Project report for PA5 in C++ Programming.

Student name: Rúnar Vestmann

Pokémon Battle Simulator

A simple C++ terminal application that simulates Pokémon battles.

Proposed items that have been fulfilled:

- You can create, edit and delete a Pokémon via the terminal (10)
- Every created Pokémon gets saved in a binary file (10)
- You can select two Pokémon from a list and make them battle each other (5)
- Battle modes:
    - Player vs Player (5)
    - Player vs Computer (5)
- When two Pokémon battle they each take turns performing attack moves (5)
- Each Pokémon can have an unlimited amount of attack moves (normal rules only allow 1-4 moves) (5)
- Each Pokémon can be of 1 or more (up to 16) different types (normal rules only allow 1 or 2 types) (5)
- Type modifiers get applied during damage calculation (10)

Additional items that have been fulfilled:

- You can create, edit and delete an attack move via the terminal (10)
- Every created attack move gets saved in a binary file (10)
- Pokémon and attack moves are cached to reduce unneccessary disk reads (2.5)
- Battle mode:
    - Computer vs Computer (5)
- Pokémon speed and attack move used, influence which Pokémon gets to attack first (2.5)

Total points: 90/100

Implementation details:

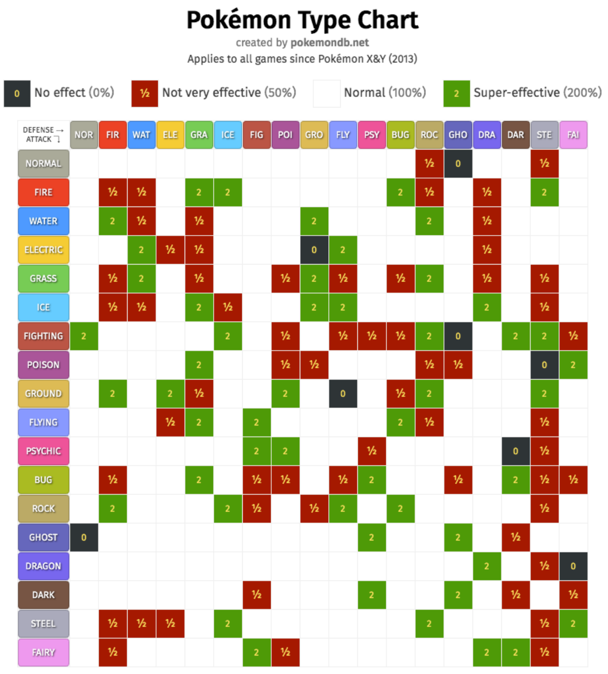
Type chart implementation

There are 18 different types (as of right now) in the Pokémon universe. A Pokémon can for example be an Electric Type (like Pikachu) or even have more than 1 type, like for example Charizard which is a Fire and Flying type Pokémon.

When Pokémon engage in a battle they have 1 or more moves they can choose to use each turn. These moves also have a type, and these different types of moves affect how damage gets calculated. For example if a Grass type Pokémon gets hit by a Fire type move, it will deal 2x damage, since Fire type moves are „super effective“ against Grass type Pokémon.

Moves can also be „not very effective“, like when a Fire type Pokémon gets hit by a Grass type move it will deal 0.5x damage. All of this can be seen in the Pokémon Type chart below.

To implement this functionality in code I simply just created that same Type chart as a 2D array of doubles, and then indexed that array to get the corresponding modifier. And if a Pokémon has many types, it‘s just a simple for loop that multiplies all the different modifiers together.



