|  |  |
| --- | --- |
| **Gerb-BMSTU_01** | **Министерство науки и высшего образования Российской Федерации**  Калужский филиал  федерального государственного бюджетного  образовательного учреждения высшего образования  ***«Московский государственный технический университет имени Н.Э. Баумана (национальный исследовательский университет)»***  ***(КФ МГТУ им. Н.Э. Баумана)*** |

**ФАКУЛЬТЕТ** \_***ИУК «Информатика и управление»*\_\_**\_\_\_\_\_\_\_\_\_\_\_\_

**КАФЕДРА** \_\_***ИУК4 «Программное обеспечение ЭВМ, информационные технологии»***

**ЛАБОРАТОРНАЯ РАБОТА №3**

**«Перегрузка операторов»**

**ДИСЦИПЛИНА: «Высокоуровневое программирование»**

|  |  |  |
| --- | --- | --- |
| Выполнил: студент гр. ИУК4-22Б | | \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ ( Карельский М.К. )  (Подпись) (Ф.И.О.) |
| Проверил: | | \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ ( Козина А.В. )  (Подпись) (Ф.И.О.) |
| Дата сдачи (защиты):  Результаты сдачи (защиты): | | |
|  | - Балльная оценка:  - Оценка: | |
| Калуга , 2021 | | |

**Цель:** приобретение практических навыков и знаний по работе с перегрузкой операторов.

**Задачи:**

1. Изучить понятия оператора и что он из себя представляет;
2. Выяснить виды и способы перегрузки операторов;
3. Научиться применять перегрузку операторов на практике;
4. Изучить методы и случаи применения перегрузок;
5. Научиться соединять пользовательские объекты с потоками ввода/вывода;
6. Познакомиться с понятием функтора.

**Вариант 8**

**Задание:**

**Общая задача**

Вам будет предложено написать программу – «Автоматизированная система диалога (чат бот)». Которая будет включать следующий функционал:

* Ведение базы пользователей
  + Создание / удаление / редактирование записей
  + Сортировка / фильтрация
* Ведение базы диалогов, тем, интересов и напоминаний
* Возможность авторизации
* Создание файлов-отчётов и сохранения состояния

**Индивидуальные задания**

*Задача 1*

Добавьте в класс меню, который вы разрабатывали на прошлой Л/Р перегрузку оператора вывода (cout). Таким образом, при выполнении команды std::cout << menu; - где menu – это объект класса Menu, меню выводилось на экран. Данная перегрузка оператора должна использовать встроенный метод вывода меню на экран.

*Задача 2*

Добавьте в свой класс меню перегрузку оператора cin, таким образом, чтобы при использовании команды: std::cin >> menu, где menu – это объект вашего класса меню, меню выполняло считывание пользовательского ввода. Данная перегрузка оператора должна использовать встроенный метод считывания пользовательского ввода.

*Задача 3*

Создайте классы – сущности данных вашей программы: пользователь, диалог, тема, интерес, напоминание. Для класса: пользователь создайте общий класс родитель – человек и унаследуйтесь от него. У каждой сущности должно быть поле уникального идентификатора – id, которое представлять из себя тип: date, либо unsigned long int, а также дата создания.

*Задача 4*

В отдельном файле функций – app.cpp (app.h) создайте функции сортировки, фильтрации, удаления, добавления и редактирования сущностей данных (пользователь, диалог, тема, интерес, напоминание). Для этих функций будет создан интерфейс, который принимает массив нужных объектов (пользователи, напоминания, темы и т д) и редактирует его (либо сортирует записи, либо добавляет новые, либо редактирует нужную, либо удаляет заданную). Для функций фильтрации данных интерфейс будет принимать константный массив объектов и возвращать новый массив с отфильтрованными данными (придумайте, как можно возвращать вместе с этим массивом его размер).

*Задача 5*

Отдельно создайте функции добавления соответствующих записей (пользователь, напоминания, темы и т д).

*Задача 6*

Соедините созданные функции и модели данных в созданном меню и протестируйте прототип вашей программы. Устраните выявленные ошибки. На данном этапе ваша программа должна уметь: добавлять пользователя / диалог / темы / напоминание / интерес. Редактировать эти записи, удалять, сортировать по заданному полю и фильтровать по заданному полю. Все сущности программы – пока не должны быть жёстко связаны между собой.

**UML-диаграмма классов:**



**Рисунок 1.1.** UML-диаграмма классов



**Рисунок 2.2.** UML-диаграмма классов



**Рисунок 3.3.** UML-диаграмма классов

**Листинг:**

**Constants.h**

#ifndef CONSTANTS\_H

#define CONSTANTS\_H

namespace KMK

{

extern const int LENGTH\_OF\_FIELD;

extern const int MAXIMUM\_NUMBER\_OF\_DIGITS\_IN\_ID;

}

#endif // !CONSTANTS\_H

**Constants.cpp**

#include "Constants.h"

const int KMK::LENGTH\_OF\_FIELD = 255;

const int KMK::MAXIMUM\_NUMBER\_OF\_DIGITS\_IN\_ID = 10;

**TypeDefinitions.h**

#ifndef TYPE\_DEFINITIONS

#define TYPE\_DEFINITIONS\_H

namespace KMK

{

typedef unsigned long int Id;

typedef unsigned short int Iteration;

}

#endif // !TYPE\_DEFINITIONS

**Entity.h**

#ifndef ENTITY\_H

#define ENTITY\_H

#include "TypeDefinitions.h"

namespace KMK

{

class Entity

{

public:

struct Date

{

unsigned short day{};

unsigned short month{};

unsigned short year{};

};

Entity(Id id, Date date);

Id GetId();

Date GetDate();

void SetId(Id id);

void SetDate(Date date);

private:

Id m\_id{};

Date m\_date{};

};

}

#endif // !ENTITY\_H

**Entity.cpp**

#include "Entity.h"

using namespace KMK;

Entity::Entity(Id id, Date date)

{

m\_id = id;

m\_date = date;

}

Id Entity::GetId() { return m\_id; }

Entity::Date Entity::GetDate() { return m\_date; }

void Entity::SetId(Id id) { m\_id = id; }

void Entity::SetDate(Date date)

{

m\_date.day = date.day;

m\_date.month = date.month;

m\_date.year= date.year;

}

**TextEntity.h**

#ifndef TEXT\_ENTITY\_H

#define TEXT\_ENTITY\_H

#include "Entity.h"

namespace KMK

{

class TextEntity : public Entity

{

public:

TextEntity(Id id, Date date, char\* content, Id ownerId);

char\* GetContent();

Id GetOwnerId();

void SetContent(char\* content);

void SetOwnerId(Id ownerId);

private:

char\* m\_content{};

Id m\_ownerId{};

};

}

#endif // !TEXT\_ENTITY\_H

**TextEntity.cpp**

#include "TextEntity.h"

#include "Constants.h"

#include <iostream>

using namespace KMK;

TextEntity::TextEntity(Id id, Date date, char\* content, Id ownerId) : Entity(id, date)

{

m\_content = new char[LENGTH\_OF\_FIELD] {};

strcpy\_s(m\_content, LENGTH\_OF\_FIELD, content);

m\_ownerId = ownerId;

}

char\* TextEntity::GetContent() { return m\_content; }

Id TextEntity::GetOwnerId() { return m\_ownerId; }

Void TextEntity::SetContent(char\* content) { strcpy\_s(m\_content, LENGTH\_OF\_FIELD, content); }

void TextEntity::SetOwnerId(Id ownerId) { m\_ownerId = ownerId; }

**Dialogue.h**

#ifndef DIALOGUE\_H

#define DIALOGUE\_H

#include "TextEntity.h"

namespace KMK

{

class Dialogue : public TextEntity

{

public:

Dialogue(Id id, Date date, char\* message, Id senderId, Id adresseeId);

Id GetAdresseeId();

void SetAdresseeId(Id adresseeId);

private:

Id m\_adresseeId{};

};

}

#endif // !DIALOGUE\_H

**Dialogue.cpp**

#include "Dialogue.h"

using namespace KMK;

Dialogue::Dialogue(Id id, Date date, char\* message, Id senderId, Id adresseeId) :

TextEntity(id, date, message, senderId)

{

m\_adresseeId = adresseeId;

}

Id Dialogue::GetAdresseeId() { return m\_adresseeId; }

void Dialogue::SetAdresseeId(Id adresseeId) { m\_adresseeId = adresseeId; }

**Interest.h**

#ifndef INTEREST\_H

#define INTEREST\_H

#include "TextEntity.h"

namespace KMK

{

class Interest : public TextEntity

{

public:

Interest(Id id, Date date, char\* interest, Id ownerId);

};

}

#endif // !INTEREST\_H

**Interest.cpp**

#include "Interest.h"

using namespace KMK;

Interest::Interest(Id id, Date date, char\* interest, Id ownerId) :

TextEntity(id, date, interest, ownerId) {}

**Reminder.h**

#ifndef REMINDER\_H

#define REMINDER\_H

#include "TextEntity.h"

namespace KMK

{

class Reminder : public TextEntity

{

public:

Reminder(Id id, Date date, char\* reminder, Id ownerId, Date reminderTime);

Date GetReminderTime();

void SetReminderTime(Date reminderTime);

private:

Date m\_reminderTime{};

};

}

#endif // !REMINDER\_H

**Reminder.cpp**

#include "Reminder.h"

using namespace KMK;

Reminder::Reminder(Id id, Date date, char\* reminder, Id ownerId, Date reminderTime) :

TextEntity(id, date, reminder, ownerId)

{

m\_reminderTime = reminderTime;

}

Reminder::Date Reminder::GetReminderTime() { return m\_reminderTime; }

void Reminder::SetReminderTime(Date reminderTime)

{

m\_reminderTime.day = reminderTime.day;

m\_reminderTime.month = reminderTime.month;

m\_reminderTime.year = reminderTime.year;

}

**Theme.h**

#ifndef THEME\_H

#define THEME\_H

#include "TextEntity.h"

namespace KMK

{

class Theme : public TextEntity

{

public:

Theme(Id id, Date date, char\* theme, Id ownerId);

};

}

#endif // !THEME\_H

**Theme.cpp**

#include "Theme.h"

using namespace KMK;

Theme::Theme(Id id, Date date, char\* theme, Id ownerId) :

TextEntity(id, date, theme, ownerId){}

**User.h**

#ifndef USER\_H

#define USER\_H

#include "Entity.h"

namespace KMK

{

class User : public Entity

{

public:

User();

User(Id id, Date date, char\* name, char\* login, char\* password);

char\* GetName();

char\* GetLogin();

char\* GetPassword();

void SetName(char\* name);

void SetLogin(char\* login);

void SetPassword(char\* password);

private:

char\* m\_name{};

char\* m\_login{};

char\* m\_password{};

};

}

#endif // !USER\_H

**User.cpp**

#include "User.h"

#include "Constants.h"

#include <iostream>

using namespace KMK;

User::User() : Entity(0, {})

{

m\_name = {};

m\_login = {};

m\_password = {};

}

User::User(Id id, Date date, char\* name, char\* login, char\* password) :

Entity(id, date)

{

m\_name = new char[LENGTH\_OF\_FIELD];

strcpy\_s(m\_name, LENGTH\_OF\_FIELD, name);

m\_login = new char[LENGTH\_OF\_FIELD];

strcpy\_s(m\_login, LENGTH\_OF\_FIELD, login);

m\_password = new char[LENGTH\_OF\_FIELD];

strcpy\_s(m\_password, LENGTH\_OF\_FIELD, password);

}

char\* User::GetName() { return m\_name; }

char\* User::GetLogin() { return m\_login; }

char\* User::GetPassword() { return m\_password; }

void User::SetName(char\* name) { strcpy\_s(m\_name, LENGTH\_OF\_FIELD, name); }

void User::SetLogin(char\* login) { strcpy\_s(m\_login, LENGTH\_OF\_FIELD, login); }

void User::SetPassword(char\* password) { strcpy\_s(m\_password, LENGTH\_OF\_FIELD, password); }

**IdCounter.h**

#ifndef ID\_COUNTER\_H

#define ID\_COUNTER\_H

#include "TypeDefinitions.h"

namespace KMK

{

class IdCounter

{

public:

IdCounter();

IdCounter(char\* storageFile);

Id operator() ();

private:

Id m\_id{};

char\* m\_storageFile = nullptr;

};

}

#endif // !ID\_COUNTER\_H

**IdCounter.cpp**

#include "IdCounter.h"

#include "Constants.h"

#include <iostream>

#include <fstream>

using namespace KMK;

IdCounter::IdCounter()

{

m\_id = 0;

m\_storageFile = nullptr;

}

IdCounter::IdCounter(char\* storageFile)

{

m\_storageFile = new char[LENGTH\_OF\_FIELD] {};

strcpy\_s(m\_storageFile, LENGTH\_OF\_FIELD, storageFile);

std::ifstream fileRead(m\_storageFile, std::ios::binary);

fileRead.read((char\*)&m\_id, sizeof(m\_id));

fileRead.close();

}

Id IdCounter::operator() ()

{

++m\_id;

std::ofstream fileWrite(m\_storageFile, std::ios::binary);

fileWrite.write((char\*)&m\_id, sizeof(m\_id));

fileWrite.close();

return m\_id;

}

**AbstractMenuItem.h**

#ifndef ABSTRACT\_MENU\_ITEM\_H

#define ABSTRACT\_MENU\_ITEM\_H

namespace KMK

{

class MenuItem

{

public:

MenuItem(char\* itemName);

virtual char\* GetItemName();

virtual void PrintItemName();

virtual int Run() = 0;

private:

char\* m\_itemName = nullptr;

};

}

#endif // !ABSTRACT\_MENU\_ITEM\_H

**AbstractMenuItem.cpp**

#include "AbstractMenuItem.h"

#include "Constants.h"

#include <iostream>

using namespace KMK;

MenuItem::MenuItem(char\* itemName)

{

m\_itemName = new char[LENGTH\_OF\_FIELD];

strcpy\_s(m\_itemName, LENGTH\_OF\_FIELD, itemName);

}

char\* MenuItem::GetItemName()

{

return m\_itemName;

}

void MenuItem::PrintItemName()

{

std::cout << m\_itemName;

}

**DialogueListItem.h**

#ifndef DIALOGUE\_LIST\_ITEM\_H

#define DIALOGUE\_LIST\_ITEM\_H

#include "AbstractMenuItem.h"

#include "Dialogue.h"

#include "IdCounter.h"

namespace KMK

{

class DialogueListItem : public MenuItem

{

public:

DialogueListItem(char\* itemName, char\* dialogueDatabase, char\* idCounterDatabase);

int Run();

private:

char\* m\_dialogueDatabase = nullptr;

char\* m\_idCounterDatabase = nullptr;

size\_t m\_size{};

Dialogue\*\* m\_dialogueList = nullptr;

IdCounter m\_idCounter{};

};

}

#endif // !DIALOGUE\_LIST\_ITEM\_H

**DialogueListItem.cpp**

#include "DialogueListItem.h"

#include "Constants.h"

#include <iostream>

#include <fstream>

#include "Add.h"

#include <Windows.h>

#include "Remove.h"

