**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра МО ЭВМ**

отчет

**по лабораторной работе №3**

**по дисциплине «ООП»**

Тема: Полиморфизм

|  |  |  |
| --- | --- | --- |
| Студент гр. 2383 |  | Лустенкова Д,Д. |
| Преподаватель |  | Жангиров Т.Р. |

Санкт-Петербург

2024

## Цель работы

Получить опыт работы с интерфейсами на языке программирования С++, реализовав интерфейс игрового события, а в классе управления игроком добавить проверку на наличие события на клетке, что является важным шагом в реализации проекта первой игры.

## Задание

а) Создать интерфейс игрового события. Интерфейс должен обеспечивать срабатывание события когда игрок наступает на клетку.

б) Реализовать интерфейс игрового события тремя конкретными событиями. Одно событие должно положительно влиять на характеристики игрока, второе должно негативно влиять на характеристики игрока, третье изменять координаты игрока на поле. При желании можно реализовать больше событий и/или события меняющие само поле (например, делать из непроходимой клетки проходимую).

в) В классе управления игроком добавить проверку на наличие события на клетке, если событие присутствует, то оно должно сработать. Срабатывание должно происходить через интерфейс события, и не должно быть никаких проверок на тип события (реализация через динамический полиморфизм)

г) Создать класс создающий поле. Предусмотреть возможность создания 2 разных уровней. По желанию можно сделать случайную генерацию уровней. Должно гарантироваться, что игрок может дойти от входа до выхода.

Примечания:

* События должны быть такими, чтобы был сценарий проигрыша игрока.
* В событиях и клетках не должно быть полей сообщающих информацию о типе события

## Выполнение работы

Для реализации интерфейса игрового события создан абстрактый класс event и 3 класса наследника, отвечающие за конкретные события.

class event

Абстрактный класс (поля отсутствуют, все методы виртуальные), реализующий интерфейс.

Методы класса *event*:

* *virtual void do\_event(controller& c) = 0* – целевой вирутуальный метод, отвечающий за реализацию события;
* *virtual event\* copy() = 0* – метод, позволяющий копировать объект данного абстрактного класса по указателю (разыменовывание указателя для абстрактного класса невозможно);
* *virtual ~event() = default* – виртуальный деструктор, задан деструткором по умолчанию, создан для того, чтобы в случае, сели у одного из классов-наследников будет поле-указатель, корректно уничтожить объект класса правильной очисткой динамической памяти;

class Healing : public event

Наследник класса *event*, часть интерфейса игрового события, отвечающая за событие, добавляющее жизни игроку в размере *HEALING* (макроопределение, заданное в заголовочном файле значением 300).

Методы класса *Healing*:

* *virtual void do\_event(controller& c)* – использует в своей реализации метод *add\_lives\_* контроллера игрока *c*;
* *virtual event\* copy()* – возвращает *new Heal\_point*.

class Damage : public event

Наследник класса *event*, часть интерфейса игрового события, отвечающая за событие, отнимающее жизни у игрока в размере *DAMAGE* (макроопределение, заданное в заголовочном файле значением -300).

Методы класса *Damage*:

* *virtual void do\_event(controller& c)* – использует в своей реализации метод *add\_lives\_* контроллера игрока *c*;
* *virtual event\* copy()* – возвращает *new Damage*.

class Rift : public event

Наследник класса *event*, часть интерфейса игрового события, отвечающая за событие, телепортирующее игрока в рандомную точку на поле.

Методы класса *Rift*:

* *virtual void do\_event(controller& c)* – использует в своей реализации функцию *rand()* и методы *get\_height\_*, *get\_width\_*, touch\_cell и *set\_cords\_* контроллера игрока *c*;
* *virtual event\* copy()* – возвращает *new Rift*.

В ранее написанные классы внесены все необходимые измененения.

Изменения в class cell

Теперь, как когда-то говорилось в условии второй лабораторной работы, в классе клетки храниться поле *event\* event\_* – указатель на событие, которое должно происходить, если игрок наступает на эту клетку.

Изменён конструктор класса: в него добавлен входной аргумент *event\* event*, по умолчанию равный *nullptr*, что значит, что события в данной клетке нет, которому присваивается значение поля *event\_*.

Теперь, когда в клетке лежит указатель, потребовалось переопределить *operator =* для копирования и написать деструктор.

