МАТФ
Универзитет у Београду
Математички факултет
11001 БЕОГРАД
Студентски трг 16
П. П. 550
Телефон: 011 20 27 801
Факс: 011 26 30 151
matf@matf.bg.ac.rs
www.matf.bg.ac.rs

8 Тестирање софтвера и развој вођен тестовима у Catch2

Упознавање са библиотеком Catch2

- Након преузимања датотеке `catch.hpp`, потребно је направити датотеку `test.cpp` са наредним садржајем

```
#define CATCH_CONFIG_MAIN
#include "catch.hpp"

TEST_CASE("...", "...")
{
    // Kod za testiranje ide ovde
}
```

- Све макрое за тестирање ћемо приказати кроз потребе тестирања наредне функције

```
#include <vector>
#include <stdexcept>
#include <iterator>

vector<unsigned long> fib(int n)
{
    if (n < 0) throw invalid_argument("n < 0");
    if (n == 0) return {0u};
    if (n == 1) return {0u, 1u};
    auto previous = fib(n-1);
    previous.push_back(*(end(previous) - 1) + *(end(previous) - 2));
    return previous;
}
```

- Макрои `TEST_CASE` и `SECTION` служе за дефинисање секција у оквиру којих се пишу тестови. Често се један јединични тест налази у једном `SECTION` делу, а више сродних јединичних тестова се групишу у један `TEST_CASE` део.
- Најосновнији макро за проверавање да ли је испуњен неки услов је `REQUIRE`.

```
TEST_CASE("Izracunavanje fibonacijevog niza", "[fib][function]")
{
    SECTION("Funkcija fib vraca prvih 6 brojeva fib. niza za ulaz 5")
    {
        vector<unsigned long> expected = {0, 1, 1, 2, 3, 5};
        REQUIRE(fib(5) == expected);
    }
}
```

- Ако променимо дефиницију фибоначијевог низа, на пример, да то буде низ 1, 1, 2, 3, 5, 8 ..., претходни тест ће сигурно пасти.
- Ако макар једна секција падне, онда цео случај пада. Ипак, све секције се извршавају независно једне од других (тако да падање једне секције не утиче на падање других секција). Тако, на пример, наредна секција пада:

```
SECTION("Funkcija fib vraca niz [1, 1] za ulaz 1")
{
    vector<unsigned long> expected = {1, 1};
    REQUIRE(fib(1) == expected);
}
```

али се зато наредна секција извршава и пролази успешно:

```
SECTION("Funkcija fib vraca niz koji ima makar jednu 1 za ulaz 1")
{
    const auto result = fib(1);
    const auto number_of_ones = count(begin(result), end(result), 1);
    const auto expected_min = 1;
    REQUIRE(number_of_ones >= expected_min);
}
```

- Остали макрои: `REQUIRE_FALSE`, `CHECK_FALSE`, `REQUIRE_NOTHROW`, `CHECK_NOTHROW`, `REQUIRE_THROWS`, `CHECK_THROWS`, `REQUIRE_THROWS_AS` и `CHECK_THROWS_AS`

```
SECTION("Prosledjivanje pozitivnog broja ne sme da izazove izuzetak")
{
    REQUIRE_NOTHROW(fib(1));
    REQUIRE_NOTHROW(fib(2));
    /* ... */
}

SECTION("Prosledjivanje negativnog broja mora da izazove izuzetak")
{
    CHECK_THROWS(fib(-1));
    CHECK_THROWS(fib(-2));
    /* ... */
}

SECTION("Pros. neg. broja mora da izazove izuzetak invalid_argument")
{
    CHECK_THROWS_AS(fib(-1), invalid_argument);
    CHECK_THROWS_AS(fib(-2), invalid_argument);
    /* ... */
}
```

Парадигме писања и именовања тестова

Све се налази на презентацији.

Развој вођен тестовима

Прећи на презентацију и објаснити концепт.

Развој класе `BigInteger`:

- Тест 1:

```
TEST_CASE("Creating the BigInteger", "[BigInteger]")
{
    SECTION("When BigInteger is default-constructed, its value is zero")
    {
        BigInteger bigInt;
    }
}
```

- Већ имамо компилаторску грешку! Стајемо и имплементирамо код:

```
class BigInteger
{};
```

- Сада када имамо конструктор, можемо да завршимо тест 1:

```
TEST_CASE("Creating the BigInteger", "[BigInteger]")
{
    SECTION("When BigInteger is default-constructed, its value is zero")
    {
        BigInteger bigInt;
        auto expected{0};
        auto result{bigInt.toInt()};
        REQUIRE(result == expected);
    }
}
```

- Имплементација:

```
class BigInteger
{
public:
    BigInteger() = default;

    int toInt()
    {
        return 0;
    }
};
```

- Наредни тест — конструкција великог целог броја од једне цифре:

```
SECTION("When BigInteger is constructed from one letter string, " \
        "its value is number in that string")
{
    BigInteger bigInt{"1"};
    auto expected{1};
    auto result{bigInt.toInt()}
    REQUIRE(result == expected);
}
```

- Имплементација:

```
public:
    BigInteger()
        : _digits({0})
    {}

    BigInteger(string number)
    {
        _digits.push_back(number.back() - '0');
    }

    int toInt()
    {
        return _digits.back();
    }
private:
    vector<unsigned> _digits;
```

- Рефакторисање:

```
public:
    BigInteger(string number) {
        extractDigits(number);
    }
private:
    void extractDigits(string number) {
        _digits.push_back(digitFromChar(number.back()));
    }

    unsigned digitFromChar(char digit) {
        return static_cast<unsigned>(digit - '0');
    }
```

- Наредни тест који не пролази:

```
SECTION("When BigInteger is constructed from more than one letter string, " \
        "the number of digits are equal to the size of the string")
{
    BigInteger bigInt{"1234"};
    auto expected{4u};
    auto result{bigInt.numberOfDigits()};
    REQUIRE(result == expected);
}
```

- Имплементација измене:

```
private:
    void extractDigits(string number) {
        _digits.push_back(digitFromChar(number.front()));
        if (number.size() == 1u) {
            return;
        }
        extractDigits(number.substr(1));
    }
```