#include "Edit.h"

#include "Sort.h"

#include "Filter.h"

#include <iomanip>

using namespace KMK;

DialogueListItem::DialogueListItem(char\* itemName, char\* dialogueDatabase, char\* idCounterDatabase) : MenuItem(itemName)

{

m\_dialogueDatabase = new char[LENGTH\_OF\_FIELD] {};

strcpy\_s(m\_dialogueDatabase, LENGTH\_OF\_FIELD, dialogueDatabase);

m\_idCounterDatabase = new char[LENGTH\_OF\_FIELD] {};

strcpy\_s(m\_idCounterDatabase, LENGTH\_OF\_FIELD, idCounterDatabase);

m\_idCounter = { idCounterDatabase };

std::ifstream fileRead(m\_dialogueDatabase, std::ios::binary);

fileRead.read((char\*)&m\_size, sizeof(size\_t));

m\_dialogueList = new Dialogue \* [m\_size] {};

for (Iteration i{}; i < m\_size; ++i)

{

Id id{};

fileRead.read((char\*)&id, sizeof(Id));

Entity::Date date{};

fileRead.read((char\*)&date, sizeof(Entity::Date));

char\* message = new char[LENGTH\_OF\_FIELD] {};

fileRead.read(message, LENGTH\_OF\_FIELD);

Id ownerId{};

fileRead.read((char\*)&ownerId, sizeof(Id));

Id adresseeId{};

fileRead.read((char\*)&adresseeId, sizeof(Id));

m\_dialogueList[i] = new Dialogue{ id, date, message, ownerId, adresseeId };

}

fileRead.close();

}

int DialogueListItem::Run()

{

enum Command

{

RESET,

ADD,

REMOVE,

EDIT,

SORT,

FILTER,

ID,

EXIT

};

unsigned short command = 0;

while (command != EXIT)

{

unsigned short maximumMessageLength = 7;

for (Iteration i{}; i < m\_size; ++i)

{

if (strlen(m\_dialogueList[i]->GetContent()) > maximumMessageLength)

{

maximumMessageLength = strlen(m\_dialogueList[i]->GetContent());

}

}

std::cout << std::setw((11 + 3 + 3 + 5 + maximumMessageLength + 1 + 11 + 11 + 6 + strlen(GetItemName())) / 2) << GetItemName() << "\n\n";

std::cout << std::setw(11) << "ID" << "|";

std::cout << std::setw(3) << "dd" << "|";

std::cout << std::setw(3) << "mm" << "|";

std::cout << std::setw(5) << "yyyy" << "|";

std::cout << std::setw(maximumMessageLength + 1) << "Message" << "|";

std::cout << std::setw(11) << "Sender ID" << "|";

std::cout << std::setw(11) << "Adressee ID";

std::cout << '\n';

for (Iteration i{}; i < m\_size; ++i)

{

std::cout << std::setw(11) << m\_dialogueList[i]->GetId() << "|";

std::cout << std::setw(3) << m\_dialogueList[i]->GetDate().day << "|";

std::cout << std::setw(3) << m\_dialogueList[i]->GetDate().month << "|";

std::cout << std::setw(5) << m\_dialogueList[i]->GetDate().year << "|";

std::cout << std::setw(maximumMessageLength + 1) << m\_dialogueList[i]->GetContent() << "|";

std::cout << std::setw(11) << m\_dialogueList[i]->GetOwnerId() << "|";

std::cout << std::setw(11) << m\_dialogueList[i]->GetAdresseeId();

std::cout << '\n';

}

std::cout << '\n';

std::cout << RESET << ". Reset list\n";

std::cout << ADD << ". Add new dialogue\n";

std::cout << REMOVE << ". Delete dialogue\n";

std::cout << EDIT << ". Edit dialogue\n";

std::cout << SORT << ". Sort list\n";

std::cout << FILTER << ". Filter list\n";

std::cout << ID << ". Choose ID\n";

std::cout << EXIT << ". Exit\n";

std::cout << "Input command: ";

std::cin >> command;

std::cin.ignore();

std::cout << '\n';

if (command == RESET)

{

std::ifstream fileRead(m\_dialogueDatabase, std::ios::binary);

fileRead.read((char\*)&m\_size, sizeof(size\_t));

m\_dialogueList = new Dialogue \* [m\_size] {};

for (Iteration i{}; i < m\_size; ++i)

{

Id id{};

fileRead.read((char\*)&id, sizeof(Id));

Entity::Date date{};

fileRead.read((char\*)&date, sizeof(Entity::Date));

char\* message = new char[LENGTH\_OF\_FIELD] {};

fileRead.read(message, LENGTH\_OF\_FIELD);

Id ownerId{};

fileRead.read((char\*)&ownerId, sizeof(Id));

Id adresseeId{};

fileRead.read((char\*)&adresseeId, sizeof(Id));

m\_dialogueList[i] = new Dialogue{ id, date, message, ownerId, adresseeId };

}

fileRead.close();

}

else if (command == ADD)

{

char\* message = new char[LENGTH\_OF\_FIELD];

std::cout << "Input message: ";

std::cin.getline(message, LENGTH\_OF\_FIELD, '\n');

Id senderId;

std::cout << "Input sender ID: ";

std::cin >> senderId;

Id adresseeId;

std::cout << "Input adressee ID: ";

std::cin >> adresseeId;

std::cin.ignore();

SYSTEMTIME systemTime;

GetLocalTime(&systemTime);

Dialogue newDialogue = Dialogue(m\_idCounter(), { systemTime.wDay, systemTime.wMonth, systemTime.wYear }, message, senderId, adresseeId);

m\_dialogueList = Add(m\_dialogueList, m\_size, newDialogue, m\_size);

}

else if (command == REMOVE)

{

std::cout << "Input ID: ";

Id id;

std::cin >> id;

std::cin.ignore();

m\_dialogueList = Remove(m\_dialogueList, m\_size, id, m\_size);

}

else if (command == EDIT)

{

std::cout << "Input ID: ";

Id id;

std::cin >> id;

std::cin.ignore();

std::cout << '\n';

std::cout << "Fields to edit\n";

std::cout << "0. ID\n";

std::cout << "1. Date\n";

std::cout << "2. Message\n";

std::cout << "3. Sender ID\n";

std::cout << "4. Adressee ID\n";

std::cout << "Choose field: ";

unsigned short fieldToChange;

std::cin >> fieldToChange;

std::cin.ignore();

std::cout << '\n';

if (fieldToChange == 0)

{

std::cout << "Input new ID: ";

Id\* newId = new Id{};

std::cin >> \*newId;

std::cin.ignore();

m\_dialogueList = Edit(m\_dialogueList, m\_size, id, (void\*)newId, EditMode::ID);

}

if (fieldToChange == 1)

{

std::cout << "Input new date\n";

std::cout << "Day:";

unsigned short day;

std::cin >> day;

std::cout << "Month:";

unsigned short month;

std::cin >> month;

std::cout << "Year:";

unsigned short year;

std::cin >> year;

std::cin.ignore();

m\_dialogueList = Edit(m\_dialogueList, m\_size, id, (void\*)new Entity::Date{ day, month, year }, EditMode::DATE);

}

if (fieldToChange == 2)

{

std::cout << "Input new message: ";

char\* message = new char[LENGTH\_OF\_FIELD] {};

std::cin.getline(message, LENGTH\_OF\_FIELD, '\n');

m\_dialogueList = Edit(m\_dialogueList, m\_size, id, (void\*)message, EditMode::CONTENT);

}

if (fieldToChange == 3)

{

std::cout << "Input new sender ID: ";

Id\* newSenderId = new Id{};

std::cin >> \*newSenderId;

std::cin.ignore();

m\_dialogueList = Edit(m\_dialogueList, m\_size, id, (void\*)newSenderId, EditMode::OWNER\_ID);

}

if (fieldToChange == 4)

{

std::cout << "Input new adressee ID: ";

Id\* newAdresseeId = new Id{};

std::cin >> \*newAdresseeId;

std::cin.ignore();

m\_dialogueList = Edit(m\_dialogueList, m\_size, id, (void\*)newAdresseeId, EditMode::OWNER\_ID);

}

}

else if (command == SORT)

{

std::cout << "Orders for sort\n";

std::cout << "0. Descending\n";

std::cout << "1. Ascending\n";

std::cout << "Choose order: ";

unsigned short order;

std::cin >> order;

std::cout << '\n';

std::cout << "Fields for sort\n";

std::cout << "0. ID\n";

std::cout << "1. Date\n";

std::cout << "2. Message\n";

std::cout << "3. Sender ID\n";

std::cout << "4. Adressee ID\n";

std::cout << "Choose field: ";

unsigned short field;

std::cin >> field;

std::cin.ignore();

SortMode sortMode = (SortMode)-1;

switch (field)

{

case 0:

sortMode = SortMode::ID;

break;

case 1:

sortMode = SortMode::DATE;

break;

case 2:

sortMode = SortMode::CONTENT;

break;

case 3:

sortMode = SortMode::OWNER\_ID;

break;

case 4:

sortMode = SortMode::ADRESSEE\_ID;

break;

}

m\_dialogueList = Sort(m\_dialogueList, m\_size, (OrderMode)order, sortMode);

}

else if (command == FILTER)

{

std::cout << "Fields for filter\n";

std::cout << "0. ID\n";

std::cout << "1. Date\n";

std::cout << "2. Message\n";

std::cout << "3. Sender ID\n";

std::cout << "4. Adressee ID\n";

std::cout << "Choose field: ";

unsigned short field;

std::cin >> field;

std::cin.ignore();

std::cout << '\n';

if (field == 0)

{

std::cout << "Input part of ID: ";

Id id;

std::cin >> id;

std::cin.ignore();

m\_dialogueList = Filter(m\_dialogueList, m\_size, (void\*)&id, FilterMode::ID, m\_size);

}

if (field == 1)

{

std::cout << "Input date (if you don't want to filter by the field, input 0)\n";

std::cout << "Day: ";

unsigned short day;

std::cin >> day;

std::cout << "Month: ";

unsigned short month;

std::cin >> month;

std::cout << "Year: ";

unsigned short year;

std::cin >> year;

std::cin.ignore();

m\_dialogueList = Filter(m\_dialogueList, m\_size, (void\*)new Entity::Date{ day, month, year }, FilterMode::DATE, m\_size);

}

if (field == 2)

{

std::cout << "Input part of message: ";

char\* message = new char[LENGTH\_OF\_FIELD] {};

std::cin.getline(message, LENGTH\_OF\_FIELD, '\n');

m\_dialogueList = Filter(m\_dialogueList, m\_size, (void\*)message, FilterMode::CONTENT, m\_size);

}

if (field == 3)

{

std::cout << "Input part of sender ID: ";

Id senderId;

std::cin >> senderId;

std::cin.ignore();

m\_dialogueList = Filter(m\_dialogueList, m\_size, (void\*)&senderId, FilterMode::OWNER\_ID, m\_size);

}

if (field == 4)

{

std::cout << "Input part of adressee ID: ";

Id adresseeId;

std::cin >> adresseeId;

std::cin.ignore();

m\_dialogueList = Filter(m\_dialogueList, m\_size, (void\*)&adresseeId, FilterMode::ADRESSEE\_ID, m\_size);

}

}

if (command == ADD || command == REMOVE || command == EDIT)

{

std::ofstream fileWrite(m\_dialogueDatabase, std::ios::binary);

fileWrite.write((char\*)&m\_size, sizeof(size\_t));

for (Iteration i{}; i < m\_size; ++i)

{

fileWrite.write((char\*)new Id{ m\_dialogueList[i]->GetId() }, sizeof(Id));

fileWrite.write((char\*)&m\_dialogueList[i]->GetDate(), sizeof(Entity::Date));

fileWrite.write(m\_dialogueList[i]->GetContent(), LENGTH\_OF\_FIELD);

fileWrite.write((char\*)new Id{ m\_dialogueList[i]->GetOwnerId() }, sizeof(Id));

fileWrite.write((char\*)new Id{ m\_dialogueList[i]->GetAdresseeId() }, sizeof(Id));

}

fileWrite.close();

}

system("cls");

}

return 0;

}

**InterestListItem.h**

#ifndef INTEREST\_LIST\_ITEM\_H

#define INTEREST\_LIST\_ITEM\_H

#include "AbstractMenuItem.h"

#include "Interest.h"

#include "IdCounter.h"

namespace KMK

{

class InterestListItem : public MenuItem

{

public:

InterestListItem(char\* itemName, char\* interestDatabase, char\* idCounterDatabase);

int Run();

private:

char\* m\_interestDatabase = nullptr;

char\* m\_idCounterDatabase = nullptr;

size\_t m\_size{};

Interest\*\* m\_interestList = nullptr;

IdCounter m\_idCounter{};

};

}

#endif // !INTEREST\_LIST\_ITEM\_H

**InterestListItem.cpp**

#include "InterestListItem.h"

#include "Constants.h"

#include <iostream>

#include <fstream>

#include "Add.h"

#include <Windows.h>

#include "Remove.h"

#include "Edit.h"

#include "Sort.h"

#include "Filter.h"

#include <iomanip>

using namespace KMK;

InterestListItem::InterestListItem(char\* itemName, char\* interestDatabase, char\* idCounterDatabase) : MenuItem(itemName)

{

m\_interestDatabase = new char[LENGTH\_OF\_FIELD] {};

strcpy\_s(m\_interestDatabase, LENGTH\_OF\_FIELD, interestDatabase);

m\_idCounterDatabase = new char[LENGTH\_OF\_FIELD] {};

strcpy\_s(m\_idCounterDatabase, LENGTH\_OF\_FIELD, idCounterDatabase);

m\_idCounter = { idCounterDatabase };

std::ifstream fileRead(m\_interestDatabase, std::ios::binary);

fileRead.read((char\*)&m\_size, sizeof(size\_t));

m\_interestList = new Interest \* [m\_size] {};

for (Iteration i{}; i < m\_size; ++i)

{

Id id{};

fileRead.read((char\*)&id, sizeof(Id));

Entity::Date date{};

fileRead.read((char\*)&date, sizeof(Entity::Date));

char\* interest = new char[LENGTH\_OF\_FIELD] {};

fileRead.read(interest, LENGTH\_OF\_FIELD);

Id ownerId{};

fileRead.read((char\*)&ownerId, sizeof(Id));

m\_interestList[i] = new Interest{ id, date, interest, ownerId };

}

fileRead.close();

}

int InterestListItem::Run()

{

enum Command

{

RESET,

ADD,

REMOVE,

EDIT,

SORT,

FILTER,

ID,

EXIT

};

unsigned short command = 0;

while (command != EXIT)

{

unsigned short maximumInterestLength = 8;

for (Iteration i{}; i < m\_size; ++i)

{

if (strlen(m\_interestList[i]->GetContent()) > maximumInterestLength)

{

maximumInterestLength = strlen(m\_interestList[i]->GetContent());