Новые методы класса *cell*:

// copy

* *void copy(const cell& other)* – метод копирования клетки, написан в полной аналогии с методом копирования поля, написанном в предыдущей лаборатороной работе: если указатель на текущий объект класса не равен указателю на объект класса, который требуется скопировать, происходит копирование полей *type\_*, *tp\_cords\_* и копирование поля *event\_* с помощью метода *event\_->copy()*;
* *cell(const cell& other)* – конструктор копирования, использует метод копирования;
* *cell& operator = (const cell& obj)* – переопределение оператора присваивания в случае копирования, использует метод копирования;

*// work with events*

* *void set\_event\_(event\* event)* – метод, меняющий событие в клетке на заданное;
* *void do\_event\_(controller& c)* – метод, вызывающий *event\_→do\_event* от принятого им контроллера, если событие не равно *nullptr*;

*// Destructor*

* *void remove\_event\_()* – метод, очизающий память по указателю event\_ и присвающий ему *nullptr*, используется в деструкторе и в методе *set\_event\_*.
* *~cell()* – деструтор класса, вызывает метод *remove\_event\_*.

К новым методам клетки реализован доступ через соответствующие им новые методы поля.

Изменения в class field

Новые методы класса *field*:

*// work with events*

* *void set\_event\_of\_cell(int x, int y, event\* event)* – доступ к методу *set\_event\_*(*event\* event*) клетки по координатам *x*, *y*;
* *void do\_event\_of\_cell(int x, int y, controller& c)* – доступ к методу *do\_event\_*(*controller& c)* клетки по координатам *x*, *y*;
* *void remove\_event\_of\_cell(int x, int y)* – доступ к методу *remove\_event\_*(*)* клетки по координатам *x*, *y*;

Изменения в class controller

В метод *void controller::move(Direction move\_direction)* добавлены 2 строчки кода, а именно выполнение события клетки, на которую было совершено перемещение, и очистка данного события после его исполнения.

Написан новый класс, создающий поле.

class field\_creator

Класс хранит в себе единственное поле *field field\_* с единственной целью сделать из пустого поля уровень и передать его выше. Класс написан по причине того, что вся лабораторная работа выполняется строго в рамках ООП и написание функций, за исключением функции *main()* строго запрещено.

Методы класса *field\_creator*:

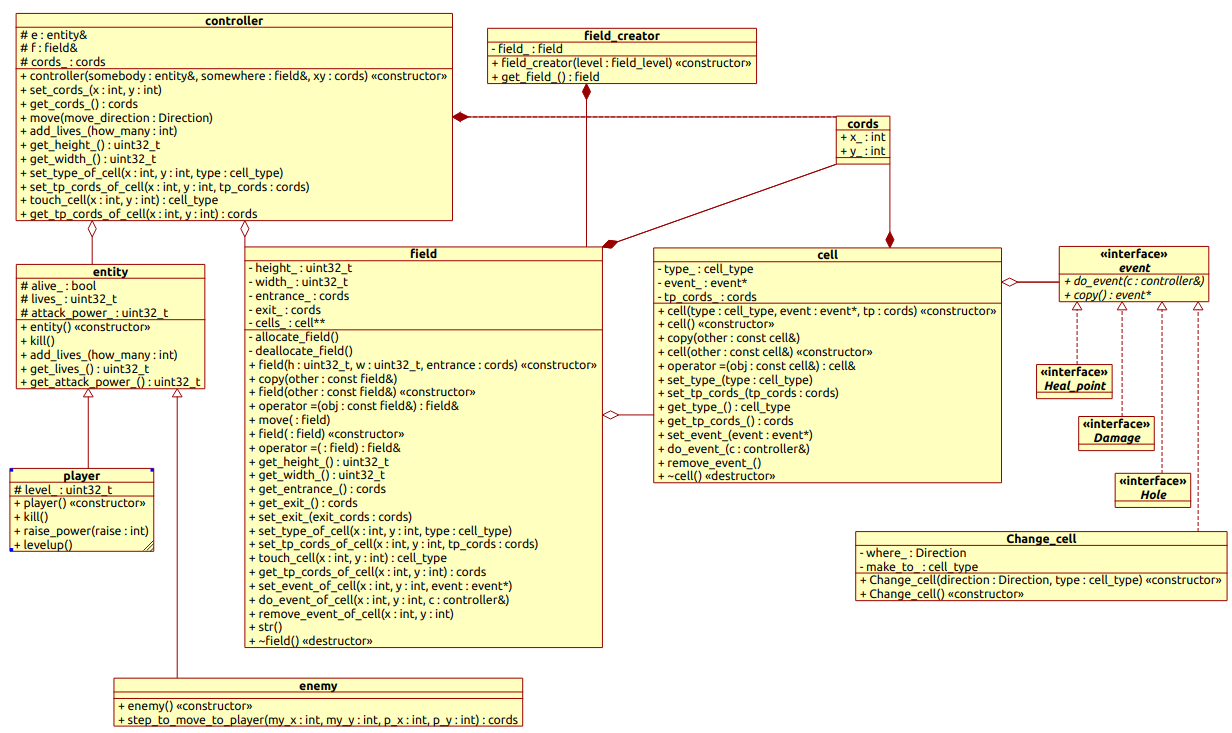
* *field\_creator(field\_level level)* – конструктор класса, инициализирует поле *field\_* полем по умолчанию и меняет типы клеток, а так же добавляет события конкретным клеткам, в зависимости от выбранного уровня (*enum field\_level {main\_level, end\_level, secret\_level};*);
* *field get\_field\_();* – возвращает *field\_*.

