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

ПНИПУ

**Лабораторная работа №9**

**«Обработка исключительных ситуаций»**

**14 вариант**

Выполнил:

Студент группы РИС-23-1б

Шароглазов Егор Алексеевич

Проверила:

Доцент кафедры ИТАС

О.А. Полякова

1. г

**Постановка задачи:**

1. Реализовать класс, перегрузить для него операции, указанные в варианте.
2. Определить исключительные ситуации.
3. Предусмотреть генерацию исключительных ситуаций.

**Анализ задачи:**

**1. Классы ошибок:**

Error: Базовый класс для всех ошибок, с методом what() для вывода сообщения об ошибке.

IndexError: Наследует от Error, описывает ошибки, связанные с индексами.

Index\_min\_error и Index\_max\_error: Наследники IndexError, указывают на выход индекса за минимальное или максимальное значение.

size\_error: Описывает ошибки, связанные с размером списка.

error\_empty: Указывает на попытку операции с пустым списком.

error\_diff\_size: Ошибка при попытке операции с списками разного размера.

**2. Структура Node:**

Представляет узел двусвязного списка с полями:

data: значение типа int.

next: указатель на следующий узел.

prev: указатель на предыдущий узел.

**3. Класс List:**

Реализует двусвязный список с методами:

Конструкторы для создания пустого списка, списка определенного размера с заданным значением или копии другого списка.

push\_back(int data): Добавляет элемент в конец списка.

push\_front(int data): Добавляет элемент в начало списка.

front(): Возвращает значение первого элемента.

back(): Возвращает значение последнего элемента.

pop\_back(): Удаляет последний элемент и возвращает его значение.

pop\_front(): Удаляет первый элемент и возвращает его значение.

is\_empty(): Проверяет, пуст ли список.

operator=: Перегруженный оператор присваивания для копирования списка.

operator[]: Перегруженный оператор индексации для доступа к элементу по индексу.

operator-: Перегруженный оператор для перемещения влево на заданное количество позиций.

operator(): Возвращает размер списка.

operator\*: Перегруженный оператор умножения для поэлементного умножения списков.

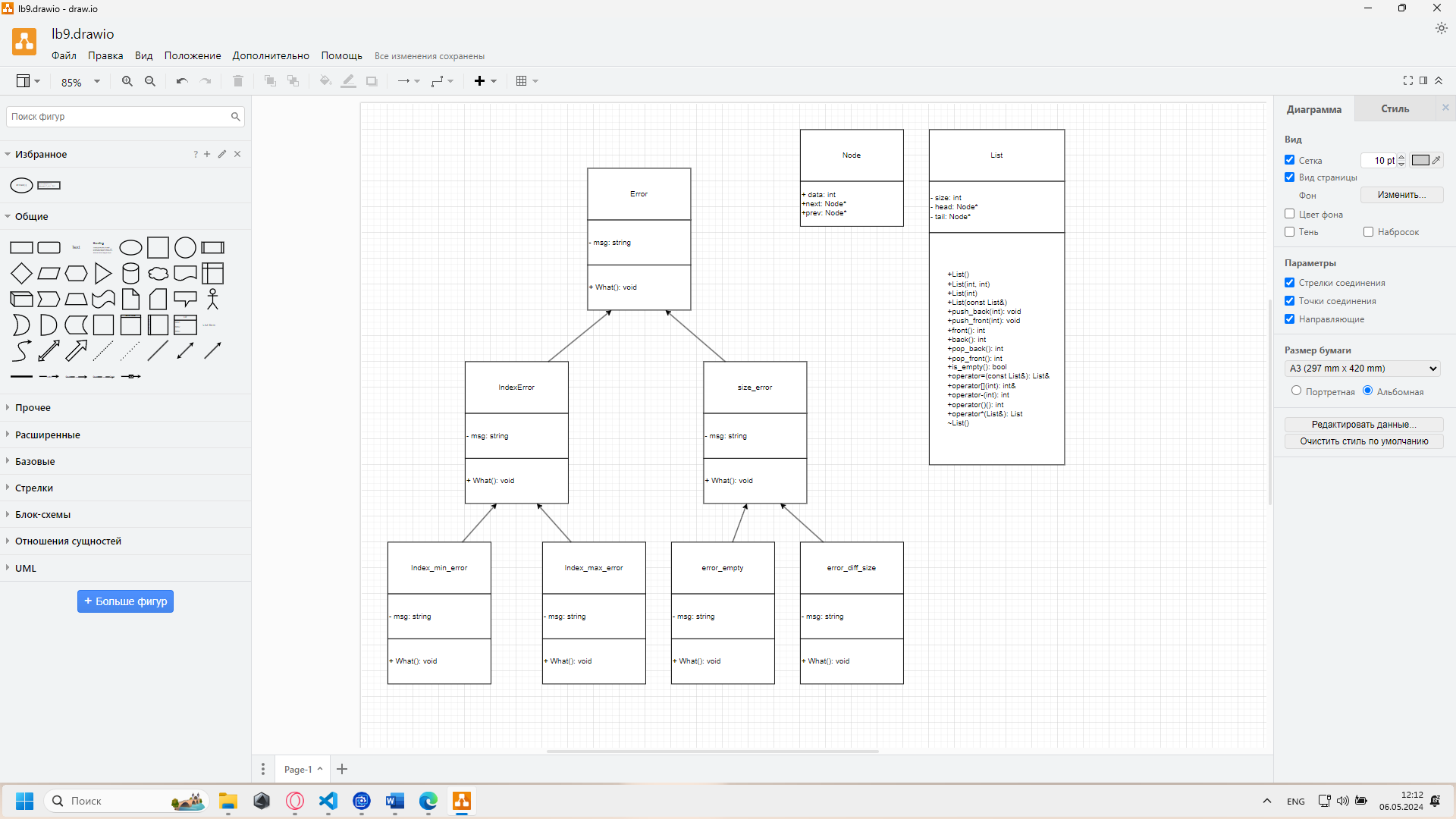
Деструктор для очистки памяти.