}

}

std::cout << std::setw((11 + 3 + 3 + 5 + maximumInterestLength + 1 + 11 + 5 + strlen(GetItemName())) / 2) << GetItemName() << "\n\n";

std::cout << std::setw(11) << "ID" << "|";

std::cout << std::setw(3) << "dd" << "|";

std::cout << std::setw(3) << "mm" << "|";

std::cout << std::setw(5) << "yyyy" << "|";

std::cout << std::setw(maximumInterestLength + 1) << "Interest" << "|";

std::cout << std::setw(11) << "Owner ID";

std::cout << '\n';

for (Iteration i{}; i < m\_size; ++i)

{

std::cout << std::setw(11) << m\_interestList[i]->GetId() << "|";

std::cout << std::setw(3) << m\_interestList[i]->GetDate().day << "|";

std::cout << std::setw(3) << m\_interestList[i]->GetDate().month << "|";

std::cout << std::setw(5) << m\_interestList[i]->GetDate().year << "|";

std::cout << std::setw(maximumInterestLength + 1) << m\_interestList[i]->GetContent() << "|";

std::cout << std::setw(11) << m\_interestList[i]->GetOwnerId();

std::cout << '\n';

}

std::cout << '\n';

std::cout << RESET << ". Reset list\n";

std::cout << ADD << ". Add new interest\n";

std::cout << REMOVE << ". Delete interest\n";

std::cout << EDIT << ". Edit interest\n";

std::cout << SORT << ". Sort list\n";

std::cout << FILTER << ". Filter list\n";

std::cout << ID << ". Choose ID\n";

std::cout << EXIT << ". Exit\n";

std::cout << "Input command: ";

std::cin >> command;

std::cin.ignore();

std::cout << '\n';

if (command == RESET)

{

std::ifstream fileRead(m\_interestDatabase, std::ios::binary);

fileRead.read((char\*)&m\_size, sizeof(size\_t));

m\_interestList = new Interest \* [m\_size] {};

for (Iteration i{}; i < m\_size; ++i)

{

Id id{};

fileRead.read((char\*)&id, sizeof(Id));

Entity::Date date{};

fileRead.read((char\*)&date, sizeof(Entity::Date));

char\* interest = new char[LENGTH\_OF\_FIELD] {};

fileRead.read(interest, LENGTH\_OF\_FIELD);

Id ownerId{};

fileRead.read((char\*)&ownerId, sizeof(Id));

m\_interestList[i] = new Interest{ id, date, interest, ownerId };

}

fileRead.close();

}

else if (command == ADD)

{

char\* interest = new char[LENGTH\_OF\_FIELD];

std::cout << "Input interest: ";

std::cin.getline(interest, LENGTH\_OF\_FIELD, '\n');

Id ownerId;

std::cout << "Input owner ID: ";

std::cin >> ownerId;

std::cin.ignore();

SYSTEMTIME systemTime;

GetLocalTime(&systemTime);

Interest newInterest = Interest(m\_idCounter(), { systemTime.wDay, systemTime.wMonth, systemTime.wYear }, interest, ownerId);

m\_interestList = Add(m\_interestList, m\_size, newInterest, m\_size);

}

else if (command == REMOVE)

{

std::cout << "Input ID: ";

Id id;

std::cin >> id;

std::cin.ignore();

m\_interestList = Remove(m\_interestList, m\_size, id, m\_size);

}

else if (command == EDIT)

{

std::cout << "Input ID: ";

Id id;

std::cin >> id;

std::cin.ignore();

std::cout << '\n';

std::cout << "Fields to edit\n";

std::cout << "0. ID\n";

std::cout << "1. Date\n";

std::cout << "2. Interest\n";

std::cout << "3. Owner ID\n";

std::cout << "Choose field: ";

unsigned short fieldToChange;

std::cin >> fieldToChange;

std::cin.ignore();

std::cout << '\n';

if (fieldToChange == 0)

{

std::cout << "Input new ID: ";

Id\* newId = new Id{};

std::cin >> \*newId;

std::cin.ignore();

m\_interestList = Edit(m\_interestList, m\_size, id, (void\*)newId, EditMode::ID);

}

if (fieldToChange == 1)

{

std::cout << "Input new date\n";

std::cout << "Day: ";

unsigned short day;

std::cin >> day;

std::cout << "Month: ";

unsigned short month;

std::cin >> month;

std::cout << "Year: ";

unsigned short year;

std::cin >> year;

std::cin.ignore();

m\_interestList = Edit(m\_interestList, m\_size, id, (void\*)new Entity::Date{ day, month, year }, EditMode::DATE);

}

if (fieldToChange == 2)

{

std::cout << "Input new interest: ";

char\* interest = new char[LENGTH\_OF\_FIELD] {};

std::cin.getline(interest, LENGTH\_OF\_FIELD, '\n');

m\_interestList = Edit(m\_interestList, m\_size, id, (void\*)interest, EditMode::CONTENT);

}

if (fieldToChange == 3)

{

std::cout << "Input new owner ID: ";

Id\* newOwnerId = new Id{};

std::cin >> \*newOwnerId;

std::cin.ignore();

m\_interestList = Edit(m\_interestList, m\_size, id, (void\*)newOwnerId, EditMode::OWNER\_ID);

}

}

else if (command == SORT)

{

std::cout << "Orders for sort\n";

std::cout << "0. Descending\n";

std::cout << "1. Ascending\n";

std::cout << "Choose order: ";

unsigned short order;

std::cin >> order;

std::cout << '\n';

std::cout << "Fields for sort\n";

std::cout << "0. ID\n";

std::cout << "1. Date\n";

std::cout << "2. Interest\n";

std::cout << "3. Owner ID\n";

std::cout << "Choose field: ";

unsigned short field;

std::cin >> field;

std::cin.ignore();

SortMode sortMode = (SortMode)-1;

switch (field)

{

case 0:

sortMode = SortMode::ID;

break;

case 1:

sortMode = SortMode::DATE;

break;

case 2:

sortMode = SortMode::CONTENT;

break;

case 3:

sortMode = SortMode::OWNER\_ID;

break;

}

m\_interestList = Sort(m\_interestList, m\_size, (OrderMode)order, sortMode);

}

else if (command == FILTER)

{

std::cout << "Fields for filter\n";

std::cout << "0. ID\n";

std::cout << "1. Date\n";

std::cout << "2. Interest\n";

std::cout << "3. Owner ID\n";

std::cout << "Choose field: ";

unsigned short field;

std::cin >> field;

std::cin.ignore();

std::cout << '\n';

if (field == 0)

{

std::cout << "Input part of ID: ";

Id id;

std::cin >> id;

std::cin.ignore();

m\_interestList = Filter(m\_interestList, m\_size, (void\*)&id, FilterMode::ID, m\_size);

}

if (field == 1)

{

std::cout << "Input date (if you don't want to filter by the field, input 0)\n";

std::cout << "Day: ";

unsigned short day;

std::cin >> day;

std::cout << "Month: ";

unsigned short month;

std::cin >> month;

std::cout << "Year: ";

unsigned short year;

std::cin >> year;

std::cin.ignore();

m\_interestList = Filter(m\_interestList, m\_size, (void\*)new Entity::Date{ day, month, year }, FilterMode::DATE, m\_size);

}

if (field == 2)

{

std::cout << "Input part of interest: ";

char\* interest = new char[LENGTH\_OF\_FIELD] {};

std::cin.getline(interest, LENGTH\_OF\_FIELD, '\n');

m\_interestList = Filter(m\_interestList, m\_size, (void\*)interest, FilterMode::CONTENT, m\_size);

}

if (field == 3)

{

std::cout << "Input part of owner ID: ";

Id ownerId;

std::cin >> ownerId;

std::cin.ignore();

m\_interestList = Filter(m\_interestList, m\_size, (void\*)&ownerId, FilterMode::OWNER\_ID, m\_size);

}

}

if (command == ADD || command == REMOVE || command == EDIT)

{

std::ofstream fileWrite(m\_interestDatabase, std::ios::binary);

fileWrite.write((char\*)&m\_size, sizeof(size\_t));

for (Iteration i{}; i < m\_size; ++i)

{

fileWrite.write((char\*)new Id{ m\_interestList[i]->GetId() }, sizeof(Id));

fileWrite.write((char\*)&m\_interestList[i]->GetDate(), sizeof(Entity::Date));

fileWrite.write(m\_interestList[i]->GetContent(), LENGTH\_OF\_FIELD);

fileWrite.write((char\*)new Id{ m\_interestList[i]->GetOwnerId() }, sizeof(Id));

}

fileWrite.close();

}

system("cls");

}

return 0;

}

**ReminderListItem.h**

#ifndef REMINDER\_LIST\_ITEM\_H

#define REMINDER\_LIST\_ITEM\_H

#include "AbstractMenuItem.h"

#include "Reminder.h"

#include "IdCounter.h"

namespace KMK

{

class ReminderListItem : public MenuItem

{

public:

ReminderListItem(char\* itemName, char\* reminderDatabase, char\* idCounterDatabase);

int Run();

private:

char\* m\_reminderDatabase = nullptr;

char\* m\_idCounterDatabase = nullptr;

size\_t m\_size{};

Reminder\*\* m\_reminderList = nullptr;

IdCounter m\_idCounter{};

};

}

#endif // !REMINDER\_LIST\_ITEM\_H

**ReminderListItem.cpp**

#include "ReminderListItem.h"

#include "Constants.h"

#include <iostream>

#include <fstream>

#include "Add.h"

#include <Windows.h>

#include "Remove.h"

#include "Edit.h"

#include "Sort.h"

#include "Filter.h"

#include <iomanip>

using namespace KMK;

ReminderListItem::ReminderListItem(char\* itemName, char\* reminderDatabase, char\* idCounterDatabase) : MenuItem(itemName)

{

m\_reminderDatabase = new char[LENGTH\_OF\_FIELD] {};

strcpy\_s(m\_reminderDatabase, LENGTH\_OF\_FIELD, reminderDatabase);

m\_idCounterDatabase = new char[LENGTH\_OF\_FIELD] {};

strcpy\_s(m\_idCounterDatabase, LENGTH\_OF\_FIELD, idCounterDatabase);

m\_idCounter = { idCounterDatabase };

std::ifstream fileRead(m\_reminderDatabase, std::ios::binary);

fileRead.read((char\*)&m\_size, sizeof(size\_t));

m\_reminderList = new Reminder \* [m\_size] {};

for (Iteration i{}; i < m\_size; ++i)

{

Id id{};

fileRead.read((char\*)&id, sizeof(Id));

Entity::Date date{};

fileRead.read((char\*)&date, sizeof(Entity::Date));

char\* reminder = new char[LENGTH\_OF\_FIELD] {};

fileRead.read(reminder, LENGTH\_OF\_FIELD);

Id ownerId{};

fileRead.read((char\*)&ownerId, sizeof(Id));

Entity::Date reminderTime{};

fileRead.read((char\*)&reminderTime, sizeof(Entity::Date));

m\_reminderList[i] = new Reminder{ id, date, reminder, ownerId, reminderTime };

}

fileRead.close();

}

int ReminderListItem::Run()

{

enum Command

{

RESET,

ADD,

REMOVE,

EDIT,

SORT,

FILTER,

ID,

EXIT

};

unsigned short command = 0;

while (command != EXIT)

{

unsigned short maximumReminderLength = 8;

for (Iteration i{}; i < m\_size; ++i)

{

if (strlen(m\_reminderList[i]->GetContent()) > maximumReminderLength)

{

maximumReminderLength = strlen(m\_reminderList[i]->GetContent());

}

}

std::cout << std::setw((11 + 3 + 3 + 5 + 11 + maximumReminderLength + 1 + 3 + 3 + 5 + 8 + strlen(GetItemName())) / 2) << GetItemName() << "\n\n";

std::cout << std::setw(11) << "ID" << "|";

std::cout << std::setw(3) << "dd" << "|";

std::cout << std::setw(3) << "mm" << "|";

std::cout << std::setw(5) << "yyyy" << "|";

std::cout << std::setw(11) << "Owner ID" << "|";

std::cout << std::setw(maximumReminderLength + 1) << "Reminder" << ":";

std::cout << std::setw(3) << "dd" << "|";

std::cout << std::setw(3) << "mm" << "|";

std::cout << std::setw(5) << "yyyy";

std::cout << '\n';

for (Iteration i{}; i < m\_size; ++i)

{

std::cout << std::setw(11) << m\_reminderList[i]->GetId() << "|";

std::cout << std::setw(3) << m\_reminderList[i]->GetDate().day << "|";

std::cout << std::setw(3) << m\_reminderList[i]->GetDate().month << "|";

std::cout << std::setw(5) << m\_reminderList[i]->GetDate().year << "|";

std::cout << std::setw(11) << m\_reminderList[i]->GetOwnerId() << "|";

std::cout << std::setw(maximumReminderLength + 1) << m\_reminderList[i]->GetContent() << ":";

std::cout << std::setw(3) << m\_reminderList[i]->GetReminderTime().day << "|";

std::cout << std::setw(3) << m\_reminderList[i]->GetReminderTime().month << "|";

std::cout << std::setw(5) << m\_reminderList[i]->GetReminderTime().year;

std::cout << '\n';

}

std::cout << '\n';

std::cout << RESET << ". Reset list\n";

std::cout << ADD << ". Add new reminder\n";

std::cout << REMOVE << ". Delete reminder\n";

std::cout << EDIT << ". Edit reminder\n";

std::cout << SORT << ". Sort list\n";

std::cout << FILTER << ". Filter list\n";

std::cout << ID << ". Choose ID\n";

std::cout << EXIT << ". Exit\n";

std::cout << "Input command: ";

std::cin >> command;

std::cin.ignore();

std::cout << '\n';

if (command == RESET)

{

std::ifstream fileRead(m\_reminderDatabase, std::ios::binary);

fileRead.read((char\*)&m\_size, sizeof(size\_t));

m\_reminderList = new Reminder \* [m\_size] {};

for (Iteration i{}; i < m\_size; ++i)

{

Id id{};

fileRead.read((char\*)&id, sizeof(Id));

Entity::Date date{};

fileRead.read((char\*)&date, sizeof(Entity::Date));

char\* reminder = new char[LENGTH\_OF\_FIELD] {};

fileRead.read(reminder, LENGTH\_OF\_FIELD);

Id ownerId{};

fileRead.read((char\*)&ownerId, sizeof(Id));

Entity::Date reminderTime{};

fileRead.read((char\*)&reminderTime, sizeof(Entity::Date));

m\_reminderList[i] = new Reminder{ id, date, reminder, ownerId, reminderTime };

}

fileRead.close();

}

else if (command == ADD)

{

Id ownerId;

std::cout << "Input owner ID: ";

std::cin >> ownerId;

std::cin.ignore();

char\* reminder = new char[LENGTH\_OF\_FIELD];

std::cout << "Input reminder: ";

std::cin.getline(reminder, LENGTH\_OF\_FIELD, '\n');

unsigned short day;

std::cout << "Input reminder day: ";

std::cin >> day;

unsigned short month;

std::cout << "Input reminder month: ";

std::cin >> month;

unsigned short year;

std::cout << "Input reminder year: ";

std::cin >> year;

std::cin.ignore();

SYSTEMTIME systemTime;

GetLocalTime(&systemTime);

