Министерство науки и высшего образования Российской Федерации Федеральное государственное автономное учреждение высшего образования «Пермский национальный исследовательский политехнический университет»

ПНИПУ

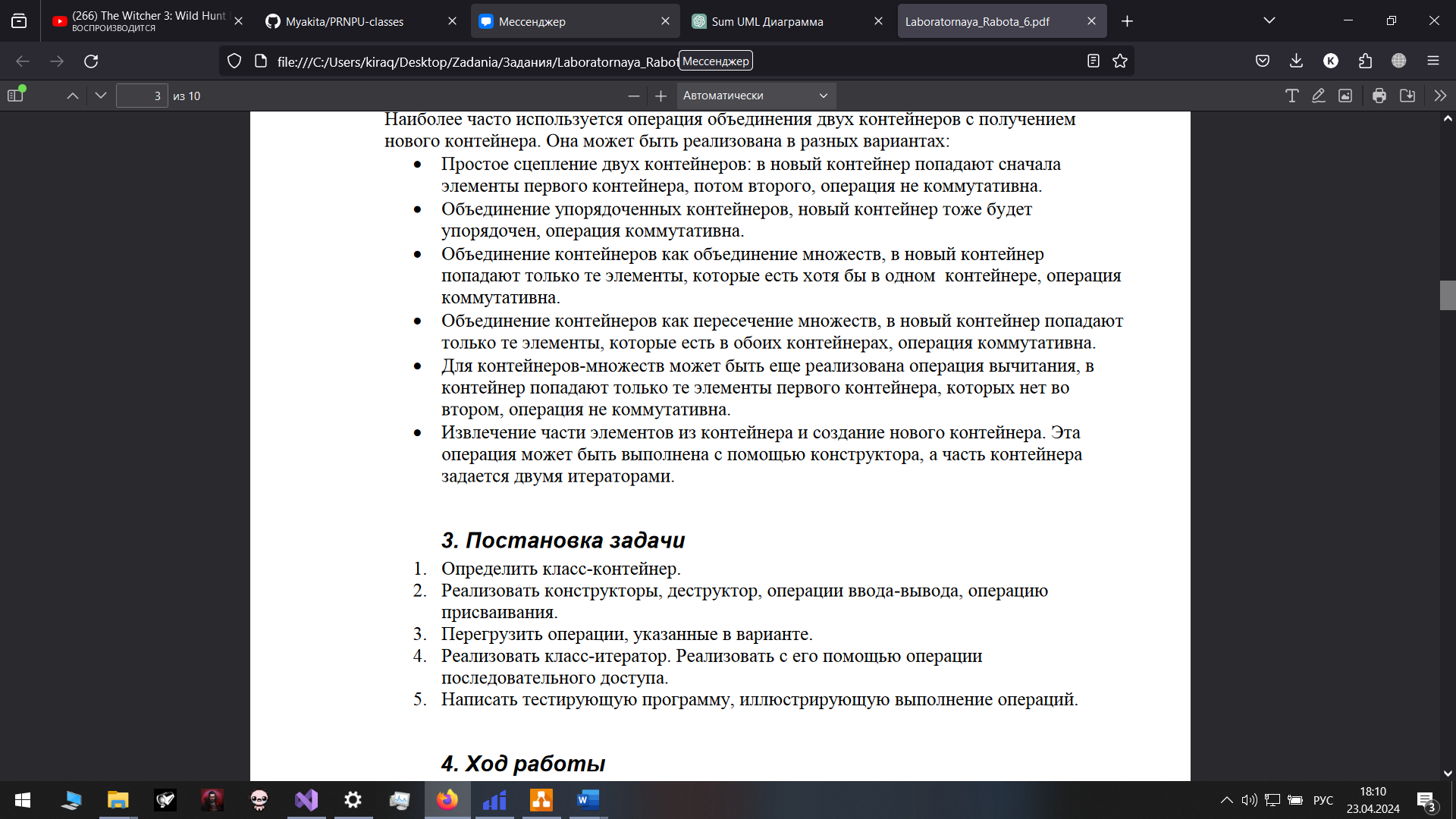
Лабораторная работа  
«Контейнеры АТД»