- Рефакторисање

```
private:
    void extractDigits(string number) {
        extractFirstDigit(number);
        extractTail(number);
    }

    void extractFirstDigit(string number) {
        _digits.push_back(digitFromChar(number.front()));
    }

    void extractTail(string number) {
        auto tail{number.substr(1)};
        if (tail.empty()) {
            return;
        }
        extractDigits(tail);
    }
```

- Сада можемо написати тест за проверу вредности великог целог броја са више од једне цифре који смо претходно ставили са стране у листу тестова

```
SECTION("When BigInteger is constructed from multi-letter string, " \
        "its value is number in that string")
{
    BigInteger bigInt{"123456789"};
    auto expected{123456789};
    auto result{bigInt.toInt()};
    REQUIRE(result == expected);
}
```

- Имплементација измене

```
public:
    int toInt()
    {
        using begin, end;
        return accumulate(begin(_digits), end(_digits), int{},
        [](const auto acc, const auto next)
        {
            return 10*acc + next;
        });
    }
```

- Задовољни смо квалитетом кода за сада, па нема потребе за рефакторисањем.

- Сада правимо тестове који се односе на одређивање знака великог броја. Хајде прво да се позабавимо подр. конструкцијом.

```
SECTION("When BigInteger is default-constructed, its sign is Zero")
{
    const BigInteger bigInt;
    const auto expected{BigInteger::Zero};
    const auto result{bigInt.sign()};
    REQUIRE(result == expected);
}
```

- Имплементација измене

```
public:
    enum Sign
    {
        Zero,
        Positive,
        Negative
    };

    BigInteger()
        : _digits({0}), _sign(Zero)
    {
    }

    Sign sign() const
    {
        return _sign;
    }

private:
    Sign _sign;
```

- Нема рефакторисања. Хајде да направимо тест за знак позитивних бројева

```
SECTION("When BigInteger is constructed with a positive number string, its sign is Positive")
{
    const BigInteger bigInt{"123456789"};
    const auto expected{BigInteger::Positive};
    const auto result{bigInt.sign()};
    REQUIRE(result == expected);
}
```

- Имплементација измене:

```
public:
    BigInteger(string number) {
        if (number.front() != '-') {
            _sign = Positive;
        }
        extractDigits(number);
    }
```

- Рефакторисање:

```
public:
    BigInteger(string number) {
        extractSign(number);
        extractDigits(number);
    }

private:
    void extractSign(string &number) {
        if (number.front() != '-') {
            _sign = Positive;
        }
    }
```

- Пишемо тест за знак негативног броја:

```
SECTION("When BigInteger is constructed with a negative number string, its sign is Negative")
{
    const BigInteger bigInt{"-123456789"};
    const auto expected{BigInteger::Negative};
    const auto result{bigInt.sign()};
    REQUIRE(result == expected);
}
```

- Имплементација измене:

```
private:
    void extractSign(string &number) {
        if (number.front() == '-') {
            _sign = Negative;
            number = number.substr(1);
        }
        else {
            _sign = Positive;
        }
    }
```

- Нема рефакторисања. Пишемо тест за знак нуле:

```
SECTION("When BigInteger is constructed with a zero, its sign is Zero")
{
    const BigInteger bigInt{"0"};
    const auto expected{BigInteger::Zero};
    const auto result{bigInt.sign()};
    REQUIRE(result == expected);
}
```

- Имплементација измене:

```
private:
    void extractSign(string &number) {
        if (number.front() == '-') {
            _sign = Negative;
            number = number.substr(1);
        }
        else if (number.front() == '0') {
            _sign = Zero;
        }
        else {
            _sign = Positive;
        }
    }
```

- Нема рефакторисања. Коначно, пишемо тест за израчунавање вредности негативних бројева:

```
SECTION("When BigInteger is constructed with a negative number string, " \
        "its value is the negative number in string")
{
    const BigInteger bigInt{"-123456789"};
    const auto expected{-123456789};
    const auto result{bigInt.toInt()};
    REQUIRE(result == expected);
}
```

- Имплементација измене:

```
private:
    int toInt() const
    {
        using begin, end;
        return SignToInt(sign()) * accumulate(begin(_digits), end(_digits), int{},
        [](const auto acc, const auto next) {
            return 10 * acc + next;
        });
    }

private:
    static int SignToInt(const Sign sign) {
        if (sign == Zero) return 0;
        if (sign == Positive) return 1;
        return -1;
    }
```

- Нема потребе за рефакторисањем. Завршили смо текући тест случај који се односи на конструкцију великих бројева и одређивање њихових вредности. Сада настављамо са имплементацијама осталих тест случајева (за домаћи: довршити развој класе).

Развој вођен понашањем

Датотека `test.cpp`:

```
#define CATCH_CONFIG_MAIN
#include "catch.hpp"

#include "HelloClass.hpp"

SCENARIO("Should be able to greet with Hello BDD! message")
{
    GIVEN("an instance of Hello class is created")
    {
        const HelloClass hello;

        WHEN("the sayHello method is invoked")
        {
            const auto result{hello.sayHello()};

            THEN("it should return 'Hello BDD!'")
            {
                const auto expected{"Hello BDD!"};

                REQUIRE(result == expected);
            }
        }
    }
}
```

Датотека `HelloClass.hpp`:

```
class HelloClass
{
public:
    const char* sayHello() const
    {
        return "Hello BDD!";
    }
};
```

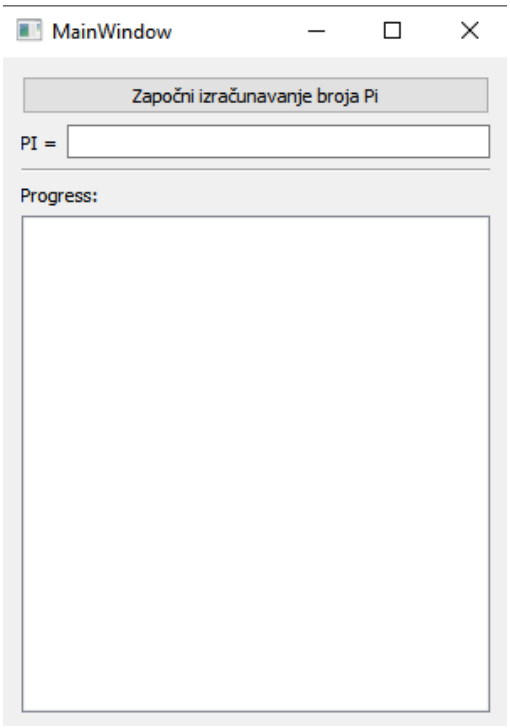
МАТФ
Универзитет у Београду
Математички факултет
11001 БЕОГРАД
Студентски трг 16
П.П. 550
Телефон: 011 20 27 801
Факс: 011 26 30 151
matf@matf.bg.ac.rs
www.matf.bg.ac.rs

