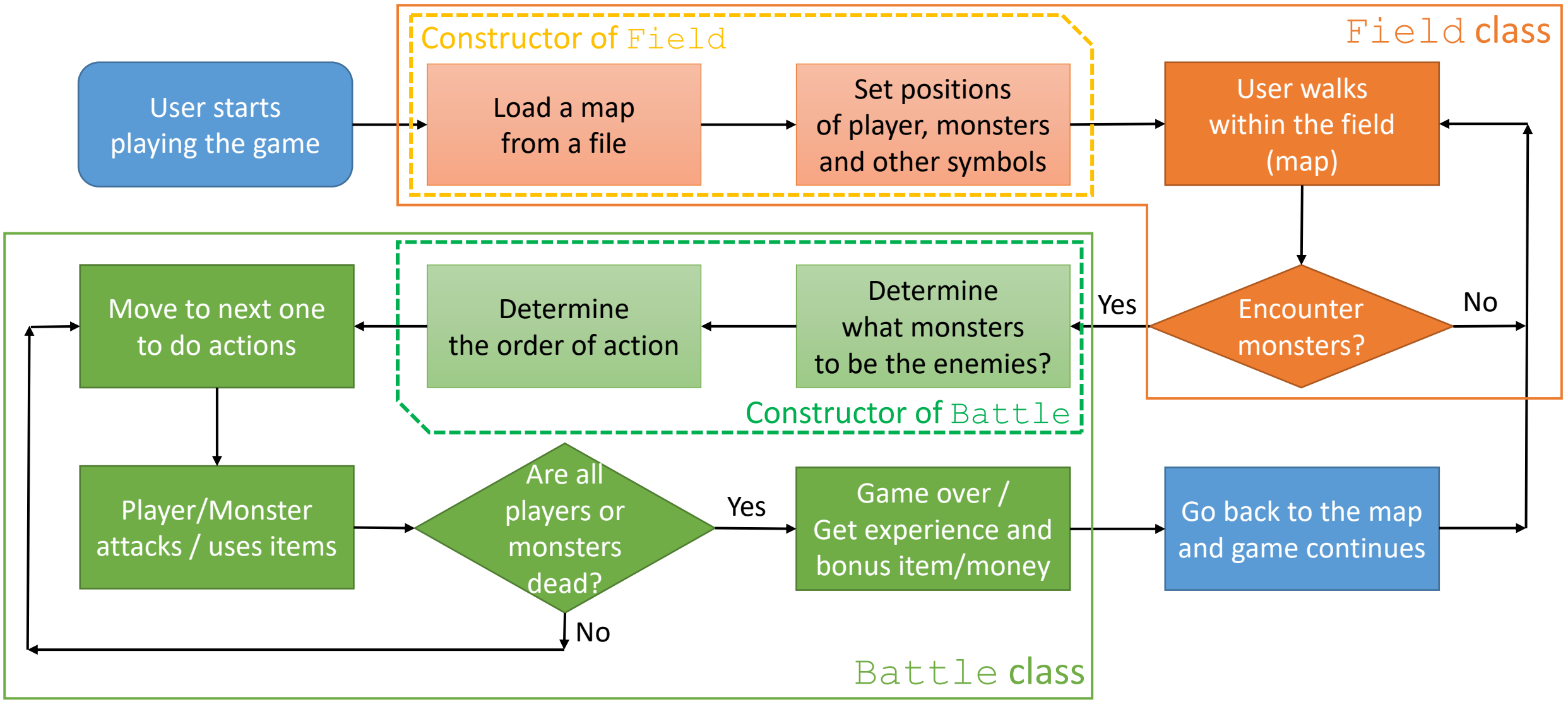
Example interface of `Battle` and `Field`

For Assignment #9 and #10

Introduction to Computers II

Simplified Game Flow



`Battle` class

`Battle` is Responsible for…

- In constructor
  - Determine which players attend this battle
    - Generally all players should attend, but you can limit to number to increase difficulty
  - Determine what monsters attend this battle as players' opponent
    - Generating by `Battle` itself
    - Generating outside, and pass them via constructor of `Battle`
      - i.e., determine by `Field` and pass them as one of the parameters of `Battle`'s constructor
  - Determine the order of action
    - Team-scale
    - Entity-scale
  - Initialize the number of turns to zero
  - Form a multi-player versus multi-monster battle

`Battle` is Responsible for…

- In general
  - Calculate the number of turns
  - Determine if it exceeds limit of number of turns
  - Move to next actor according to the order of action
  - Return the pointer of instance of current actor
  - Provide information of the battle
    - Current number of turns
    - Limit on number of turns
    - …