int main()

В функции *main()* на данный момент реализации проекта находятся тестирование класса *field\_creator* и тестирование прохождения клеток каждого типа и прохождения всех клеток с событиями.

Программный код приведен в приложении А.

## UML-диаграмма классов



## Тестирование

Результаты тестирования представлены в табл. 1.

Таблица 1 – Результаты тестирования.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| № п/п | Входные данные | Выходные данные | | Комментарии |
|  | field\_creator creator(main\_level);  field f1 = creator.get\_field\_();  f1.str();  std::cout << '\n'; | 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2  2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2  2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2  2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2  2 0 0 0 3 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2  2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2  2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2  2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2  2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 2  2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 2  2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 0 2  2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 2  2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 2  2 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 2  2 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 2  2 0 0 0 0 0 0 0 0 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 2  2 0 0 0 0 0 0 0 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 2  2 0 0 0 0 0 0 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2  2 0 0 0 0 0 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2  2 0 0 0 0 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2  2 0 0 0 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 3 0 0 0 2  2 0 0 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2  2 0 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2  2 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2  2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 | | field\_creator корректно считывает типы клеток поле из текстового файла. |
|  | player p1;  p1.add\_lives\_(1);  controller c\_p1(p1, f1, f1.get\_entrance\_());  // Spikes  std::cout << "check Spikes:" << '\n';  c\_p1.set\_cords\_(11, 16);  std::cout << "p1: " << c\_p1.get\_cords\_().x\_ << ' ' << c\_p1.get\_cords\_().y\_ << '\n';  std::cout << "lives\_ = " << p1.get\_lives\_() << '\n';  c\_p1.move(Bottom);  std::cout << "p1: " << c\_p1.get\_cords\_().x\_ << ' ' << c\_p1.get\_cords\_().y\_ << '\n';  std::cout << "lives\_ = " << p1.get\_lives\_() << '\n';  // Healing  std::cout << "check Healing:" << '\n';  c\_p1.move(Bottom);  std::cout << "p1: " << c\_p1.get\_cords\_().x\_ << ' ' << c\_p1.get\_cords\_().y\_ << '\n';  std::cout << "lives\_ = " << p1.get\_lives\_() << '\n';  // Damage  std::cout << "check Damage:" << '\n';  c\_p1.set\_cords\_(18, 9);  std::cout << "p1: " << c\_p1.get\_cords\_().x\_ << ' ' << c\_p1.get\_cords\_().y\_ << '\n';  std::cout << "lives\_ = " << p1.get\_lives\_() << '\n';  c\_p1.move(Right);  std::cout << "p1: " << c\_p1.get\_cords\_().x\_ << ' ' << c\_p1.get\_cords\_().y\_ << '\n';  std::cout << "lives\_ = " << p1.get\_lives\_() << '\n';  // Teleport  std::cout << "check Teleport:" << '\n';  c\_p1.set\_cords\_(4, 3);  std::cout << "p1: " << c\_p1.get\_cords\_().x\_ << ' ' << c\_p1.get\_cords\_().y\_ << '\n';  c\_p1.move(Top);  std::cout << "p1: " << c\_p1.get\_cords\_().x\_ << ' ' << c\_p1.get\_cords\_().y\_ << '\n';  // Rift  std::cout << "check Rift:" << '\n';  c\_p1.set\_cords\_(8, 12);  std::cout << "p1: " << c\_p1.get\_cords\_().x\_ << ' ' << c\_p1.get\_cords\_().y\_ << '\n';  c\_p1.move(Bottom);  std::cout << "p1: " << c\_p1.get\_cords\_().x\_ << ' ' << c\_p1.get\_cords\_().y\_ << '\n';  return 0; | | check Spikes:  p1: 11 16  lives\_ = 101  p1: 11 15  lives\_ = 101  check Healing:  p1: 11 14  lives\_ = 101  check Damage:  p1: 18 9  lives\_ = 101  p1: 19 9  lives\_ = 101  check Teleport:  p1: 4 3  p1: 4 4  check Rift:  p1: 8 12  p1: 8 11 | От прохождения клеток каждого типа и от прохождения всех клеток с событиями получен требуемый результат (положительное или отрицательное изменение жизней игрока и изменение его координат в пространстве поля). |