Reminder newReminder = Reminder(m\_idCounter(), { systemTime.wDay, systemTime.wMonth, systemTime.wYear }, reminder, ownerId, {day, month, year});

m\_reminderList = Add(m\_reminderList, m\_size, newReminder, m\_size);

}

else if (command == REMOVE)

{

std::cout << "Input ID: ";

Id id;

std::cin >> id;

std::cin.ignore();

m\_reminderList = Remove(m\_reminderList, m\_size, id, m\_size);

}

else if (command == EDIT)

{

std::cout << "Input ID: ";

Id id;

std::cin >> id;

std::cin.ignore();

std::cout << '\n';

std::cout << "Fields to edit\n";

std::cout << "0. ID\n";

std::cout << "1. Date\n";

std::cout << "2. Owner ID\n";

std::cout << "3. Reminder\n";

std::cout << "4. Reminder time\n";

std::cout << "Choose field: ";

unsigned short fieldToChange;

std::cin >> fieldToChange;

std::cin.ignore();

std::cout << '\n';

if (fieldToChange == 0)

{

std::cout << "Input new ID: ";

Id\* newId = new Id{};

std::cin >> \*newId;

std::cin.ignore();

m\_reminderList = Edit(m\_reminderList, m\_size, id, (void\*)newId, EditMode::ID);

}

if (fieldToChange == 1)

{

std::cout << "Input new date\n";

std::cout << "Day: ";

unsigned short day;

std::cin >> day;

std::cout << "Month: ";

unsigned short month;

std::cin >> month;

std::cout << "Year: ";

unsigned short year;

std::cin >> year;

std::cin.ignore();

m\_reminderList = Edit(m\_reminderList, m\_size, id, (void\*)new Entity::Date{ day, month, year }, EditMode::DATE);

}

if (fieldToChange == 2)

{

std::cout << "Input new owner ID: ";

Id\* newOwnerId = new Id{};

std::cin >> \*newOwnerId;

std::cin.ignore();

m\_reminderList = Edit(m\_reminderList, m\_size, id, (void\*)newOwnerId, EditMode::OWNER\_ID);

}

if (fieldToChange == 3)

{

std::cout << "Input new reminder: ";

char\* reminder = new char[LENGTH\_OF\_FIELD] {};

std::cin.getline(reminder, LENGTH\_OF\_FIELD, '\n');

m\_reminderList = Edit(m\_reminderList, m\_size, id, (void\*)reminder, EditMode::CONTENT);

}

if (fieldToChange == 4)

{

std::cout << "Input new reminder time\n";

std::cout << "Day: ";

unsigned short day;

std::cin >> day;

std::cout << "Month: ";

unsigned short month;

std::cin >> month;

std::cout << "Year: ";

unsigned short year;

std::cin >> year;

std::cin.ignore();

m\_reminderList = Edit(m\_reminderList, m\_size, id, (void\*)new Entity::Date{ day, month, year }, EditMode::REMINDER\_TIME);

}

}

else if (command == SORT)

{

std::cout << "Orders for sort\n";

std::cout << "0. Descending\n";

std::cout << "1. Ascending\n";

std::cout << "Choose order: ";

unsigned short order;

std::cin >> order;

std::cout << '\n';

std::cout << "Fields for sort\n";

std::cout << "0. ID\n";

std::cout << "1. Date\n";

std::cout << "2. Owner ID\n";

std::cout << "3. Reminder\n";

std::cout << "4. Reminder time\n";

std::cout << "Choose field: ";

unsigned short field;

std::cin >> field;

std::cin.ignore();

SortMode sortMode = (SortMode)-1;

switch (field)

{

case 0:

sortMode = SortMode::ID;

break;

case 1:

sortMode = SortMode::DATE;

break;

case 2:

sortMode = SortMode::OWNER\_ID;

break;

case 3:

sortMode = SortMode::CONTENT;

break;

case 4:

sortMode = SortMode::REMINDER\_TIME;

break;

}

m\_reminderList = Sort(m\_reminderList, m\_size, (OrderMode)order, sortMode);

}

else if (command == FILTER)

{

std::cout << "Fields for filter\n";

std::cout << "0. ID\n";

std::cout << "1. Date\n";

std::cout << "2. Owner ID\n";

std::cout << "3. Reminder\n";

std::cout << "4. Reminder time\n";

std::cout << "Choose field: ";

unsigned short field;

std::cin >> field;

std::cin.ignore();

std::cout << '\n';

if (field == 0)

{

std::cout << "Input part of ID: ";

Id id;

std::cin >> id;

std::cin.ignore();

m\_reminderList = Filter(m\_reminderList, m\_size, (void\*)&id, FilterMode::ID, m\_size);

}

if (field == 1)

{

std::cout << "Input date (if you don't want to filter by the field, input 0)\n";

std::cout << "Day: ";

unsigned short day;

std::cin >> day;

std::cout << "Month: ";

unsigned short month;

std::cin >> month;

std::cout << "Year: ";

unsigned short year;

std::cin >> year;

std::cin.ignore();

m\_reminderList = Filter(m\_reminderList, m\_size, (void\*)new Entity::Date{ day, month, year }, FilterMode::DATE, m\_size);

}

if (field == 2)

{

std::cout << "Input part of owner ID: ";

Id ownerId;

std::cin >> ownerId;

std::cin.ignore();

m\_reminderList = Filter(m\_reminderList, m\_size, (void\*)&ownerId, FilterMode::OWNER\_ID, m\_size);

}

if (field == 3)

{

std::cout << "Input part of reminder: ";

char\* reminder = new char[LENGTH\_OF\_FIELD] {};

std::cin.getline(reminder, LENGTH\_OF\_FIELD, '\n');

m\_reminderList = Filter(m\_reminderList, m\_size, (void\*)reminder, FilterMode::CONTENT, m\_size);

}

if (field == 4)

{

std::cout << "Input reminder time (if you don't want to filter by the field, input 0)\n";

std::cout << "Day: ";

unsigned short day;

std::cin >> day;

std::cout << "Month: ";

unsigned short month;

std::cin >> month;

std::cout << "Year: ";

unsigned short year;

std::cin >> year;

std::cin.ignore();

m\_reminderList = Filter(m\_reminderList, m\_size, (void\*)new Entity::Date{ day, month, year }, FilterMode::REMINDER\_TIME, m\_size);

}

}

if (command == ADD || command == REMOVE || command == EDIT)

{

std::ofstream fileWrite(m\_reminderDatabase, std::ios::binary);

fileWrite.write((char\*)&m\_size, sizeof(size\_t));

for (Iteration i{}; i < m\_size; ++i)

{

fileWrite.write((char\*)new Id{ m\_reminderList[i]->GetId() }, sizeof(Id));

fileWrite.write((char\*)&m\_reminderList[i]->GetDate(), sizeof(Entity::Date));

fileWrite.write(m\_reminderList[i]->GetContent(), LENGTH\_OF\_FIELD);

fileWrite.write((char\*)new Id{ m\_reminderList[i]->GetOwnerId() }, sizeof(Id));

fileWrite.write((char\*)&m\_reminderList[i]->GetReminderTime(), sizeof(Entity::Date));

}

fileWrite.close();

}

system("cls");

}

return 0;

}

**ThemeListItem.h**

#ifndef THEME\_LIST\_ITEM\_H

#define THEME\_LIST\_ITEM\_H

#include "AbstractMenuItem.h"

#include "Theme.h"

#include "IdCounter.h"

namespace KMK

{

class ThemeListItem : public MenuItem

{

public:

ThemeListItem(char\* itemName, char\* themeDatabase, char\* idCounterDatabase);

int Run();

private:

char\* m\_themeDatabase = nullptr;

char\* m\_idCounterDatabase = nullptr;

size\_t m\_size{};

Theme\*\* m\_themeList = nullptr;

IdCounter m\_idCounter{};

};

}

#endif // !THEME\_LIST\_ITEM\_H

**ThemeListItem.cpp**

#include "ThemeListItem.h"

#include "Constants.h"

#include <iostream>

#include <fstream>

#include "Add.h"

#include <Windows.h>

#include "Remove.h"

#include "Edit.h"

#include "Sort.h"

#include "Filter.h"

#include <iomanip>

using namespace KMK;

ThemeListItem::ThemeListItem(char\* itemName, char\* themeDatabase, char\* idCounterDatabase) : MenuItem(itemName)

{

m\_themeDatabase = new char[LENGTH\_OF\_FIELD] {};

strcpy\_s(m\_themeDatabase, LENGTH\_OF\_FIELD, themeDatabase);

m\_idCounterDatabase = new char[LENGTH\_OF\_FIELD] {};

strcpy\_s(m\_idCounterDatabase, LENGTH\_OF\_FIELD, idCounterDatabase);

m\_idCounter = { idCounterDatabase };

std::ifstream fileRead(m\_themeDatabase, std::ios::binary);

fileRead.read((char\*)&m\_size, sizeof(size\_t));

m\_themeList = new Theme \* [m\_size] {};

for (Iteration i{}; i < m\_size; ++i)

{

Id id{};

fileRead.read((char\*)&id, sizeof(Id));

Entity::Date date{};

fileRead.read((char\*)&date, sizeof(Entity::Date));

char\* theme = new char[LENGTH\_OF\_FIELD] {};

fileRead.read(theme, LENGTH\_OF\_FIELD);

Id ownerId{};

fileRead.read((char\*)&ownerId, sizeof(Id));

m\_themeList[i] = new Theme{ id, date, theme, ownerId };

}

fileRead.close();

}

int ThemeListItem::Run()

{

enum Command

{

RESET,

ADD,

REMOVE,

EDIT,

SORT,

FILTER,

ID,

EXIT

};

unsigned short command = 0;

while (command != EXIT)

{

unsigned short maximumThemeLength = 5;

for (Iteration i{}; i < m\_size; ++i)

{

if (strlen(m\_themeList[i]->GetContent()) > maximumThemeLength)

{

maximumThemeLength = strlen(m\_themeList[i]->GetContent());

}

}

std::cout << std::setw((11 + 3 + 3 + 5 + maximumThemeLength + 1 + 11 + 5 + strlen(GetItemName())) / 2) << GetItemName() << "\n\n";

std::cout << std::setw(11) << "ID" << "|";

std::cout << std::setw(3) << "dd" << "|";

std::cout << std::setw(3) << "mm" << "|";

std::cout << std::setw(5) << "yyyy" << "|";

std::cout << std::setw(maximumThemeLength + 1) << "Theme" << "|";

std::cout << std::setw(11) << "Owner ID";

std::cout << '\n';

for (Iteration i{}; i < m\_size; ++i)

{

std::cout << std::setw(11) << m\_themeList[i]->GetId() << "|";

std::cout << std::setw(3) << m\_themeList[i]->GetDate().day << "|";

std::cout << std::setw(3) << m\_themeList[i]->GetDate().month << "|";

std::cout << std::setw(5) << m\_themeList[i]->GetDate().year << "|";

std::cout << std::setw(maximumThemeLength + 1) << m\_themeList[i]->GetContent() << "|";

std::cout << std::setw(11) << m\_themeList[i]->GetOwnerId();

std::cout << '\n';

}

std::cout << '\n';

std::cout << RESET << ". Reset list\n";

std::cout << ADD << ". Add new theme\n";

std::cout << REMOVE << ". Delete theme\n";

std::cout << EDIT << ". Edit theme\n";

std::cout << SORT << ". Sort list\n";

std::cout << FILTER << ". Filter list\n";

std::cout << ID << ". Choose ID\n";

std::cout << EXIT << ". Exit\n";

std::cout << "Input command: ";

std::cin >> command;

std::cin.ignore();

std::cout << '\n';

if (command == RESET)

{

std::ifstream fileRead(m\_themeDatabase, std::ios::binary);

fileRead.read((char\*)&m\_size, sizeof(size\_t));

m\_themeList = new Theme \* [m\_size] {};

for (Iteration i{}; i < m\_size; ++i)

{

Id id{};

fileRead.read((char\*)&id, sizeof(Id));

Entity::Date date{};

fileRead.read((char\*)&date, sizeof(Entity::Date));

char\* theme = new char[LENGTH\_OF\_FIELD] {};

fileRead.read(theme, LENGTH\_OF\_FIELD);

Id ownerId{};

fileRead.read((char\*)&ownerId, sizeof(Id));

m\_themeList[i] = new Theme{ id, date, theme, ownerId };

}

fileRead.close();

}

else if (command == ADD)

{

char\* theme = new char[LENGTH\_OF\_FIELD];

std::cout << "Input theme: ";

std::cin.getline(theme, LENGTH\_OF\_FIELD, '\n');

Id ownerId;

std::cout << "Input owner ID: ";

std::cin >> ownerId;

std::cin.ignore();

SYSTEMTIME systemTime;

GetLocalTime(&systemTime);

Theme newTheme = Theme(m\_idCounter(), { systemTime.wDay, systemTime.wMonth, systemTime.wYear }, theme, ownerId);

m\_themeList = Add(m\_themeList, m\_size, newTheme, m\_size);

}

else if (command == REMOVE)

{

std::cout << "Input ID: ";

Id id;

std::cin >> id;

std::cin.ignore();

m\_themeList = Remove(m\_themeList, m\_size, id, m\_size);

}

else if (command == EDIT)

{

std::cout << "Input ID: ";

Id id;

std::cin >> id;

std::cin.ignore();

std::cout << '\n';

std::cout << "Fields to edit\n";

std::cout << "0. ID\n";

std::cout << "1. Date\n";

std::cout << "2. Theme\n";

std::cout << "3. Owner ID\n";

std::cout << "Choose field: ";

unsigned short fieldToChange;

std::cin >> fieldToChange;

std::cin.ignore();

std::cout << '\n';

if (fieldToChange == 0)

{

std::cout << "Input new ID: ";

Id\* newId = new Id{};

std::cin >> \*newId;

std::cin.ignore();

m\_themeList = Edit(m\_themeList, m\_size, id, (void\*)newId, EditMode::ID);

}

if (fieldToChange == 1)

{

std::cout << "Input new date\n";

std::cout << "Day: ";

unsigned short day;

std::cin >> day;

std::cout << "Month: ";

unsigned short month;

std::cin >> month;

std::cout << "Year: ";

unsigned short year;

std::cin >> year;

std::cin.ignore();

m\_themeList = Edit(m\_themeList, m\_size, id, (void\*)new Entity::Date{ day, month, year }, EditMode::DATE);

}

if (fieldToChange == 2)

{

std::cout << "Input new theme: ";

char\* theme = new char[LENGTH\_OF\_FIELD] {};

std::cin.getline(theme, LENGTH\_OF\_FIELD, '\n');

m\_themeList = Edit(m\_themeList, m\_size, id, (void\*)theme, EditMode::CONTENT);

}

if (fieldToChange == 3)

{

std::cout << "Input new owner ID: ";

Id\* newOwnerId = new Id{};

std::cin >> \*newOwnerId;

std::cin.ignore();

m\_themeList = Edit(m\_themeList, m\_size, id, (void\*)newOwnerId, EditMode::OWNER\_ID);

}

}

else if (command == SORT)

