

МИНИСТЕРСТВО ЦИФРОВОГО РАЗВИТИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ
ОБРАЗОВАТЕЛЬНОЕ

УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
УНИВЕРСИТЕТ ТЕЛЕКОММУНИКАЦИЙ им. М. А.

Бонч-Бруевича»
(СПБГУТ им. М. А. Бонч-Бруевича)

Кафедра программной инженерии и вычислительной техники

Курсовая работа
по дисциплине: Объектно ориентированное программирование

Выполнил:
студент группы ИКПИ-33
Коломиец Александр Романович

Санкт-Петербург
2024 г.

Оглавление

1. Описание задачи	2
2. Описание структуры данных	3
3. Анализ задачи	5
4. Использование Qt для создания пользовательского интерфейса	7
5. Анализ файлов и классов	9
6. Инструкция для пользователя	12
7. Выводы	14

1. Описание задачи

Данная курсовая работа посвящена созданию приложения для управления базой данных фильмов без использования стандартных контейнеров STL. Основная цель – разработать программу, которая позволяет пользователю:

- Добавлять, удалять, редактировать записи о фильмах;
- Выполнять поиск по заданным критериям (например, по режиссёру или году);
- Сохранять данные на диск и загружать их из файла;
- Объединять две разные базы данных фильмов.

Для реализации хранения данных создаётся собственный контейнер, написанный «с нуля», с применением динамического выделения памяти и ручного управления размером массива. Интерфейс программы создан с использованием библиотеки Qt.

2. Описание структуры данных

Основой для базы данных служит собственный контейнер **MyContainer**. В отличие от подхода с использованием STL (**std::vector**), здесь вся функциональность реализуется вручную:

- **Хранение данных:**
Внутри контейнера хранится динамически выделяемый массив элементов типа **Movie**.
При добавлении новых фильмов, при необходимости, происходит реаллокация массива (выделение большего блока памяти, копирование существующих элементов и освобождение старого блока).
- **Итераторы:**
Реализованы простейшие итераторы, предоставляющие доступ к элементам массива. Итераторы позволяют последовательно обходить элементы контейнера.
- **Основные операции:**
 - Добавление элемента (**push_back**):
При добавлении нового фильма, если размер массива достигнут, контейнер увеличивает свой размер, выделяя больше памяти и копируя существующие элементы.
 - Удаление элемента по критерию (**erase_if**):
Для удаления элементов применяется перебор массива с последующим смещением элементов или уменьшением размера.
 - Доступ к элементам по индексу (**operator[]**), а также методы **size()** и **empty()**.

Данная структура данных не зависит от STL-контейнеров, что соответствует условию. Это усложняет реализацию, но даёт полную контроль над памятью и поведением.

Структура данных **Movie**:

- Название фильма (**QString**)
- Режиссёр (**QString**)
- Год выпуска (**int**)
- Жанр (**QString**)

Класс **Movie** предоставляет геттеры и сеттеры для этих полей, а также

средства для сериализации/десериализации данных (чтения и записи в поток).

3. Анализ задачи

Выбор собственной структуры данных

Отказ от использования STL-контейнеров приводит к необходимости ручной реализации динамического массива. Такой подход требует реализации:

- Логике увеличения размера массива;
- Логике удаления элементов;
- Собственных итераторов.

Преимущество такого подхода – полный контроль над поведением контейнера. Недостаток – увеличение сложности кода. Однако, в рамках учебной задачи это позволяет лучше понять механизмы работы со структурой данных.

Типы данных

Movie – основной тип, хранящий всю информацию о фильмах.

MyContainer<Movie> – контейнер, содержащий объекты **Movie**.

Над данными выполняются операции:

- Добавление (**addMovie**)
- Удаление (**removeMovie**)
- Редактирование (**editMovie**)
- Поиск (**searchByDirector**, **searchByYear**)
- Сохранение/загрузка (**saveToFile**, **loadFromFile**)
- Объединение двух баз данных (**mergeWith**)

4. Использование Qt для создания пользовательского интерфейса

Для графического интерфейса применяется библиотека Qt. Основные

используемые классы:

- **QApplication** – запуск приложения.
- **MainWindow** – создание главного окна.
- **QMenuBar, QMenu, QAction** – реализация меню для команд (File, Edit, Help).
- **QToolBar** – панель инструментов.
- **QStatusBar** – строка состояния для вывода статуса операций.
- **QTableWidget** – визуализация списка фильмов в табличном формате.
- **QInputDialog** – ввод строк, чисел.
- **MessageBox** – вывод информационных и предупредительных сообщений.
- **FileDialog** – выбор файлов для загрузки/сохранения.
- **QDialog** (производный класс **AboutDialog**) – окно «О программе».

