same idea and even have some functions named similarly, I don't think useful to make a base class and derive both from this new class. After all, I am not going to use the two views in a polymorphic way.

```
<<GameView class>>=
class GameView
{
    public:
        <<Cursor type definition>>

        <<GameView constructor>>

        <<GameView public functions>>

    private:
        <<GameView constants>>

        <<GameView private attributes>>
};
```

Like `StatusView`, `GameView` has a curses window that is created in its constructor as well as the reference to the `Game` model. In this case, though, the constructor also receives the *maximum* number of columns that this view can use. `GameView` uses this width, as well as the terminal's height, to compute the maximum number of cells, not characters, that the view can show. Then I can create the window given this number and each cell's size in characters.

When computing the number of cells from the maximum columns, I have to reserve enough space for the view's border. Also, the view's usable area on the screen can be bigger or smaller than the board, so I always have to use the lowest of the usable screen's are or the board's size as the view size.

```
<<GameView constants>>=
static const int kCellWidth = 3; // characters
static const int kCellHeight = 1; // characters
```

```
<<GameView private attributes>>=
const Game &game_;
int view_height_;
int view_width_;
Window window_;
```

```
<<GameView constructor>>=
GameView(const Game &game, int maximum_width):
    game_(game),
    view_height_(std::min((LINES - 2) / kCellHeight, static_cast<int>(game.height()))),
    view_width_(std::min((maximum_width - 2) / kCellWidth, static_cast<int>(game.width()))) ←
    ,
    window_(view_height_ * kCellHeight + 2, view_width_ * kCellWidth + 2, 0, 0),
    <<other GameView initialization>>
{
}
```

As hinted before, `GameView` is able to handle boards that are bigger than what can be seen on the screen by scrolling the board. To scroll, I need a couple of variables that tell to the view which is the board's first column and first row that are visible on the screen, which by default are column 0 and row 0.

```
<<GameView private attributes>>=
size_t scroll_x_;
size_t scroll_y_;
```

```
<<other GameView initialization>>=
scroll_x_(0),
scroll_y_(0),
```

To know whether the view needs to scroll or not, I must allow the player to move around the view. This is what is termed a *cursor*, which is a 2D position on the view.

```
<<Cursor type definition>>=
struct Cursor
{
    size_t x;
    size_t y;

    Cursor():
        x(0),
        y(0)
    {
    }

    Cursor(size_t x, size_t y):
        x(x),
        y(y)
    {
    }
};
```

```
<<GameView private attributes>>=
Cursor view_cursor_;
```

```
<<other GameView initialization>>=
view_cursor_()
```

However, this view has **two** cursors: one is the actual cursor visible on the screen and the other is the position over the game board where the view cursor is on. This cursor isn't stored; is computed from the visible cursor and the current scroll.

```
<<GameView public functions>>=
Cursor BoardCursor() const
{
    return Cursor(view_cursor_.x + scroll_x_, view_cursor_.y + scroll_y_);
}
```

But a static cursor if of no use whatsoever, so I have to add functions that move the cursor around the screen and that also update the scroll variables when the player tries to move outside the view's limits, if possible. These scroll variable can't have a value that allows the view to draw outside the board's limits.

```
<<GameView public functions>>=
size_t Height() const
{
    return view_height_;
}

size_t Width() const
{
    return view_width_;
}

void MoveDown()
{
    if (view_cursor_.y < Height() - 1)
    {
```

```

        ++view_cursor_.y;
    }
    else if (scroll_y_ + Height() < game_.height())
    {
        ++scroll_y_;
    }
}

void MoveLeft()
{
    if (view_cursor_.x > 0)
    {
        --view_cursor_.x;
    }
    else if (scroll_x_ > 0)
    {
        --scroll_x_;
    }
}

void MoveRight()
{
    if (view_cursor_.x < Width() - 1)
    {
        ++view_cursor_.x;
    }
    else if (scroll_x_ + Width() < game_.width())
    {
        ++scroll_x_;
    }
}

void MoveUp()
{
    if (view_cursor_.y > 0)
    {
        --view_cursor_.y;
    }
    else if (scroll_y_ > 0)
    {
        --scroll_y_;
    }
}