## Выводы

Реализован интерфейс игрового события, в класс клетки добавлен указатель на игровое событие, все необходимые в связи с этим изменения внесены в проект. Реализован класс field\_creator, создающий поле с заданными типами клеток и событиями.

Сделан третий шаг в реализации проекта игры на языке программирования С++.

# **Приложение А Исходный код программы**

Название файла: event.h

#pragma once

#define HEALING 200

#define DAMAGE -200

class controller;

class event

{

public:

virtual void do\_event(controller & c) = 0;

virtual event\* copy() = 0;

virtual ~event() = default;

};

Название файла: Healing.h

#pragma once

#include "event.h"

// A positive event for the entity

class Healing : public event

{ // Concentration of continuum energy (allows entity to heal)

public:

void do\_event(controller& c) final;

event\* copy() final;

};

Название файла: Healing.cpp

#include "Healing.h"  
#include "../controller.h"  
  
// A positive event for the entity  
  
void Healing::do\_event(controller& c)  
{  
 c.add\_lives\_(HEALING);  
}  
  
event\* Healing::copy()  
{  
 return new Healing;  
}

Название файла: Damage.h

#pragma once

#include "event.h"

// A negative event for the entity

class Damage : public event

{ // Continuum shift of energies

public:

void do\_event(controller& c) final;

event\* copy() final;

};

Название файла: Damage.cpp

#include "Damage.h"  
#include "../controller.h"  
  
// A negative event for the entity  
  
void Damage::do\_event(controller& c)  
{  
 c.add\_lives\_(DAMAGE);  
}  
  
event\* Damage::copy()  
{  
 return new Damage;  
}

Название файла: Rift.h

#pragma once

#include "event.h"

// Moving around the map

class Rift : public event

{ // A rift in the continuum teleporting to an indeterminate point

public:

void do\_event(controller& c) final;

event\* copy() final;

};

Название файла: Rift.cpp

#include "Rift.h"  
#include "../controller.h"  
  
#define DELTA 15  
  
// Moving around the map  
  
void Rift::do\_event(controller& c)  
{  
 uint32\_t start\_x = rand() % (c.get\_width\_() - DELTA) + 1;  
 uint32\_t start\_y = rand() % (c.get\_height\_() - DELTA) + 1;  
 bool br\_check = false;  
 for (int y = start\_y; y < c.get\_height\_() - 1; y++)  
 {  
 for (int x = start\_x; x < c.get\_width\_() - 1; x++)  
 {  
 if (c.touch\_cell(x, y) != Barrier && c.touch\_cell(x, y) != Teleport)  
 {  
 //\*this = new event;  
 c.set\_cords\_(x, y);  
 br\_check = true;  
 break;  
 }  
 }  
 if (br\_check) { break; }  
 }  
}  
  
event\* Rift::copy()  
{  
 return new Rift;  
}

Название файла: cell\_type.h

#pragma once

#ifndef CELL\_TYPE\_H

#define CELL\_TYPE\_H

enum Direction { Top, Bottom, Right, Left };

enum cell\_type { Floor, Spikes, Barrier, Teleport, Exit\_door };

#endif

Название файла: cords.h

#include <iostream>

struct cords

{

int x\_;

int y\_;