Сигналы и слоты Qt связывают действия пользователя с методами программы. Например, нажатие пункта меню «Add Movie» вызывает соответствующий слот в **MainWindow**, который инициирует добавление фильма.

5. Анализ файлов и классов

- **main.cpp**:
Точка входа в программу: создаёт **QApplication**, экземпляр **MainWindow** и запускает цикл событий **app.exec()**.
- **movie.h / movie.cpp**:
Класс **Movie**:
 - Атрибуты: **title, director, year, genre**.
 - Методы: геттеры/сеттеры, операторы для сериализации (чтения/записи из **QDataStream**).

Подробное описание класса **Movie**

Назначение класса:

Класс `Movie` представляет собой модель данных, описывающую один фильм.

Атрибуты:

- `QString title;` – название фильма. Тип `QString` удобен для работы с Unicode-строками.
- `QString director;` – режиссёр фильма.
- `int year;` – год выпуска фильма. Целое число, например, 1995.
- `QString genre;` – жанр фильма (драма, комедия, фантастика и т.д.).

Конструкторы:

- Конструктор по умолчанию `Movie()` устанавливает значения атрибутов по умолчанию. Например, `year = 0`, а строки пустые.
- Конструктор с параметрами `Movie(const QString &title, const QString &director, int year, const QString &genre)` позволяет сразу создать объект фильма с заданными параметрами.

Методы доступа (геттеры):

- `QString getTitle() const;`
- `QString getDirector() const;`
- `int getYear() const;`
- `QString getGenre() const;`

Эти методы возвращают значения соответствующих полей. Они `const`, чтобы гарантировать, что вызов не модифицирует объект фильма.

Методы изменения (сеттеры):

- `void setTitle(const QString &newTitle);`

- `void setDirector(const QString &newDirector);`
- `void setYear(int newYear);`
- `void setGenre(const QString &newGenre);`

Эти методы меняют значения полей. Это обеспечивает инкапсуляцию и контроль над тем, как данные модифицируются.

Сериализация: Для сохранения и загрузки данных в файл реализованы оператор вывода и ввода в `QDataStream`:

- `QDataStream &operator<<(QDataStream &out, const Movie &m);`
- `QDataStream &operator>>(QDataStream &in, Movie &m);`

Благодаря сериализации можно сохранить объекты `Movie` в бинарный файл и затем их считать. При записи в поток записываются последовательно название, режиссёр, год и жанр. При чтении – эти данные считываются в том же порядке и устанавливаются в объект `Movie`.

- **mycontainer.h / mycontainer.cpp:**

Класс `MyContainer<T>` реализует динамический массив без использования STL:

- Указатель на динамически выделенный массив.
- Переменная размера и вместимости.
- `push_back` (добавление элемента с возможной реаллокацией памяти).
- `erase_if` (удаление по критерию).
- Итераторы `Iterator` и `ConstIterator`, позволяющие обходить элементы массива.
- `size()`, `empty()`, `operator[]` для управления данными.

- **moviecontainer.h / moviecontainer.cpp:**

Класс `MovieContainer`:

- Содержит `MyContainer<Movie>`.

- Реализует операции над базой данных: `addMovie`, `removeMovie`, `editMovie`, `searchByDirector`, `searchByYear`, `saveToFile`, `loadFromFile`, `mergeWith`.
- **mainwindow.h / mainwindow.cpp:**
Класс `MainWindow` (наследует `QMainWindow`):
 - Формирует GUI: меню, тулбар, таблицу.
 - Слоты для операций: добавление, удаление, редактирование, поиск, загрузка, сохранение, объединение.
 - `refreshTable` – обновляет отображение базы.
- **aboutdialog.h / aboutdialog.cpp:**
Класс `AboutDialog`:
 - Отображает информацию о программе в модальном окне.

6. Инструкция для пользователя

При запуске приложения отображается главное окно со следующими элементами:

- **Меню File:**
 - Load... – загрузить базу данных из файла;
 - Save... – сохранить текущую базу;
 - Merge... – объединить с другой базой.
- **Меню Edit:**
 - Add Movie – добавление нового фильма (ввод через диалоговые окна);
 - Remove Movie – удаление фильма по названию;
 - Edit Movie – редактирование данных существующего фильма;
 - Search Movie – поиск по режиссёру или году. Результат отображается в таблице.
- **Меню Help:**
 - About – информация о программе.
- **Панель инструментов** дублирует основные действия для удобного доступа.
- **Таблица** отображает текущие данные о фильмах.

Рекомендуется перед выходом сохранить изменения. При загрузке или объединении с другой базой данные обновляются в таблице.