```

GameView also has convenient functions to move to the first column, the first row, the last column, and the last row.

```

<<GameView public functions>>=
void GoToFirstCol()
{
    view_cursor_.x = 0;
    scroll_x_ = 0;
}

void GoToLastCol()
{
    view_cursor_.x = Width() - 1;
    scroll_x_ = game_.width() - Width();
}

void GoToFirstRow()
{
    view_cursor_.y = 0;
}

```

```

    scroll_y_ = 0;
}

void GoToLastRow()
{
    view_cursor_.y = Height() - 1;
    scroll_y_ = game_.height() - Height();
}

```

Another convenient function allows to move to the next yet unrevealed cell. Here I don't need to check if there is any remaining unrevealed cell, because if the game has called this function, as a result of a key press, I can assume that there is at least one invisible cell, otherwise the game would have ended already.

```

<<GameView public functions>>=
void GoToNextInvisibleCell()
{
    do
    {
        if (view_cursor_.x < Width() - 1 || scroll_x_ + Width() < game_.width())
        {
            MoveRight();
        }
        else
        {
            GoToFirstCol();
            if (view_cursor_.y < Height() - 1 || scroll_y_ + Height() < game_.height())
            {
                MoveDown();
            }
            else
            {
                GoToFirstRow();
            }
        }
    }
    while (game_.Get(
        scroll_x_ + view_cursor_.x,
        scroll_y_ + view_cursor_.y).is_visible);
}

```

The last remaining bit for GameView to do is to draw itself. Using the `scroll_x_` and `scroll_y_` variables I can know the position in the game's board to start reading cells. If the cell is visible, then I draw the cell's value either as a character, if it is a monster, or as an integer, except for empty cells with a value of 0, that I show as blank spaces. For unrevealed cells, I draw two hyphens.

To show where the cursor is, when I am going to draw the cell where the cursor is over, I'll reverse the character colors and make that cell distinguishable.

When the game is completed, either because the player lost all the hit points or because has revealed all the cells, I also draw all monsters, regardless of whether is revealed or not.

Here again the monster character can have color attributes. However, I also need the space in front of the monster character to be of the same color as the monster, otherwise when I draw the cursor in reverse mode, it has two colors and looks weird. As here the whole string needs to have the same color, I can extract the color information from the monster's character, if it has any, with a bitwise-and of the character and `A_COLOR` and apply the result attribute to the curses window with `wattron` and `wattroff` before and after printing the string, respectively. In this case, there is no need to split the output of the two characters in two different calls as it was with `StatusView`.

```

<<GameView public functions>>=
void Draw()
{
    bool show_monsters = game_.IsDone();
}

```

```

for(size_t y = 0, board_y = scroll_y_, cell_y = 1 ;
    y < Height() ;
    ++y, ++board_y, cell_y += kCellHeight)
{
    for(size_t x = 0, board_x = scroll_x_, cell_x = 1 ;
        x < Width() ;
        ++x, ++board_x, cell_x += kCellWidth)
    {
        bool show_cursor = y == view_cursor_.y && x == view_cursor_.x;
        if (show_cursor)
        {
            wattron(window_.Get(), A_REVERSE);
        }

        const Game::Cell &cell(game_.Get(board_x, board_y));
        if (cell.is_visible || (cell.is_monster && show_monsters))
        {
            if (cell.is_monster)
            {
                wattron(window_.Get(), cell.value & A_COLOR);
                mvwprintw(window_.Get(), cell_y, cell_x, "%c", cell.value);
                wattroff(window_.Get(), cell.value & A_COLOR);
            }
            else if (cell.value > 0)
            {
                mvwprintw(window_.Get(), cell_y, cell_x, "%2d", cell.value);
            }
            else
            {
                mvwprintw(window_.Get(), cell_y, cell_x, " ");
            }
        }
        else
        {
            mvwprintw(window_.Get(), cell_y, cell_x, "--");
        }

        if (show_cursor)
        {
            wattroff(window_.Get(), A_REVERSE);
        }
    }
}

