Jakub Piotrowicz 318350

Keyword: Hospital

1. **Premise**
   The keyword 'hospital' was given. I decided to create 4 classes, in order to divide objects into sub-objects. Hospitals will be divided into wards, wards into employees, and employees into patients.

   **Hospital class:**
   - contains: the name of the hospital, a vector containing all wards in the particular hospital, a vector containing all employees within the particular hospital and a vector containing all employees within the particular hospital,
   - wards, employees and patients can be added and removed,
   - a copy constructor allows for easy copying of all data from one hospital to another.

   **Ward class:**
   - contains: the name of the ward, a pointer to the hospital the ward belongs to, a vector containing all employees within this particular ward, a vector containing all patients within this ward,
   - employees and patients can be added and removed.

   **Employee class:**

   - contains: the name of the employee, the ID of the employee, the name of the hospital this employee belongs to, the name of the ward this employee belongs to, a vector containing all patients assigned to this employee,
   - patients can be added and remove.

   **Patient class:**

   - contains: the name of the patient, the ID of the patient, the name of the hospital this patient belongs to, the name of the ward this patient belongs to, the name of the employee this patient is assigned to.

2. **Class declarations**

   **Hospital class:**

```
#pragma once
#ifndef HOSPITAL_H
#define HOSPITAL_H

#include "Ward.h"
#include "Employee.h"
```

```cpp
#include "Patient.h"
#include <vector>

class Hospital
{
private:
    char* name;
    std::vector <Ward*> wards;
    std::vector <Employee*> employees;
    std::vector <Patient*> patients;

public:
    Hospital() = default;
    /**
     * @brief Copy constructor, constructs a new object of this class with the same pareameters as the given object
     *
     * @param other_obj
     */
    Hospital(const Hospital& other_obj);
    ~Hospital();

    /**
     * @brief Get this object's name
     *
     * @return char*
     */
    char* get_name();

    /**
     * @brief Set this object's name
     *
     * @param name
     */
    void set_name(char* name);

    /**
     * @brief adds a new ward to this hospital
     *
     * @param name
     * @return true if the ward is added successfully
     * @return false if the ward cannot be added
     */
    bool add_ward(char* name);

    /**
     * @brief removes a ward from this hospital
     *
     * @param name
```

```
     * @return true if the ward is removed successfully
     * @return false if the ward cannot be removed
     */
    bool remove_ward(char* name);

    /**
     * @brief adds an employee to this hospital
     *
     * @param name
     * @param ID
     * @param ward_name
     * @return true if the employee is added successfully
     * @return false if the employee cannot be added
     */
    bool add_employee(char* name, int ID, char* ward_name);

    /**
     * @brief removes an employee from this hospital
     *
     * @param name
     * @return true if the employee is removed successfully
     * @return false if the employee cannot be removed
     */
    bool remove_employee(char* name);

    /**
     * @brief adds a patient to this hospital
     *
     * @param name
     * @param ID
     * @param ward_name
     * @param employee_name
     * @return true if the patient is addeed successfully
     * @return false if the patient cannot be added
     */
    bool add_patient(char* name, int ID, char* ward_name, char* em-
ployee_name);

    /**
     * @brief removes a patient from this hospital
     *
     * @param name
     * @return true if the patient is removed successfully
     * @return false if the patient cannot be removed
     */
    bool remove_patient(char* name);

    /**
     * @brief prints all the information about this hospital
```

```
     *
     */
    void print() const;
};

#endif
```

**Ward class:**

```
#pragma once
#ifndef WARD_H
#define WARD_H

#include <vector>

class Hospital;
class Employee;
class Patient;

class Ward
{
private:
    char* name;
    Hospital* hospital_name;
    std::vector <Employee*> employees;
    std::vector <Patient*> patients;

public:
    Ward() = default;
    Ward(const Ward& other_obj);
    ~Ward();

    /**
     * @brief Get this object's name
     *
     * @return char*
     */
    char* get_name() const;

    /**
     * @brief Set this object's name
     *
     * @param name
     */
    void set_name(char* name);

    /**
     * @brief Get the name of the hospital this object belongs to
```

```
     *
     * @return Hospital*
     */
    Hospital* get_hospital_name() const;

    /**
     * @brief Set this object's hospital's name
     *
     * @param hospital_name
     */
    void set_hospital_name(Hospital* hospital_name);

    /**
     * @brief adds an employee to this ward
     *
     * @param name
     * @param ID
     * @return true if the employee is added successfully
     * @return false if the employee cannot be added
     */
    bool add_employee(char* name, int ID);

    /**
     * @brief removes an employee from this ward
     *
     * @param name
     * @return true if the employee is removed successfully
     * @return false if the employee cannot be removed
     */
    bool remove_employee(char* name);

    /**
     * @brief adds a patient to this ward
     *
     * @param name
     * @param ID
     * @param employee_name
     * @return true if the patient is added successfully
     * @return false if the patient cannot be added
     */
    bool add_patient(char* name, int ID, char* employee_name);

    /**
     * @brief removes a patient from this ward
     *
     * @param name
     * @return true if the patient is removed successsfully
     * @return false if the patient cannot be removed
     */
```

```
    bool remove_patient(char* name);

    /**
     * @brief prints all information about this ward
     *
     */
    void print() const;
};

#endif
```