7. Тестирование

Добавление фильма в список:

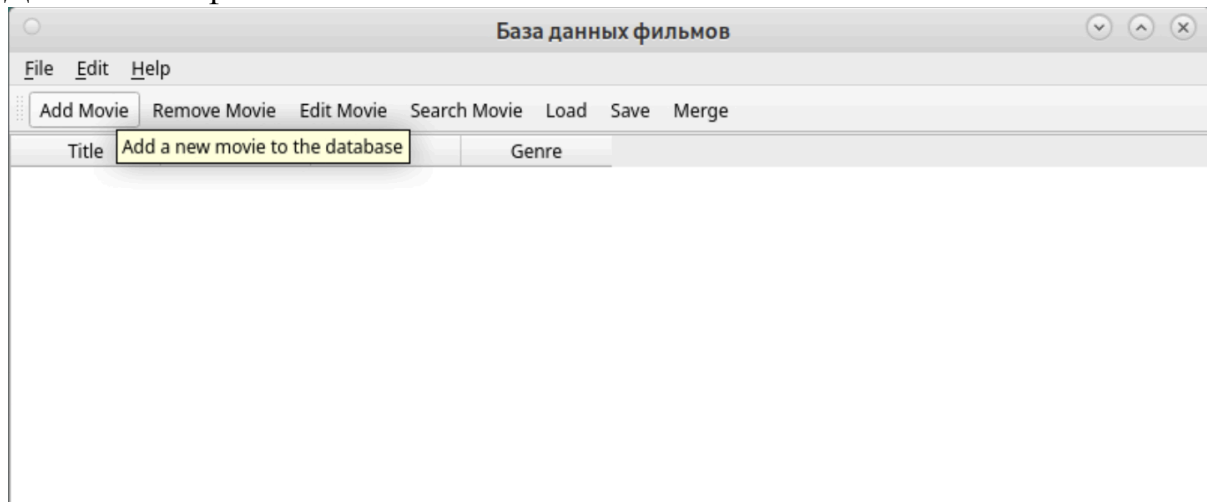


рисунок 1.

На *рисунке 1* представлена панель инструментов на которой мы можем выбрать 'Add Movie' чтобы добавить в список необходимый фильм. После нажатия на Add Movie у нас запрашивают данные на ввод в базу данных представлено на *рисунке 2*

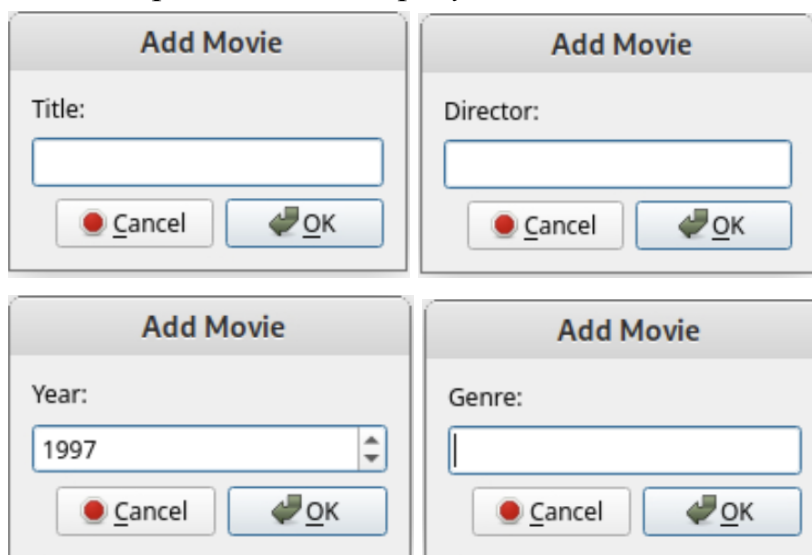


рисунок 2.

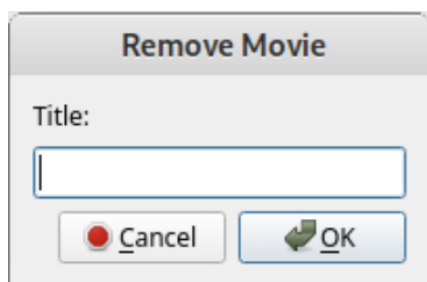
После ввода данных фильм добавляется в базу данных представлено на *рисунке 3*



File Edit Help				
Add Movie Remove Movie Edit Movie Search Movie Load Save Merge				
	Title	Director	Year	Genre
1	Титаник	Джеймс ...	1997	драма

рисунок 3.

