МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ

УЧРЕЖДЕНИЕ ОБРАЗОВАНИЯ

“БРЕСТСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ”

КАФЕДРА ИИТ

ОТЧЁТ

по лабораторной работе №8

**«Обобщение знаний.»**

Выполнил:

студент 3 курса

группы ПО-9

Мисиюк Алексей Сергеевич

Проверил:

Козик И. Д.

**Цель работы**: написать программу, используя знания, полученные в предыдущих лабораторных работах.

**Вариант №3**

Выполнение действий должно происходить в разных потоках. Все классы должны быть подключены, используя DLL. В каждом варианте должно быть использовано наследование и должна быть дополнительная программа-клиент, написанная, используя WinAPI, служащая для заполнения данных (основная программа - консольная).

Создать симулятор больницы.

Необходимые классы: разные виды врачей, пациенты. Заполняемые с помощью WinAPI данные: вся информация о врачах и пациентах. Суть работы: врачи посещают пациентов, за которыми закреплены и лечат определённые симптомы в течение некоторого времени в зависимости от специальности. У врача есть максимальное время лечения, после которого он уходит к следующему пациенту. Два врача не могут одновременно обслуживать одного пациента. Как только пациент вылечен – он выписывается из больницы и врачи больше к нему не ходят. Программа завершается, когда все пациенты вылечены.

**Код программы**

**Dll: HospitalDll.h**

#pragma once

#include <iostream>

#include <vector>

#include <thread>

#include <mutex>

#include <chrono>

//A good tutorial: https://learn.microsoft.com/en-us/cpp/build/walkthrough-creating-and-using-a-dynamic-link-library-cpp?view=msvc-170

#ifdef MAINLIBRARY\_EXPORTS

#define HOSPITALLIB\_API \_\_declspec(dllexport)

#else

#define HOSPITALLIB\_API \_\_declspec(dllimport)

#endif

std::mutex coutMutex;

std::mutex patientsMutex;

extern "C++" HOSPITALLIB\_API class Patient {

public:

std::string name;

int treatmentTime;

bool isTreated;

enum class PatientType {

Base,

Adult,

Child

};

Patient(const std::string& name, const int treatmentTime = 500)

: name(name), treatmentTime(treatmentTime), isTreated(false) {}

void receiveTreatment(int time) {

std::unique\_lock<std::mutex> coutLock(coutMutex);

std::cout << "Patient " << name << " is receiving treatment (" << treatmentTime << " left).\n\n";

coutLock.unlock();

std::this\_thread::sleep\_for(std::chrono::milliseconds(time));

treatmentTime -= time;

coutLock.lock();

if (treatmentTime <= 0) {

isTreated = true;

std::cout << "Patient " << name << " has been successfully treated.\n";

}

else {

std::cout << "Patient " << name << " need additional attention (" << treatmentTime << " left).\n";

}

coutLock.unlock();

}

virtual PatientType getType() const {

return PatientType::Base;

}

};

extern "C++" HOSPITALLIB\_API class AdultPatient : public Patient {

public:

AdultPatient(const std::string& name, const int treatmentTime = 500)

: Patient(name, treatmentTime) {}

virtual PatientType getType() const {

return PatientType::Adult;

}

};

extern "C++" HOSPITALLIB\_API class ChildPatient : public Patient {

public:

ChildPatient(const std::string& name, const int treatmentTime = 500)

: Patient(name, treatmentTime) {}

virtual PatientType getType() const {

return PatientType::Child;

}

};

extern "C++" HOSPITALLIB\_API class Doctor {

public:

std::string name;

int maxTreatmentTime;

Doctor(const std::string& name, int maxTreatmentTime = 300) : name(name), maxTreatmentTime(maxTreatmentTime) {}

void treatPatient(Patient& patient) {

std::unique\_lock<std::mutex> coutLock(coutMutex);

std::cout << "Doctor " << name << " is treating patient " << patient.name << ".\n";

coutLock.unlock();

patient.receiveTreatment((((maxTreatmentTime) < (patient.treatmentTime)) ? (maxTreatmentTime) : (patient.treatmentTime)));

coutLock.lock();

std::cout << "Doctor " << name << " has finished treating patient " << patient.name << ".\n";

coutLock.unlock();

}

void work(std::vector<std::reference\_wrapper<Patient>>& patientsQueue) {

while (true) {

std::unique\_lock<std::mutex> lock(patientsMutex);

if (!patientsQueue.empty()) {

Patient& patient = getNextPatient(patientsQueue);

lock.unlock();

if (patient.isTreated) {

std::this\_thread::sleep\_for(std::chrono::milliseconds(10));

continue;

}

treatPatient(patient);

if (!patient.isTreated) {

patientsQueue.push\_back(patient);

}

}

else {

lock.unlock();

break; // No more patients to treat

}

std::this\_thread::sleep\_for(std::chrono::milliseconds(10));

}

}

virtual Patient& getNextPatient(std::vector<std::reference\_wrapper<Patient>>& patientsQueue) {

if (!patientsQueue.empty()) {

Patient& patient = patientsQueue.front().get();

patientsQueue.erase(patientsQueue.begin());

return patient;

}

// Return null if no patient is found

static Patient nullPatient("", 0);

nullPatient.isTreated = true;

return nullPatient;

}

};

extern "C++" HOSPITALLIB\_API class AdultDoctor : public Doctor {

public:

