Министерство высшего образования и науки Российской Федерации

Федеральное государственное бюджетное образовательное учреждение высшего образования

**«Пермский национальный исследовательский политехнический университет» (ПНИПУ)**

Электротехнический факультет

Кафедра «Информационные технологии и автоматизированные системы»

ОТЧЁТ

по лабораторной работе №9

Тема: «Обработка исключительных ситуаций.»

Выполнил

Студент группы РИС-22-2б

Малых В.О.

Проверил доц. Кафедры ИТАС

Полякова О.А.

Пермь 2023

# Постановка задачи

# Реализовать класс, перегрузить для него операции, указанные в варианте.

# Определить исключительные ситуации.

# Предусмотреть генерацию исключительных ситуаций.

ВАРИАНТ 15:

Класс- контейнер СПИСОК с ключевыми значениями типа int.

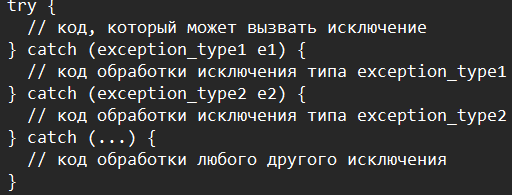
Реализовать операции: [] – доступа по индексу; int() – определение размера списка; \* вектор – умножение элементов списков a[i]\*b[i]; +n - переход вправо к элементу с номером n.

**Контрольные вопросы**

*1. Что представляет собой исключение в С++?*

Исключение в C++ представляет собой механизм обработки ошибок, который позволяет программе сообщать об ошибках, возникших во время выполнения программы, и перехватывать эти ошибки для их корректной обработки.

Когда происходит ошибка, генерируется исключение, которое может быть отправлено на верхний уровень вызова функции или метода, где может быть обработано с помощью конструкции try-catch. Конструкция try-catch позволяет программе перехватывать исключения, выбрасываемые программой во время выполнения.



*2. На какие части исключения позволяют разделить вычислительный процесс? Достоинства такого подхода?*

Исключения в C++ позволяют разделить вычислительный процесс на две части: код, который может вызывать исключения, и код, который обрабатывает исключения.

При возникновении исключительной ситуации происходит переход управления к ближайшей точке обработки исключения, где выполняется серия инструкций, специфичных для данной ситуации, и затем управление возвращается обратно в выполнение программы.

Достоинства такого подхода включают:

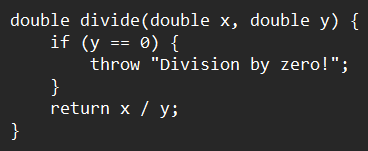
1. Разделение алгоритмов. Благодаря исключениям, код, который производит вычисления, можно разделить на отдельные блоки, которые могут выполняться в разных потоках, функциях и классах. Исключения позволяют обрабатывать ошибки в каждом из блоков и перехватывать их в одном месте, что позволяет сделать код более читабельным, поддерживаемым и надежным.

2. Обработка ошибок. Использование исключений позволяет программистам написать более чистый и надежный код, поскольку традиционная практика обработки ошибок с помощью возвращаемых значений может быстро привести к появлению сложного и неразборчивого кода. Использование исключений вынуждает программиста более тщательно следить за ошибками и обработкой их соответствующим образом.

3. Упрощение отладки. Отладка приложения, которое использует исключения, может быть упрощена, поскольку код, который ищет ошибки, может быть разделен на две части: код, который ищет ошибки в выполнении вычислений, например, некорректные аргументы, и код, который обрабатывает ошибки, если они возникнут.

*3. Какой оператор используется для генерации исключительной ситуации?*

Для генерации исключительной ситуации в C++ используется оператор throw. Он позволяет бросить значение любого типа данных, которое может быть обработано в блоке try-catch.

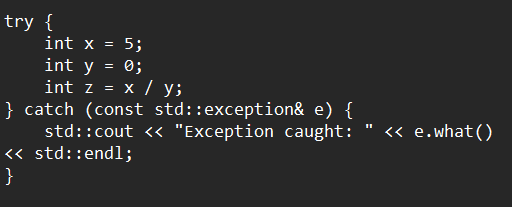


*4. Что представляет собой контролируемый блок? Для чего он нужен?*

Контролируемый блок в C++ (или блок try-catch) представляет собой участок кода, в котором можно обрабатывать исключительные ситуации.