Для удаления фильма из списка надо просто нажать на ‘Remove Movie’ в инструментарии и ввести название фильма что хотите удалить из списка представлено на *рисунке 4*

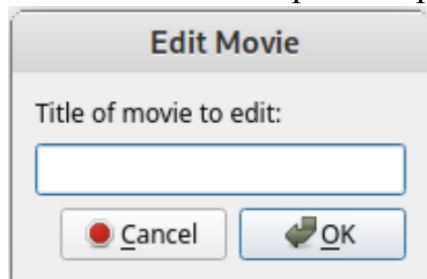


Remove Movie

Title:

рисунок 4.

Редактирование уже добавленного фильма также производится через поиск по названию фильма *рисунок 5* после ввода которого тебя просят ввести новые (все) данные для фильма в будущем можно добавить выбор того что хотите отредактировать в списке (режиссера/название/год выхода)



Edit Movie

Title of movie to edit:

рисунок 5.

Также есть возможность поиска фильма в инструментари 'Search Movie', при нажатии на эту кнопку нас спрашивают по каким критериям мы будем искать *рисунок 6* после выбора критерия мы просто вводим что хотим и находим нужный фильм (необходимо ввести полное название/имя)

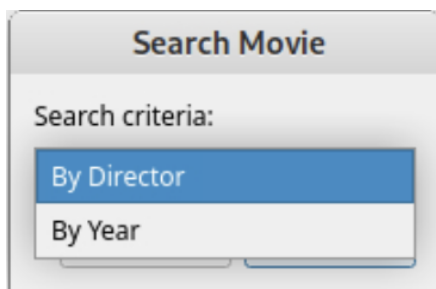


рисунок 6.

Возможность открыть сохраненную базу данных в инструментари 'Load' просто выбираем файл который нас интересует *рисунок 7*

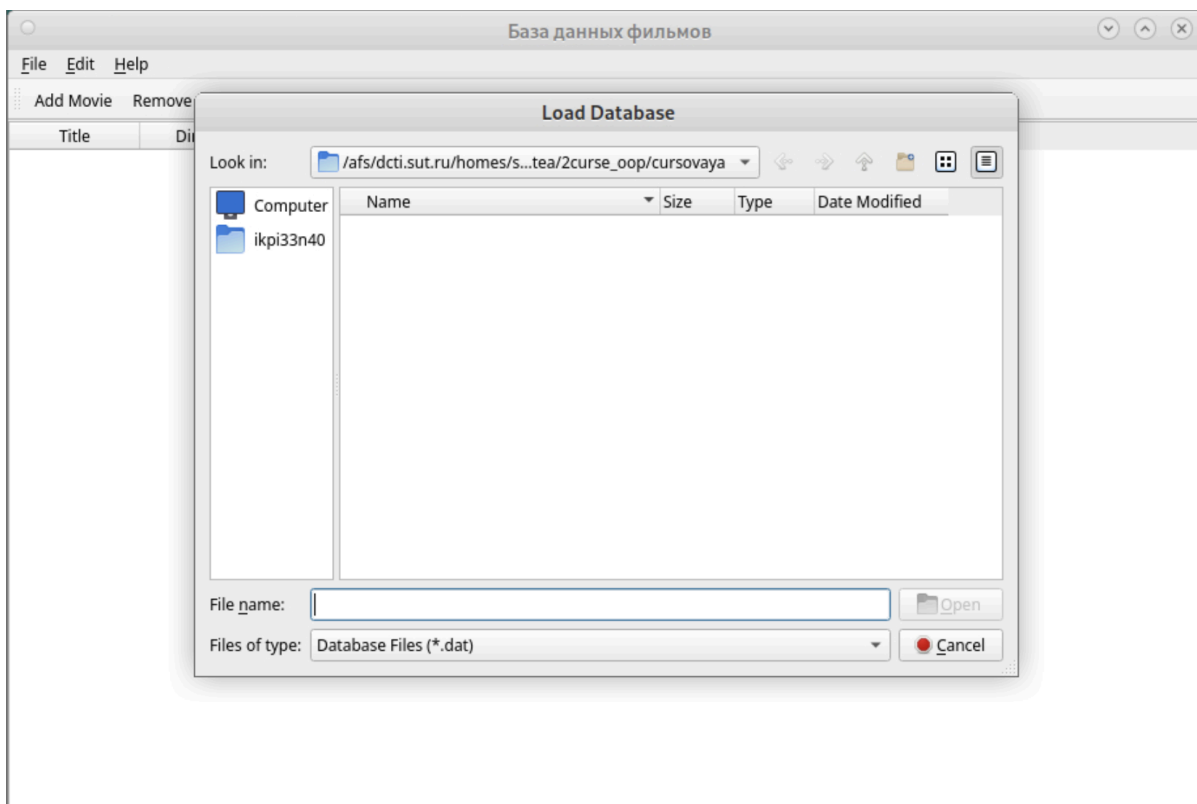


рисунок 7.