**4. Перегруженные операторы ввода и вывода:**

operator<<: Выводит элементы списка.

operator>>: Считывает элементы списка.

**UML – диаграмма**



**Код**

#include<string>

#include <iostream>

#include <vector>

using namespace std;

class Error{

protected:

    string msg;

public:

    virtual void what(){};

};

class IndexError:public Error{

protected:

    string msg;

public:

    IndexError(){msg="Index Error\n";}

    virtual void what(){cout<<msg;}

};

class Index\_min\_error:public IndexError{

protected:

    string msg\_;

public:

    Index\_min\_error(){IndexError();msg\_="Index<MinIndex\n";}

    virtual void what(){cout<<msg<<msg\_;}

};

class Index\_max\_error:public IndexError{

protected:

    string msg\_;

public:

    Index\_max\_error(){IndexError();msg\_="Index>MaxIndex\n";}

    virtual void what(){cout<<msg<<msg\_;}

};

class size\_error:public Error{

protected:

    string msg;

public:

    size\_error(){msg="Index Error\n";}

    virtual void what(){cout<<msg;}

};

class error\_empty :public size\_error

{

protected:

    string msg\_;

public:

    error\_empty(){size\_error();msg\_ = "empty list";}

    virtual void what(){cout<<msg<<msg\_;}

};

class error\_diff\_size :public size\_error

{

protected:

    string msg\_;

public:

    error\_diff\_size(){size\_error();msg\_= "Different size of list";}

    virtual void what(){cout<<msg<<msg\_;}

};

struct Node {

public:

    int data;

    Node\* next;

    Node\* prev;

};

class List {

private:

    int size;

    Node\* head;

    Node\* tail;

    friend ostream& operator<<(ostream& out, const List& list);

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

public:

    List() : size(0), head(nullptr),tail(nullptr) {}

    void push\_back(int data){

        Node\* new\_node = new Node;

        new\_node->data=data;

        new\_node->next=nullptr;

        if(this->head==nullptr)

        {

            this->head=new\_node;

            this->tail=new\_node;

        }

        else

        {

            tail->next = new\_node;

            new\_node->prev = tail;

            tail=new\_node;

        }

        this->size++;

    }

    void push\_front(int data){

        Node\* new\_node = new Node;

        new\_node->data=data;

        if(this->head==nullptr)

        {

            this->head=new\_node;

            this->tail=new\_node;

            this->size++;