};

Название файла: cell.h

#pragma once

#include "cords.h"

#include "../events/event.h"

#include "../events/Healing.h"

#include "../events/Damage.h"

#include "../events/Rift.h"

#include "cell\_type.h"

// a cell is a unit of a field

class cell

{

private:

cell\_type type\_;

cords tp\_cords\_;

event\* event\_;

protected:

public:

cell(cell\_type type, event\* event = nullptr, cords tp = { 1, 1 });

cell();

// copy

void copy(const cell& other);

cell(const cell& other);

cell& operator = (const cell& obj);

// getters and setters

void set\_type\_(cell\_type type);

cell\_type get\_type\_();

void set\_tp\_cords\_(cords tp\_cords);

cords get\_tp\_cords\_();

// work with events

void set\_event\_(event\* event);

void do\_event\_(controller& c);

// Destructor

void remove\_event\_();

~cell();

};

Название файла: cell.cpp

#include "cell.h"  
  
cell::cell(cell\_type type, event\* event, cords tp)  
 : type\_(type), event\_(event), tp\_cords\_({ -1, -1 })  
{  
 if (type\_ == Teleport)  
 tp\_cords\_ = tp;  
}  
  
cell::cell() : cell(Floor) {}  
  
// copying  
  
void cell::copy(const cell& other)  
{  
 if (this != &other)  
 {  
 type\_ = other.type\_;  
 tp\_cords\_ = other.tp\_cords\_;  
 // copy \*event  
 if (other.event\_ == nullptr)  
 event\_ = nullptr;  
 else  
 event\_ = other.event\_->copy();  
 }  
}  
  
cell::cell(const cell& other)  
{  
 copy(other);  
}  
  
cell& cell::operator = (const cell& obj)  
{  
 copy(obj);  
 return \*this;  
}  
  
// getters and setters  
  
void cell::set\_type\_(cell\_type type)  
{  
 type\_ = type;  
}  
  
cell\_type cell::get\_type\_()  
{  
 return type\_;  
}  
  
void cell::set\_tp\_cords\_(cords tp\_cords)  
{  
 tp\_cords\_ = tp\_cords;  
}  
  
cords cell::get\_tp\_cords\_()  
{  
 if (type\_ == Teleport)  
 return tp\_cords\_;  
}  
  
  
// events  
  
void cell::set\_event\_(event\* event)  
{  
 remove\_event\_();  
 event\_ = event;  
}  
  
void cell::do\_event\_(controller& c)  
{  
 if (event\_ != nullptr)  
 {  
 event\_->do\_event(c);  
 }  
}  
  
// Destructor  
  
void cell::remove\_event\_()  
{  
 delete event\_;  
 event\_ = nullptr;  
}  
  
cell::~cell()  
{  
 remove\_event\_();  
}

Название файла: field.h

#pragma once

#include "cell.h"

#include <cstdint>

class field

{

private:

// memory operations

void allocate\_field();

void deallocate\_field();

//

uint32\_t height\_;

uint32\_t width\_;

cords entrance\_;

cords exit\_;

cell\*\* cells\_;

protected:

public:

field(uint32\_t h = 25, uint32\_t w = 25, cords entrance = { 12, 1 });

//copying

void copy(const field& other);

field(const field& other);

field& operator = (const field& obj);

// move

void move(field&& other);

field(field&& other);

field& operator = (field&& obj);

// getters and setters

uint32\_t get\_height\_();

uint32\_t get\_width\_();

cords get\_entrance\_();

cords get\_exit\_();

void set\_exit\_(cords exit\_cords);

void set\_type\_of\_cell(int x, int y, cell\_type type);

cell\_type touch\_cell(int x, int y);

cords get\_tp\_cords\_of\_cell(int x, int y);

void set\_tp\_cords\_of\_cell(int x, int y, cords tp\_cords);

// events

void set\_event\_of\_cell(int x, int y, event\* event);

void do\_event\_of\_cell(int x, int y, controller& c);

void remove\_event\_of\_cell(int x, int y);

// std::cout

void str();

~field();

};

Название файла: field.cpp

#include "field.h"  
  