Pokémon Type Chart

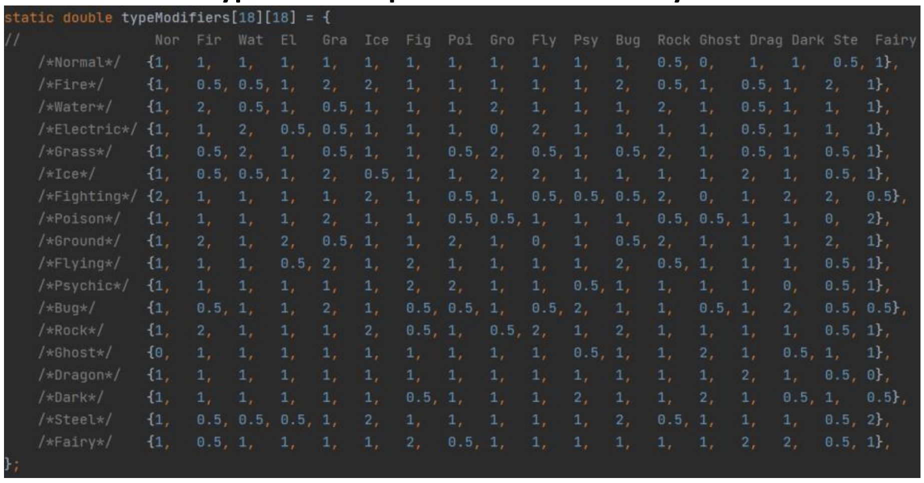
created by pokemondb.net

Applies to all games since Pokémon X&Y (2013)

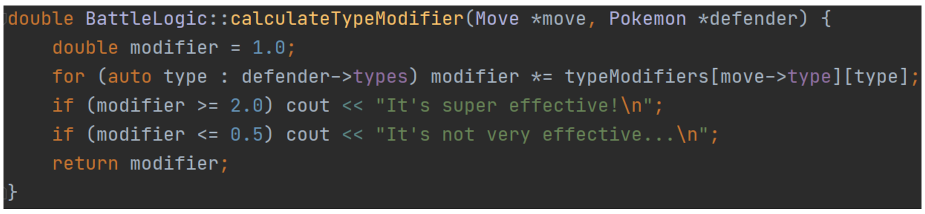
0 No effect (0%) · ½ Not very effective (50%) · Normal (100%) · 2 Super-effective (200%)

DEFENSE → ATTACK ↴	NOR	FIR	WAT	ELE	GRA	ICE	FIG	POI	GRO	FLY	PSY	BUG	ROC	GHO	DRA	DAR	STE	FAI
NORMAL													½	0			½	
FIRE		½	½		2	2						2	½		½		2	
WATER		2	½		½				2				2		½			
ELECTRIC			2	½	½				0	2					½			
GRASS		½	2		½			½	2	½		½	2		½		½	
ICE		½	½		2	½			2	2					2		½	
FIGHTING	2					2		½		½	½	½	2	0		2	2	½
POISON					2			½	½				½	½			0	2
GROUND		2		2	½			2		0		½	2				2	
FLYING				½	2		2					2	½				½	
PSYCHIC							2	2			½					0	½	
BUG		½			2		½	½		½	2			½		2	½	½
ROCK		2				2	½		½	2		2					½	
GHOST	0										2			2		½		
DRAGON															2		½	0
DARK							½				2			2		½		½
STEEL		½	½	½		2							2				½	2
FAIRY		½					2	½							2	2	½	