Тоже самое что и 'Load' есть возможность сохранения базы данных 'Save'
рисунок 8 вводим название файла и сохраняем где нам удобно

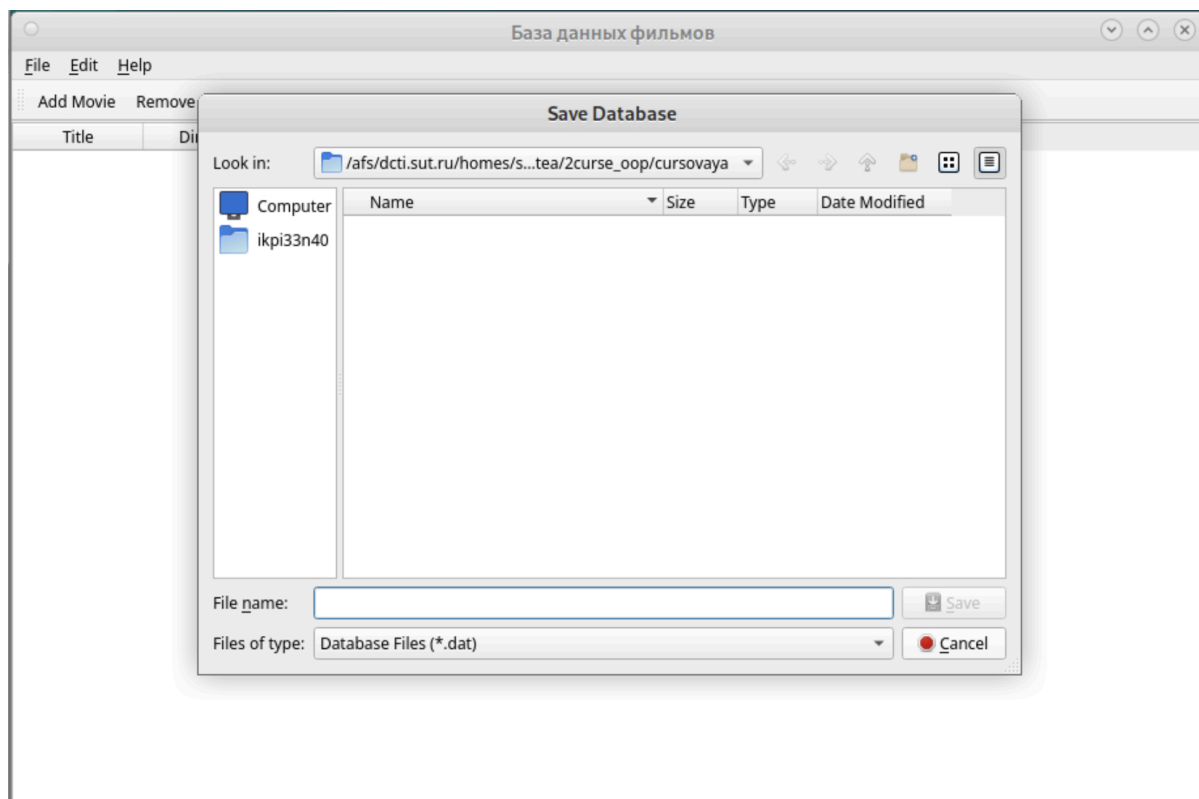
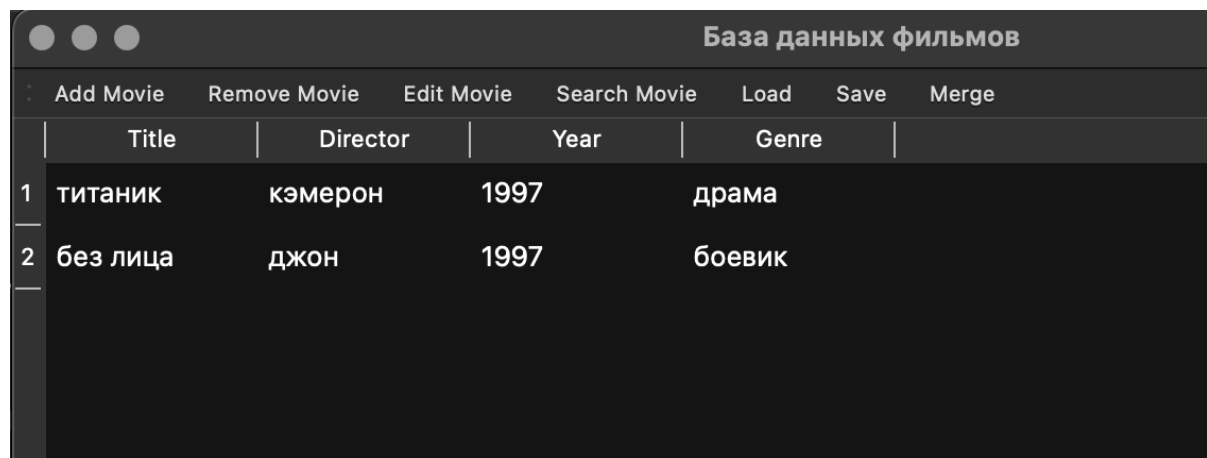


рисунок 8.