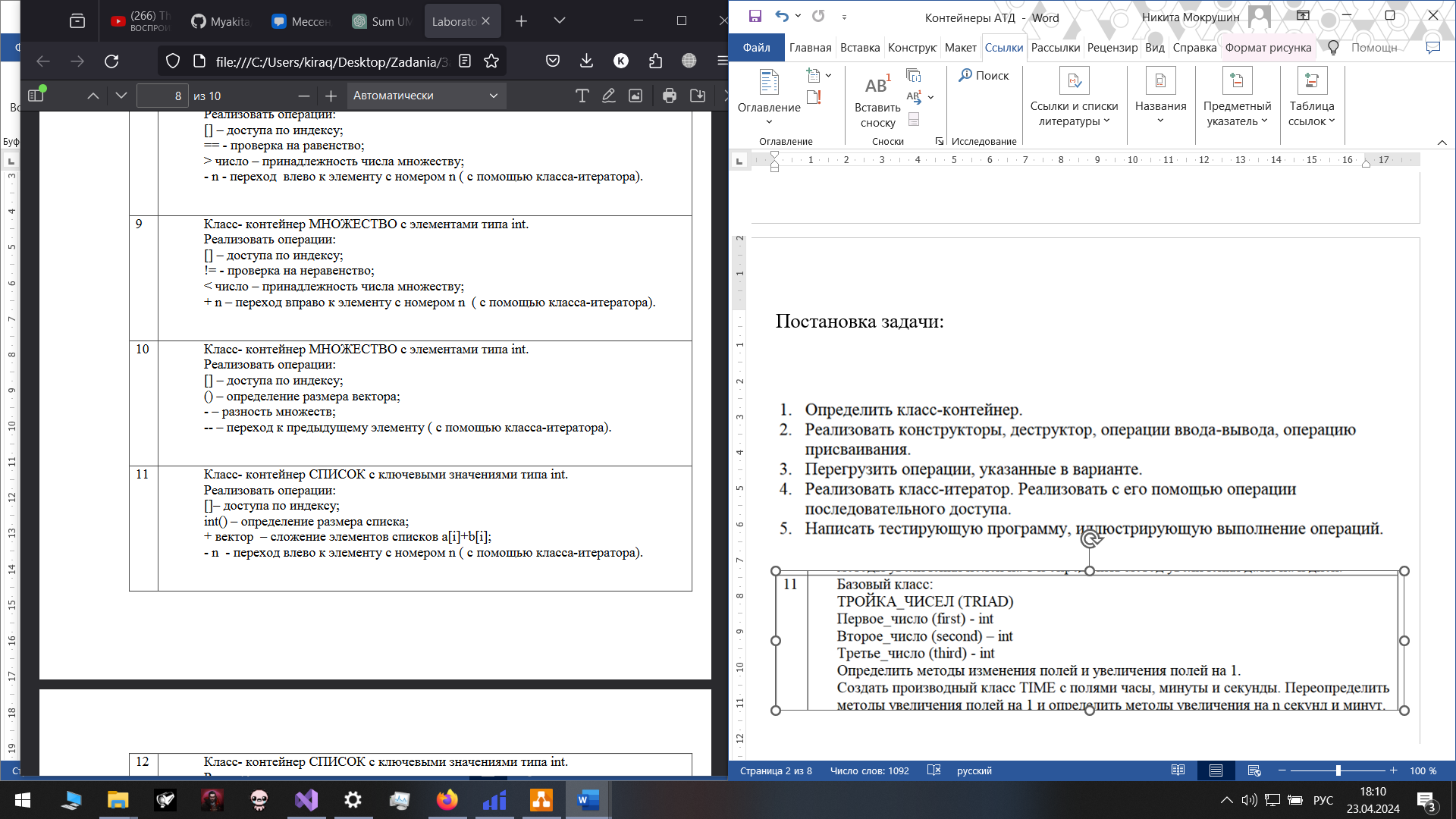
Выполнил:   
студент группы РИС-23-1б   
Мокрушин Никита Дмитриевич

Проверила:   
доцент кафедры ИТАС   
О.А. Полякова

Пермь 2024 г.