Pokémon Type Chart implemented as an array



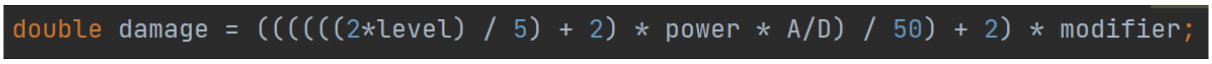
```
static double typeModifiers[18][18] = {
//              Nor  Fir  Wat  El   Gra  Ice  Fig  Poi  Gro  Fly  Psy  Bug  Rock Ghost Drag Dark Ste  Fairy
    /*Normal*/   {1,  1,   1,   1,   1,   1,   1,   1,   1,   1,   1,   1,   0.5, 0,    1,   1,   0.5, 1},
    /*Fire*/     {1,  0.5, 0.5, 1,   2,   2,   1,   1,   1,   1,   1,   2,   0.5, 1,    0.5, 1,   2,   1},
    /*Water*/    {1,  2,   0.5, 1,   0.5, 1,   1,   1,   2,   1,   1,   1,   2,   1,    0.5, 1,   1,   1},
    /*Electric*/ {1,  1,   2,   0.5, 0.5, 1,   1,   1,   0,   2,   1,   1,   1,   1,    0.5, 1,   1,   1},
    /*Grass*/    {1,  0.5, 2,   1,   0.5, 1,   1,   0.5, 2,   0.5, 1,   0.5, 2,   1,    0.5, 1,   0.5, 1},
    /*Ice*/      {1,  0.5, 0.5, 1,   2,   0.5, 1,   1,   2,   2,   1,   1,   1,   1,    2,   1,   0.5, 1},
    /*Fighting*/ {2,  1,   1,   1,   1,   2,   1,   0.5, 1,   0.5, 0.5, 0.5, 2,   0,    1,   2,   2,   0.5},
    /*Poison*/   {1,  1,   1,   1,   2,   1,   1,   0.5, 0.5, 1,   1,   1,   0.5, 0.5,  1,   1,   0,   2},
    /*Ground*/   {1,  2,   1,   2,   0.5, 1,   1,   2,   1,   0,   1,   0.5, 2,   1,    1,   1,   2,   1},
    /*Flying*/   {1,  1,   1,   0.5, 2,   1,   2,   1,   1,   1,   1,   2,   0.5, 1,    1,   1,   0.5, 1},
    /*Psychic*/  {1,  1,   1,   1,   1,   1,   2,   2,   1,   1,   0.5, 1,   1,   1,    1,   0,   0.5, 1},
    /*Bug*/      {1,  0.5, 1,   1,   2,   1,   0.5, 0.5, 1,   0.5, 2,   1,   1,   0.5,  1,   2,   0.5, 0.5},
    /*Rock*/     {1,  2,   1,   1,   1,   2,   0.5, 1,   0.5, 2,   1,   2,   1,   1,    1,   1,   0.5, 1},
    /*Ghost*/    {0,  1,   1,   1,   1,   1,   1,   1,   1,   1,   0.5, 1,   1,   2,    1,   0.5, 1,   1},
    /*Dragon*/   {1,  1,   1,   1,   1,   1,   1,   1,   1,   1,   1,   1,   1,   1,    2,   1,   0.5, 0},
    /*Dark*/     {1,  1,   1,   1,   1,   1,   0.5, 1,   1,   1,   2,   1,   1,   2,    1,   0.5, 1,   0.5},
    /*Steel*/    {1,  0.5, 0.5, 0.5, 1,   2,   1,   1,   1,   1,   1,   2,   0.5, 1,    1,   1,   0.5, 2},
    /*Fairy*/    {1,  0.5, 1,   1,   1,   1,   2,   0.5, 1,   1,   1,   1,   1,   1,    2,   2,   0.5, 1},
};
```



```
double BattleLogic::calculateTypeModifier(Move *move, Pokemon *defender) {
    double modifier = 1.0;
    for (auto type : defender->types) modifier *= typeModifiers[move->type][type];
    if (modifier >= 2.0) cout << "It's super effective!\n";
    if (modifier <= 0.5) cout << "It's not very effective...\n";
    return modifier;
}
```

Damage calculation

$$Damage = \left(\frac{\left(\frac{2 \times Level}{5} + 2\right) \times Power \times A/D}{50} + 2 \right) \times Modifier$$



```
double damage = (((((2*level) / 5) + 2) * power * A/D) / 50) + 2) * modifier;
```

3 Layered architecture

The project was split up in the following way:

- Model classes
- UI classes
- Logic classes
- Data classes

Model classes are responsible for storing data as well as being able to serialize and deserialize that data.

UI classes are responsible for displaying the text user interface and reading user input.

Logic classes are classes that connect the UI layer and Data layer together and also deal with all of the complex logic that didn't relate to the UI.

Data classes are responsible for reading and writing data to and from binary files.

Furthermore the functionality was split up into three sections, mainly, the Battle section, Pokémon section and Move section.

Computer battles

The computer can take control of a Pokémon and choose what moves it will perform during a battle. What moves it will pick is completely random.

References

Damage calculation method:

https://bulbapedia.bulbagarden.net/wiki/Damage

Correct stats for Pokémon:

http://www.psypokes.com/dex/stats.php

Different attack moves and stats:

https://serebii.net/

The Pokémon Type Chart which was used to calculate type modifiers:

https://pokemondb.net/type