Последняя возможность это 'Merge' для соединения двух файлов надо открыть один из них к примеру 'титаник_тест.dat' и нажать на 'Merge' после чего тебя перекинет на файлы проекта для выбора базы данных с которой ты хочешь соединить эту, я выбрал 'безлица_тест.dat' и получившееся можно посмотреть на *рисунке 9*



рисунк 9.

8. Приложение

mainwindow.h:

```
#ifndef MAINWINDOW_H
#define MAINWINDOW_H

#include <QMainWindow>
#include <QMenu>
#include <QAction>
#include <QMenuBar>
#include <QToolBar>
#include <QStatusBar>
#include <QTableWidget>
#include "moviecontainer.h"

class MainWindow : public QMainWindow {
    Q_OBJECT

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

protected:
    void contextMenuEvent(QContextMenuEvent *event) override;
    void closeEvent(QCloseEvent *event) override;

private slots:
    void showAbout();
    void addMovie();
    void removeMovie();
    void searchMovie();
    void editMovie();
    void loadFromFile();
    void saveToFile();
    void mergeWithAnotherDB();
}
```

```

private:
    void createMenu();
    void createToolBar();
    void createContextMenu();
    void createStatusBar();
    void refreshTable();
    bool maybeSave();

    QMenu *fileMenu;
    QMenu *editMenu;
    QMenu *helpMenu;

    QAction *aboutAction;
    QAction *addAction;
    QAction *removeAction;
    QAction *searchAction;
    QAction *editAction;
    QAction *loadAction;
    QAction *saveAction;
    QAction *mergeAction;

    QToolBar *toolbar;
    QTableWidget *table;

    MovieContainer movieDB;
};

#endif // MAINWINDOW_H

```

mainwindow.cpp:

```

#include "mainwindow.h"
#include "aboutdialog.h"
#include <QInputDialog>
#include <QFileDialog>
#include <QMessageBox>
#include <QContextMenuEvent>
#include <QCloseEvent>
#include <QStringList>

MainWindow::MainWindow(QWidget *parent)
    : QMainWindow(parent)
{
    setWindowTitle(tr("База данных фильмов"));
    resize(800, 600);

    table = new QTableWidget(this);
    table->setColumnCount(4);
    QStringList headers;
    headers << tr("Title") << tr("Director") << tr("Year") << tr("Genre");
    table->setHorizontalHeaderLabels(headers);
    setCentralWidget(table);

    createMenu();
    createToolBar();
}

```

```

        createStatusBar();
        createContextMenu();
    }

MainWindow::~MainWindow() {}

void MainWindow::createMenu() {
    fileMenu = menuBar()->addMenu(tr("&File"));
    loadAction = new QAction(tr("Load..."), this);
    loadAction->setToolTip(tr("Load database from file"));
    saveAction = new QAction(tr("Save..."), this);
    saveAction->setToolTip(tr("Save database to file"));
    mergeAction = new QAction(tr("Merge..."), this);
    mergeAction->setToolTip(tr("Merge with another database file"));
    fileMenu->addAction(loadAction);
    fileMenu->addAction(saveAction);
    fileMenu->addAction(mergeAction);

    connect(loadAction, &QAction::triggered, this, &MainWindow::loadFromFile);
    connect(saveAction, &QAction::triggered, this, &MainWindow::saveToFile);
    connect(mergeAction, &QAction::triggered, this,
    &MainWindow::mergeWithAnotherDB);

    editMenu = menuBar()->addMenu(tr("&Edit"));
    addAction = new QAction(tr("Add Movie"), this);
    addAction->setToolTip(tr("Add a new movie to the database"));
    removeAction = new QAction(tr("Remove Movie"), this);
    removeAction->setToolTip(tr("Remove a movie by title"));
    searchAction = new QAction(tr("Search Movie"), this);
    searchAction->setToolTip(tr("Search movies by director or year"));
    editAction = new QAction(tr("Edit Movie"), this);
    editAction->setToolTip(tr("Edit an existing movie"));
    editMenu->addAction(addAction);
    editMenu->addAction(removeAction);
    editMenu->addAction(editAction);
    editMenu->addAction(searchAction);

    connect(addAction, &QAction::triggered, this, &MainWindow::addMovie);
    connect(removeAction, &QAction::triggered, this, &MainWindow::removeMovie);
    connect(searchAction, &QAction::triggered, this, &MainWindow::searchMovie);
    connect(editAction, &QAction::triggered, this, &MainWindow::editMovie);

    helpMenu = menuBar()->addMenu(tr("&Help"));
    aboutAction = new QAction(tr("About"), this);
    helpMenu->addAction(aboutAction);
    connect(aboutAction, &QAction::triggered, this, &MainWindow::showAbout);
}

void MainWindow::createToolBar() {
    toolbar = addToolBar(tr("Main Toolbar"));
    toolbar->addAction(addAction);
    toolbar->addAction(removeAction);
    toolbar->addAction(editAction);
    toolbar->addAction(searchAction);
    toolbar->addAction(loadAction);
}