Он нужен для того, чтобы избежать аварийного завершения программы при возникновении непредвиденных ситуаций, таких как ошибки ввода-вывода, деление на ноль, и т.д. Вместо того, чтобы прекращать работу программы или выводить ошибки на экран, мы можем описать блок кода, в котором обрабатываются исключения, связанные с этой ситуацией. Это позволяет программе корректно завершить работу и предоставить пользователю информацию об ошибке.

Блок try-catch состоит из двух частей: блока try и блока catch. Блок try содержит код, который может приводить к возникновению исключений. Блок catch содержит код, который обрабатывает исключения, генерируемые блоком try. В блоке catch указывается тип исключения, который нужно обработать.



*5. Что представляет собой секция-ловушка? Для чего она нужна?*

*6. Какие формы может иметь спецификация исключения в секции ловушке? В каких ситуациях используются эти формы?*

Блок кода, заключенный в секцию try, может содержать инструкции, которые могут привести к исключениям во время выполнения программы. В таком случае, управление передается в ближайший блок catch, который может обрабатывать это исключение и принимать необходимые меры по восстановлению или завершению программы.

Использование секции-ловушки позволяет написать более надежный и устойчивый код, который может корректно обрабатывать ошибки и исключения, которые могут возникнуть во время выполнения программы.

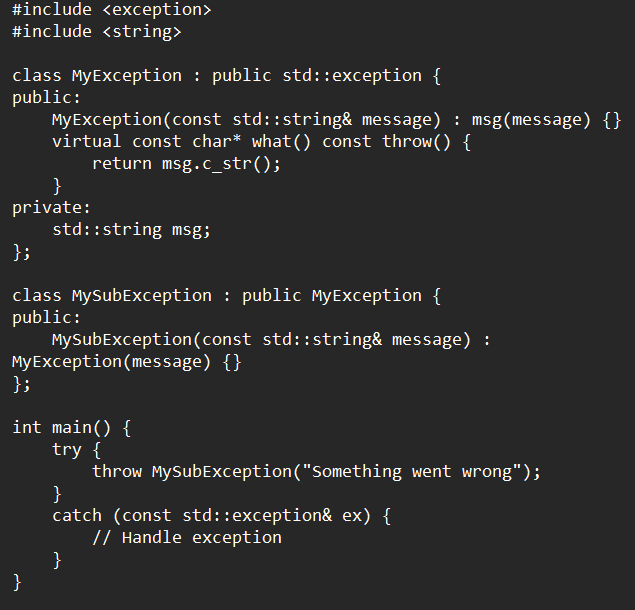
*7. Какой стандартный класс можно использовать для создания собственной иерархии исключений?*

Для создания собственной иерархии исключений в языке C++ можно использовать стандартный класс `std::exception` как базовый класс и наследовать от него свои собственные классы исключений.

Например, для создания исключения, которое будет бросаться при попытке деления на ноль, можно определить свой класс, наследующийся от `std::exception`, следующим образом:

*8. Каким образом можно создать собственную иерархию исключений?*

Для создания собственной иерархии исключений в языке C++ необходимо создать класс исключения, производный от класса std::exception. Затем можно создавать подклассы исключений, добавляя дополнительную информацию или переопределяя методы базового класса.



*9. Если спецификация исключений имеет вид: void f1()throw(int,double); то какие исключения может прождать функция f1()?*

Функция f1() может ожидать исключения типа int и double. Эта спецификация ограничивает типы исключений, которые могут быть брошены из функции f1(). Если функция бросает исключение, которое не соответствует этой спецификации, то будет вызвана функция std::unexpected(), которая обычно вызывается по умолчанию, если исключение не было обработано в соответствующем catch блоке.

*10. Если спецификация исключений имеет вид: void f1()throw(); то какие исключения может прождать функция f1()?*

Функция f1() не будет бросать никаких исключений, потому что ее спецификация исключений имеет вид void f1() throw();. Это означает, что функция не может выбрасывать никаких исключений, включая исключения типа std::exception или других пользовательских исключений. Если в процессе выполнения функции произойдет исключительная ситуация, то она будет обработана в стандартном для языка C++ режиме "stack unwinding".

*11. В какой части программы может генерироваться исключение?*

Исключения могут генерироваться в любой части программы, где происходит ошибка или неожиданное поведение. Исключения помогают сообщить о возникшей проблеме и предупредить об этом вызывающий код, что позволяет корректно обрабатывать ошибки и не допустить нежелательного поведения программы. В C++ исключения генерируются с помощью оператора throw.