What you should do with `Battle`

- Calculate elapsed number of turns from a particular action
- Display players' and monsters' information
  - Name, current HP/MP, attack, defense, etc.
- Determine which team wins
  - Players or monsters are all dead
  - Turn limit exceeds
  - Boss is dead
  - …
- Give player experience and bonus items/money after they win

The Order of Action

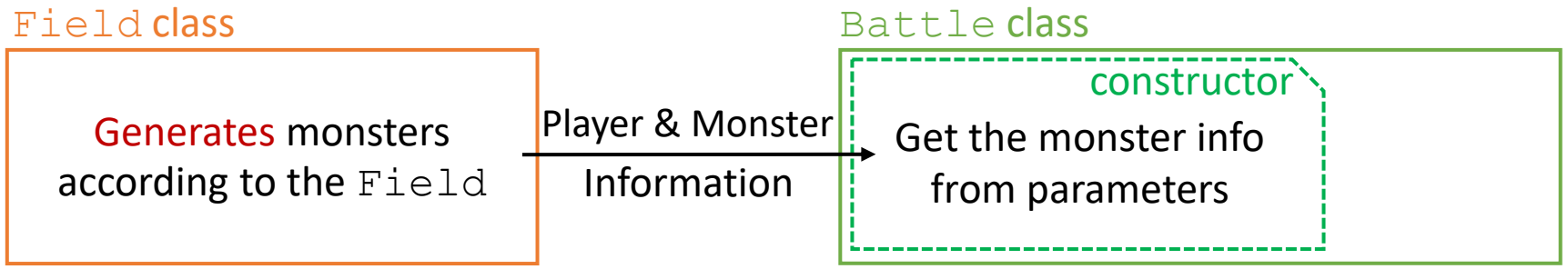
- Assuming there are 2 players versus 3 monsters
- Entity-scale
  - P1 → M1 → P2 → M2 → P1 → M3 → P2 → M1 → P1 → M2 → P2 → M3 → …

    1st Turn　2nd Turn　… nth Turn
- Team-scale
  - P1 → P2 → M1 → M2 → M3 → P1 → P2 → M1 → M2 → M3 → P1 → P2 → …

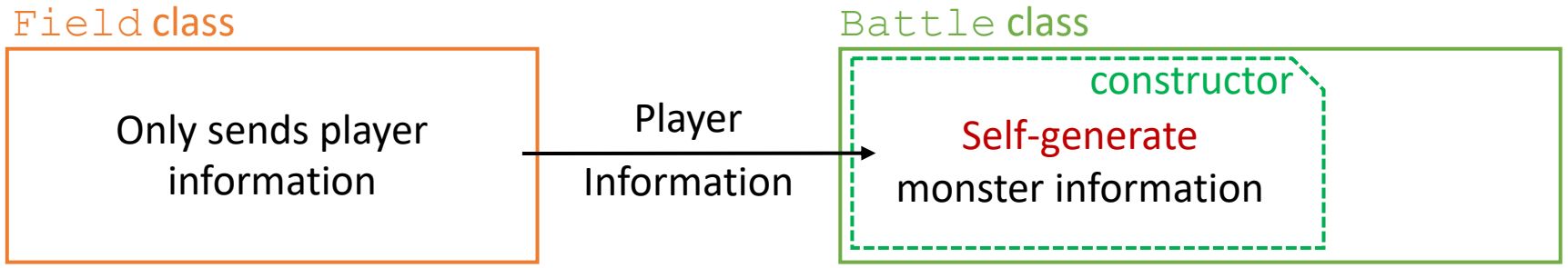
    1st Turn　2nd Turn　… nth Turn
- P: Player
- M: Monster

Generate Monster Information

- Type A



- Type B



Character Structure

```
struct Character {
    char type; // monster('m') or player('p')?
    bool alive; // alive(true) or dead(false)?
    void *instance; // pointer to instance
}
```

- We use this structure to manage characters internally
- A void pointer can point(convert) to any type of variables/instances
  - Sometimes it is called generic pointer as well
- To convert from/to void pointer, please use static_cast function

Data Members of `Battle`

- All members are `private`
- `int n_turn;`
  - The number of turn, should be initialized to 0 in constructor
- `int turn_limit;`
  - Maximum number of turn, 0 for no limit
- `Character* action_list;`
  - An ordered action list, indicates the order of action

Constructors of `Battle` (Type A)

- `Battle(NovicePlayer**, BaseMonster**, int, int, int);`
  - First parameter indicates the array/list of players to attend
  - The second one is for monsters
  - The third and forth parameter indicates the number of players and monsters
  - The fifth parameter represents the limit on number of turns
- `Battle(NovicePlayer**, BaseMonster**, int, int);`
  - The fifth parameter is omitted, set `turn_limit` to 0
  - This means this battle has no limit on number of turns