```

```

        toolbar->addAction(saveAction);
        toolbar->addAction(mergeAction);
        toolbar->setToolTip(tr("Use the toolbar to manage movies"));
    }

void MainWindow::createContextMenu() {
    setContextMenuPolicy(Qt::DefaultContextMenu);
}

void MainWindow::contextMenuEvent(QContextMenuEvent *event) {
    QMenu menu(this);
    menu.addAction(addAction);
    menu.addAction(removeAction);
    menu.addAction(editAction);
    menu.addAction(searchAction);
    menu.addAction(loadAction);
    menu.addAction(saveAction);
    menu.addAction(mergeAction);
    menu.exec(event->globalPos());
}

void MainWindow::createStatusBar() {
    statusBar()->showMessage(tr("Ready"));
}

void MainWindow::showAbout() {
    AboutDialog dlg(this);
    dlg.exec();
}

void MainWindow::addMovie() {
    bool ok;
    QString title = QInputDialog::getText(this, tr("Add Movie"), tr("Title:"),
    QLineEdit::Normal, "", &ok);
    if (!ok || title.isEmpty()) return;
    QString director = QInputDialog::getText(this, tr("Add Movie"),
    tr("Director:"), QLineEdit::Normal, "", &ok);
    if (!ok) return;
    int year = QInputDialog::getInt(this, tr("Add Movie"), tr("Year:"), 2000, 1800,
    2100, 1, &ok);
    if (!ok) return;
    QString genre = QInputDialog::getText(this, tr("Add Movie"), tr("Genre:"),
    QLineEdit::Normal, "", &ok);
    if (!ok) return;

    Movie m(title, director, year, genre);
    movieDB.addMovie(m);
    refreshTable();
    statusBar()->showMessage(tr("Movie added"), 2000);
}

void MainWindow::removeMovie() {
    bool ok;
    QString title = QInputDialog::getText(this, tr("Remove Movie"), tr("Title:"),
    QLineEdit::Normal, "", &ok);

```



```

        if (!ok || title.isEmpty()) return;
        movieDB.removeMovie(title);
        refreshTable();
        statusBar()->showMessage(tr("Movie removed"), 2000);
    }

void MainWindow::editMovie() {
    bool ok;
    QString oldTitle = QInputDialog::getText(this, tr("Edit Movie"), tr("Title of
movie to edit:"), QLineEdit::Normal, "", &ok);
    if (!ok || oldTitle.isEmpty()) return;

    QString newTitle = QInputDialog::getText(this, tr("Edit Movie"), tr("New
Title:"), QLineEdit::Normal, "", &ok);
    if (!ok) return;
    QString newDirector = QInputDialog::getText(this, tr("Edit Movie"), tr("New
Director:"), QLineEdit::Normal, "", &ok);
    if (!ok) return;
    int newYear = QInputDialog::getInt(this, tr("Edit Movie"), tr("New Year:"),
2000, 1800, 2100, 1, &ok);
    if (!ok) return;
    QString newGenre = QInputDialog::getText(this, tr("Edit Movie"), tr("New
Genre:"), QLineEdit::Normal, "", &ok);
    if (!ok) return;

    Movie m(newTitle, newDirector, newYear, newGenre);
    if (movieDB.editMovie(oldTitle, m)) {
        refreshTable();
        statusBar()->showMessage(tr("Movie edited"), 2000);
    } else {
        QMessageBox::warning(this, tr("Edit Movie"), tr("Movie not found"));
    }
}

void MainWindow::searchMovie() {
    QStringList options;
    options << tr("By Director") << tr("By Year");
    bool ok;
    QString choice = QInputDialog::getItem(this, tr("Search Movie"), tr("Search
criteria:"), options, 0, false, &ok);
    if (!ok || choice.isEmpty()) return;

    if (choice == tr("By Director")) {
        QString director = QInputDialog::getText(this, tr("Search by Director"),
tr("Director:"), QLineEdit::Normal, "", &ok);
        if (!ok) return;
        MyContainer<Movie> result = movieDB.searchByDirector(director);
        table->clearContents();
        table->setRowCount(result.size());
        for (int row = 0; row < result.size(); ++row) {
            const Movie &mov = result[row];
            table->setItem(row, 0, new QTableWidgetItem(mov.getTitle()));
            table->setItem(row, 1, new QTableWidgetItem(mov.getDirector()));
            table->setItem(row, 2, new
QTableWidgetItem(QString::number(mov.getYear())));