9 Конкурентно програмирање у Qt5

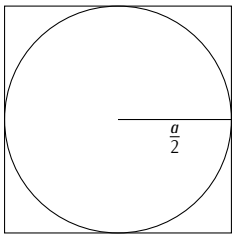
Пример 1 - PiCalc

Направити нови Qt5 пројекат назива PiCalc. Главни прозор назвати `MainWindow` и подесити да наследи `QWidget`. Креирати ГКИ као на слици поред. Користити `QVBoxLayout` за корени `QWidget`. Сместити у корени виџет редом наредне контроле:

- `QPushButton` назива `pbStartCalculating` са текстом „Започни израчунавање броја Пи”
- `QHBoxLayout` у оквиру којег се смештају наредне контроле:
  - `QLabel` са текстом „ПИ =”
  - `QLineEdit` назива `lePi`
- `Line` хоризонталне оријентације
- `QLabel` са текстом „Прогрес:”
- `QListWidget` назива `lwProgress`



Апроксимација броја π алгоритмом Монте-Карло се ради тако што се генерише велики број тачака $(x, y) \in [-1, 1]^2 \subset \mathbf{R}^2$. Затим се за сваку такву тачку проверава да ли се налази у јединичном кругу (тј. да ли је испуњено $x^2 + y^2 \leq 1$). Забележити број таквих тачака као `hits`, док је укупан број итерација `count`. Апроксимација се добија израчунавањем $4 \cdot$ `hits`/`count`.



Површина квадрата је једнака

$$P_{kv} = a^2$$

Површина круга је једнака

$$P_{kr} = \frac{a^2}{4}\pi$$

Одавде добијамо да важи

$$P_{kr} = \frac{P_{kv}}{4}\pi,$$

одакле се лако добија поменута формула за апроксимацију

$$\pi = \frac{4 \cdot P_{kr}}{P_{kv}} = \frac{4 \cdot hits}{count}$$

Сада можемо да дефинишемо слот метод у оквиру класе `MainWindow` који ће се позвати када корисник кликне на дугме. Желимо да имплементирамо да се на сваких 10% итерација у `QListWidget`-у приказује нова ставка која приказује колико процената итерација је програм извршио у симулацији.

Приметимо да се симулација извршава у главној петљи догађаја, тј. у петљи догађаја у којој ГКИ живи. Када се започне израчунавање вредности броја π у главној петљи догађаја, тада ГКИ „замрзне", односно, није респонзиван све док се извршавање не заврши. И тек у том тренутку видимо све проценте исписане, уместо да буду приказани један по један.

Код 1: Датотека MainWindow.h (ажурирања)

```
class MainWindow : public QWidget
{
// ...
private slots:
    void calculatePiValue();
};
```

Код 2: Датотека MainWindow.cpp (ажурирања)

```
// ...
#include <QRandomGenerator>

MainWindow::MainWindow(QWidget *parent)
    : QWidget(parent)
    , ui(new Ui::MainWindow)
{
    ui->setupUi(this);

    connect(ui->pbStartCalculating, &QPushButton::clicked, this, &MainWindow::calculatePiValue);
}

void MainWindow::calculatePiValue()
{
    auto hits = 0u, count = 0u;
    const auto iterations = 100'000'000u;
    const auto iterUpdate = 10'000'000u;
    ui->lwProgress->clear(); // Cistimo widget za naredni prikaz progresa

    while (count < iterations) {
        const auto x = QRandomGenerator::global()->generateDouble() * 2 - 1;
        const auto y = QRandomGenerator::global()->generateDouble() * 2 - 1;

        if (x * x + y * y <= 1.0) {
            ++hits;
        }
        ++count;

        if (count % iterUpdate == 10) {
            ui->lwProgress->addItem(QString::number(count / iterUpdate * 10u) + "%");
        }
    }

    auto pi = 4.0 * hits / count;
    ui->lePi->setText(QString::number(pi));
}
// ...
```

Пример 2 - PiCalcThreaded

Пример показује једноставан начин за решавање проблема из претходног примера коришћењем конкурентног програмирања. Програм креира једну нит која се одвојено извршава од главне нити (у којој ради ГКИ) и у тој нити израчунава апроксимацију броја π. Након што одвојена нит заврши са радом, она емитује сигнал који је претходно био повезан са слотом у главној нити (у прозору) и прослеђује израчунату вредност. Програм ту вредност приказује у прозору. Разлика у односу на претходни пример је то што ГКИ није „замрзнут".

У **вишенитним** апликацијама, ГКИ се извршава у својој нити, док се скупа израчунавања извршавају у произвољном броју засебних нити. Ово резултује у апликацијама које имају респонзивне ГКИ чак и током захтевних израчунавања. Додатни бенефит вишенитности је да вишепроцесни систем може да изврши неколико нити дословно паралелно на различитим процесорима, што резултује бољим перформансама (осим када су израчунавања тривијална, те је време потребно за одржавање нити и њихову синхронизацију скупље од укупног извршавања).

Креирање и покретање нити

Да бисмо креирали вишенитне апликације и библиотеци Qt, довољно је наследити класу `QThread` и превазићи метод `void run()` (он је `protected`, пошто се не позива директно из спољашњег контекста). Када креирамо објекте класа које су наслеђене из `QThread`, потребно је да позовемо метод `void start()` над њима да би се метод `run()` позвао. Када се метод `start()` покрене, он емитује сигнал `started` и извршава метод `run()`, а када метод `run()` заврши са извршавањем, нит емитује сигнал `finished`.