**Employee class:**

```
#pragma once
#ifndef EMPLOYEE_H
#define EMPLOYEE_H

#include <vector>

class Hospital;
class Ward;
class Patient;

class Employee
{
private:
    char* name;
    int ID;
    Hospital *hospital_name;
    Ward *ward_name;
    std::vector <Patient*> patients;

public:
    Employee() = default;
    Employee(const Employee& other_obj);
    ~Employee();

    /**
     * @brief Get this object's name
     *
     * @return char*
     */
    char* get_name() const;

    /**
     * @brief Set this object's name
     *
     * @param name
```

```
     */
    void set_name(char* name);

    /**
     * @brief Get this object's ID
     *
     * @return int
     */
    int get_ID() const;

    /**
     * @brief Set this object's ID
     *
     * @param ID
     */
    void set_ID(int ID);

    /**
     * @brief Get the name of the hospital this employee belongs to
     *
     * @return Hospital*
     */
    Hospital* get_hospital_name() const;

    /**
     * @brief Set this object's hospital's name
     *
     * @param hospital_name
     */
    void set_hospital_name(Hospital* hospital_name);

    /**
     * @brief Get the name of the ward this employee belongs to
     *
     * @return Ward*
     */
    Ward* get_ward_name() const;

    /**
     * @brief Set this object's ward name
     *
     * @param ward_name
     */
    void set_ward_name(Ward* ward_name);

    /**
     * @brief adds a patient to this employee
     *
     * @param name
```

```
     * @param ID
     * @return true if the patient is added successsfully
     * @return false if the patient cannot be added
     */
    bool add_patient(char* name, int ID);

    /**
     * @brief removes a patient from this employee
     *
     * @param name
     * @return true if the patient is removed successfully
     * @return false if the patient cannot be removed
     */
    bool remove_patient(char* name);

    /**
     * @brief prints all information about this employee
     *
     */
    void print() const;
};

#endif
```

**Patient class:**

```
#pragma once
#ifndef PATIENT_H
#define PATIENT_H

class Hospital;
class Ward;
class Employee;

class Patient
{
private:
    char* name;
    int ID;
    Hospital *hospital_name;
    Ward *ward_name;
    Employee *doctor_name;

public:
    Patient() = default;
    Patient(const Patient& other_obj);
    ~Patient();
```

```
    /**
     * @brief Get this object's name
     *
     * @return char*
     */
    char* get_name() const;

    /**
     * @brief Set this object's name
     *
     * @param name
     */
    void set_name(char* name);

    /**
     * @brief Get this object's ID
     *
     * @return int
     */
    int get_ID() const;

    /**
     * @brief Set this object's ID
     *
     * @param ID
     */
    void set_ID(int ID);

    /**
     * @brief Get the name of the hospital this patient belongs to
     *
     * @return Hospital*
     */
    Hospital* get_hospital_name() const;

    /**
     * @brief Set this object's hospital's name
     *
     * @param hospital_name
     */
    void set_hospital_name(Hospital* hospital_name);

    /**
     * @brief Get the name of the ward this patient belongs to
     *
     * @return Ward*
     */
    Ward* get_ward_name() const;
```

```
    /**
     * @brief Set this object's ward's name
     *
     * @param ward_name
     */
    void set_ward_name(Ward* ward_name);

    /**
     * @brief Get the name of the employee this patient belongs to
     *
     * @return Employee*
     */
    Employee* get_doctor_name() const;

    /**
     * @brief Set this object's employee's name
     *
     * @param employee_name
     */
    void set_doctor_name(Employee* employee_name);

    /**
     * @brief print all information about this patient
     *
     */
    void print() const;
};

#endif
```