{

std::cout << "Orders for sort\n";

std::cout << "0. Descending\n";

std::cout << "1. Ascending\n";

std::cout << "Choose order: ";

unsigned short order;

std::cin >> order;

std::cout << '\n';

std::cout << "Fields for sort\n";

std::cout << "0. ID\n";

std::cout << "1. Date\n";

std::cout << "2. Theme\n";

std::cout << "3. Owner ID\n";

std::cout << "Choose field: ";

unsigned short field;

std::cin >> field;

std::cin.ignore();

SortMode sortMode = (SortMode)-1;

switch (field)

{

case 0:

sortMode = SortMode::ID;

break;

case 1:

sortMode = SortMode::DATE;

break;

case 2:

sortMode = SortMode::CONTENT;

break;

case 3:

sortMode = SortMode::OWNER\_ID;

break;

}

m\_themeList = Sort(m\_themeList, m\_size, (OrderMode)order, sortMode);

}

else if (command == FILTER)

{

std::cout << "Fields for filter\n";

std::cout << "0. ID\n";

std::cout << "1. Date\n";

std::cout << "2. Theme\n";

std::cout << "3. Owner ID\n";

std::cout << "Choose field: ";

unsigned short field;

std::cin >> field;

std::cin.ignore();

std::cout << '\n';

if (field == 0)

{

std::cout << "Input part of ID: ";

Id id;

std::cin >> id;

std::cin.ignore();

m\_themeList = Filter(m\_themeList, m\_size, (void\*)&id, FilterMode::ID, m\_size);

}

if (field == 1)

{

std::cout << "Input date (if you don't want to filter by the field, input 0)\n";

std::cout << "Day: ";

unsigned short day;

std::cin >> day;

std::cout << "Month: ";

unsigned short month;

std::cin >> month;

std::cout << "Year: ";

unsigned short year;

std::cin >> year;

std::cin.ignore();

m\_themeList = Filter(m\_themeList, m\_size, (void\*)new Entity::Date{ day, month, year }, FilterMode::DATE, m\_size);

}

if (field == 2)

{

std::cout << "Input part of theme: ";

char\* theme = new char[LENGTH\_OF\_FIELD] {};

std::cin.getline(theme, LENGTH\_OF\_FIELD, '\n');

m\_themeList = Filter(m\_themeList, m\_size, (void\*)theme, FilterMode::CONTENT, m\_size);

}

if (field == 3)

{

std::cout << "Input part of owner ID: ";

Id ownerId;

std::cin >> ownerId;

std::cin.ignore();

m\_themeList = Filter(m\_themeList, m\_size, (void\*)&ownerId, FilterMode::OWNER\_ID, m\_size);

}

}

if (command == ADD || command == REMOVE || command == EDIT)

{

std::ofstream fileWrite(m\_themeDatabase, std::ios::binary);

fileWrite.write((char\*)&m\_size, sizeof(size\_t));

for (Iteration i{}; i < m\_size; ++i)

{

fileWrite.write((char\*)new Id{ m\_themeList[i]->GetId() }, sizeof(Id));

fileWrite.write((char\*)&m\_themeList[i]->GetDate(), sizeof(Entity::Date));

fileWrite.write(m\_themeList[i]->GetContent(), LENGTH\_OF\_FIELD);

fileWrite.write((char\*)new Id{ m\_themeList[i]->GetOwnerId() }, sizeof(Id));

}

fileWrite.close();

}

system("cls");

}

return 0;

}

**UserListItem.h**

#ifndef USER\_LIST\_ITEM\_H

#define USER\_LIST\_ITEM\_H

#include "AbstractMenuItem.h"

#include "User.h"

#include "IdCounter.h"

namespace KMK

{

class UserListItem : public MenuItem

{

public:

UserListItem(char\* itemName, char\* userDatabase, char\* idCounterDatabase);

int Run();

private:

char\* m\_userDatabase = nullptr;

char\* m\_idCounterDatabase = nullptr;

size\_t m\_size{};

User\*\* m\_userList = nullptr;

IdCounter m\_idCounter{};

};

}

#endif // !USER\_LIST\_ITEM\_H

**UserListItem.cpp**

#include "UserListItem.h"

#include "Constants.h"

#include <iostream>

#include <fstream>

#include "Add.h"

#include <Windows.h>

#include "Remove.h"

#include "Edit.h"

#include "Sort.h"

#include "Filter.h"

#include <iomanip>

using namespace KMK;

UserListItem::UserListItem(char\* itemName, char\* userDatabase, char\* idCounterDatabase) : MenuItem(itemName)

{

m\_userDatabase = new char[LENGTH\_OF\_FIELD] {};

strcpy\_s(m\_userDatabase, LENGTH\_OF\_FIELD, userDatabase);

m\_idCounterDatabase = new char[LENGTH\_OF\_FIELD] {};

strcpy\_s(m\_idCounterDatabase, LENGTH\_OF\_FIELD, idCounterDatabase);

m\_idCounter = { idCounterDatabase };

std::ifstream fileRead(m\_userDatabase, std::ios::binary);

fileRead.read((char\*)&m\_size, sizeof(size\_t));

m\_userList = new User \* [m\_size] {};

for (Iteration i{}; i < m\_size; ++i)

{

Id id{};

fileRead.read((char\*)&id, sizeof(Id));

Entity::Date date{};

fileRead.read((char\*)&date, sizeof(Entity::Date));

char\* name = new char[LENGTH\_OF\_FIELD] {};

fileRead.read(name, LENGTH\_OF\_FIELD);

char\* login = new char[LENGTH\_OF\_FIELD] {};

fileRead.read(login, LENGTH\_OF\_FIELD);

char\* password = new char[LENGTH\_OF\_FIELD] {};

fileRead.read(password, LENGTH\_OF\_FIELD);

m\_userList[i] = new User{ id, date, name, login, password };

}

fileRead.close();

}

int UserListItem::Run()

{

enum Command

{

RESET,

ADD,

REMOVE,

EDIT,

SORT,

FILTER,

ID,

EXIT

};

unsigned short command = 0;

while (command != EXIT)

{

unsigned short maximumNameLength = 4;

unsigned short maximumLoginLength = 5;

unsigned short maximumPasswordLength = 8;

for (Iteration i{}; i < m\_size; ++i)

{

if (strlen(m\_userList[i]->GetName()) > maximumNameLength)

{

maximumNameLength = strlen(m\_userList[i]->GetName());

}

if (strlen(m\_userList[i]->GetLogin()) > maximumLoginLength)

{

maximumLoginLength = strlen(m\_userList[i]->GetLogin());

}

if (strlen(m\_userList[i]->GetPassword()) > maximumPasswordLength)

{

maximumPasswordLength = strlen(m\_userList[i]->GetPassword());

}

}

std::cout << std::setw((11 + 3 + 3 + 5 + maximumNameLength + 1 + maximumLoginLength + 1 + maximumPasswordLength + 1 + 6 + strlen(GetItemName())) / 2) << GetItemName() << "\n\n";

std::cout << std::setw(11) << "ID" << "|";

std::cout << std::setw(3) << "dd" << "|";

std::cout << std::setw(3) << "mm" << "|";

std::cout << std::setw(5) << "yyyy" << "|";

std::cout << std::setw(maximumNameLength + 1) << "Name" << "|";

std::cout << std::setw(maximumLoginLength + 1) << "Login" << "|";

std::cout << std::setw(maximumPasswordLength + 1) << "Password";

std::cout << '\n';

for (Iteration i{}; i < m\_size; ++i)

{

std::cout << std::setw(11) << m\_userList[i]->GetId() << "|";

std::cout << std::setw(3) << m\_userList[i]->GetDate().day << "|";

std::cout << std::setw(3) << m\_userList[i]->GetDate().month << "|";

std::cout << std::setw(5) << m\_userList[i]->GetDate().year << "|";

std::cout << std::setw(maximumNameLength + 1) << m\_userList[i]->GetName() << "|";

std::cout << std::setw(maximumLoginLength + 1) << m\_userList[i]->GetLogin() << "|";

std::cout << std::setw(maximumPasswordLength + 1) << m\_userList[i]->GetPassword();

std::cout << '\n';

}

std::cout << '\n';

std::cout << RESET << ". Reset list\n";

std::cout << ADD << ". Add new user\n";

std::cout << REMOVE << ". Delete user\n";

std::cout << EDIT << ". Edit user\n";

std::cout << SORT << ". Sort list\n";

std::cout << FILTER << ". Filter list\n";

std::cout << ID << ". Choose ID\n";

std::cout << EXIT << ". Exit\n";

std::cout << "Input command: ";

std::cin >> command;

std::cin.ignore();

std::cout << '\n';

if (command == RESET)

{

std::ifstream fileRead(m\_userDatabase, std::ios::binary);

fileRead.read((char\*)&m\_size, sizeof(size\_t));

m\_userList = new User \* [m\_size] {};

for (Iteration i{}; i < m\_size; ++i)

{

Id id{};

fileRead.read((char\*)&id, sizeof(Id));

Entity::Date date{};

fileRead.read((char\*)&date, sizeof(Entity::Date));

char\* name = new char[LENGTH\_OF\_FIELD] {};

fileRead.read(name, LENGTH\_OF\_FIELD);

char\* login = new char[LENGTH\_OF\_FIELD] {};

fileRead.read(login, LENGTH\_OF\_FIELD);

char\* password = new char[LENGTH\_OF\_FIELD] {};

fileRead.read(password, LENGTH\_OF\_FIELD);

m\_userList[i] = new User{ id, date, name, login, password };

}

fileRead.close();

}

else if (command == ADD)

{

char\* name = new char[LENGTH\_OF\_FIELD];

std::cout << "Input name: ";

std::cin.getline(name, LENGTH\_OF\_FIELD, '\n');

char\* login = new char[LENGTH\_OF\_FIELD];

std::cout << "Input login: ";

std::cin.getline(login, LENGTH\_OF\_FIELD, '\n');

char\* password = new char[LENGTH\_OF\_FIELD];

std::cout << "Input password: ";

std::cin.getline(password, LENGTH\_OF\_FIELD, '\n');

SYSTEMTIME systemTime;

GetLocalTime(&systemTime);

User newUser = User(m\_idCounter(), { systemTime.wDay, systemTime.wMonth, systemTime.wYear }, name, login, password);

m\_userList = Add(m\_userList, m\_size, newUser, m\_size);

}

else if (command == REMOVE)

{

std::cout << "Input ID: ";

Id id;

std::cin >> id;

std::cin.ignore();

m\_userList = Remove(m\_userList, m\_size, id, m\_size);

}

else if (command == EDIT)

{

std::cout << "Input ID: ";

Id id;

std::cin >> id;

std::cin.ignore();

std::cout << '\n';

std::cout << "Fields to edit\n";

std::cout << "0. ID\n";

std::cout << "1. Date\n";

std::cout << "2. Name\n";

std::cout << "3. Login\n";

std::cout << "4. Password\n";

std::cout << "Choose field: ";

unsigned short fieldToChange;

std::cin >> fieldToChange;

std::cin.ignore();

std::cout << '\n';

if (fieldToChange == 0)

{

std::cout << "Input new ID: ";

Id\* newId = new Id{};

std::cin >> \*newId;

std::cin.ignore();

m\_userList = Edit(m\_userList, m\_size, id, (void\*)newId, EditMode::ID);

}

if (fieldToChange == 1)

{

std::cout << "Input new date\n";

std::cout << "Day: ";

unsigned short day;

std::cin >> day;

std::cout << "Month: ";

unsigned short month;

std::cin >> month;

std::cout << "Year: ";

unsigned short year;

std::cin >> year;

std::cin.ignore();

m\_userList = Edit(m\_userList, m\_size, id, (void\*)new Entity::Date{ day, month, year }, EditMode::DATE);

}

if (fieldToChange == 2)

{

std::cout << "Input new name: ";

char\* name = new char[LENGTH\_OF\_FIELD] {};

std::cin.getline(name, LENGTH\_OF\_FIELD, '\n');

m\_userList = Edit(m\_userList, m\_size, id, (void\*)name, EditMode::NAME);

}

if (fieldToChange == 3)

{

std::cout << "Input new login: ";

char\* login = new char[LENGTH\_OF\_FIELD] {};

std::cin.getline(login, LENGTH\_OF\_FIELD, '\n');

m\_userList = Edit(m\_userList, m\_size, id, (void\*)login, EditMode::LOGIN);

}

if (fieldToChange == 4)

{

std::cout << "Input new password: ";

char\* password = new char[LENGTH\_OF\_FIELD] {};

std::cin.getline(password, LENGTH\_OF\_FIELD, '\n');

m\_userList = Edit(m\_userList, m\_size, id, (void\*)password, EditMode::PASSWORD);

}

}

else if (command == SORT)

{

std::cout << "Orders for sort\n";

std::cout << "0. Descending\n";

std::cout << "1. Ascending\n";

std::cout << "Choose order: ";

unsigned short order;

std::cin >> order;

std::cout << '\n';

std::cout << "Fields for sort\n";

std::cout << "0. ID\n";

std::cout << "1. Date\n";

std::cout << "2. Name\n";

std::cout << "3. Login\n";

std::cout << "4. Password\n";

std::cout << "Choose field: ";

unsigned short field;

std::cin >> field;

std::cin.ignore();

SortMode sortMode = (SortMode)-1;

switch (field)

{

case 0:

sortMode = SortMode::ID;

break;

case 1:

sortMode = SortMode::DATE;

break;

case 2:

sortMode = SortMode::NAME;

break;

case 3:

sortMode = SortMode::LOGIN;

break;

case 4:

sortMode = SortMode::PASSWORD;

break;

}

m\_userList = Sort(m\_userList, m\_size, (OrderMode)order, sortMode);

}

else if (command == FILTER)

{

std::cout << "Fields for filter\n";

std::cout << "0. ID\n";

std::cout << "1. Date\n";

std::cout << "2. Name\n";

std::cout << "3. Login\n";

std::cout << "4. Password\n";

std::cout << "Choose field: ";

unsigned short field;

std::cin >> field;

std::cin.ignore();

std::cout << '\n';

if (field == 0)

{

std::cout << "Input part of ID: ";

Id id;

std::cin >> id;

std::cin.ignore();

m\_userList = Filter(m\_userList, m\_size, (void\*)&id, FilterMode::ID, m\_size);

}

if (field == 1)

{

std::cout << "Input date (if you don't want to filter by the field, input 0)\n";

std::cout << "Day: ";

unsigned short day;

std::cin >> day;

std::cout << "Month: ";

unsigned short month;

std::cin >> month;

std::cout << "Year: ";

unsigned short year;

std::cin >> year;

std::cin.ignore();

m\_userList = Filter(m\_userList, m\_size, (void\*)new Entity::Date{day, month, year}, FilterMode::DATE, m\_size);

}

if (field == 2)

{

std::cout << "Input part of name: ";

char\* name = new char[LENGTH\_OF\_FIELD] {};

std::cin.getline(name, LENGTH\_OF\_FIELD, '\n');

m\_userList = Filter(m\_userList, m\_size, (void\*)name, FilterMode::NAME, m\_size);

}

if (field == 3)

{

std::cout << "Input part of login: ";

char\* login = new char[LENGTH\_OF\_FIELD] {};

std::cin.getline(login, LENGTH\_OF\_FIELD, '\n');

m\_userList = Filter(m\_userList, m\_size, (void\*)login, FilterMode::LOGIN, m\_size);