Постановка задачи:





Анализ задачи:

**Анализ задачи по пунктам:**

1. **Структура данных**: Реализация двусвязного списка.
2. **Итератор**: Класс Iterator позволяет перемещаться по элементам списка и взаимодействовать с ними, таким образом, предоставляя удобный способ обхода списка.
3. **Класс узла списка**: Node представляет узел списка с полями для хранения данных и указателями на предыдущий и следующий элементы.
4. **Класс списка**: List содержит указатели на начало и конец списка, а также размер списка. Он реализует методы для работы со списком, такие как добавление, удаление элементов, доступ к первому и последнему элементам и перегруженные операторы.
5. **Ввод-вывод**: Программа демонстрирует работу списка через ввод и вывод элементов.
6. **Использование дружественных классов**: Классы List и Iterator объявлены дружественными, что позволяет классу Iterator иметь доступ к закрытым членам класса List.

Код:

#include <iostream>

using namespace std;

struct Node

{

Node\* prev = nullptr;

Node\* next = nullptr;

int data;

};

class Iterator

{

friend class List;

public:

Iterator() { elem = nullptr; };

Iterator(const Iterator& i) { elem = i.elem; };

Iterator operator=(const Iterator& a) { elem = a.elem; return \*this; }

bool operator==(const Iterator& i) { return elem == i.elem; }

bool operator!=(const Iterator& i) { return elem != i.elem; }

Iterator& operator++() { elem = elem->next; return \*this; }

Iterator& operator--() { elem = elem->prev; return \*this; }

Iterator& operator+(const int& a)

{

for (int i = 0; i < a; ++i)

{

elem = elem->next;

}

}

Iterator& operator-(const int& a)

{

for (int i = 0; i < a; ++i)

{

elem = elem->prev;

}

}

int& operator\*() { return(elem->data); }

private:

Node\* elem;

};

class List

{

public:

List(int s, int k = 0);

List(const List& a);

~List();

int front();

int back();

void pushback(int data);

void pushfront(int data);

void popback();

void popfront();

bool empty();

List& operator=(const List& a);

int& operator[](int index);

int& operator()();

List operator+(List& a);

friend ostream& operator<<(ostream& os, const List& a);

friend istream& operator>>(istream& in, const List& a);

Iterator first() { return beg; }

Iterator last() { return end; }

private:

int size;

Node\* head = nullptr;

Node\* tail = nullptr;

Iterator beg;

Iterator end;

};

List::List(int s, int k)

{

size = s;

Node\* node = new Node;

node->data = k;

head = node;

tail = node;

for (int i = 0; i < size - 1; i++)

{

Node\* node = new Node;

node->data = k + 2 + i;

tail->next = node;

node->prev = tail;

tail = node;

}

tail->next = nullptr;

beg.elem = head;

end.elem = tail->next;

}

List::List(const List& a)

{

Node\* node = a.head;

while (node != nullptr)

{

pushback(node->data);

node = node->next;

}

beg = a.beg;

end = a.end;

}

List::~List()

{

Node\* curnode = head;

while (curnode != nullptr)

{

head = curnode->next;

delete curnode;

curnode = head;

}

}

void List::pushback(int data)

{

Node\* newnode = new Node;

newnode->data = data;

if (head == nullptr)

{

head = newnode;

tail = newnode;

++size;

end.elem = tail->next;

}

else

{

tail->next = newnode;

newnode->prev = tail;

tail = newnode;