// memory operations  
  
void field::allocate\_field()  
{  
 cells\_ = new cell \* [height\_];  
 for (int y = 0; y < height\_; y++)  
 {  
 cells\_[y] = new cell[width\_];  
 }  
}  
  
void field::deallocate\_field()  
{  
 if (cells\_ != nullptr)  
 {  
 for (int y = 0; y < height\_; y++)  
 {  
 for (int x = 0; x < width\_; x++)  
 {  
 cells\_[y][x].remove\_event\_();  
 }  
 delete[] cells\_[y];  
 }  
 delete[] cells\_;  
 }  
 cells\_ = nullptr;  
}  
  
//  
  
field::field(uint32\_t h, uint32\_t w, cords entrance)  
 : height\_(h), width\_(w), entrance\_(entrance), exit\_({ -1, -1 })  
{  
 // Exception  
 if (h < 5 || w < 5)  
 {  
 std::cerr << "The size of the field is not correct" << '\n';  
 }  
 // Normal work  
 allocate\_field();  
 for (int y = 0; y < height\_; y++)  
 {  
 for (int x = 0; x < width\_; x++)  
 {  
 if (x == 0 || y == 0 || x == width\_ - 1 || y == height\_ - 1)  
 cells\_[y][x].set\_type\_(Barrier);  
 }  
 }  
}  
  
//copying  
  
void field::copy(const field& other)  
{  
 if (this != &other)  
 {  
 height\_ = other.height\_;  
 width\_ = other.width\_;  
 entrance\_ = other.entrance\_;  
 exit\_ = other.exit\_;  
  
 allocate\_field();  
  
 for (int y = 0; y < height\_; y++)  
 {  
 for (int x = 0; x < width\_; x++)  
 cells\_[y][x] = other.cells\_[y][x];  
 }  
 }  
}  
  
field::field(const field& other)  
{  
 copy(other);  
}  
  
field& field::operator = (const field& obj)  
{  
 copy(obj);  
 return \*this;  
}  
  
//  
  
// move  
  
void field::move(field&& other)  
{  
 if (this != &other)  
 {  
 height\_ = other.height\_;  
 width\_ = other.width\_;  
 entrance\_ = other.entrance\_;  
 exit\_ = other.exit\_;  
 std::swap(cells\_, other.cells\_);  
 other.cells\_ = nullptr;  
 }  
}  
  
field::field(field&& other)  
{  
 move(std::move(other));  
}  
  
field& field::operator = (field&& obj)  
{  
 move(std::move(obj));  
 return \*this;  
}  
  
//  
  
// getters and setters  
  
uint32\_t field::get\_height\_()  
{  
 return height\_;  
}  
  
uint32\_t field::get\_width\_()  
{  
 return width\_;  
}  
  
cords field::get\_entrance\_()  
{  
 return entrance\_;  
}  
  
cords field::get\_exit\_()  
{  
 return exit\_;  
}  
  
void field::set\_exit\_(cords exit\_cords)  
{  
 exit\_ = exit\_cords;  
 cells\_[exit\_cords.y\_][exit\_cords.x\_].set\_type\_(Exit\_door);  
}  
  
void field::set\_type\_of\_cell(int x, int y, cell\_type type)  
{  
 cells\_[y][x].set\_type\_(type);  
}  
  
cell\_type field::touch\_cell(int x, int y)   
{  
 return cells\_[y][x].get\_type\_();  
}  
  
cords field::get\_tp\_cords\_of\_cell(int x, int y)  
{  
 return cells\_[y][x].get\_tp\_cords\_();  
}  
  
void field::set\_tp\_cords\_of\_cell(int x, int y, cords tp\_cords) {  
 cells\_[y][x].set\_tp\_cords\_(tp\_cords);  
}  
  
//  
  
// events  
  
void field::set\_event\_of\_cell(int x, int y, event\* event) {  
 cells\_[y][x].set\_event\_(event);  
}  
  
void field::do\_event\_of\_cell(int x, int y, controller& c) {  
 cells\_[y][x].do\_event\_(c);  
}  
  
void field::remove\_event\_of\_cell(int x, int y) {  
 cells\_[y][x].remove\_event\_();  
}  
  
// std::cout  
  
void field::str()  
{  
 for (int y = 0; y < height\_; y++)  
 {  
 for (int x = 0; x < width\_; x++)  
 {  
 std::cout << cells\_[y][x].get\_type\_() << ' ';  
 }  
 std::cout << '\n';  
 }  
}  
//  
  