Петља догађаја у нитима

С обзиром да нит може да користи систем сигнала и слотова, разумно је претпоставити да свака нит има своју петљу догађаја у оквиру које се могу обрадити догађаји изван главне петље догађаја (тј. ГКИ петље догађаја). Уколико се не превазиђе, метод `run()` позива метод `QThread::exec()` која започиње петљу догађаја за дату нит. С обзиром да ми превазилазимо метод `run()`, могуће је позвати метод `exec()` у превазиђеној имплементацији и тиме започети петљу догађаја у нити.

Треба бити опрезан са овиме. Наиме, сваки објекат има свој **афинитет нити**. У зависности од афинитета нити, нит можда неће обрадити догађај за неке догађаје, ако сигнали и слотови нису исправно повезани. Ово се регулише 5. аргументом метода `QObject::connect` што је `Qt::ConnectionType type = Qt::AutoConnection` и који може бити:

- `Qt::AutoConnection`: Ако прималац живи у нити која емитује сигнал, користи се `Qt::DirectConnection`. Иначе, користи се `Qt::QueuedConnection`. Тип конекције се одређује када се сигнал емитује.
- `Qt::DirectConnection`: Слот се одмах позива када је сигнал емитован и он се извршава у оквиру нити из које се сигнализира.
- `Qt::QueuedConnection`: Слот се позива када се контрола врати петљи догађаја нити примаоца и он се извршава у оквиру нити примаоца.
- `Qt::BlockingQueuedConnection`: Овај тип конекције је исти као `Qt::QueuedConnection`, са изузетком да се нит из које се сигнализира блокира све док се слот не заврши. Овај тип конекције се не сме користити ако прималац живи у нити из које се сигнализира или ће доћи до мртве петље у нити.
- `Qt::UniqueConnection`: Ово је заставица која се може комбиновати са било којим претходним типовима конекције помоћи битовског ИЛИ. Када је `Qt::UniqueConnection` постављен, позив метода `QObject::connect()` ће пасти ако већ постоји иста конекција (тј. ако је исти сигнал већ повезан на исти слот за исти пар објеката).

Хајде да прво имплементирамо класу `PiThread` која дефинише нит која ће израчунавати вредност броја π. Ова класа мора да наследи класу `QThread`, да садржи макро `Q_OBJECT` и да превазиђе метод `run()`. Додатно, декларишемо и два сигнала. Сигнал `progressIsMade(QString)` биће емитован када се направи наредни прогрес од 10% у симулацији и уз њега придружујемо ниску која садржи проценат који треба да буде приказан у виџету листе. Сигнал `piIsCalculated(double)` биће емитован када нит заврши симулацију и уз њега придружујемо апроксимацију броја π.

Код 3: Датотека PiThread.h

```
#ifndef PITHREAD_H
#define PITHREAD_H

#include <QThread>
```

```
#include <QString>

class PiThread : public QThread
{
    Q_OBJECT
public:
    PiThread(QObject *parent = nullptr);

protected:
    void run() override;

signals:
    void progressIsMade(QString percentage);
    void piIsCalculated(double piValue);
};

#endif // PITHREAD_H
```

Код 4: Датотека PiThread.cpp

```
#include "PiThread.h"

#include <QRandomGenerator>

PiThread::PiThread(QObject *parent)
    : QThread(parent)
{
}

void PiThread::run()
{
    auto hits = 0u, count = 0u;
    const auto iterations = 100'000'000u;
    const auto iterUpdate = 10'000'000u;

    while (count < iterations)
    {
        const auto x = QRandomGenerator::global()->generateDouble() * 2 - 1;
        const auto y = QRandomGenerator::global()->generateDouble() * 2 - 1;

        if (x * x + y * y <= 1.0)
        {
            ++hits;
        }
        ++count;

        // Kada je ostvaren dovoljni progres, emitujemo dogadjaj main niti da azurira prikaz.
        // Primetimo da ce ovoga puta svaki procenat biti prikazan u ispravnom trenutku,
        // a ne samo na kraju, kao u prethodnom primeru.
        if (count % iterUpdate == 10)
        {
            const auto percentage = QString::number(count / iterUpdate * 10u) + "%";
            emit progressIsMade(percentage);
        }
    }
```

```
    auto pi = 4.0 * hits / count;

    // Kada smo zavrsili izracunavanje, emitujemo dobijenu vrednost
    emit piIsCalculated(pi);
}
```

У главном прозору сада имамо нешто више слот метода. Слот `calculatePiValue()` се и даље позива када корисник кликне на дугме, али овога пута он само припрема контроле, креира и започиње нит и извршава одговарајуће конекције сигнала и слотова. Слот `onProgressIsMade(QString)` се позива када нит пошаље сигнал за прогрес и он само додаје нову ниску и вицет листе. Коначно, слот `onPiIsCalculated(double)` се позива ката нит пошаље сигнал за крај симулације и он исписује резултат симулације у једнолинијско текстуално поље и ресетује стање одговарајућих контрола у ГКИ.

Код 5: Датотека MainWindow.h

```
#ifndef MAINWINDOW_H
#define MAINWINDOW_H

#include <QWidget>

QT_BEGIN_NAMESPACE
namespace Ui { class MainWindow; }
QT_END_NAMESPACE

class MainWindow : public QWidget
{
    Q_OBJECT

public:
    MainWindow(QWidget *parent = nullptr);
    ~MainWindow();

private slots:
    void calculatePiValue();
    void onProgressIsMade(QString percentage);
    void onPiIsCalculated(double piValue);

private:
    Ui::MainWindow *ui;
};
#endif // MAINWINDOW_H
```

Код 6: Датотека MainWindow.cpp

```
#include "MainWindow.h"
#include "ui_MainWindow.h"
#include "PiThread.h"

MainWindow::MainWindow(QWidget *parent)
    : QWidget(parent)
    , ui(new Ui::MainWindow)
{
    ui->setupUi(this);

    connect(ui->pbStartCalculating, &QPushButton::clicked, this, &MainWindow::calculatePiValue);
}

MainWindow::~MainWindow()
```