```

```

        table->setItem(row, 3, new QTableWidgetItem(mov.getGenre()));
    }
    statusBar()->showMessage(tr("Search completed"), 2000);
} else {
    int year = QInputDialog::getInt(this, tr("Search by Year"), tr("Year:"),
    2000, 1800, 2100, 1, &ok);
    if (!ok) return;
    MyContainer<Movie> result = movieDB.searchByYear(year);
    table->clearContents();
    table->setRowCount(result.size());
    for (int row = 0; row < result.size(); ++row) {
        const Movie &mov = result[row];
        table->setItem(row, 0, new QTableWidgetItem(mov.getTitle()));
        table->setItem(row, 1, new QTableWidgetItem(mov.getDirector()));
        table->setItem(row, 2, new
QTableWidgetItem(QString::number(mov.getYear())));
        table->setItem(row, 3, new QTableWidgetItem(mov.getGenre()));
    }
    statusBar()->showMessage(tr("Search completed"), 2000);
}
}

void MainWindow::loadFromFile() {
    QString filename = QFileDialog::getOpenFileName(this, tr("Load Database"), "",
tr("Database Files (*.dat)"));
    if (filename.isEmpty()) return;
    if (movieDB.loadFromFile(filename)) {
        refreshTable();
        statusBar()->showMessage(tr("Database loaded"), 2000);
    } else {
        QMessageBox::warning(this, tr("Load Database"), tr("Cannot load the
file"));
    }
}

void MainWindow::saveToFile() {
    QString filename = QFileDialog::getSaveFileName(this, tr("Save Database"), "",
tr("Database Files (*.dat)"));
    if (filename.isEmpty()) return;
    if (movieDB.saveToFile(filename)) {
        statusBar()->showMessage(tr("Database saved"), 2000);
    } else {
        QMessageBox::warning(this, tr("Save Database"), tr("Cannot save the
file"));
    }
}

void MainWindow::mergeWithAnotherDB() {
    QString filename = QFileDialog::getOpenFileName(this, tr("Merge with
Database"), "", tr("Database Files (*.dat)"));
    if (filename.isEmpty()) return;
    MovieContainer temp;
    if (temp.loadFromFile(filename)) {
        movieDB.mergeWith(temp);
        refreshTable();
    }
}

```

```

        statusBar()->showMessage(tr("Databases merged"), 2000);
    } else {
        QMessageBox::warning(this, tr("Merge Database"), tr("Cannot load the file
to merge"));
    }
}

void MainWindow::refreshTable() {
    table->clearContents();
    const MyContainer<Movie> &all = movieDB.getAllMovies();
    table->setRowCount(all.size());
    for (int row = 0; row < all.size(); ++row) {
        const Movie &mov = all[row];
        table->setItem(row, 0, new QTableWidgetItem(mov.getTitle()));
        table->setItem(row, 1, new QTableWidgetItem(mov.getDirector()));
        table->setItem(row, 2, new
QTableWidgetItem(QString::number(mov.getYear())));
        table->setItem(row, 3, new QTableWidgetItem(mov.getGenre()));
    }
}

bool MainWindow::maybeSave() {
    // Можно реализовать проверку изменения данных и предложение сохранить.
    return true;
}

void MainWindow::closeEvent(QCloseEvent *event) {
    if (maybeSave()) {
        event->accept();
    } else {
        event->ignore();
    }
}
}

```

9. Выводы

В рамках данной курсовой работы была реализована система управления базой данных фильмов без использования стандартных контейнеров STL. Вместо этого создан собственный контейнер на основе динамического массива и ручного управления памятью. Данный подход позволил глубже понять принципы работы структур данных.

Приложение сочетает в себе принципы ООП, собственную структуру данных и удобный графический интерфейс на Qt. Результат – гибкая, расширяемая система, которую можно доработать, добавив новые функции и улучшив критерии поиска.