}

if (field == 4)

{

std::cout << "Input part of password: ";

char\* password = new char[LENGTH\_OF\_FIELD] {};

std::cin.getline(password, LENGTH\_OF\_FIELD, '\n');

m\_userList = Filter(m\_userList, m\_size, (void\*)password, FilterMode::PASSWORD, m\_size);

}

}

if (command == ADD || command == REMOVE || command == EDIT)

{

std::ofstream fileWrite(m\_userDatabase, std::ios::binary);

fileWrite.write((char\*)&m\_size, sizeof(size\_t));

for (Iteration i{}; i < m\_size; ++i)

{

fileWrite.write((char\*)new Id{ m\_userList[i]->GetId() }, sizeof(Id));

fileWrite.write((char\*)&m\_userList[i]->GetDate(), sizeof(Entity::Date));

fileWrite.write(m\_userList[i]->GetName(), LENGTH\_OF\_FIELD);

fileWrite.write(m\_userList[i]->GetLogin(), LENGTH\_OF\_FIELD);

fileWrite.write(m\_userList[i]->GetPassword(), LENGTH\_OF\_FIELD);

}

fileWrite.close();

}

system("cls");

}

return 0;

}

**CopyList.h**

#ifndef COPY\_LIST\_H

#define COPY\_LIST\_H

#include "Dialogue.h"

#include "Interest.h"

#include "Reminder.h"

#include "Theme.h"

#include "User.h"

namespace KMK

{

Dialogue\*\* CopyList(Dialogue\*\* list, size\_t size);

Interest\*\* CopyList(Interest\*\* list, size\_t size);

Reminder\*\* CopyList(Reminder\*\* list, size\_t size);

Theme\*\* CopyList(Theme\*\* list, size\_t size);

User\*\* CopyList(User\*\* list, size\_t size);

}

#endif // !COPY\_LIST\_H

**CopyList.cpp**

#include "CopyList.h"

using namespace KMK;

Dialogue\*\* KMK::CopyList(Dialogue\*\* list, size\_t size)

{

Dialogue\*\* temp = new Dialogue \* [size];

for (Iteration i{}; i < size; ++i)

{

temp[i] = new Dialogue{ \*list[i] };

}

return temp;

}

Interest\*\* KMK::CopyList(Interest\*\* list, size\_t size)

{

Interest\*\* temp = new Interest \* [size];

for (Iteration i{}; i < size; ++i)

{

temp[i] = new Interest{ \*list[i] };

}

return temp;

}

Reminder\*\* KMK::CopyList(Reminder\*\* list, size\_t size)

{

Reminder\*\* temp = new Reminder \* [size];

for (Iteration i{}; i < size; ++i)

{

temp[i] = new Reminder{ \*list[i] };

}

return temp;

}

Theme\*\* KMK::CopyList(Theme\*\* list, size\_t size)

{

Theme\*\* temp = new Theme \* [size];

for (Iteration i{}; i < size; ++i)

{

temp[i] = new Theme{ \*list[i] };

}

return temp;

}

User\*\* KMK::CopyList(User\*\* list, size\_t size)

{

User\*\* temp = new User \* [size];

for (Iteration i{}; i < size; ++i)

{

temp[i] = new User{ \*list[i] };

}

return temp;

}

**Add.h**

#ifndef ADD\_H

#define ADD\_H

#include "Dialogue.h"

#include "Interest.h"

#include "Reminder.h"

#include "Theme.h"

#include "User.h"

namespace KMK

{

Dialogue\*\* Add(Dialogue\*\* dialogs, size\_t size, Dialogue newElement, size\_t& newSize);

Interest\*\* Add(Interest\*\* interests, size\_t size, Interest newElement, size\_t& newSize);

Reminder\*\* Add(Reminder\*\* reminders, size\_t size, Reminder newElement, size\_t& newSize);

Theme\*\* Add(Theme\*\* themes, size\_t size, Theme newElement, size\_t& newSize);

User\*\* Add(User\*\* users, size\_t size, User newElement, size\_t& newSize);

}

#endif // !ADD\_H

**Add.cpp**

#include "Add.h"

#include "CopyList.h"

using namespace KMK;

Dialogue\*\* KMK::Add(Dialogue\*\* dialogs, size\_t size, Dialogue newElement, size\_t &newSize)

{

Dialogue\*\* temp = new Dialogue \* [size + 1];

for (Iteration i{}; i < size; ++i)

{

temp[i] = new Dialogue{ \*dialogs[i] };

}

temp[size] = new Dialogue{ newElement };

++newSize;

return temp;

}

Interest\*\* KMK::Add(Interest\*\* interests, size\_t size, Interest newElement, size\_t& newSize)

{

Interest\*\* temp = new Interest \* [size + 1];

for (Iteration i{}; i < size; ++i)

{

temp[i] = new Interest{ \*interests[i] };

}

temp[size] = new Interest{ newElement };

++newSize;

return temp;

}

Reminder\*\* KMK::Add(Reminder\*\* reminders, size\_t size, Reminder newElement, size\_t& newSize)

{

Reminder\*\* temp = new Reminder \* [size + 1];

for (Iteration i{}; i < size; ++i)

{

temp[i] = new Reminder{ \*reminders[i] };

}

temp[size] = new Reminder{ newElement };

++newSize;

return temp;

}

Theme\*\* KMK::Add(Theme\*\* themes, size\_t size, Theme newElement, size\_t& newSize)

{

Theme\*\* temp = new Theme \* [size + 1];

for (Iteration i{}; i < size; ++i)

{

temp[i] = new Theme{ \*themes[i] };

}

temp[size] = new Theme{ newElement };

++newSize;

return temp;

}

User\*\* KMK::Add(User\*\* users, size\_t size, User newElement, size\_t& newSize)

{

User\*\* temp = new User \* [size + 1];

for (Iteration i{}; i < size; ++i)

{

temp[i] = new User{ \*users[i] };

}

temp[size] = new User{ newElement };

++newSize;

return temp;

}

**Edit.h**

#ifndef EDIT\_H

#define EDIT\_H

#include "Dialogue.h"

#include "Interest.h"

#include "Reminder.h"

#include "Theme.h"

#include "User.h"

namespace KMK

{

enum class EditMode

{

ID,

DATE,

CONTENT,

OWNER\_ID,

ADRESSEE\_ID,

REMINDER\_TIME,

NAME,

LOGIN,

PASSWORD

};

Dialogue\*\* Edit(Dialogue\*\* dialogs, size\_t size, Id idToEdit, void\* newField, EditMode mode);

Interest\*\* Edit(Interest\*\* interests, size\_t size, Id idToEdit, void\* newField, EditMode mode);

Reminder\*\* Edit(Reminder\*\* reminders, size\_t size, Id idToEdit, void\* newField, EditMode mode);

Theme\*\* Edit(Theme\*\* themes, size\_t size, Id idToEdit, void\* newField, EditMode mode);

User\*\* Edit(User\*\* users, size\_t size, Id idToEdit, void\* newField, EditMode mode);

}

#endif // !EDIT\_H

**Edit.cpp**

#include "Edit.h"

#include "CopyList.h"

using namespace KMK;

Dialogue\*\* KMK::Edit(Dialogue\*\* dialogs, size\_t size, Id idToEdit, void\* newField, EditMode mode)

{

Dialogue\*\* temp = CopyList(dialogs, size);

unsigned short dialogueNumber = 0;

while (dialogueNumber < size)

{

if (temp[dialogueNumber]->GetId() != idToEdit)

{

++dialogueNumber;

}

else

{

break;

}

}

if (dialogueNumber < size)

{

if (mode == EditMode::ID)

{

temp[dialogueNumber]->SetId(\*(Id\*)newField);

}

if (mode == EditMode::DATE)

{

temp[dialogueNumber]->SetDate(\*(Entity::Date\*)newField);

}

if (mode == EditMode::CONTENT)

{

temp[dialogueNumber]->SetContent((char\*)newField);

}

if (mode == EditMode::OWNER\_ID)

{

temp[dialogueNumber]->SetOwnerId(\*(Id\*)newField);

}

if (mode == EditMode::ADRESSEE\_ID)

{

temp[dialogueNumber]->SetAdresseeId(\*(Id\*)newField);

}

}

return temp;

}

Interest\*\* KMK::Edit(Interest\*\* interests, size\_t size, Id idToEdit, void\* newField, EditMode mode)

{

Interest\*\* temp = CopyList(interests, size);

unsigned short interestNumber = 0;

while (interestNumber < size)

{

if (temp[interestNumber]->GetId() != idToEdit)

{

++interestNumber;

}

else

{

break;

}

}

if (interestNumber < size)

{

if (mode == EditMode::ID)

{

temp[interestNumber]->SetId(\*(Id\*)newField);

}

if (mode == EditMode::DATE)

{

temp[interestNumber]->SetDate(\*(Entity::Date\*)newField);

}

if (mode == EditMode::CONTENT)

{

temp[interestNumber]->SetContent((char\*)newField);

}

if (mode == EditMode::OWNER\_ID)

{

temp[interestNumber]->SetOwnerId(\*(Id\*)newField);

}

}

return temp;

}

Reminder\*\* KMK::Edit(Reminder\*\* reminders, size\_t size, Id idToEdit, void\* newField, EditMode mode)

{

Reminder\*\* temp = CopyList(reminders, size);

unsigned short reminderNumber = 0;

while (reminderNumber < size)

{

if (temp[reminderNumber]->GetId() != idToEdit)

{

++reminderNumber;

}

else

{

break;

}

}

if (reminderNumber < size)

{

if (mode == EditMode::ID)

{

temp[reminderNumber]->SetId(\*(Id\*)newField);

}

if (mode == EditMode::DATE)

{

temp[reminderNumber]->SetDate(\*(Entity::Date\*)newField);

}

if (mode == EditMode::CONTENT)

{

temp[reminderNumber]->SetContent((char\*)newField);

}

if (mode == EditMode::OWNER\_ID)

{

temp[reminderNumber]->SetOwnerId(\*(Id\*)newField);

}

if (mode == EditMode::REMINDER\_TIME)

{

temp[reminderNumber]->SetReminderTime(\*(Entity::Date\*)newField);

}

}

return temp;

}

Theme\*\* KMK::Edit(Theme\*\* themes, size\_t size, Id idToEdit, void\* newField, EditMode mode)

{

Theme\*\* temp = CopyList(themes, size);

unsigned short themeNumber = 0;

while (themeNumber < size)

{

if (temp[themeNumber]->GetId() != idToEdit)

{

++themeNumber;

}

else

{

break;

}

}

if (themeNumber < size)

{

if (mode == EditMode::ID)

{

temp[themeNumber]->SetId(\*(Id\*)newField);

}

if (mode == EditMode::DATE)

{

temp[themeNumber]->SetDate(\*(Entity::Date\*)newField);

}

if (mode == EditMode::CONTENT)

{

temp[themeNumber]->SetContent((char\*)newField);

}

if (mode == EditMode::OWNER\_ID)

{

temp[themeNumber]->SetOwnerId(\*(Id\*)newField);

}

}

return temp;

}

User\*\* KMK::Edit(User\*\* users, size\_t size, Id idToEdit, void\* newField, EditMode mode)

{

User\*\* temp = CopyList(users, size);

unsigned short userNumber = 0;

while (userNumber < size)

{

if (temp[userNumber]->GetId() != idToEdit)

{

++userNumber;

}

else

{

break;

}

}

if (userNumber < size)

{

if (mode == EditMode::ID)

{

temp[userNumber]->SetId(\*(Id\*)newField);

}

if (mode == EditMode::DATE)

{

temp[userNumber]->SetDate(\*(Entity::Date\*)newField);

}

if (mode == EditMode::NAME)

{

temp[userNumber]->SetName((char\*)newField);

}

if (mode == EditMode::LOGIN)

{

temp[userNumber]->SetLogin((char\*)newField);

}

if (mode == EditMode::PASSWORD)

{

temp[userNumber]->SetPassword((char\*)newField);

}

}

return temp;

}

**Filter.h**

#ifndef FILTER\_H

#define FILTER\_H

#include "Dialogue.h"

#include "Interest.h"

#include "Reminder.h"

#include "Theme.h"

#include "User.h"

namespace KMK

{

enum class FilterMode

{

ID,

DATE,

CONTENT,

OWNER\_ID,

ADRESSEE\_ID,

REMINDER\_TIME,

NAME,

LOGIN,

PASSWORD

};

Dialogue\*\* Filter(Dialogue\*\* dialogs, size\_t size, void\* fieldForSearch, FilterMode mode, size\_t& newSize);

Interest\*\* Filter(Interest\*\* interests, size\_t size, void\* fieldForSearch, FilterMode mode, size\_t& newSize);

Reminder\*\* Filter(Reminder\*\* reminders, size\_t size, void\* fieldForSearch, FilterMode mode, size\_t& newSize);

Theme\*\* Filter(Theme\*\* themes, size\_t size, void\* fieldForSearch, FilterMode mode, size\_t& newSize);

User\*\* Filter(User\*\* users, size\_t size, void\* fieldForSearch, FilterMode mode, size\_t& newSize);

}

#endif // !FILTER\_H

**Filter.cpp**

#include "Filter.h"

#include "Constants.h"

#include <cmath>

#include <iostream>

#include "CopyList.h"

using namespace KMK;

void RemoveUnwanted(Entity\*\* &entities, Id\* fields, Id\* fieldForSearch, size\_t &size)

{

unsigned long int tens = 10;

unsigned short numberOfDigits = 1;

while (\*fieldForSearch / tens != 0)

{

tens \*= 10;

++numberOfDigits;

}

bool\* indexes = new bool[size] {};

size\_t newSize = 0;

for (Iteration i{}; i < MAXIMUM\_NUMBER\_OF\_DIGITS\_IN\_ID - numberOfDigits + 1; ++i)

{

for (Iteration j{}; j < size; j++)

{

if (\*fieldForSearch == (fields[j] / (int)pow(10, i)) % tens)

{

if (indexes[j] != true)

{

indexes[j] = true;

++newSize;

}

}

}

}

Entity\*\* filteredEntities = new Entity \* [newSize] {};

Iteration numberOfEntity = 0;

for (Iteration i{}; i < size; ++i)

{

if (indexes[i] == true)

{

filteredEntities[numberOfEntity] = entities[i];

++numberOfEntity;

}

}

delete[] entities;

entities = filteredEntities;

size = newSize;

}

void RemoveUnwanted(Entity\*\* &entities, char\*\* fields, char\* fieldForSearch, size\_t &size)

{

unsigned short fieldForSearchLength = strlen(fieldForSearch);

bool\* indexes = new bool[size] {};

int newSize = 0;

for (Iteration i{}; i < size; ++i)

{

for (Iteration j{}; j < strlen(fields[i]) - fieldForSearchLength + 1; ++j)

{

char\* temp = new char[fieldForSearchLength + 1] {};

for (Iteration k{}; k < fieldForSearchLength; ++k)

{

temp[k] = fields[i][j + k];

}

if (strcmp(temp, fieldForSearch) == 0)