Constructors of `Battle` (Type B)

- `Battle(NovicePlayer**, int);`
- `Battle(NovicePlayer**, int, int);`
  - Similar to Type A, but the monster information is omitted since the monster information of Type B is generated by `Battle` itself
  - The second parameter is the number of players
  - The third parameter is the limit on number of turns; if this is omitted, set `turn_limit` to 0
  - So, constructors of Type B are responsible to generate monster information

Public Methods of `Battle`

- `bool nextActor(void);`
  - Move to next actor, if all character were done, `n_turn++`
  - The return value indicates whether it exceeds limit on number of turns
- `int getTurnCount(void) const;`
  - Get the current number of turn
- `int getTurnLimit(void) const;`
  - Get the limit on number of turns, 0 for no limit

Public Methods of `Battle`

- `int getPlayerCount(void) const;`
- `int getPlayerCount(bool) const;`
  - Get the current number of players
  - If the second parameter set to true, only alive players will be counted
  - If the second parameter is omitted, return count for all players
- `int getMonsterCount(void) const;`
- `int getMonsterCount(bool) const;`
  - Similar to above ones, but this set of methods return the information of monsters

Public Methods of `Battle`

- `Character getCurrentActor(void);`
  - Get the current actor within the action list
  - Note that the type of return value is struct Character
- `Character* getPlayers(void);`
  - Get a full list of players, no matter dead of alive ones
- `Character* getMonsters(void);`
  - Get a full list of monsters, no matter dead of alive ones

`Field` class

`Field` is Responsible for…

- In constructor
  - Load a map from a file
    - You can self-define the format or refer to Assignment #3
  - Initialize positions of all items/symbols on a map
    - You can randomly place some items (such as treasure chests)
    - This step may skip if the file already included the information
  - Set the player's position

`Field` is Responsible for…

- In general
  - Let player moves (i.e., change the position of player)
  - Change player and other item/place's position
  - Let user know the current position and which map he/she is in
    - `"You are now at (104, 252) of Summoner's Rift"`
  - Display the map to user
    - But most of the maps are too large to display, so we only display part
    - Generally, the player will on the center of a window

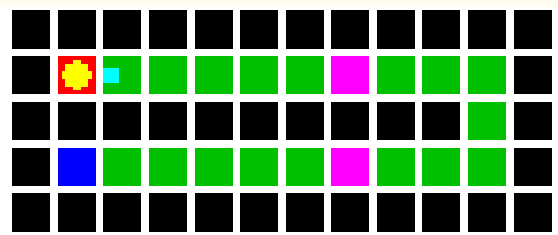
What should you do with `Field`

- Determine if the player encounters monsters
  - Randomly encounter
  - By dialogue or entering a particular position
- Trigger an event
  - Entering shop
  - A dialogue
  - A battle
  - Way point
  - Save point
  - …

A Simple Format of Map

- Symbols
  - Walls as 1
  - Pavements as 0
  - A starting point 200
  - A destination 201
  - Bonus points 202+
- Format
  - # of columns, # of rows
  - (data)

- Example



```
12,5
1,1,1,1,1,1,1,1,1,1,1,1
1,200,0,0,0,0,0,202,0,0,0,1
1,1,1,1,1,1,1,1,1,1,0,1
1,201,0,0,0,0,0,203,0,0,0,1
1,1,1,1,1,1,1,1,1,1,1,1
```