++size;

end.elem = tail->next;

}

}

void List::pushfront(int data)

{

Node\* newnode = new Node;

newnode->data = data;

if (head == nullptr)

{

head = newnode;

tail = newnode;

++size;

beg.elem = head;

}

else

{

head->prev = newnode;

newnode->next = head;

head = newnode;

++size;

beg.elem = head;

}

}

int List::front()

{

return head->data;

}

int List::back()

{

return tail->data;

}

void List::popback()

{

if (head != nullptr)

{

Node\* curnode = tail;

tail = curnode->prev;

delete curnode;

tail->next = nullptr;

--size;

end.elem = tail->next;

}

}

void List::popfront()

{

if (head != nullptr)

{

Node\* curnode = head;

head = curnode->next;

delete curnode;

head->prev = nullptr;

--size;

beg.elem = head;

}

else

{

cout << "Список пуст";

}

}

bool List::empty()

{

return size == 0;

}

int& List::operator()()

{

return size;

}

int& List::operator[](int index)

{

if (index < size && index >= 0)

{

Node\* cur = head;

for (int i = 0; i != index; ++i)

{

cur = cur->next;

}

return cur->data;

}

else

{

if (index >= size)

{

cout << "ERR: Индекс больше или равен размеру списка" << endl;

exit(0);

}

else

{

cout << "ERR: Отрицательный индекс" << endl;

exit(0);

}

}

}

List List::operator+(List& a)

{

int sizetmp = ((size > a.size) ? a.size : size);

List temp(sizetmp);

for (int i = 0; i < sizetmp; i++)

{

temp[i] = (\*this)[i] + a[i];

}

return temp;

}

ostream& operator<<(ostream& os, const List& a)

{

os << endl << "--------Вывод элементов списка-------" << endl;

Node\* cur = a.head;

while (cur != nullptr)

{

cout << cur->data << ' ';

cur = cur->next;

}

os << endl << "-----Вывод элементов списка окончен-----" << endl;

return os;

}

istream& operator>>(istream& in, const List& a)

{

cout << endl << "--------Ввод элементов списка-------" << endl;

Node\* cur = a.head;

while (cur != nullptr)

{

in >> cur->data;

cur = cur->next;

}

cout << endl << "-----Ввод элементов списка окончен-----" << endl;

return in;

}

List& List::operator=(const List& a)

{

cout << "Оператор приравнивания" << endl;

if (this == &a)

{

return \*this;

}

Node\* node = head;

while (node != nullptr)

{

head = node->next;

delete node;

node = head;

--size;

}

Node\* curnode = a.head;

while (curnode != nullptr)

{

pushback(curnode->data);

curnode = curnode->next;

}

beg = a.beg;

end = a.end;

return \*this;

}

int main()

{

setlocale(0, "");

List a(10);

cout << endl << a << endl;

cout << "Первый элемент списка a: " << \*(a.first()) << " / Последний элемент списка a: " << a.back() << endl;

for (int i = 0; i < a(); ++i)

{

a[i] = i \* i - 24 + i;

}

cout << endl << a << endl;

cout << "Первый элемент списка a: " << \*(a.first()) << " / Последний элемент списка a: " << a.back() << endl;

List b(a);

cout << endl << b << endl;

cout << "Первый элемент списка b: " << \*(b.first()) << " / Последний элемент списка b: " << b.back() << endl;

b.pushback(1488), b.popfront();

cout << endl << b << endl;

cout << "Первый элемент списка b: " << \*(b.first()) << " / Последний элемент списка b: " << b.back() << endl;

List c(6);

cin >> c;

cout << endl << c << endl;

List d = c + b;

cout << endl << d << endl;

cout << "Первый элемент списка d: " << \*(d.first()) << " / Последний элемент списка d: " << d.back() << endl;

cout << "Вывод списка с помощью итераторов: " << endl;

Iterator it = d.first();

cout << "Первый элемент списка d: " << \*it << endl;