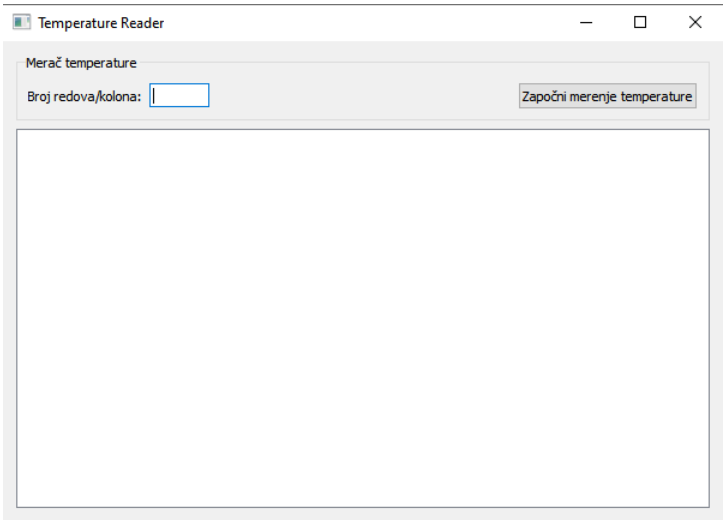
```
{
    delete ui;
}

void MainWindow::calculatePiValue()
{
    ui->pbStartCalculating->setEnabled(false);

    // Cistimo widget za naredni prikaz progresa
    ui->lwProgress->clear();

    // Kreiramo nit koja ce racunati vrednost broja Pi
    PiThread *thread = new PiThread(this);

    // Povezujemo odgovarajuce signale i slotove:
    // 1. Kada nit emituje da smo napravili progres,
    // treba da azuriramo progres u widgetu
    connect(thread, &PiThread::progressIsMade, this, &MainWindow::onProgressIsMade);
    // 2. Kada nit emituje da je zavrsila za izracunavanjem vrednosti,
    // treba da tu vrednost prikazemo
    connect(thread, &PiThread::piIsCalculated, this, &MainWindow::onPiIsCalculated);
    // 3. Kada je nit zavrsena,
    // potrebno ju je unistiti
    connect(thread, &PiThread::finished, thread, &PiThread::deleteLater);

    // Konacno, pokrecemo nit kako bi se izvrsila
    thread->start();
}

void MainWindow::onProgressIsMade(QString percentage)
{
    ui->lwProgress->addItem(percentage);
}

void MainWindow::onPiIsCalculated(double piValue)
{
    ui->lePi->setText(QString::number(piValue));
    ui->pbStartCalculating->setEnabled(true);
}
```

Пример 3 - TemperatureThreaded

Направити нови Qt5 пројекат назива TemperatureThreaded. Главни прозор назвати `TemperatureReader` и подесити да наследи `QWidget`. Креирати ГКИ као на слици у наставку. Користити `QVBoxLayout` за корени `QWidget`. Убацити једну групу контрола `QGroupBox`, поставити јој наслов на „Мерач температуре”, поредак контрола на `QHBoxLayout` и у њу сместити редом наредне контроле:

- `QLabel` са текстом „Број редова/колона:”
- `QLineEdit` назива `leNumOfRc` и фиксне ширине 50px.
- `Spacer` хоризонталне оријентације
- `QPushButton` назива `pbStartReading` са текстом „Започни мерење температуре”

Испод ове групе контрола сместити `QTableWidget` назива `twRoomMatrix`.



Имплементацију ћемо започети креирањем класе `Room` која ће чувати информације о соби чију температуру меримо. Додајмо нову класу `Room` у Qt пројекат. Собу ћемо реализовати као дводимензионалну матрицу бројева у покретном зарезу (температура једне ћелије), па је потребно да додамо одговарајући атрибут класи. Изменимо и конструктор тако што ћемо проследити број редова/колона на основу којег ћемо иницијализовати температуру у соби. Потребно је да забранимо копирање и померање собе, па бришемо конструкторе копирања и померања, као и одговарајуће операторе доделе. Додатно, имплементирајмо методе `numberOfRowsAndColumns()` и `cellValue()` за дохватање броја редова/колона и дохватање вредности конкретног поља, редом.

Код 7: Датотека Room.h

```
#ifndef ROOM_H
#define ROOM_H

#include <QObject>
#include <QVector>

class Room : public QObject
{
    Q_OBJECT
public:
    explicit Room(int n, QObject *parent = nullptr);

    // Zabranjujemo kopiranje i pomeranje sobe
    Room(const Room &) = delete;
    Room& operator=(const Room &) = delete;
    Room(Room &&) = delete;
    Room& operator=(Room &&) = delete;

    int numberOfRowsAndColumns() const;
    double cellValue(int i, int j) const;

private:
    QVector<QVector<double>> _matrix;
};

#endif // ROOM_H
```

Код 8: Датотека Room.cpp

```
#include "Room.h"

#include <QRandomGenerator>
```

```
Room::Room(int n, QObject *parent)
    : QObject(parent)
{
    for (auto i = 0; i < n; ++i)
    {
        QVector<double> row;
        for (auto j = 0; j < n; ++j)
        {
            // Generisemo nasumicnu temperaturu u intervalu [15, 30)
            const auto randomTemp = QRandomGenerator::global()->generateDouble() * 15 + 15;
            row.push_back(randomTemp);
        }
        _matrix.push_back(row);
    }
}

int Room::numberOfRowsAndColumns() const
{
    return _matrix.size();
}

double Room::cellValue(int i, int j) const
{
    return _matrix[i][j];
}
```

Вратимо се сада на класу `TemperatureReader`. Имплементирати и јавни слот који ће се позвати када корисник притисне на дугме. Нека за сада овај слот само проверава унос у једнолинијском пољу.