```

I also want to draw a box around the GameView view, but this time I can't use box function and have to use border, because I want to display a different border if the board can be scrolled in any of the four directions. When a side can be scrolled, instead of vertical or horizontal lines, I show arrows pointing to the direction that can be scrolled.

```

<<GameView public functions>>=
    wborder(window_.Get(),
        (scroll_x_ > 0) ? ACS_LARROW : ACS_VLINE, // Left side,
        (scroll_x_ < game_.width() - Width()) ? ACS_RARROW : ACS_VLINE, // Right side
        (scroll_y_ > 0) ? ACS_UARROW : ACS_HLINE, // Top side
        (scroll_y_ < game_.height() - Height()) ? ACS_DARROW : ACS_HLINE, // Bottom side
        ACS_ULCORNER, ACS_URCORNER, ACS_LLCORNER, ACS_LRCORNER);

    wrefresh(window_.Get());
}

```

5 Difficulty Modes

The game must ask the user which difficulty mode she wants to play. As it would be too cumbersome to expect the user to enter each monster's level, experience points, attack points, etc., *Monster Sweeper* has a number of difficulty levels pre-configured.

The game shows a new window at the screen's center with the list of options and then waits until the player enters a valid choice. Then, based on that choice, I just return a new `Game` object set up according to the selected mode.