++it;

cout << "Вторйо элемент списка d: " << \*it << endl;

for (Iterator iter = d.first(); iter != d.last(); ++iter)

{

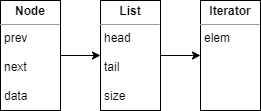
cout << \*iter << ' ';

}

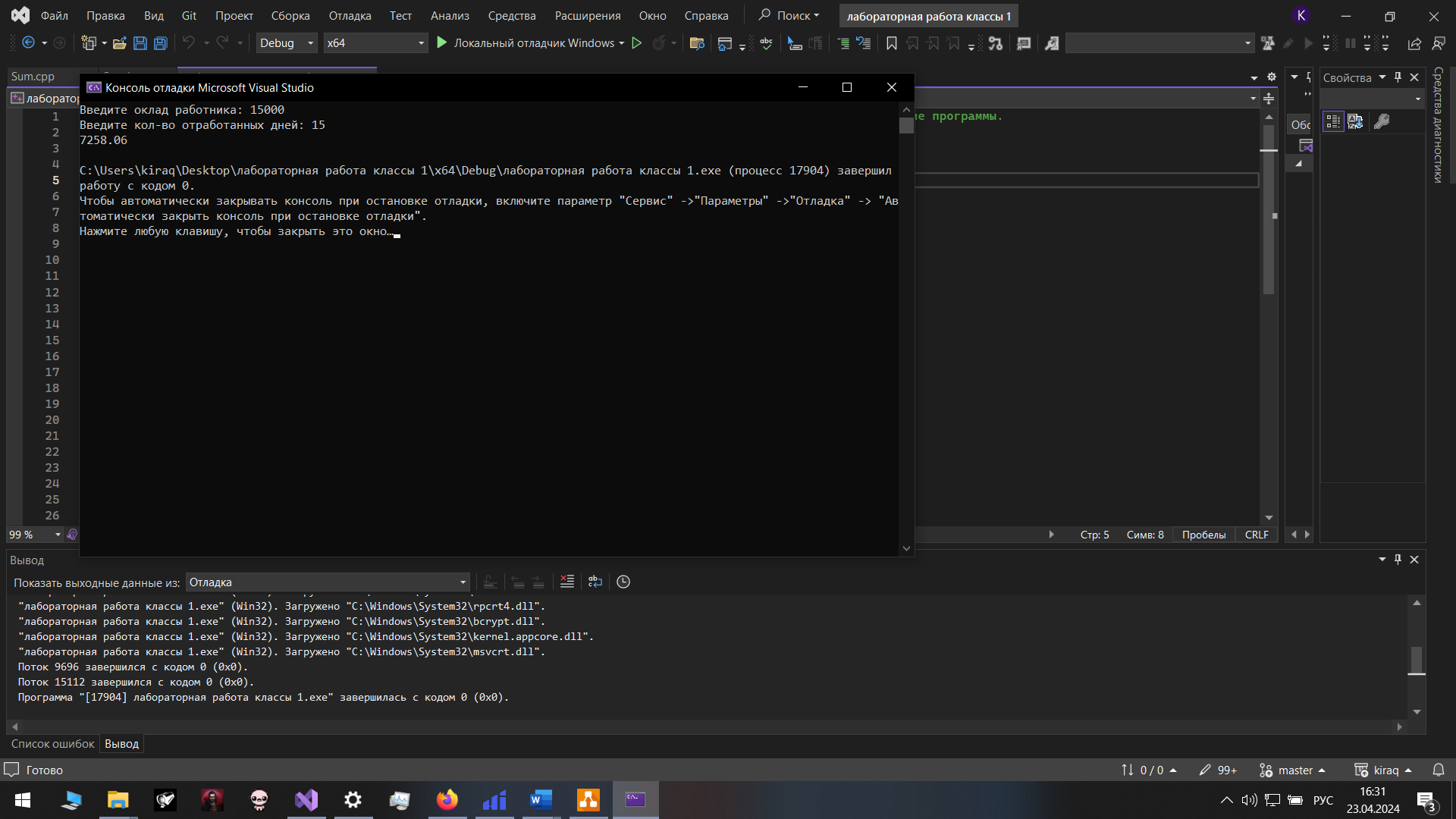
return 0;