Код 9: Датотека TemperatureReader.h

```
#ifndef TEMPERATUREREADER_H
#define TEMPERATUREREADER_H

#include <QWidget>

QT_BEGIN_NAMESPACE
namespace Ui { class TemperatureReader; }
QT_END_NAMESPACE

class TemperatureReader : public QWidget
{
    Q_OBJECT

public:
    TemperatureReader(QWidget *parent = nullptr);
    ~TemperatureReader();

public slots:
    void onPbStartReading();

private:
    Ui::TemperatureReader *_ui;
};
#endif // TEMPERATUREREADER_H
```

Код 10: Датотека TemperatureReader.cpp

```
#include "TemperatureReader.h"
#include "ui_TemperatureReader.h"

#include <QMessageBox>

TemperatureReader::TemperatureReader(QWidget *parent)
    : QWidget(parent)
    , _ui(new Ui::TemperatureReader)
{
    _ui->setupUi(this);

    // Povezivanje signala i slot-metoda u okviru formulara
    QObject::connect(_ui->pbStartReading, &QPushButton::clicked,
                     this, &TemperatureReader::onPbStartReading);
}

TemperatureReader::~TemperatureReader()
{
    // Brisemo auto-generisanu klasu koja implementira formular
    delete _ui;
}

void TemperatureReader::onPbStartReading()
{
    // Dohvatamo broj iz QLineEdit polja i pokusavamo da ga parsiramo u celi broj
    const auto number_str = _ui->leNumOfRc->text();
    bool parsed;
    const auto rcNumber = number_str.toInt(&parsed);

    // Ako je korisnik uneo nesto sto nije broj ili je uneo nepozitivan broj,
    // onda mu treba prikazati poruku da ispravi unos
    if (!parsed || rcNumber <= 0)
    {
        QMessageBox msgBox;
        msgBox.setText("Morate uneti ispravan pozitivan broj");
        msgBox.exec();
        return;
    }

    // Iskljucujemo sve kontrole za unos podataka
    _ui->groupBox->setDisabled(true);
}
```

Додајмо овој класи атрибут који представља показивач на објекат класе `Room`. Одабрали смо показивач пошто ће се објекат класе креирати динамички, када корисник унесе број редова/колона и притисне на дугме. Имплементираћемо и додатни метод `populateTableWidget()` који ће иницијализовати `QTableWidget`. Вицет табеле се иницијализује тако што се прво дефинише број врста и колона, а затим се сваком пољу табеле додељује по један објекат класе `QTableWidgetItem`. Ајтемима табеле није потребно поставити родитеља, зато што ће се то аутоматски урадити када се ајтем табеле постави као поље вицета табеле. Оно о чему треба водити рачуна јесте да вицет табеле неће аутоматски обрисати стари ајтем ако се у неком пољу постави нови ајтем. Због тога се попуњавање табеле ради само једном, а измена вредности ајтема се врши позивањем метода `setText()` над ајтемима у табели. Ово ће нам бити значајно касније, када будемо ажурирали податке у табели.

Код 11: Датотека TemperatureThreaded.h (ажурирања)

```
// ...
class Room;

class TemperatureReader : public QWidget
{
// ...
public:
    // Ovde je neophodno da vracamo reference kako ne bi doslo do kopiranja u nitima!!!
    // Obratite paznju da suvisna kopiranja ovog tipa na ispitu nose negativne poene.
    Room &getRoom();
private:
    void populateTableWidget();

    Room *_room;
// ...
};
```

Код 12: Датотека TemperatureThreaded.cpp (ажурирања)

```
// ...
Room &TemperatureReader::getRoom()
{
    return *_room;
}

void TemperatureReader::onPbStartReading()
{
    // ...

    // Kreiramo objekat sobe na osnovu broja celija u redu/koloni
    // i pripremamo QTableWidget za prikazivanje
    _room = new Room(rcNumber, this);
    populateTableWidget();
}

void TemperatureReader::populateTableWidget()
{
    const auto rcNumber = _room->numberOfRowsAndColumns();

    _ui->twRoomMatrix->setRowCount(rcNumber);
    _ui->twRoomMatrix->setColumnCount(rcNumber);
    for (auto i = 0; i < rcNumber; ++i)
    {
        for (auto j = 0; j < rcNumber; ++j)
        {
            auto cellValue = _room->cellValue(i, j);
            // QTableWidgetItem-i se kreiraju bez roditelja,
            // ali kad se dodaju nekom QTableWidget-u, onda on preuzima vlasnistvo nad njima
            auto tableItem = new QTableWidgetItem(QString::number(cellValue));
            _ui->twRoomMatrix->setItem(i, j, tableItem);
        }
    }
}
// ...
```

Пре него што креирамо и покренемо нити, неопходно је да креирамо класу која представља једну нит. Поразмислимо шта је свакој нити потребно од информација да би исправно радила. Једна нит би требало да израчуна нову вредност температуре у једној ћелији собе. С обзиром да објекте класе `Room` не можемо копирати нити померати, да би ћелија добила информацију о конкретној ћелији, али и о њеним суседним ћелијама, потребно је да нити проследимо индексе те ћелије у матрици собе. Додатно, потребно је проследимо показивач на прозор (`TemperatureReader`), како би нит могла да дохвати референцу на објекат собе. Један начин да се ово уради јесте да прозор проследимо као родитељски објекат за нит, што и има смисла из погледа имплементације хијерархије `QObject`-а. Дакле, креирамо нову класу `CellThread`.

Код 13: Датотека CellThread.h

```
#ifndef CELLTHREAD_H
#define CELLTHREAD_H

#include <QThread>

class CellThread : public QThread
{
    Q_OBJECT

public:
    CellThread(const int i, const int j, QObject *parent = nullptr);

protected:
    void run() override;

signals:
    void threadFinished(int i, int j);

private:
    int _i;
    int _j;
};
#endif // CELLTHREAD_H
```

Код 14: Датотека CellThread.cpp

```
#include "CellThread.h"
#include "TemperatureReader.h"
#include "Room.h"

CellThread::CellThread(const int i, const int j, QObject *parent)
    : QThread(parent)
    , _i(i)
    , _j(j)
{
}

void CellThread::run()
{
    // Znamo da ce nit kao roditelja dobiti pokazivac na prozor,
    // pa mozemo izvrsiti dinamicko kastovanje u prozor,
    // kako bismo mogli da dohvatimo objekat sobe
    TemperatureReader *window = qobject_cast<TemperatureReader *>(parent());

    // Ponovo, obratite paznju da se cuva referenca da bi se izbeglo kopiranje.
    auto &room = window->getRoom();
```

```
    // Racunamo razliku u temperaturi izmedju tekuce celije i okolnih celija
    auto tempDiff = 0.0;

    // Dodajemo razliku u temperaturi od okolnih celija
    tempDiff += room.getTemperatureDiffFromCellAbove(_i, _j);
    tempDiff += room.getTemperatureDiffFromCellBelow(_i, _j);
    tempDiff += room.getTemperatureDiffFromCellLeft(_i, _j);
    tempDiff += room.getTemperatureDiffFromCellRight(_i, _j);

    // Azuriramo temperaturu
    room.updateNewTemperatureForCell(_i, _j, tempDiff);

    // Emitujemo da smo zavrsili posao
    emit threadFinished(_i, _j);
}
```

Наравно, неопходно је да имплементирамо нове методе у класи Room које се користе у овој имплементацији.