The only special case is when the player selects to quit the game. In this case, as I do when pressing `q` mid-game, I throw a `quit_exception` because I don't have any `Game` object to return.

```
<<function that asks for the difficulty>>=
Game AskDifficulty()
{
    const int kDialogWidth = 19;
    const int kDialogHeight = 11; // The 8 modes, a blank line,
                                   // and quit plus the border.

    <<monster types>>

    Window dialog(kDialogHeight, kDialogWidth,
        LINES / 2 - kDialogHeight / 2, COLS / 2 - kDialogWidth / 2);

    int line = 1;
    mvwprintw(dialog.Get(), line++, 2, "1. Easy");
    mvwprintw(dialog.Get(), line++, 2, "2. Normal");
    mvwprintw(dialog.Get(), line++, 2, "3. Huge");
    mvwprintw(dialog.Get(), line++, 2, "4. Extreme");
    mvwprintw(dialog.Get(), line++, 2, "5. Blind");
    mvwprintw(dialog.Get(), line++, 2, "6. Extreme Huge");
    mvwprintw(dialog.Get(), line++, 2, "7. Blind Huge");
    ++line;
    mvwprintw(dialog.Get(), line++, 2, "q. Quit");
    box(dialog.Get(), ACS_VLINE, ACS_HLINE);
    wrefresh(dialog.Get());
    for (;;)
    {
        int key = getch();
        switch (key)
        {
            case ERR:
                throw std::runtime_error("couldn't read from input");
                break;

            case '1':
            {
                Game easy(16, 16, 10);
                easy.AddMonster(kSlime, 1, 1, 1, 10, 7);
                easy.AddMonster(kGoblin, 2, 2, 2, 8, 20);
                easy.AddMonster(kLizard, 3, 3, 4, 6, 50);
                easy.AddMonster(kGolem, 4, 4, 8, 4, 82);
                easy.AddMonster(kDragon, 5, 5, 16, 2, 9999);
                return easy;
            }
            break;

            case '2':
            {
                Game normal(30, 16, 10);
                normal.AddMonster(kSlime, 1, 1, 1, 33, 10);
                normal.AddMonster(kGoblin, 2, 2, 2, 27, 50);
                normal.AddMonster(kLizard, 3, 3, 4, 20, 167);
                normal.AddMonster(kGolem, 4, 4, 8, 13, 271);
```

```
        normal.AddMonster(kDragon, 5, 5, 16, 6, 9999);
        return normal;
    }
    break;

case '3':
{
    Game huge(50, 25, 20);
    huge.AddMonster(kSlime, 1, 1, 1, 52, 10);
    huge.AddMonster(kGoblin, 2, 2, 2, 46, 90);
    huge.AddMonster(kLizard, 3, 3, 4, 40, 202);
    huge.AddMonster(kGolem, 4, 4, 8, 36, 400);
    huge.AddMonster(kDragon, 5, 5, 16, 30, 1072);
    huge.AddMonster(kDemon, 6, 6, 32, 24, 1840);
    huge.AddMonster(kNinja, 7, 7, 64, 18, 2292);
    huge.AddMonster(kDragonZombie, 8, 8, 128, 13, 4656);
    huge.AddMonster(kSatan, 9, 9, 256, 1, 9999);
    return huge;
}
break;

case '4':
{
    Game extreme(30, 16, 10);
    extreme.AddMonster(kSlime, 1, 1, 1, 25, 10);
    extreme.AddMonster(kGoblin, 2, 2, 2, 25, 50);
    extreme.AddMonster(kLizard, 3, 3, 4, 25, 175);
    extreme.AddMonster(kGolem, 4, 4, 8, 25, 375);
    extreme.AddMonster(kDragon, 5, 5, 16, 25, 9999);
    return extreme;
}
break;

case '5':
{
    Game blind(30, 16, 1, false);
    blind.AddMonster(kSlime, 1, 1, 1, 33, 9999);
    blind.AddMonster(kGoblin, 2, 2, 2, 27, 9999);
    blind.AddMonster(kLizard, 3, 3, 4, 20, 9999);
    blind.AddMonster(kGolem, 4, 4, 8, 13, 9999);
    blind.AddMonster(kDragon, 5, 5, 16, 6, 9999);
    return blind;
}
break;

case '6':
{
    Game hugeExtreme(50, 25, 10);
    hugeExtreme.AddMonster(kSlime, 1, 1, 1, 36, 3);
    hugeExtreme.AddMonster(kGoblin, 2, 2, 2, 36, 10);
    hugeExtreme.AddMonster(kLizard, 3, 3, 4, 36, 150);
    hugeExtreme.AddMonster(kGolem, 4, 4, 8, 36, 540);
    hugeExtreme.AddMonster(kDragon, 5, 5, 16, 36, 1116);
    hugeExtreme.AddMonster(kDemon, 6, 6, 32, 36, 2268);
    hugeExtreme.AddMonster(kNinja, 7, 7, 64, 36, 4572);
    hugeExtreme.AddMonster(kDragonZombie, 8, 8, 128, 36, 9180);
    hugeExtreme.AddMonster(kSatan, 9, 9, 256, 36, 9999);
    return hugeExtreme;
}

case '7':
{
```



```

        Game hugeBlind(50, 25, 1, false);
        hugeBlind.AddMonster(kSlime, 1, 1, 1, 52, 9999);
        hugeBlind.AddMonster(kGoblin, 2, 2, 2, 46, 9999);
        hugeBlind.AddMonster(kLizard, 3, 3, 4, 40, 9999);
        hugeBlind.AddMonster(kGolem, 4, 4, 8, 36, 9999);
        hugeBlind.AddMonster(kDragon, 5, 5, 16, 30, 9999);
        hugeBlind.AddMonster(kDemon, 6, 6, 32, 24, 9999);
        hugeBlind.AddMonster(kNinja, 7, 7, 64, 18, 9999);
        hugeBlind.AddMonster(kDragonZombie, 8, 8, 128, 13, 9999);
        hugeBlind.AddMonster(kSatan, 9, 9, 256, 1, 9999);
        return hugeBlind;
    }
    break;

    case 'q':
        throw quit_exception();
        break;
}
}
}

```

The monster types used here are just characters, but with the appropriate color attribute.

```

<<monster types>>=
const chtype kSlime = 'A' | COLOR_PAIR(1);
const chtype kGoblin = 'B' | COLOR_PAIR(2);
const chtype kLizard = 'C' | COLOR_PAIR(3);
const chtype kGolem = 'D' | COLOR_PAIR(4);
const chtype kDragon = 'E' | COLOR_PAIR(5);
const chtype kDemon = 'F' | COLOR_PAIR(6);
const chtype kNinja = 'G' | COLOR_PAIR(7);
const chtype kDragonZombie = 'H' | COLOR_PAIR(8);
const chtype kSatan = 'I' | COLOR_PAIR(9);

```

To be able to see the monsters in colors, first I have to initialize *curses*' color support as well as defining the *color pairs*. The color pairs are the definition of the foreground and background for each of the colors attributes used in the monster types used above.