AdultDoctor(const std::string& name, int maxTreatmentTime = 300) : Doctor(name, maxTreatmentTime) {}

Patient& getNextPatient(std::vector<std::reference\_wrapper<Patient>>& patientsQueue) override {

auto it = std::find\_if(patientsQueue.begin(), patientsQueue.end(),

[](const std::reference\_wrapper<Patient>& patient) {

return patient.get().getType() == Patient::PatientType::Adult;

});

if (it != patientsQueue.end()) {

Patient& patient = it->get();

patientsQueue.erase(it);

return patient;

}

// Return null if no patient is found

static Patient nullPatient("", 0);

nullPatient.isTreated = true;

return nullPatient;

}

};

extern "C++" HOSPITALLIB\_API class Pediatrician : public Doctor {

public:

Pediatrician(const std::string& name, int maxTreatmentTime = 300) : Doctor(name, maxTreatmentTime) {}

Patient& getNextPatient(std::vector<std::reference\_wrapper<Patient>>& patientsQueue) override {

auto it = std::find\_if(patientsQueue.begin(), patientsQueue.end(),

[](const std::reference\_wrapper<Patient>& patient) {

return patient.get().getType() == Patient::PatientType::Child;

});

if (it != patientsQueue.end()) {

Patient& patient = it->get();

patientsQueue.erase(it);

return patient;

}

// Return null if no patient is found

static Patient nullPatient("", 0);

nullPatient.isTreated = true;

return nullPatient;

}

};

**console.cpp**

// Выполнение действий должно происходить в разных потоках.

// Все классы должны быть подключены, используя DLL.

// В каждом варианте должно быть использовано наследование

// и должна быть дополнительная программа - клиент, написанная, используя WinAPI,

// служащая для заполнения данных(основная программа - консольная).

// Создать симулятор больницы.

// Необходимые классы : разные виды врачей, пациенты.

// Заполняемые с помощью WinAPI данные : вся информация о врачах и пациентах.

// Суть работы : врачи посещают пациентов, за которыми закреплены и лечат определённые

// симптомы в течение некоторого времени в зависимости от специальности.

// У врача есть максимальное время лечения, после которого он уходит к следующему пациенту.

// Два врача не могут одновременно обслуживать одного пациента.

// Как только пациент вылечен – он выписывается из больницы и врачи больше к нему не ходят.

// Программа завершается, когда все пациенты вылечены.

#include <iostream>

#include <vector>

#include <thread>

#include "HospitalLib.h"

int main() {

std::vector<std::reference\_wrapper<Patient>> patientsQueue;

// Create adult patients

std::vector<AdultPatient> adultPatients = { {"AdultPatient1", 4000}, {"AdultPatient2", 1000}, {"AdultPatient3", 7500} };

for (auto& adultPatient : adultPatients) {

patientsQueue.push\_back(std::ref(adultPatient));

}

// Create child patients

std::vector<ChildPatient> childPatients = { {"ChildPatient1", 3000}, {"ChildPatient2", 2000}, {"ChildPatient3", 5000} };

for (auto& childPatient : childPatients) {

patientsQueue.push\_back(std::ref(childPatient));

}

std::vector<std::unique\_ptr<Doctor>> doctors;

doctors.push\_back(std::make\_unique<Doctor>("Doctor1", 2500));

doctors.push\_back(std::make\_unique<AdultDoctor>("AdultDoctor2", 4000));

doctors.push\_back(std::make\_unique<Pediatrician>("Pediatrician3", 3500));

std::vector<std::thread> threads;

for (auto& doctor : doctors) {

threads.emplace\_back(&Doctor::work, doctor.get(), std::ref(patientsQueue));

}

for (auto& thread : threads) {

thread.join();

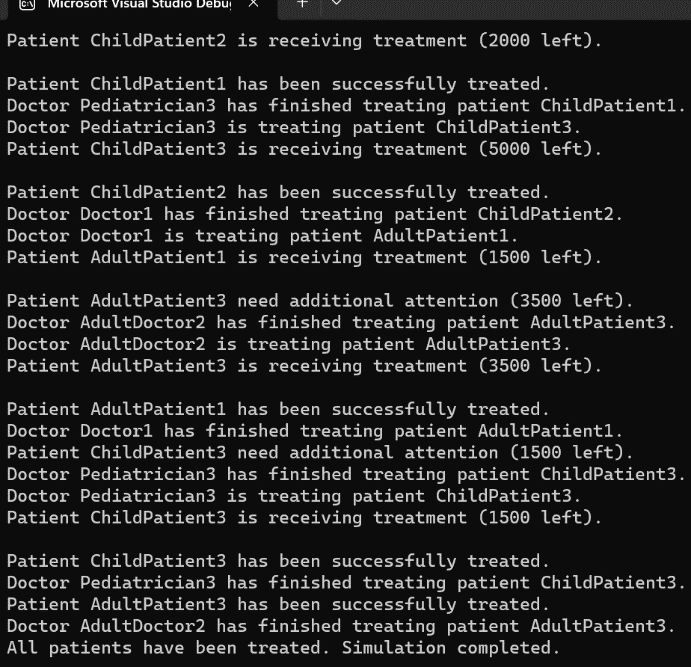
}

std::cout << "All patients have been treated. Simulation completed.\n";

return 0;

}

**Пример работы**

****