#include <iostream>

#include <string>

#include <array>

#include "multilist.hpp"

#include <locale>

#include <windows.h> // для SetConsoleCP и SetConsoleOutputCP

struct Student {

std::string surname;

std::string location;

std::array<int, 3> grades{};

bool has\_excellence{};

bool is\_accomodation\_required{};

};

auto operator==(Student const& lhs, Student const& rhs) -> bool

{

return lhs.surname == rhs.surname and lhs.location == rhs.location;

}

auto operator<<(std::ostream& out, Student const& s) -> std::ostream&

{

return out << s.surname << ' '

<< s.location << ' '

<< s.grades[0] << ' ' << s.grades[1] << ' ' << s.grades[2] << ' '

<< s.has\_excellence << ' ' << s.is\_accomodation\_required;

}

auto operator>>(std::istream& in, Student& s) -> std::istream&

{

return in >> s.surname

>> s.location

>> s.grades[0] >> s.grades[1] >> s.grades[2]

>> s.has\_excellence >> s.is\_accomodation\_required;

}

auto is\_all\_grades\_5(Student const& student) -> bool

{

return student.grades[0] == 5 and student.grades[1] == 5 and student.grades[2] == 5;

}

auto has\_excellence(Student const& student) -> bool

{

return student.has\_excellence;

}

auto is\_outside(Student const& student) -> bool

{

return student.location != "Орёл";

}

auto is\_accomodation\_required(Student const& student) -> bool

{

return student.is\_accomodation\_required;

}

using namespace std;

int main() {

SetConsoleCP(1251);

SetConsoleOutputCP(1251);

int action, kriteriy;

using List = Multilist<Student, 4>;

array<List::Predicate\*, 4> predicates{ is\_all\_grades\_5, has\_excellence, is\_outside, is\_accomodation\_required };

List list{ predicates };

Student student;

do {

cout

<< "Выберите действие: " << endl

<< "1. Ввод записи о студенте" << endl

<< "2. Вывод по критерию" << endl

<< "3. Удаление записи" << endl

<< "4. Удаление всех записей" << endl;

cin >> action;

switch (action) {

case 1: { //Ввод

cout << "Введите через пробел фамилию,название населённого пункта студента, оценки за 3 экзамена, наличие аттестата с отличием (0/1) и необходимость предоставления общежития (0/1)" << endl;

cin >> student;

list.enlist(student);

} break;

case 2: { //Вывод

cout

<< "Выберите критерий вывода: " << endl

<< "0. Все экзамены сданы на 5" << endl

<< "1. Имеется аттестат с отличием" << endl

<< "2. Проживает за пределами населённого пункта, в котором расположен университет" << endl

<< "3. Нуждается в общежитии" << endl

<< "4. Всё " << endl;

cin >> kriteriy;

if ((kriteriy >= 0) && (kriteriy < 5))

{

list.print\_to\_cout(kriteriy);

}

else cout << "Ошибка, попробуйте ввести другое число." << endl;

} break;

case 3: { //Удаление записи

cout << "Введите фамилию и название населённого пункта студента:" << endl;

cin >> student.surname >> student.location;

list.delist(student);

} break;

case 4: { //Удаление всех записей

list.clear();

} break;

default: cout << "Ошибка, попробуйте ввести другое число." << endl;

}

} while (true);

}  
  
  
  
#include <array>

#include <iostream>

template<typename El\_type, int size>

struct Multilist

{

using Predicate = bool(El\_type const&);

static constexpr int common\_list\_index{ size };

Multilist(std::array<Predicate\*, size> const& predicates)

: m\_predicates{ predicates } {}

~Multilist()

{

clear();

}

void clear()

{

for\_each\_node(common\_list\_index, [](Node\_pair node\_pair) -> bool

{ delete node\_pair.node;

return true; });

for (auto& descriptor : m\_descriptors)

{

descriptor.first = nullptr;

descriptor.last = nullptr;

}

}

auto is\_empty(int list\_index) const noexcept -> bool

{

return m\_descriptors[list\_index].first == nullptr;

}

auto enlist(El\_type const& element) -> Multilist&

{

auto add\_to\_list = [this](int list\_index, Node\* new\_node)

{ auto& descriptor = m\_descriptors[list\_index];

if (is\_empty(list\_index))

{

descriptor.first = new\_node;

descriptor.last = new\_node;

}

else {

descriptor.last->next[list\_index] = new\_node;

descriptor.last = new\_node;

}

};

auto new\_node = new Node{ element };

add\_to\_list(common\_list\_index, new\_node);

for (int i = 0; i < static\_cast<int>(m\_predicates.size()); ++i)

{

if (m\_predicates[i](new\_node->data)) add\_to\_list(i, new\_node);

}

return \*this;

}

auto delist(El\_type const& element) -> bool

{

auto find\_in\_lists = [this](El\_type const& element)->std::array<Node\_pair, size + 1>

{ std::array<Node\_pair, size + 1> result;

for (int i = 0; i < static\_cast<int>(m\_descriptors.size()); ++i)

{

for\_each\_node(i, [&](Node\_pair node\_pair)

{ if (node\_pair.node->data == element)

{

result[i] = node\_pair;

return false;

}

else {

return true;

}

});

}

return result;

};

auto unlink\_from\_list = [this](int list\_index, Node\_pair node\_pair)

{ if (m\_descriptors[list\_index].first == node\_pair.node)

m\_descriptors[list\_index].first = node\_pair.node->next[list\_index];

else node\_pair.prev->next[list\_index] = node\_pair.node->next[list\_index];

if (m\_descriptors[list\_index].last == node\_pair.node)

m\_descriptors[list\_index].last = nullptr;

node\_pair.node->next[list\_index] = nullptr;

};

Node\* node{};

auto node\_pairs = find\_in\_lists(element);

for (int i{}; i < static\_cast<int>(node\_pairs.size()); ++i)

{

if (node\_pairs[i].node != nullptr)

{

node = node\_pairs[i].node;

unlink\_from\_list(i, node\_pairs[i]);

}

}

if (node != nullptr)

{

delete node;

return true;

}

else

return false;

}

// Принимает функцию f типа El\_type const& -> bool, где возвращаемое значение означает:

// true - продолжить обход

// false - завершить обход

template<typename Function\_El\_type\_to\_bool>

void for\_each(int list\_index, Function\_El\_type\_to\_bool f)

{

for\_each\_node(list\_index, [f](Node\_pair node\_pair)

{ return f(node\_pair.node->data); });

}

void print\_to\_cout(int list\_index)

{

for\_each\_node(list\_index, [](Node\_pair node\_pair) -> bool

{ std::cout << node\_pair.node->data << '\n';

return true;

});

}

private:

struct Node {

El\_type data;

std::array<Node\*, size + 1> next{};

};

struct Node\_pair {

Node\* node{};

Node\* prev{};

};

struct List\_descriptor {

Node\* first{};

Node\* last{};

};

// Принимает функцию f типа Node\_pair -> bool, где возвращаемое значение означает:

// true - продолжить обход

// false - завершить обход

template<typename Function\_Node\_pair\_to\_bool>

void for\_each\_node(int list\_index, Function\_Node\_pair\_to\_bool f)

{

List\_descriptor descriptor{ m\_descriptors[list\_index] };

Node\* curr{ descriptor.first };

Node\* next{};

Node\* prev{};

for (; curr != nullptr; prev = curr, curr = next)

{

next = curr->next[list\_index];

if (!f(Node\_pair{ curr, prev })) return;

}

}

std::array<List\_descriptor, size + 1> m\_descriptors;

std::array<Predicate\*, size> m\_predicates{};

};