I wanted to use a different color for each monster, but most terminals have only support for 8 different colors, hence I had to reuse some of the colors. I tried to use the same color for the characters that resemble the less, to avoid confusion, but it is not at all perfect.

```

<<initialize curses colors>>=
start_color();
init_pair(1, COLOR_CYAN, COLOR_BLACK);
init_pair(2, COLOR_RED, COLOR_BLACK);
init_pair(3, COLOR_BLUE, COLOR_BLACK);
init_pair(4, COLOR_YELLOW, COLOR_BLACK);
init_pair(5, COLOR_GREEN, COLOR_BLACK);
init_pair(6, COLOR_RED, COLOR_BLACK);
init_pair(7, COLOR_CYAN, COLOR_BLACK);
init_pair(8, COLOR_GREEN, COLOR_BLACK);
init_pair(9, COLOR_MAGENTA, COLOR_BLACK);

```

6 Initialization and Cleanup of ncurses

Although I could use directly the C API offered by *curses*, I usually prefer to keep initialization and release of resources in a RAII object that automatically takes care of their life cycle. This is specially true when there are exceptions involved in the mix.

```
<<Curses class>>=
class Curses
{
    public:
        <<Curses constructor>>

        <<Curses destructor>>

        <<make the Curses class uncopyable>>
};
```

In this case, the RAI object doesn't actually hold any pointer or reference to any resource, because *curses* keeps all its status in global variables. Thus, in the constructor I need to call *initscr*, which initializes the *stdscr* global variable and aborts the program if there is any error initializing the library.

```
<<Curses constructor>>=
Curses()
{
    initscr();
    <<initialize curses colors>>
```

After the library is initialized, then I set up the terminal according to the needs in *Monster Sweeper*. Particularly, I want to use the keyboard arrow keys.

```
<<Curses constructor>>=
    keypad(stdscr, true);
```

But I don't want neither to show the input keys nor the cursor on the screen. This cursor is a curses cursor and has nothing to do with the cursor defined in *GameView*, except for the name.

```
<<Curses constructor>>=
    noecho();
    curs_set(0); // 0 == invisible
```

Finally, I have to refresh the screen to apply all the changes.

```
<<Curses constructor>>=
    refresh();
}
```

The destructor for this class only needs to call *endwin* which releases all the memory used by the library and restores the terminal to its previous configuration. Usually, I would have to worry about any possibly created windows, but in this game all the windows are created by objects whose destructor are called before the destructor of *Curses*, hence I don't really have to worry about leaked windows.

```
<<Curses destructor>>=
~Curses()
{
    endwin();
}
```

This class is not meant to be copied around, because if I did I would end up calling *endwin* twice or more. To discourage this behaviour, the best is to declare its copy constructor and assignment operator as private and left them undefined, hence rendering the class *uncopyable*.

```
<<make the Curses class uncopyable>>=
protected:
    Curses(const Curses &);
    Curses& operator=(const Curses &);
```

7 Main

With all the building blocks that I've made in the preceding sections, the game's main function remains reasonably neat and tidy. It only has to initialize the curses library by creating a new instance of the `Curses` class, initialize the random number generator that `random_shuffle` uses, and, inside an infinite loop, call the `AskDifficulty` function and pass its result to the `Play` function.

The only other thing that the main function has to keep an eye on is the “exceptional” cases: the game has been aborted, which must continue the next loop iteration; or the player wants to quit, which should quit from the game without error. These two are implemented as exceptions because there is no reasonable return value from neither `Play` nor `AskDifficulty` to signal these situations.

As said, then, I must catch `abort_game_exception` inside the same loop where I call `Play` and `quit_exception` outside. That catch for `quit_exception` must also destroy the `Curses` object to clean up the curses library before quitting.