}

UML Диаграммы:



Вывод:



программа работает корректно и выдаёт требуемый результат.  
Ответы на вопросы:

**Абстрактный тип данных (АТД)** - это математическая модель данных, определяющая тип данных и набор операций, которые можно выполнить с этими данными, скрывая при этом детали их реализации. Примеры АТД:

Стек: операции push (добавление элемента), pop (удаление элемента), top (получение верхнего элемента).

Очередь: операции enqueue (добавление элемента), dequeue (удаление элемента), front (получение первого элемента).

Словарь: операции insert (вставка пары ключ-значение), find (поиск значения по ключу), erase (удаление значения по ключу).

**Примеры абстракции через параметризацию**:

**Шаблоны в C++**: например, std::vector<T> - контейнер, который может хранить элементы любого типа T.

**Шаблоны функций**: например, функция сортировки sort<T>(), которая может сортировать массивы различных типов.

**Примеры абстракции через спецификацию**:

**Интерфейсы в Java или C#**: определяют набор методов без их реализации, обеспечивая только спецификацию того, что класс, реализующий этот интерфейс, должен делать.

**Контейнер** - это структура данных, которая хранит коллекцию элементов одного типа и предоставляет методы для их управления. Примеры контейнеров:

Вектор (std::vector) в C++.

Список (LinkedList) в Java.

Словарь (Dictionary) в Python.

**Группы операций в контейнерах**:

Операции доступа к элементам.

Операции вставки и удаления элементов.

Операции изменения размера контейнера.

Операции поиска элементов.

Операции сортировки и перемешивания элементов.

**Виды доступа к элементам контейнера**:

**Последовательный доступ**: последовательный перебор элементов контейнера с использованием итераторов или циклов.

**Произвольный доступ**: доступ к элементам по индексу или ключу. Примеры: массивы, ассоциативные массивы.

**Итератор** - это объект, который позволяет последовательно перебирать элементы контейнера, обеспечивая доступ к каждому элементу.

**Реализации итератора**:

**Указатель**: итератор может быть реализован как указатель, который указывает на текущий элемент в контейнере.

**Специальный класс**: итератор может быть реализован как специальный класс, который хранит текущее состояние и позволяет перемещаться между элементами.

**Объединение контейнеров** может быть организовано путем создания нового контейнера, содержащего элементы из двух или более исходных контейнеров.

**Доступ к элементам контейнера, состоящего из элементов "ключ-значение"**, обычно осуществляется с использованием ключа для получения соответствующего значения. Примеры таких контейнеров: словари, ассоциативные массивы.

**Контейнер, в котором вставка и удаление элементов выполняется на одном конце контейнера**, называется **стеком**.

Ответ: d. int mas[100] - является контейнером. Он представляет собой статический массив, который может содержать до 100 элементов типа int.

Ответ: a. int a[]={1,2,3,4,5} - не является контейнером. Это обычный статический массив фиксированного размера, который содержит пять элементов типа int. В то время как контейнер представляет собой абстракцию, обычно с динамическим размером или способностью изменять размер по мере необходимости.

**Доступ к элементам контейнера**, реализованного как динамический массив с операцией доступа по индексу, будет **произвольным**. Поскольку элементы в таком контейнере хранятся в виде массива, можно обращаться к элементам по их индексам, что обеспечивает произвольный доступ.

**Доступ к элементам контейнера**, реализованного как линейный список, будет **последовательным**. Поскольку в линейном списке элементы хранятся в виде связанных узлов, доступ к элементам происходит последовательно, начиная с начала списка и двигаясь по одному элементу за раз.