3. **Testing**

```
//creating an object of the 'Hospital' class
    Hospital h1;

    //setting the name of the Hospital
    h1.set_name("Saint George's Hospital");
    //adding a Ward
    h1.add_ward("Oncology");

    //trying to add the same Ward again
    cout << "Test 1: ";
    if(h1.add_ward("Oncology") == false)
    {
        cout << "Failed to add the Ward\n";
    }

    //adding a second, different Ward
    h1.add_ward("Dermatology");

    //trying to add an Employee to a Ward that doesn't exist
    cout << "Test 2: ";
    h1.add_employee("John Swanson", 1, "ER");

    //adding Employees to a Wards which exists
    h1.add_employee("John Swanson", 1, "Oncology");
    h1.add_employee("Johnnie Walker", 2, "Dermatology");

    //trying to assign a Patient to an Employee who doesn't exist
    cout << "Test 3: ";
    h1.add_patient("Teo", 1, "Oncology","Babe Ruth");

    //trying to assign a Patient to a Ward that doesn't exist
    cout << "Test 4: ";
    h1.add_patient("Teo", 1, "ER", "John Swanson");

    //assigning Patients to Employees
    h1.add_patient("Teo", 1, "Oncology","John Swanson");
    h1.add_patient("Sareen", 1, "Dermatology", "Johnnie Walker");

    //displaying information about the hospital
    h1.print();

    //creating a second object with the copy constructor
    Hospital h2(h1);

    //printing information about the two objects
    h1.print();
```

```
    h2.print();

    //removing and adding patients, employees and wards from the second
hospital to introduce changes
    cout << "Test 5: \n";
    h2.remove_employee("Teo");
    h2.add_ward("ER");
    h2.add_employee("Booba Fett", 3, "ER");

    //displaying the new information
    h1.print();
    h2.print();

    //using the overloaded '=' operator to copy data from h2 to h1
    h1 = h2;

    //printing information about both Hospitals
    h1.print();
    h2.print();
```

4. **Feedback**
   The following feedback was given:
   - Creating new objects with constructors instead of dedicated functions,
   - Adding wards, employees and patients through references to objects instead of text,
   - Changing some confusing data and function names,
   - Changing destructors,
   - Implementing a more independent design instead of a 'Hospital' super-class.
5. **Feedback implementation:**
   <u>Creating new objects with constructors instead of dedicated functions:</u>
   As it stands, the constructors are not utilized enough in the project. Instead, addition functions are used to initialize the objects. This is redundant design, which can be rectified by rewriting the constructors to be more encompassing. Several different constructors based on the number of given parameters would eliminate the need for additional functions.

   <u>Adding wards, employees and patients through references to objects instead of text:</u>
   This is the original implementation:

```
h1.add_employee("John Swanson", 1, "ER");
```

   Instead, a different approach could be taken:

```
add_employee("John Swanson",1,&ward1);
```

   where ward1 is an object of the 'Ward' class, this makes the program more universal and easier to operate. Several functions could be written to facilitate different scenarios, and in turn make the whole structure more flexible.

   <u>Changing some confusing data and function names:</u>
   In 'Ward', 'Employee' and 'Patient' classes, the names of the higher-order objects are called e.g. 'hospital_name' or 'ward_name' , instead, they should simply be called 'hospital', 'ward' etc. to avoid confusion.

Changing destructors:

With a new, decentralized approach, new constructors ought to be implemented into the project. The new constructors would have to make sure that the deletion of an object would be reflected within objects of different classes connected to it, e.g.:

```
Hospital::~Hospital() {

    for(int i = 0; i < this->wards.size(); i++)
    {
        this->wards.at(i)->clear_hospital_name(this);
    }

    for(int i = 0; i < this->employees.size(); i++)
    {
        this->employees.at(i)->clear_hospital_name(this);
    }

    for(int i = 0; i < this->patients.size(); i++)
    {
        this->patients.at(i)->clear_hospital_name(this);
    }

    delete[] this->name;
    this->wards.clear();
    this->employees.clear();
    this->patients.clear();
}
```

Implementing a more independent design instead of a 'Hospital' super-class:

As it stands, the 'Hospital' class is a super-class that menages all the objects of the other three classes. This is a valid, but a limiting approach. Instead, the classes could be rewritten to facilitate more independence between objects of the three classes besides the 'Hospital' one.

6. **Final Conclusions**

Summing up, this project taught me how to implement object-oriented solutions to a problem. The given feedback pointed me towards more decentralized solutions, which I will make sure to use in any future projects.