        }

        else

        {

            head->prev = new\_node;

            new\_node->next = head;

            head=new\_node;

            this->size++;

        }

    }

    int front(){

        return this->head->data;

    }

    int back(){

        return this->tail->data;

    }

    int pop\_back(){

        int temp;

        if(this->tail!=nullptr){

            Node \* current\_node= this->tail;

            tail=current\_node->prev;

            temp=current\_node->data;

            tail->next=nullptr;

            this->size--;

        }

        return temp;

    }

    int pop\_front(){

        int temp;

        if(this->head!=nullptr){

            Node \* current\_node= this->head;

            head=current\_node->next;

            temp=current\_node->data;

            head->prev=nullptr;

            this->size--;

        }

        return temp;

    }

    bool is\_empty(){

        return this->size==0;

    }

    List& operator = (const List& list){

        cout<<"Operator ="<<endl;

        if(this == &list){

            return \*this;

        }

        while (head!=nullptr){

            Node\* temp=head;

            head=head->next;

            delete temp;

        }

        size =0;

        Node\* current\_node=list.head;

        while(current\_node!=nullptr){

            push\_back(current\_node->data);

            current\_node=current\_node->next;

        }

        return \*this;

    }

    int& operator [](int index){

        if (index < 0) throw Index\_min\_error();

        if (index >= size) throw Index\_max\_error();

        if(index <this->size&&index>=0){

            Node\* current\_node = this->head;

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

                current\_node=current\_node->next;

            }

            return current\_node->data;

        }

        else{

            cerr<<"index out of range";

            exit(0);

        }

    }

    int operator-(int h) {

    if (h < 0) throw Index\_min\_error();

    if (h >= size) throw Index\_max\_error();

    Node\* c = head;

    while (c->next != nullptr) {

        c = c->next;

    }

    for (int i = 0; i < h; ++i) {

        c = c->prev;

    }

    return c->data;

}

    int operator () (){

        if(size==0)throw error\_empty();

        return this->size;

    }

    List operator \* (List& list){

        if (this->size != list.size)

        {

            throw error\_diff\_size();

        }

        int temp\_size;

        if(this->size>list.size){

            temp\_size=list.size;

        }

        else

        {

            temp\_size=this->size;

        }

        List temp(temp\_size,0);

        for(int i=0;i<temp\_size;i++){

            temp[i]=(\*this)[i]\*list[i];

        }

        return temp;

    }

    List(int size, int data){

        this->size=size;

        if(size>0){

            Node\* node = new Node;

            node->data=data;

            this->head=node;

            this->tail=node;

            for(int i=1;i<size; i++){

                Node\* newNode = new Node;

                newNode->data=data;

                tail->next = newNode;

                newNode->prev = tail;

                tail=newNode;

            }

            tail->next=nullptr;

        }

        else

        {

            this->head=nullptr;

            this->tail=nullptr;

        }

    }

    List(const List& list)

    {

        this->head=nullptr;

        this->tail=nullptr;

        this->size=0;

        Node\* current\_node=list.head;

        while(current\_node!=nullptr){

            push\_back(current\_node->data);

            current\_node=current\_node->next;

        }

    };

    List(int size){

        this->size=size;

        if(size>0){

            Node\* node = new Node;

            this->head=node;

            this->tail=node;

            for(int i=1;i<size; i++){

                Node\* newNode = new Node;

                tail->next = newNode;

                newNode->prev = tail;

                tail=newNode;

            }

            tail->next=nullptr;

        }

        else

        {

            this->head=nullptr;

            this->tail=nullptr;

        }

    };

    ~List()

    {

        Node\* current\_node=head;

        while(current\_node!=nullptr){

            Node\* next=current\_node->next;

            delete current\_node;

            current\_node=next;

            }

        head=nullptr;

    };

};

    ostream& operator<<(ostream& out, const List& list){

        out <<endl<< "Elements of list"<<endl;

        Node\* current\_node = list.head;

        while (current\_node!=nullptr)

        {

            out<<current\_node->data<< " ";

            current\_node=current\_node->next;

        }

        out<<endl<<"conclusion end"<<endl;

        return out;

    }

    istream& operator >> (istream& in, const List& list){

        cout<<endl<<"Enter element of list"<<endl;