// Destructor  
  
field::~field()  
{  
 deallocate\_field();  
}

Название файла: field\_creator.h

#include "field.h"

enum field\_level { main\_level, end\_level, secret\_level };

class field\_creator

{

private:

field field\_;

protected:

public:

field\_creator(field\_level level);

field get\_field\_();

};

Название файла: field\_creator.cpp

#include "field\_creator.h"  
  
#include <fstream>  
  
field\_creator::field\_creator(field\_level level) : field\_()  
{  
 std::string level\_id;  
 if (level == main\_level)  
 {  
 level\_id = "main";  
 }  
 if (level == end\_level)  
 {  
 level\_id = "end";  
 }  
 if (level == secret\_level)  
 {  
 level\_id = "secret";  
 field\_ = field(5, 7, { 2, 1 });  
 }  
 std::string file\_name = "../field/levels/" + level\_id + "\_level.txt";  
 std::ifstream file(file\_name);  
 if (!file.is\_open())  
 {  
 std::cerr << "field\_creator(field\_level level): !file.is\_open()" << '\n';  
 return;  
 }  
 int current\_id;  
 for (int y = 0; y < field\_.get\_height\_(); y++)  
 {  
 for (int x = 0; x < field\_.get\_width\_(); x++)  
 {  
 file >> current\_id;  
 // -3, -2, -1 is Floor with event  
 if (current\_id == -3)  
 field\_.set\_event\_of\_cell(x, y, new Rift);  
 if (current\_id == -2)  
 field\_.set\_event\_of\_cell(x, y, new Damage);  
 if (current\_id == -1)  
 field\_.set\_event\_of\_cell(x, y, new Healing);  
 // 0 is just Floor  
 if (current\_id == 1)  
 field\_.set\_type\_of\_cell(x, y, Spikes);  
 if (current\_id == 2)  
 field\_.set\_type\_of\_cell(x, y, Barrier);  
 if (current\_id == 3)  
 field\_.set\_type\_of\_cell(x, y, Teleport);  
 if (current\_id == 4)  
 field\_.set\_type\_of\_cell(x, y, Exit\_door);  
 }  
 }  
 if (level == main\_level)  
 {  
 field\_.set\_tp\_cords\_of\_cell(4, 4, { 20, 20 });  
 field\_.set\_tp\_cords\_of\_cell(20, 20, { 4, 4 });  
 }  
 if (level == end\_level)  
 {  
 field\_.set\_tp\_cords\_of\_cell(6, 6, { 6, 18 });  
 field\_.set\_tp\_cords\_of\_cell(6, 18, { 6, 6 });  
 }  
}  
  
field field\_creator::get\_field\_()  
{  
 return field\_;  
}

Название файла: controller.h

#include "entities/player.h"

#include "entities/enemy.h"

class controller

{

private:

protected:

entity& e;

field& f;

cords cords\_;

public:

controller(entity& somebody, field& somewhere, cords xy);

cords get\_cords\_();

void set\_cords\_(int x, int y);

void move(Direction move\_direction);

// the entity methods needed above

void add\_lives\_(int how\_many);

// the field methods needed above

uint32\_t get\_height\_();

uint32\_t get\_width\_();

void set\_type\_of\_cell(int x, int y, cell\_type type);

void set\_tp\_cords\_of\_cell(int x, int y, cords tp\_cords);

cell\_type touch\_cell(int x, int y);

cords get\_tp\_cords\_of\_cell(int x, int y);

};

Название файла: controller.cpp

#include "controller.h"

controller::controller(entity& somebody, field& somewhere, cords xy)

: e(somebody), f(somewhere), cords\_(xy)

{

// Now we can manage events

}

cords controller::get\_cords\_()

{

return cords\_;

}

void controller::set\_cords\_(int x, int y)

{

cords\_.x\_ = x;

cords\_.y\_ = y;

}

void controller::move(Direction move\_direction)

{

if (move\_direction == Top) {

if (f.touch\_cell(cords\_.x\_, cords\_.y\_ + 1) != Barrier)

cords\_.y\_ += 1;

}

if (move\_direction == Bottom) {

if (f.touch\_cell(cords\_.x\_, cords\_.y\_ - 1) != Barrier)

cords\_.y\_ -= 1;

}

if (move\_direction == Right) {

if (f.touch\_cell(cords\_.x\_ + 1, cords\_.y\_) != Barrier)

cords\_.x\_ += 1;