Код 15: Датотека Room.h (ажурирања)

```
// ...
class Room : public QObject
{
// ...
public:
    double getTemperatureDiffFromCellAbove(int i, int j) const;
    double getTemperatureDiffFromCellBelow(int i, int j) const;
    double getTemperatureDiffFromCellLeft(int i, int j) const;
    double getTemperatureDiffFromCellRight(int i, int j) const;
    void updateNewTemperatureForCell(int i, int j, double tempDiff);
// ...
};
#endif // ROOM_H
```

Код 16: Датотека Room.cpp (ажурирања)

```
// ...
double Room::getTemperatureDiffFromCellAbove(int i, int j) const
{
    if (i <= 0)
    {
        return 0.0;
    }
    return TEMP_COEFFICIENT * (_matrix[i-1][j] - _matrix[i][j]);
}

double Room::getTemperatureDiffFromCellBelow(int i, int j) const
{
    if (i >= _matrix.size() - 1)
    {
        return 0.0;
    }
    return TEMP_COEFFICIENT * (_matrix[i+1][j] - _matrix[i][j]);
}

double Room::getTemperatureDiffFromCellLeft(int i, int j) const
{
```

```
    if (j <= 0)
    {
        return 0.0;
    }
    return TEMP_COEFFICIENT * (_matrix[i][j-1] - _matrix[i][j]);
}

double Room::getTemperatureDiffFromCellRight(int i, int j) const
{
    if (j >= _matrix.size() - 1)
    {
        return 0.0;
    }
    return TEMP_COEFFICIENT * (_matrix[i][j+1] - _matrix[i][j]);
}

void Room::updateNewTemperatureForCell(int i, int j, double tempDiff)
{
    _matrix[i][j] += tempDiff;
}
// ...
```

Сада можемо да довршимо имплементацију метода у класи прозора за покретање нити и оперисање над резултатима тих нити. Потребно је да додамо и нови атрибут класе `TemperatureReader` који означава тајмер који ће одбројавати 1,5 секунду до новог покретања нити.

Код 17: Датотека TemperatureReader.h (ажурирања)

```
// ...
#include <QTimer>
// ...
class TemperatureReader : public QWidget
{
// ...
public slots:
    void onThreadFinished(int i, int j);

private slots:
    void startThreads();

private:
    QTimer _timer;
// ...
};
#endif // TEMPERATUREREADER_H
```

Датотека `TemperatureReader.cpp`:

```
// ...
TemperatureReader::TemperatureReader(QWidget *parent)
    // ...
    , _timer(this)
{
    // ...
}

void TemperatureReader::onPbStartReading()
{
```

```
    // ...

    // Pravimo niti koje ce racunati vrednost jednog polja u matrici na osnovu susednih polja.
    startThreads();

    // Postavljamo tajmer koji ce pokrenuti niti na svaku 1.5 sekundu
    QObject::connect(&_timer, &QTimer::timeout,
                     this, &TemperatureReader::startThreads);
    _timer.start(1500);
}

void TemperatureReader::startThreads()
{
    for (auto i = 0; i < _room->numberOfRowsAndColumns(); ++i)
    {
        for (auto j = 0; j < _room->numberOfRowsAndColumns(); ++j)
        {
            auto thread = new CellThread(i, j, this);
            QObject::connect(thread, &CellThread::threadFinished,
                             this, &TemperatureReader::onThreadFinished);
            QObject::connect(thread, &CellThread::finished,
                             thread, &CellThread::deleteLater);
            thread->start();
        }
    }
}

void TemperatureReader::onThreadFinished(int i, int j)
{
    // Azuriramo prikaz matrice u prikazu.
    // S obzirom da smo u metodu populateTableWidget() konstruisali tabelu i popunili je item-ima,
    // sada mozemo jednostavno da menjamo sadrzaj tih item-a.
    auto cellValue = _room->cellValue(i, j);
    _ui->twRoomMatrix->item(i, j)->setText(QString::number(cellValue));
}
// ...
```

Синхронизација нити помоћу QMutex и QMutexLocker

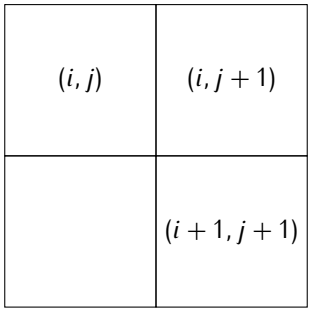
Оно о чему нисмо дискутовали до сада то су проблеми који се могу јавити када више нити конкурентно приступа истим подацима. У том случају, могу да се јаве две врсте проблема:

- Проблем конкурентног читања-писања се јавља када постоји макар једна нит која приступа податку за читање и макар једна нит која приступа истом том податку за писање, при чему се ове операције извршавају у исто време.
- Проблем конкурентног писања-писања се јавља када постоје макар две нити које приступају истом податку за писање, при чему се ове операције извршавају у исто време.