Of course, if there is any other exception, I print it to the screen and return with an error.

```
<<headers>>=  
#include <cstdlib>  
#include <iostream>
```

```
<<main function>>=  
int main()  
{  
    try  
    {  
        Curses curses;  
        std::srand(std::time(NULL));  
        for(;;)  
        {  
            try  
            {  
                Play(AskDifficulty());  
            }  
            catch (abort_game_exception &e)  
            {  
                continue;  
            }  
        }  
    }  
    catch (quit_exception &e)  
    {  
        return EXIT_SUCCESS;  
    }  
    catch (std::exception &e)  
    {  
        std::cerr << "Error: " << e.what() << std::endl;  
    }  
    catch (...)  
    {  
        std::cerr << "Unknown error" << std::endl;  
    }  
    return EXIT_FAILURE;  
}
```

8 monstersweeper.cpp

All the snippets defined above can be written in a single C++ module ready to be compiled.

```
<<monstersweeper.cpp>>=
/*
<<license>>
*/
<<headers>>

<<quit_exception class>>

<<abort_game_exception class>>

<<Game class>>

<<Window class>>

<<StatusView class>>

<<GameView class>>

<<function that asks for the difficulty>>

<<function that plays the game>>

<<Curses class>>

<<main function>>
```

9 Makefile

Being a simple application, an small Makefile would be sufficient to build and link *Monster Sweeper* from the source document.

The first thing that needs to be done is to extract and tangle the C++ source code from the AsciiDoc document using atangle. It is necessary, therefore, to have atangle installed.

```
<<extract cpp source code>>=
monstersweeper.cpp: monstersweeper.txt
    atangle -r $@ $< > $@
```

Then is possible to link the executable from the extracted C++ source code. But I have to take into account the platform executable suffix. For GNU/Linux and other UNIX systems, the suffix is the empty string. But for Windows, I need to append *.exe* to the executable name. Also, for GNU/Linux and other UNIX systems, the curses library is named *ncurses*, but for Windows I'll use *pdcurses*. Therefore, I need to determine on which platform I am building.

To know which system is the executable being build, I'll use the `uname -s` command, available both in GNU/Linux and also in **MinGW** or **Cygwin** for Windows. In this case, I only detect the presence of MinGW, because I don't want to add yet another dependency to Cygwin's DLL.

```
<<determine executable suffix>>=
UNAME = $(shell uname -s)
MINGW = $(findstring MINGW32, $(UNAME))
```

Later, I check if the substring *MINGW32* is contained in the output of `uname`. If the `findstring` call's result is the empty string, then we assume we are building in a platform that doesn't have a executable suffix.

```
<<determine executable suffix>>=
ifeq ($(MINGW),)
EXE :=
CURSES := -lncurses
else
EXE := .exe
```

```
CURSES := -lpdcurses -Wl,--enable-auto-import
endif
```

With the suffix setup, I can now build the final executable. Of course, I need to link also to the curses library for it to work.

```
<<build monstersweeper executable>>=
monstersweeper$(EXE): monstersweeper.cpp
    g++ -o $@ $< $(CURSES)
```

Sometimes, it is convenient to remove the executable as well as the intermediary build artifacts. For this reason, I added a target named `clean` that removes all the files built by the Makefile and only leaves the original document. I have to mark this target as `PHONY` in case there is a file named `clean` in the same directory as the Makefile.

```
<<clean build artifacts>>=
.PHONY: clean

clean:
    rm -f monstersweeper$(EXE) monstersweeper.cpp
```

As the first defined target is the Makefile's default target, I write the executable first in the Makefile and then all the dependences. After all the source code targets, it comes the `clean` target. This is not required by the Makefile, but a personal choice. The final Makefile's structure is thus the following.

```
<<Makefile>>=
<<determine executable suffix>>

<<build monstersweeper executable>>

<<extract cpp source code>>

<<clean build artifacts>>
```

10 License

This program is distributed under the terms of the GNU General Public License (GPL) version 2.0 as follows:

```
<<license>>=
Monster Sweeper - An ASCII demake of Hojamaka Games' Mamono Sweeper.
Copyright (c) Jordi Fita <jfita@geishastudios.com>

This program is free software; you can redistribute it and/or modify
it under the terms of the GNU General Public License version 2.0 as
published by the Free Software Foundation.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA
```