}

if (move\_direction == Left) {

if (f.touch\_cell(cords\_.x\_ - 1, cords\_.y\_) != Barrier)

cords\_.x\_ -= 1;

}

// if Spikes

if (f.touch\_cell(cords\_.x\_, cords\_.y\_) == Spikes)

{

e.add\_lives\_(-100);

}

// if Teleport

if (f.touch\_cell(cords\_.x\_, cords\_.y\_) == Teleport)

{

cords\_ = f.get\_tp\_cords\_of\_cell(cords\_.x\_, cords\_.y\_);

}

// events

f.do\_event\_of\_cell(cords\_.x\_, cords\_.y\_, \*this);

f.remove\_event\_of\_cell(cords\_.x\_, cords\_.y\_);

}

// the entity methods needed above

void controller::add\_lives\_(int how\_many) {

e.add\_lives\_(how\_many);

}

// the field methods needed above

uint32\_t controller::get\_height\_() {

return f.get\_height\_();

}

uint32\_t controller::get\_width\_() {

return f.get\_width\_();

}

void controller::set\_type\_of\_cell(int x, int y, cell\_type type) {

f.set\_type\_of\_cell(x, y, type);

}

void controller::set\_tp\_cords\_of\_cell(int x, int y, cords tp\_cords) {

f.set\_tp\_cords\_of\_cell(x, y, tp\_cords);

}

cell\_type controller::touch\_cell(int x, int y) {

return f.touch\_cell(x, y);

}

cords controller::get\_tp\_cords\_of\_cell(int x, int y) {

return f.get\_tp\_cords\_of\_cell(x, y);

}

Название файла: main.cpp

#include "field/field\_creator.h"

#include "controller.h"

int main()

{

field\_creator creator(main\_level);

field f1 = creator.get\_field\_();

f1.str();

std::cout << '\n';

player p1;

p1.add\_lives\_(1);

controller c\_p1(p1, f1, f1.get\_entrance\_());

// Spikes

std::cout << "check Spikes:" << '\n';

c\_p1.set\_cords\_(11, 16);

std::cout << "p1: " << c\_p1.get\_cords\_().x\_ << ' ' << c\_p1.get\_cords\_().y\_ << '\n';

std::cout << "lives\_ = " << p1.get\_lives\_() << '\n';

c\_p1.move(Bottom);

std::cout << "p1: " << c\_p1.get\_cords\_().x\_ << ' ' << c\_p1.get\_cords\_().y\_ << '\n';

std::cout << "lives\_ = " << p1.get\_lives\_() << '\n';

// Healing

std::cout << "check Heal\_point:" << '\n';

c\_p1.move(Bottom);

std::cout << "p1: " << c\_p1.get\_cords\_().x\_ << ' ' << c\_p1.get\_cords\_().y\_ << '\n';

std::cout << "lives\_ = " << p1.get\_lives\_() << '\n';

// Damage

std::cout << "check Damage:" << '\n';

c\_p1.set\_cords\_(18, 9);

std::cout << "p1: " << c\_p1.get\_cords\_().x\_ << ' ' << c\_p1.get\_cords\_().y\_ << '\n';

std::cout << "lives\_ = " << p1.get\_lives\_() << '\n';

c\_p1.move(Right);

std::cout << "p1: " << c\_p1.get\_cords\_().x\_ << ' ' << c\_p1.get\_cords\_().y\_ << '\n';

std::cout << "lives\_ = " << p1.get\_lives\_() << '\n';

// Teleport

std::cout << "check Teleport:" << '\n';

c\_p1.set\_cords\_(4, 3);

std::cout << "p1: " << c\_p1.get\_cords\_().x\_ << ' ' << c\_p1.get\_cords\_().y\_ << '\n';

c\_p1.move(Top);

std::cout << "p1: " << c\_p1.get\_cords\_().x\_ << ' ' << c\_p1.get\_cords\_().y\_ << '\n';

// Rift

std::cout << "check Rift:" << '\n';

c\_p1.set\_cords\_(8, 12);

std::cout << "p1: " << c\_p1.get\_cords\_().x\_ << ' ' << c\_p1.get\_cords\_().y\_ << '\n';

c\_p1.move(Bottom);

std::cout << "p1: " << c\_p1.get\_cords\_().x\_ << ' ' << c\_p1.get\_cords\_().y\_ << '\n';

return 0;

}