{

indexes[i] = true;

++newSize;

break;

}

}

}

Entity\*\* newList = new Entity \* [newSize] {};

unsigned short numberOfElement = 0;

for (Iteration i{}; i < size; ++i)

{

if (indexes[i] == true)

{

newList[numberOfElement] = entities[i];

++numberOfElement;

}

}

delete[] entities;

entities = newList;

size = newSize;

}

void RemoveUnwanted(Entity\*\*& entities, Entity::Date\* fields, Entity::Date\* fieldForSearch, size\_t& size)

{

bool\* indexes = new bool[size] {};

unsigned short newSize = 0;

for (Iteration i{}; i < size; ++i)

{

if ((fields[i].day == fieldForSearch->day || fieldForSearch->day == 0) &&

(fields[i].month == fieldForSearch->month || fieldForSearch->month == 0) &&

(fields[i].year == fieldForSearch->year || fieldForSearch->year == 0))

{

indexes[i] = true;

++newSize;

}

}

Entity\*\* newList = new Entity \* [newSize] {};

unsigned short numberOfElement = 0;

for (Iteration i{}; i < size; ++i)

{

if (indexes[i] == true)

{

newList[numberOfElement] = entities[i];

++numberOfElement;

}

}

delete[] entities;

entities = newList;

size = newSize;

}

void FilterEntities(Entity\*\* &entities, size\_t &size, void\* fieldForSearch, FilterMode mode)

{

if (mode == FilterMode::ID)

{

Id\* fields = new Id[size]{};

for (Iteration i{}; i < size; ++i)

{

fields[i] = entities[i]->GetId();

}

RemoveUnwanted(entities, fields, (Id\*)fieldForSearch, size);

}

else if (mode == FilterMode::DATE)

{

Entity::Date\* fields = new Entity::Date[size]{};

for (Iteration i{}; i < size; ++i)

{

fields[i] = entities[i]->GetDate();

}

RemoveUnwanted(entities, fields, (Entity::Date\*)fieldForSearch, size);

}

}

void FilterTextEntities(TextEntity\*\*& entities, size\_t& size, void\* fieldForSearch, FilterMode mode)

{

if (mode == FilterMode::CONTENT)

{

char\*\* fields = new char\* [size] {};

for (Iteration i{}; i < size; ++i)

{

fields[i] = entities[i]->GetContent();

}

RemoveUnwanted((Entity\*\*&)entities, fields, (char\*)fieldForSearch, size);

}

else if (mode == FilterMode::OWNER\_ID)

{

Id\* fields = new Id[size]{};

for (Iteration i{}; i < size; ++i)

{

fields[i] = entities[i]->GetOwnerId();

}

RemoveUnwanted((Entity\*\*&)entities, fields, (Id\*)fieldForSearch, size);

}

}

Dialogue\*\* KMK::Filter(Dialogue\*\* dialogs, size\_t size, void\* fieldForSearch, FilterMode mode, size\_t& newSize)

{

Dialogue\*\* temp = CopyList(dialogs, size);

newSize = size;

if (mode == FilterMode::ID || mode == FilterMode::DATE)

{

FilterEntities((Entity\*\*&)temp, newSize, fieldForSearch, mode);

}

else if (mode == FilterMode::CONTENT || mode == FilterMode::OWNER\_ID)

{

FilterTextEntities((TextEntity\*\*&)temp, newSize, fieldForSearch, mode);

}

else if (mode == FilterMode::ADRESSEE\_ID)

{

Id\* fields = new Id[size]{};

for (Iteration i{}; i < size; ++i)

{

fields[i] = dialogs[i]->GetAdresseeId();

}

RemoveUnwanted((Entity\*\*&)temp, fields, (Id\*)fieldForSearch, newSize);

}

return temp;

}

Interest\*\* KMK::Filter(Interest\*\* interests, size\_t size, void\* fieldForSearch, FilterMode mode, size\_t &newSize)

{

Interest\*\* temp = CopyList(interests, size);

newSize = size;

if (mode == FilterMode::ID || mode == FilterMode::DATE)

{

FilterEntities((Entity\*\*&)temp, newSize, fieldForSearch, mode);

}

else if (mode == FilterMode::CONTENT || mode == FilterMode::OWNER\_ID)

{

FilterTextEntities((TextEntity\*\*&)temp, newSize, fieldForSearch, mode);

}

return temp;

}

Reminder\*\* KMK::Filter(Reminder\*\* reminders, size\_t size, void\* fieldForSearch, FilterMode mode, size\_t& newSize)

{

Reminder\*\* temp = CopyList(reminders, size);

newSize = size;

if (mode == FilterMode::ID || mode == FilterMode::DATE)

{

FilterEntities((Entity\*\*&)temp, newSize, fieldForSearch, mode);

}

else if (mode == FilterMode::CONTENT || mode == FilterMode::OWNER\_ID)

{

FilterTextEntities((TextEntity\*\*&)temp, newSize, fieldForSearch, mode);

}

else if (mode == FilterMode::REMINDER\_TIME)

{

Entity::Date\* fields = new Entity::Date[size]{};

for (Iteration i{}; i < size; ++i)

{

fields[i] = reminders[i]->GetReminderTime();

}

RemoveUnwanted((Entity\*\*&)temp, fields, (Entity::Date\*)fieldForSearch, newSize);

}

return temp;

}

Theme\*\* KMK::Filter(Theme\*\* themes, size\_t size, void\* fieldForSearch, FilterMode mode, size\_t& newSize)

{

Theme\*\* temp = CopyList(themes, size);

newSize = size;

if (mode == FilterMode::ID || mode == FilterMode::DATE)

{

FilterEntities((Entity\*\*&)temp, newSize, fieldForSearch, mode);

}

else if (mode == FilterMode::CONTENT || mode == FilterMode::OWNER\_ID)

{

FilterTextEntities((TextEntity\*\*&)temp, newSize, fieldForSearch, mode);

}

return temp;

}

User\*\* KMK::Filter(User\*\* users, size\_t size, void\* fieldForSearch, FilterMode mode, size\_t& newSize)

{

User\*\* temp = CopyList(users, size);

newSize = size;

if (mode == FilterMode::ID || mode == FilterMode::DATE)

{

FilterEntities((Entity\*\*&)temp, newSize, fieldForSearch, mode);

}

else if (mode == FilterMode::NAME || mode == FilterMode::LOGIN || mode == FilterMode::PASSWORD)

{

char\*\* fields = new char\* [size] {};

if (mode == FilterMode::NAME)

{

for (Iteration i{}; i < size; ++i)

{

fields[i] = users[i]->GetName();

}

}

if (mode == FilterMode::LOGIN)

{

for (Iteration i{}; i < size; ++i)

{

fields[i] = users[i]->GetLogin();

}

}

if (mode == FilterMode::PASSWORD)

{

for (Iteration i{}; i < size; ++i)

{

fields[i] = users[i]->GetPassword();

}

}

RemoveUnwanted((Entity\*\*&)temp, fields, (char\*)fieldForSearch, newSize);

}

return temp;

}

**Remove.h**

#ifndef REMOVE\_H

#define REMOVE\_H

#include "Dialogue.h"

#include "Interest.h"

#include "Reminder.h"

#include "Theme.h"

#include "User.h"

namespace KMK

{

Dialogue\*\* Remove(Dialogue\*\* dialogs, size\_t size, Id idToRemove, size\_t& newSize);

Interest\*\* Remove(Interest\*\* interests, size\_t size, Id idToRemove, size\_t& newSize);

Reminder\*\* Remove(Reminder\*\* reminders, size\_t size, Id idToRemove, size\_t& newSize);

Theme\*\* Remove(Theme\*\* themes, size\_t size, Id idToRemove, size\_t& newSize);

User\*\* Remove(User\*\* users, size\_t size, Id idToRemove, size\_t& newSize);

}

#endif // !REMOVE\_H

**Remove.cpp**

#include "Remove.h"

using namespace KMK;

Dialogue\*\* KMK::Remove(Dialogue\*\* dialogs, size\_t size, Id idToRemove, size\_t& newSize)

{

bool found = false;

for (Iteration i{}; i < size; ++i)

{

if (idToRemove == dialogs[i]->GetId())

{

found = true;

}

}

if (found == true)

{

Dialogue\*\* temp = new Dialogue \* [size - 1];

unsigned short tempElementNumber = 0;

for (Iteration i{}; i < size; ++i)

{

if (dialogs[i]->GetId() != idToRemove)

{

temp[tempElementNumber] = new Dialogue{ \*dialogs[i] };

++tempElementNumber;

}

}

--newSize;

return temp;

}

else

{

return dialogs;

}

}

Interest\*\* KMK::Remove(Interest\*\* interests, size\_t size, Id idToRemove, size\_t& newSize)

{

bool found = false;

for (Iteration i{}; i < size; ++i)

{

if (idToRemove == interests[i]->GetId())

{

found = true;

}

}

if (found == true)

{

Interest\*\* temp = new Interest \* [size - 1];

unsigned short tempElementNumber = 0;

for (Iteration i{}; i < size; ++i)

{

if (interests[i]->GetId() != idToRemove)

{

temp[tempElementNumber] = new Interest{ \*interests[i] };

++tempElementNumber;

}

}

--newSize;

return temp;

}

else

{

return interests;

}

}

Reminder\*\* KMK::Remove(Reminder\*\* reminders, size\_t size, Id idToRemove, size\_t& newSize)

{

bool found = false;

for (Iteration i{}; i < size; ++i)

{

if (idToRemove == reminders[i]->GetId())

{

found = true;

}

}

if (found == true)

{

Reminder\*\* temp = new Reminder \* [size - 1];

unsigned short tempElementNumber = 0;

for (Iteration i{}; i < size; ++i)

{

if (reminders[i]->GetId() != idToRemove)

{

temp[tempElementNumber] = new Reminder{ \*reminders[i] };

++tempElementNumber;

}

}

--newSize;

return temp;

}

else

{

return reminders;

}

}

Theme\*\* KMK::Remove(Theme\*\* themes, size\_t size, Id idToRemove, size\_t& newSize)

{

bool found = false;

for (Iteration i{}; i < size; ++i)

{

if (idToRemove == themes[i]->GetId())

{

found = true;

}

}

if (found == true)

{

Theme\*\* temp = new Theme \* [size - 1];

unsigned short tempElementNumber = 0;

for (Iteration i{}; i < size; ++i)

{

if (themes[i]->GetId() != idToRemove)

{

temp[tempElementNumber] = new Theme{ \*themes[i] };

++tempElementNumber;

}

}

--newSize;

return temp;

}

else

{

return themes;

}

}

User\*\* KMK::Remove(User\*\* users, size\_t size, Id idToRemove, size\_t& newSize)

{

bool found = false;

for (Iteration i{}; i < size; ++i)

{

if (idToRemove == users[i]->GetId())

{

found = true;

}

}

if (found == true)

{

User\*\* temp = new User \* [size - 1];

unsigned short tempElementNumber = 0;

for (Iteration i{}; i < size; ++i)

{

if (users[i]->GetId() != idToRemove)

{

temp[tempElementNumber] = new User{ \*users[i] };

++tempElementNumber;

}

}

--newSize;

return temp;

}

else

{

return users;

}

}

**Sort.h**

#ifndef SORT\_H

#define SORT\_H

#include "Dialogue.h"

#include "Interest.h"

#include "Reminder.h"

#include "Theme.h"

#include "User.h"

namespace KMK

{

enum class OrderMode

{

DESCENDING,

ASCENDING

};

enum class SortMode

{

ID,

DATE,

CONTENT,

OWNER\_ID,

ADRESSEE\_ID,

REMINDER\_TIME,

NAME,

LOGIN,

PASSWORD

};

Dialogue\*\* Sort(Dialogue\*\* dialogs, size\_t size, OrderMode order, SortMode mode);

Interest\*\* Sort(Interest\*\* interests, size\_t size, OrderMode order, SortMode mode);

Reminder\*\* Sort(Reminder\*\* reminders, size\_t size, OrderMode order, SortMode mode);

Theme\*\* Sort(Theme\*\* themes, size\_t size, OrderMode order, SortMode mode);

User\*\* Sort(User\*\* users, size\_t size, OrderMode order, SortMode mode);

}

#endif // !SORT\_H

**Sort.cpp**

#include "Sort.h"

#include <iostream>

#include "CopyList.h"

using namespace KMK;

void JustSwap(Entity\* &firstEntity, Entity\* &secondEntity)

{

Entity\* temp = firstEntity;

firstEntity = secondEntity;

secondEntity = temp;

}

void JustSwap(Id& firstField, Id& secondField)

{

Id temp = firstField;

firstField = secondField;

secondField = temp;

}

void JustSwap(Entity::Date& firstField, Entity::Date& secondField)

{

Entity::Date temp = firstField;

firstField = secondField;

secondField = temp;

}

void JustSwap(char\* &firstField, char\* &secondField)

{

char\* temp = firstField;

firstField = secondField;

secondField = temp;

}

void Swap(Entity\* &firstEntity, Entity\* &secondEntity,

Id& firstField, Id& secondField, OrderMode order)

{

if (order == OrderMode::DESCENDING)

{

if (firstField < secondField)

{

JustSwap(firstEntity, secondEntity);

JustSwap(firstField, secondField);

}

}

else if (order == OrderMode::ASCENDING)

{

if (firstField > secondField)

{

JustSwap(firstEntity, secondEntity);

JustSwap(firstField, secondField);

}

}

}

void Swap(Entity\* &firstEntity, Entity\* &secondEntity,

Entity::Date& firstField, Entity::Date& secondField, OrderMode order)

{

if (order == OrderMode::DESCENDING)

{

if (firstField.year < secondField.year)

{

JustSwap(firstEntity, secondEntity);

JustSwap(firstField, secondField);

}

else if (firstField.year == secondField.year)

{

if (firstField.month < secondField.month)

{

JustSwap(firstEntity, secondEntity);

JustSwap(firstField, secondField);

}

else if (firstField.month == secondField.month)

{

if (firstField.day < secondField.day)

{

JustSwap(firstEntity, secondEntity);

JustSwap(firstField, secondField);

}

}

}

}

else if (order == OrderMode::ASCENDING)

{

if (firstField.year > secondField.year)

{

JustSwap(firstEntity, secondEntity);

JustSwap(firstField, secondField);

}

else if (firstField.year == secondField.year)

{

if (firstField.month > secondField.month)

{

JustSwap(firstEntity, secondEntity);

JustSwap(firstField, secondField);

}

else if (firstField.month == secondField.month)

{

if (firstField.day > secondField.day)

{

JustSwap(firstEntity, secondEntity);

JustSwap(firstField, secondField);

}

}

}

}

}

void Swap(Entity\* &firstEntity, Entity\* &secondEntity,

char\* &firstField, char\* &secondField, OrderMode order)

{

Iteration letter = 0;

while (firstField[letter] == secondField[letter] &&

letter < strlen(firstField) - 1 && letter < strlen(secondField) - 1)

{

++letter;

}

if (order == OrderMode::DESCENDING)

{

if (firstField[letter] < secondField[letter])

{

JustSwap(firstEntity, secondEntity);

JustSwap(firstField, secondField);