Using `enum` to Increase Readability

- Before

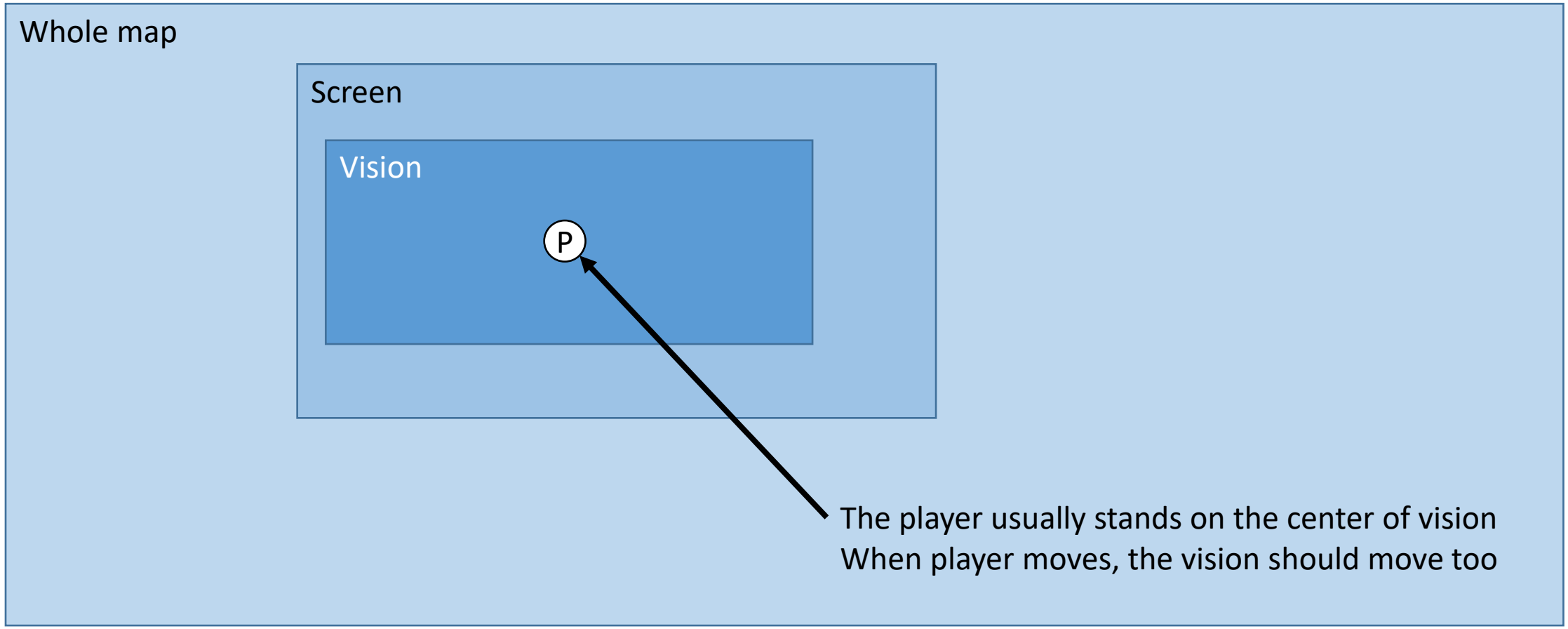
```
…
switch(next_position) {
   case 1:
      // do something
      break;
   case 202:
      // do something
      break;
   …
}
…
```

- After

```
enum{WALL=1, BOUNS=202};
…
switch(next_position) {
   case WALL:
      // do something
      break;
   case BONUS:
      // do something
      break;
   …
}
…
```

Vision

- Most of the maps are too large to display, so we only display part



Data Members of `Field`

- Basic members, all members are `private`
  - `int** map_data;`
    - The actual map data with symbols and items
    - This is a double dimension array of integers
  - `int current_position_x;`
  - `int current_position_y;`
    - Current position of player
    - This can be used to display the map (player is always on the center)
  - `string map_name;`
    - The name of this map

Data Members of `Field`

- Basic members, all members are `private`
  - `int vision_width;`
  - `int vision_height;`
    - The width and height of vision

- Please add more data members you want!
  - Type of monsters
  - Probability of encountering monsters
  - …

Constructors of `Field`

- `Field(int**, int, int, int, int);`
  - The first parameter is the map data
  - The second and third parameter are the current place (x,y) of player
  - The forth and fifth parameter are the (width, height) of vision
- `Field(const char*, int, int, int, int);`
  - Similar to first one, but the first parameter is name of file that stores map data

Basic Public Methods of `Field`

- `bool move(char);`
  - Move player to next position, the parameter is the direction
  - The return value indicates whether this move is legal or not
- `bool moveUp(void);`
- `bool moveDown(void);`
- `bool moveLeft(void);`
- `bool moveRight(void);`
  - The same as `move(char)`, move player to next position
  - But the direction is determined

Basic Public Methods of `Field`

- These methods get the information of the map
  - `int getCurrentPositionX(void) const;`
  - `int getCurrentPositionY(void) const;`
  - `int getVisionWidth(void) const;`
  - `int getVisionHeight(void) const;`
  - `string getMapName(void) const;`
  - `int getMapSymbol(int, int);`
    - Parameter is the position (x,y)

Basic Public Methods of `Field`

- `void setPosition(int, int);`
  - Set the position of player, prameters are position (x,y)
- `void setMapSymbol(int, int, int);`
  - Set the symbol on a specific position
  - The first parameter is symbol, the rest are position (x,y)
- `void setVisionSize(int, int);`
  - Set the size of vision, parameters are (width, height)
- `void display(void) const;`
  - Displaying the map

- Please add more methods according to your own features!

Notice

- If you did `news` within constructors, please also remember to do `deletes` within destructors to prevent memory leak
- You can change the internal data representation as you want
  - E.g., pointer-based to STL-based
- You can add more data members and methods according to your needs as well