        Node\* current\_node=list.head;

        while (current\_node!=nullptr){

            in>>current\_node->data;

            current\_node=current\_node->next;

        }

        cout<<endl<<"Enter element stop"<<endl;

        return in;

    }

int main(){

    List sp1, sp2;

    int count, x, index,h;

    cout << "Enter element count: ";

    cin >> count;

    try

    {

        if (count < 1) throw error\_empty();

        cout << "Enter element of list: ";

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

        {

            cin >> x;

            sp1.push\_back(x);

        }

        cout << "Enter index of list number 1 for calling: ";

        cin >> index;

        cout << "Enter element with this index: " << sp1[index] << endl;

        cout << "Enter index one of the left element of list sp1: ";

        cin >> h;

        cout << "element with this index: " << sp1-h << endl;

        cout << "Size of lsit : " << sp1() << endl;

        cout << "Enter count of element of list number 2: ";

        cin >> count;

        if (count < 1) throw error\_empty();

        cout << "Enter element of list: ";

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

        {

            cin >> x;

            sp2.push\_back(x);

        }

        sp1\* sp2;

        cout << "List obtained by multiplying two lists: ";

        cout<<sp1<<endl;

    }

    catch (Error& a)

    {

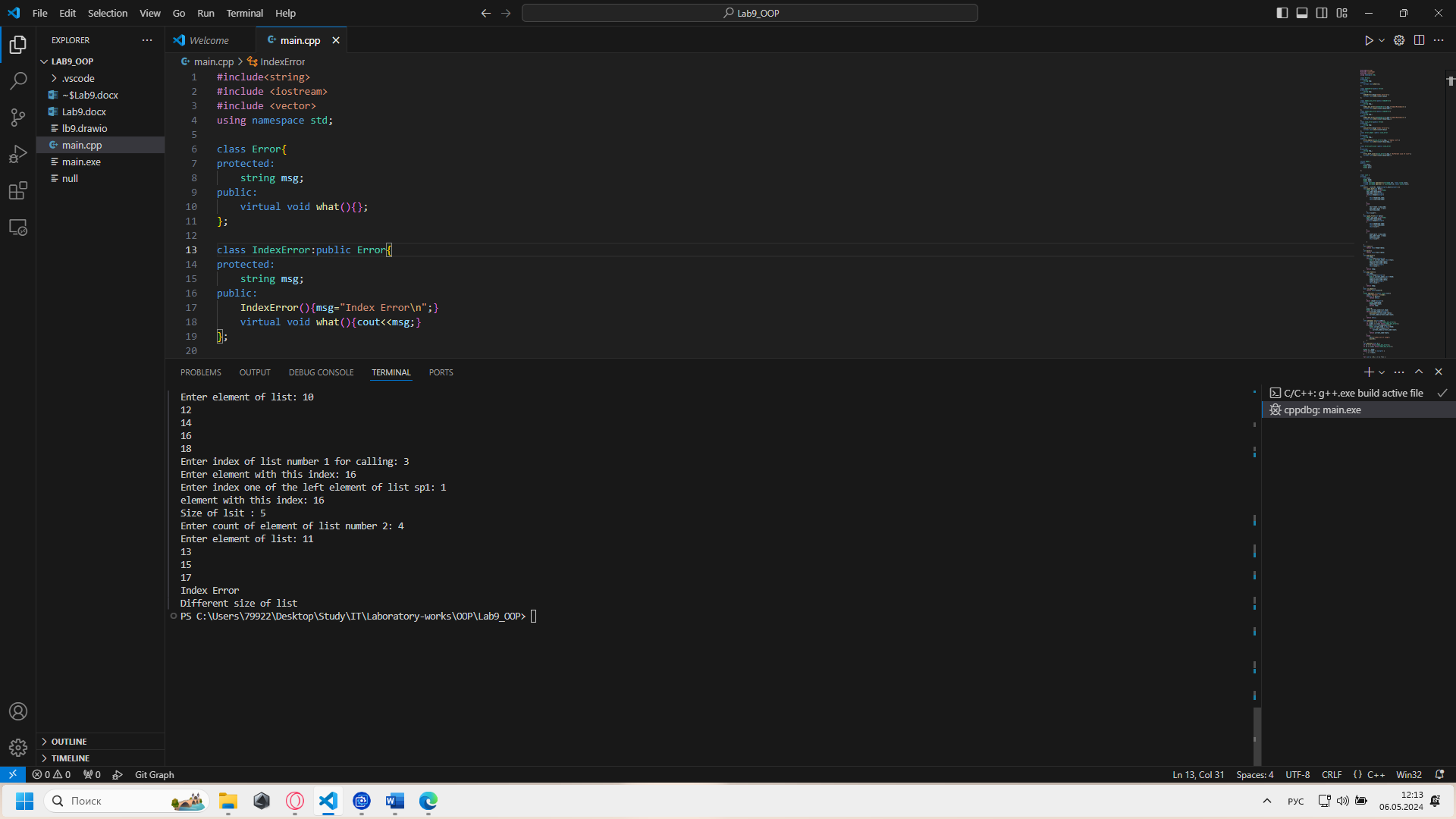
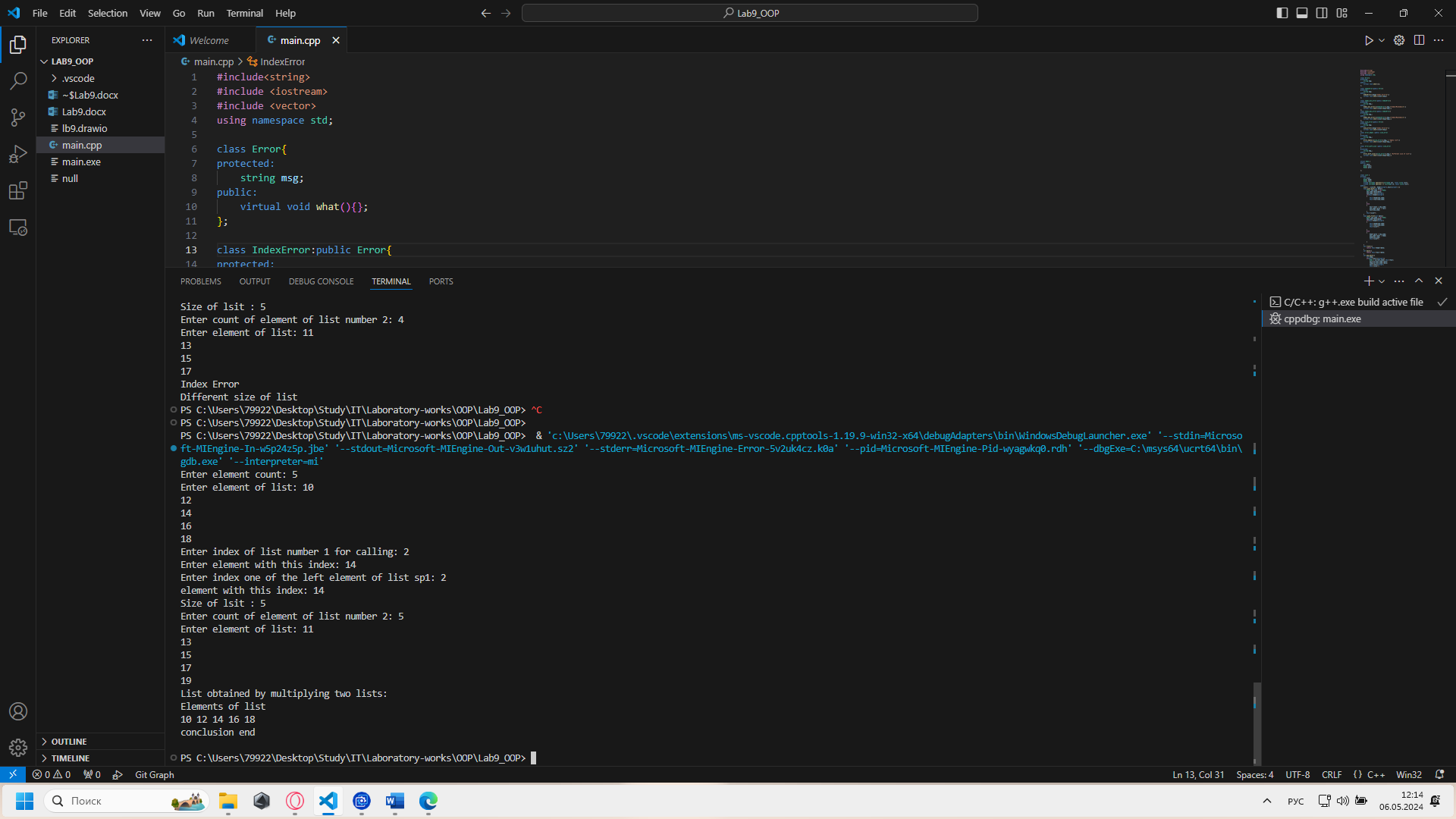
        a.what();