}

}

else if (order == OrderMode::ASCENDING)

{

if (firstField[letter] > secondField[letter])

{

JustSwap(firstEntity, secondEntity);

JustSwap(firstField, secondField);

}

}

}

void SortByShaker(Entity\*\* entities, Id\* fields, size\_t size, OrderMode order)

{

short bottomBorder = 0;

short upperBorder = size - 1;

while (upperBorder - bottomBorder > 0)

{

for (Iteration i = bottomBorder; i < upperBorder; ++i)

{

Swap(entities[i], entities[i + 1], fields[i], fields[i + 1], order);

}

--upperBorder;

for (Iteration i = upperBorder; i > bottomBorder; --i)

{

Swap(entities[i - 1], entities[i], fields[i - 1], fields[i], order);

}

++bottomBorder;

}

}

void SortByShaker(Entity\*\* entities, Entity::Date\* fields, size\_t size, OrderMode order)

{

short bottomBorder = 0;

short upperBorder = size - 1;

while (upperBorder - bottomBorder > 0)

{

for (Iteration i = bottomBorder; i < upperBorder; ++i)

{

Swap(entities[i], entities[i + 1], fields[i], fields[i + 1], order);

}

--upperBorder;

for (Iteration i = upperBorder; i > bottomBorder; --i)

{

Swap(entities[i - 1], entities[i], fields[i - 1], fields[i], order);

}

++bottomBorder;

}

}

void SortByShaker(Entity\*\* entities, char\*\* fields, size\_t size, OrderMode order)

{

short bottomBorder = 0;

short upperBorder = size - 1;

while (upperBorder - bottomBorder > 0)

{

for (Iteration i = bottomBorder; i < upperBorder; ++i)

{

Swap(entities[i], entities[i + 1], fields[i], fields[i + 1], order);

}

--upperBorder;

for (Iteration i = upperBorder; i > bottomBorder; --i)

{

Swap(entities[i - 1], entities[i], fields[i - 1], fields[i], order);

}

++bottomBorder;

}

}

void SortEntities(Entity\*\* entities, size\_t size, OrderMode order, SortMode mode)

{

if (mode == SortMode::ID)

{

Id\* fields = new Id[size] {};

for (Iteration i = 0; i < size; ++i)

{

fields[i] = entities[i]->GetId();

}

SortByShaker(entities, fields, size, order);

}

else if (mode == SortMode::DATE)

{

Entity::Date\* fields = new Entity::Date[size] {};

for (Iteration i = 0; i < size; ++i)

{

fields[i] = entities[i]->GetDate();

}

SortByShaker(entities, fields, size, order);

}

}

void SortTextEntities(TextEntity\*\* entities, size\_t size, OrderMode order, SortMode mode)

{

if (mode == SortMode::CONTENT)

{

char\*\* fields = new char\*[size] {};

for (Iteration i = 0; i < size; ++i)

{

fields[i] = entities[i]->GetContent();

}

SortByShaker((Entity\*\*)entities, fields, size, order);

}

else if (mode == SortMode::OWNER\_ID)

{

Id\* fields = new Id[size] {};

for (Iteration i = 0; i < size; ++i)

{

fields[i] = entities[i]->GetOwnerId();

}

SortByShaker((Entity\*\*)entities, fields, size, order);

}

}

Dialogue\*\* KMK::Sort(Dialogue\*\* dialogs, size\_t size, OrderMode order, SortMode mode)

{

if (size > 1)

{

Dialogue\*\* temp = CopyList(dialogs, size);

if (mode == SortMode::ID || mode == SortMode::DATE)

{

SortEntities((Entity\*\*)temp, size, order, mode);

}

else if (mode == SortMode::CONTENT || mode == SortMode::OWNER\_ID)

{

SortTextEntities((TextEntity\*\*)temp, size, order, mode);

}

else if (mode == SortMode::ADRESSEE\_ID)

{

Id\* fields = new Id[size]{};

for (Iteration i = 0; i < size; ++i)

{

fields[i] = temp[i]->GetAdresseeId();

}

SortByShaker((Entity\*\*)temp, fields, size, order);

}

return temp;

}

else

{

return dialogs;

}

}

Interest\*\* KMK::Sort(Interest\*\* interests, size\_t size, OrderMode order, SortMode mode)

{

if (size > 1)

{

Interest\*\* temp = CopyList(interests, size);

if (mode == SortMode::ID || mode == SortMode::DATE)

{

SortEntities((Entity\*\*)temp, size, order, mode);

}

else if (mode == SortMode::CONTENT || mode == SortMode::OWNER\_ID)

{

SortTextEntities((TextEntity\*\*)temp, size, order, mode);

}

return temp;

}

else

{

return interests;

}

}

Reminder\*\* KMK::Sort(Reminder\*\* reminders, size\_t size, OrderMode order, SortMode mode)

{

if (size > 1)

{

Reminder\*\* temp = CopyList(reminders, size);

if (mode == SortMode::ID || mode == SortMode::DATE)

{

SortEntities((Entity\*\*)temp, size, order, mode);

}

else if (mode == SortMode::CONTENT || mode == SortMode::OWNER\_ID)

{

SortTextEntities((TextEntity\*\*)temp, size, order, mode);

}

else if (mode == SortMode::REMINDER\_TIME)

{

Entity::Date\* fields = new Entity::Date[size]{};

for (Iteration i = 0; i < size; ++i)

{

fields[i] = temp[i]->GetReminderTime();

}

SortByShaker((Entity\*\*)temp, fields, size, order);

}

return temp;

}

else

{

return reminders;

}

}

Theme\*\* KMK::Sort(Theme\*\* themes, size\_t size, OrderMode order, SortMode mode)

{

if (size > 1)

{

Theme\*\* temp = CopyList(themes, size);

if (mode == SortMode::ID || mode == SortMode::DATE)

{

SortEntities((Entity\*\*)temp, size, order, mode);

}

else if (mode == SortMode::CONTENT || mode == SortMode::OWNER\_ID)

{

SortTextEntities((TextEntity\*\*)temp, size, order, mode);

}

return temp;

}

else

{

return themes;

}

}

User\*\* KMK::Sort(User\*\* users, size\_t size, OrderMode order, SortMode mode)

{

if (size > 1)

{

User\*\* temp = CopyList(users, size);

if (mode == SortMode::ID || mode == SortMode::DATE)

{

SortEntities((Entity\*\*)temp, size, order, mode);

}

else if (mode == SortMode::NAME)

{

char\*\* fields = new char\* [size] {};

for (Iteration i = 0; i < size; ++i)

{

fields[i] = temp[i]->GetName();

}

SortByShaker((Entity\*\*)temp, fields, size, order);

}

else if (mode == SortMode::LOGIN)

{

char\*\* fields = new char\* [size] {};

for (Iteration i = 0; i < size; ++i)

{

fields[i] = temp[i]->GetLogin();

}

SortByShaker((Entity\*\*)temp, fields, size, order);

}

else if (mode == SortMode::PASSWORD)

{

char\*\* fields = new char\* [size] {};

for (Iteration i = 0; i < size; ++i)

{

fields[i] = temp[i]->GetPassword();

}

SortByShaker((Entity\*\*)temp, fields, size, order);

}

return temp;

}

else

{

return users;

}

}

**Menu.h**

#ifndef MENU\_H

#define MENU\_H

#include "AbstractMenuItem.h"

#include <iostream>

namespace KMK

{

class Menu

{

public:

Menu(char\* title, MenuItem\*\* items, size\_t count);

int GetSelect();

bool GetRunning();

char\* GetTitle();

size\_t GetCount();

MenuItem\*\* GetItems();

void Print() const;

int RunCommand() const;

friend std::ostream& operator<<(std::ostream& out, const Menu& menu);

friend std::istream& operator>>(std::istream& in, const Menu& menu);

private:

int m\_select = -1;

bool m\_running = false;

char\* m\_title = nullptr;

size\_t m\_count{};

MenuItem\*\* m\_items = nullptr;

};

std::ostream& operator<<(std::ostream& out, const Menu& menu);

std::istream& operator>>(std::istream& in, const Menu& menu);

}

#endif // !MENU\_H

**Menu.cpp**

#include "Menu.h"

#include "Constants.h"

#include <iostream>

#include "TypeDefinitions.h"

using namespace KMK;

Menu::Menu(char\* title, MenuItem\*\* items, size\_t count)

{

m\_title = new char[LENGTH\_OF\_FIELD];

strcpy\_s(m\_title, LENGTH\_OF\_FIELD, title);

m\_items = items;

m\_count = count;

}

int Menu::GetSelect() { return m\_select; }

bool Menu::GetRunning() { return m\_running; }

char\* Menu::GetTitle() { return m\_title; }

size\_t Menu::GetCount() { return m\_count; }

MenuItem\*\* Menu::GetItems() { return m\_items; }

void Menu::Print() const

{

std::cout << '\t' << m\_title << "\n\n";

for (Iteration i = 0; i < m\_count; i++)

{

std::cout << i << ". ";

m\_items[i]->PrintItemName();

std::cout << '\n';

}

}

int Menu::RunCommand() const

{

Print();

std::cout << m\_count << ". Exit\n";

std::cout << "Enter command: ";

unsigned short command;

std::cin >> command;

system("cls");

if (command != m\_count)

{

return m\_items[command]->Run();

}

else

{

return 1;

}

}

std::ostream& KMK::operator<<(std::ostream& out, const Menu& menu)

{

menu.Print();

return out;

}

std::istream& KMK::operator>>(std::istream& in, const Menu& menu)

{

int code = 0;

while (code == 0)

{

code = menu.RunCommand();

system("cls");

}

return in;

}

**Main.cpp**

#include "Menu.h"

#include "UserListItem.h"

#include "DialogueListItem.h"

#include "InterestListItem.h"

#include "ReminderListItem.h"

#include "ThemeListItem.h"

#include <iostream>

using namespace KMK;

int main()

{

UserListItem users = UserListItem((char\*)"User list", (char\*)"User database.dat", (char\*)"User IDs.dat");

DialogueListItem dialogs = DialogueListItem((char\*)"Dialogue list", (char\*)"Dialogue database.dat", (char\*)"Dialogue IDs.dat");

InterestListItem interests = InterestListItem((char\*)"Interest list", (char\*)"Interst database.dat", (char\*)"Interst IDs.dat");

ReminderListItem reminders = ReminderListItem((char\*)"Reminder list", (char\*)"Reminder database.dat", (char\*)"Reminder IDs.dat");

ThemeListItem themes = ThemeListItem((char\*)"Theme list", (char\*)"Theme database.dat", (char\*)"Theme IDs.dat");

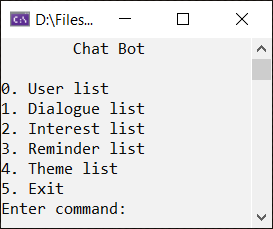
Menu menu = Menu((char\*)"Chat Bot", new MenuItem\*[5] { &users, &dialogs, &interests, &reminders, &themes }, 5);

std::cin >> menu;

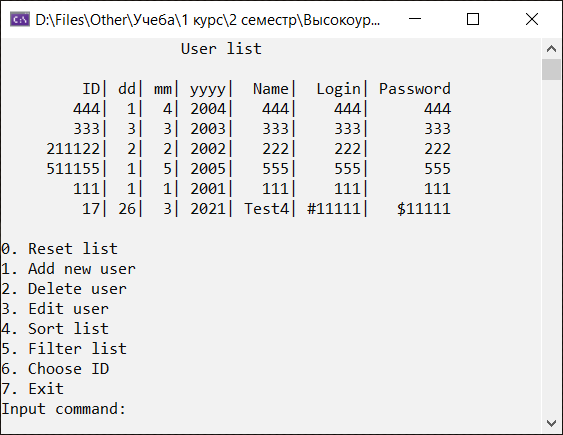
return 0;

}

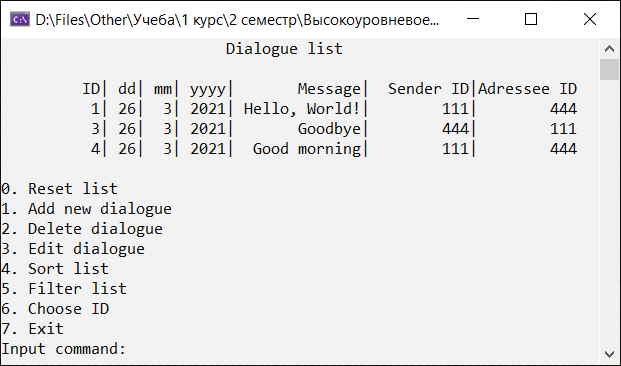
**Демонстрация:**



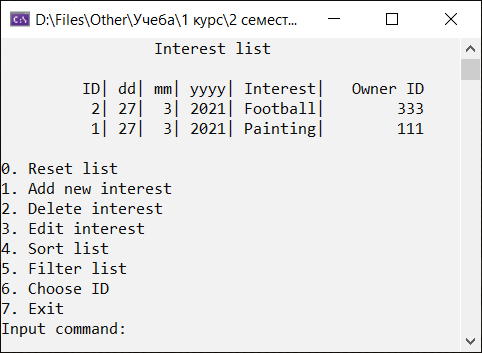
**Рисунок 2.** Главное меню



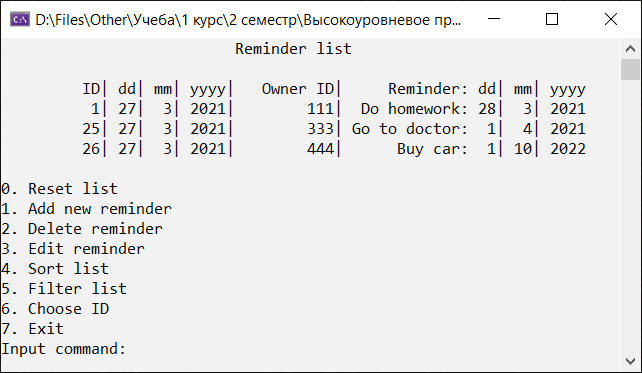
**Рисунок 3.** База данных пользователей



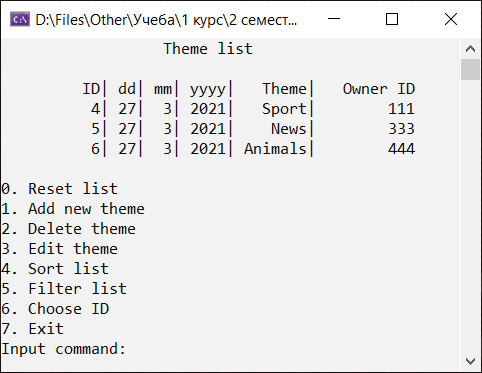
**Рисунок 4.** База данных диалогов



**Рисунок 5.** База данных интересов



**Рисунок 6.** База данных напоминаний



**Рисунок 7.** База данных тем

**Вывод:** в ходе выполнения лабораторной работы были получены практические навыки перегрузки операторов “<<”, “>>” и “()” через дружественные функции, создания функторов, работы с базами данных, сортировки и фильтрации массивов данных, создания, удаления и изменения данных.