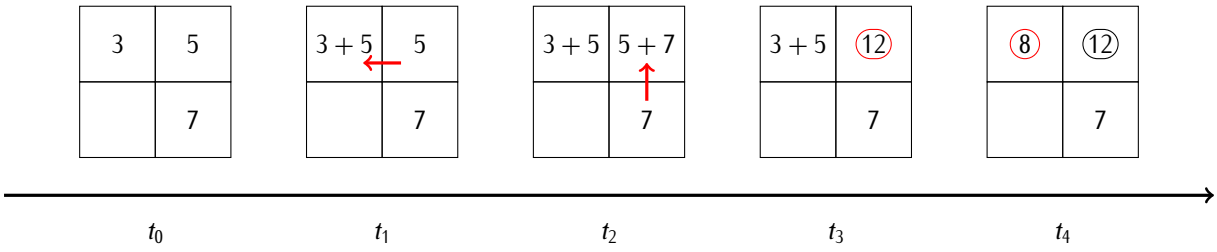
У нашој апликацији имамо појаву првог проблема. Нека имамо ситуацију да наредне нити израчунавају вредности нових температура одговарајућих поља: нит H1 израчунава за (i, j) и нит H2 израчунава за $(i, j + 1)$. Нека имамо наредно конкурентно извршавање ових нити:

- У тренутку t_1, H1 чита вредност суседног поља $(i, j + 1)$.
- У тренутку t_2, H2 чита вредност суседног поља $(i + 1, j + 1)$.
- У тренутку t_3, H2 пише вредност свог поља $(i, j + 1)$.

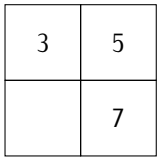
- У тренутку t_4, H1 пише вредност свог поља (i, j).



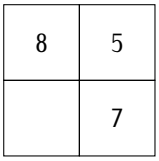
У оваквом моделу извршавања, H1 је израчунала нову вредност свог поља на основу температуре која је прочитана, али је у међувремену H2 ажурирала вредност тог истог поља, што доводи до некоректног ажурирања вредности поља (i, j). Дакле, неопходно је синхронизовати рад нити.



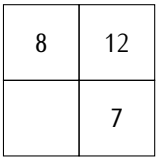
Посматрано из перспективе система у којем бисмо очекивали да нити извршавају атомичко мењање матрице, након завршетка H1, очекивали бисмо да резултат буде као на наредној слици лево, а добили смо резултат десно, што очигледно није идентично.



Почетно стање



Очекивано стање



Добијено стање

Најједноставнији начин да се то уради јесте коришћењем мутекса. Библиотека Qt нуди две класе за рад са мутексима и то су `QMutex` и `QMutexLocker`. Прва класа се користи за закључавање одговарајућег ресурса, а друга се користи за управљање мутексима. Наиме, класа `QMutex` се користи тако што се над њом позивају методи `lock()` и `unlock()`. Када једна нит покуша да закључа неки мутекс, постоје два случаја која се могу десити:

- Ако нема ниједна нит која је претходно закључала тај исти мутекс, онда га текућа нит успешно закључава и може да настави даље са извршавањем.
- Ако постоји макар једна нит која је претходно закључала тај исти мутекс, онда се текућа нит успављује све док та друга нит не откључа тај мутекс.

Проблем који се може јавити јесте да нит закључа мутекс и да заборави да га откључа (или се деси нешто због чега мутекс остане закључан, на пример, испали се изузетак између позива метода `lock()` и `unlock()`). Уколико нит не откључа мутекс, онда ће остале нити чекати бесконачно дуго на откључавање мутекса и тако се добија **мртва петља**. Да бисмо избегли ову ситуацију, уместо позива метода `lock()` и `unlock()`, креирамо објекат класе `QMutexLocker` (најчешће са локалним аутоматским животним веком) и прослеђујемо му показивач на мутекс. Он ће користити механизам RAII за закључавање (приликом конструкције) и откључавање (приликом деструкције) тог мутекса.

Направити пример за наредно понашање: У системима који омогућују **успављивање** нити, може се јавити још један проблем. Уколико нит закључа мутекс, па се затим успава, мутекс ће се откључати, па ће нека друга нит моћи да га закључа. Када се прва нит поново активира, онда покушава да поново закључа исти мутекс. Решење је да се успављивање нити врши пре закључавања, кад год је то могуће. Алтернативно, користити `QTimer`, сигнале и слотове уместо успављивања.

Хајде да ажурирамо класе тако да користе описани механизам за синхронизацију нити. Прво ажурирамо класу прозора у оквиру којег ћемо чувати један мутекс и имплементирати метод за његово дохватање изван те класе.

Код 18: Датотека TemperatureReader.h (ажурирања)

```
// ...
#include <QMutex>
// ...

class TemperatureReader : public QWidget
{
// ...
public:
    // Ovde je neophodno da vracamo reference kako ne bi doslo do kopiranja u nitima!!!
    // Obratite paznju da suvisna kopiranja ovog tipa na ispitu nose negativne poene.
    QMutex &getMutexForRoom();

private:
    QMutex _mutexForRoom;
};
#endif // TEMPERATUREREADER_H
```

Датотека `TemperatureReader.cpp`:

```
// ...
#include <QMutexLocker>

QMutex &TemperatureReader::getMutexForRoom()
{
    return _mutexForRoom;
}

void TemperatureReader::startThreads()
{
    // Ovde pristupamo objektu sobe za citanje broja kolona,
    // medjutim, moguce je da neka nit jos uvek radi,
    // pa imamo konkurentno citanje, te moramo sinhronizovati niti.
    QMutexLocker lock(&_mutexForRoom);

    // ...
}

void TemperatureReader::onThreadFinished(int i, int j)
{
    // I ovde imamo konkurentno citanje, te moramo sinhronizovati niti.
    QMutexLocker lock(&_mutexForRoom);

    // ...
}
// ...
```

Слично овоме, потребно је допунити имплементацију метода `run()` у нити.

Код 19: Датотека CellThread.cpp (ажурирања)

```
void CellThread::run()
{
    // Znamo da ce nit kao roditelja dobiti pokazivac na prozor,
```

```
    // pa mozemo izvrsiti dinamicko kastovanje u prozor,
    // kako bismo mogli da dohvatimo objekat sobe i muteks radi sinhronizacije
    TemperatureReader *window = qobject_cast<TemperatureReader *>(parent());

    // Zakljucavamo muteks kako bismo sprecili da druge niti konkurentno pristupaju podacima
    QMutexLocker lock(&window->getMutexForRoom());

    // ...

    // Dodajemo razliku u temperaturi od okolnih celija (konkurentno citanje)
    tempDiff += room.getTemperatureDiffFromCellAbove(_i, _j);
    // ...

    // Azuriramo temperaturu (konkurentno pisanje)
    room.updateNewTemperatureForCell(_i, _j, tempDiff);

    // ...
}
```