    }

    return 0;

}

**Вывод**

**Ответы на вопросы**

· **Исключение в C++** представляет собой объект, который передается во время выполнения программы, чтобы указать на ошибку или необычное состояние, которое может повлиять на нормальное продолжение работы программы.

· **Части исключения**: Разделяются на генерацию и обработку. Генерация исключения происходит в том месте кода, где возникает проблема или ошибка. Обработка исключения происходит в блоке кода, который реагирует на возникшее исключение. Преимущество такого подхода заключается в том, что он позволяет отделить код, который может вызвать ошибку, от кода, который обрабатывает эту ошибку, повышая читаемость и модульность программы.

· Для генерации исключительной ситуации используется оператор throw.

· **Контролируемый блок** - это участок кода, в котором может произойти исключение. Он нужен для того, чтобы определить область, в которой должна быть обработка исключения.

· **Секция-ловушка** - это блок кода, который обрабатывает исключение. Она нужна для того, чтобы предотвратить прекращение выполнения программы из-за исключения и выполнить необходимые действия по его обработке.

· Формы спецификации исключения в секции ловушке могут быть следующими:

* catch (...) - обрабатывает любое исключение, не указанное явно;
* catch (int ex) - обрабатывает исключение типа int;
* catch (double ex) - обрабатывает исключение типа double. Эти формы используются в ситуациях, когда необходимо обработать конкретные типы исключений или все исключения вообще.

· Стандартный класс std::exception можно использовать для создания собственной иерархии исключений.

· Для создания собственной иерархии исключений необходимо унаследоваться от класса std::exception или его производного класса и переопределить метод what().

· Функция f1() может принимать исключения типа int и double.

· Функция f1() не может принимать никаких исключений.

· Исключение может генерироваться в любой части программы, где возникает ошибка или необычное состояние, которое не может быть обработано локально.

· cpp

#include <iostream>

#include <cmath>

#include <stdexcept>

// Без спецификации исключений

double areaNoException(double a, double b, double c) {

double p = (a + b + c) / 2;

return sqrt(p \* (p - a) \* (p - b) \* (p - c));

}

// Со спецификацией throw()

double areaWithThrow(double a, double b, double c) throw() {

double p = (a + b + c) / 2;

if (p \* (p - a) \* (p - b) \* (p - c) < 0)

throw std::domain\_error("Недопустимые значения сторон");

return sqrt(p \* (p - a) \* (p - b) \* (p - c));

}

// С конкретной спецификацией с подходящим стандартным исключением

double areaWithStandardException(double a, double b, double c) throw(std::domain\_error) {

double p = (a + b + c) / 2;

if (p \* (p - a) \* (p - b) \* (p - c) < 0)

throw std::domain\_error("Недопустимые значения сторон");

return sqrt(p \* (p - a) \* (p - b) \* (p - c));

}

// Спецификация с собственным реализованным исключением

class InvalidTriangleException : public std::exception {

public:

const char\* what() const noexcept override {

return "Невозможно вычислить площадь: недопустимые значения сторон";

}

};

double areaWithCustomException(double a, double b, double c) throw(InvalidTriangleException) {

double p = (a + b + c) / 2;

if (p \* (p - a) \* (p - b) \* (p - c) < 0)

throw InvalidTriangleException();

return sqrt(p \* (p - a) \* (p - b) \* (p - c));

}

int main() {

double a, b, c;

std::cout << "Введите стороны треугольника: ";

std::cin >> a >> b >> c;

try {

double area = areaNoException(a, b, c);

std::cout << "Площадь треугольника: " << area << std::endl;

} catch (const std::exception& e) {

std::cout << "Ошибка: " << e.what() << std::endl;

}

return 0;

}