*12. Написать функцию, которая вычисляет площадь треугольника по трем сторонам (формула Герона). Функцию реализовать в 4 вариантах: без спецификации исключений; со спецификацией throw(); с конкретной спецификацией с подходящим стандартным исключением; спецификация с собственным реализованным исключением.*

#include <iostream>

#include <cmath>

#include <stdexcept>

using namespace std;

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

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

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

}

// без спецификации исключений

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

if (a <= 0 || b <= 0 || c <= 0) {

cout << "Invalid input. Side value should be positive." << endl;

return -1.0;

}

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

if (area < 0) {

cout << "Invalid input. The sides cannot form a triangle." << endl;

return -1.0;

}

return area;

}

// со спецификацией throw()

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

if (a <= 0 || b <= 0 || c <= 0) {

throw invalid\_argument("Invalid input. Side value should be positive.");

}

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

if (area < 0) {

throw runtime\_error("Invalid input. The sides cannot form a triangle.");

}

return area;

}

// с конкретной спецификацией исключений

double triangleAreaSpec(double a, double b, double c) throw(invalid\_argument, runtime\_error) {

if (a <= 0 || b <= 0 || c <= 0) {

throw invalid\_argument("Invalid input. Side value should be positive.");

}

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

if (area < 0) {

throw runtime\_error("Invalid input. The sides cannot form a triangle.");

}

return area;

}

// спецификация с собственным реализованным исключением

class TriangleException : public exception {

public:

virtual const char\* what() const throw() {

return "Invalid input. The sides cannot form a triangle.";

}

};

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

if (a <= 0 || b <= 0 || c <= 0) {

throw invalid\_argument("Invalid input. Side value should be positive.");

}

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

if (area < 0) {

throw TriangleException();

}

return area;

}

int main() {

double a = 3.0, b = 4.0, c = 5.0;

try {

cout << "Triangle area: " << triangleArea(a, b, c) << endl;

cout << "Triangle area with throw(): " << triangleAreaThrow(a, b, c) << endl;

cout << "Triangle area with specific exception: " << triangleAreaSpec(a, b, c) << endl;

cout << "Triangle area with custom exception: " << triangleAreaException(a, b, c) << endl;

} catch (const exception& ex) {

cout << "Exception caught: " << ex.what() << endl;

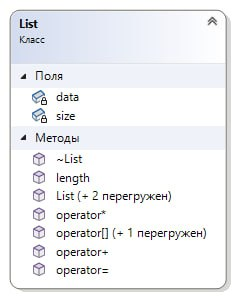
}

return 0;

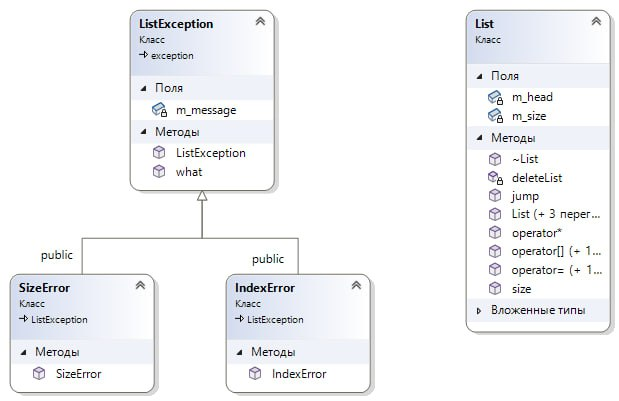
}

UML – диаграмма

9.1.



9.3.



Код программы

9.1.

**list.h**

#pragma once

#ifndef LIST\_H

#define LIST\_H

#include <iostream>

#include <stdexcept>

class List {

private:

int\* data;

int size;

public:

List();

List(int size);

List(const List& other);

~List();

List& operator=(const List& other);

int& operator[](int index);

const int& operator[](int index) const;

int operator\*(const List& other) const;

List operator+(int n) const;

int length() const;

friend std::ostream& operator<<(std::ostream& os, const List& list);

};

#endif

**list.cpp**

#include "list.h"

List::List() : data(nullptr), size(0) {}

List::List(int size) : size(size) {

data = new int[size];

for (int i = 0; i < size; ++i) {

data[i] = 0;

}

}

List::List(const List &other) : size(other.size) {

data = new int[size];

for (int i = 0; i < size; ++i) {

data[i] = other.data[i];

}

}

List::~List() {

delete[] data;

}

List &List::operator=(const List &other) {

if (this != &other) {

delete[] data;

size = other.size;

data = new int[size];

for (int i = 0; i < size; ++i) {

data[i] = other.data[i];

}

}

return \*this;

}

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

if (index < 0 || index >= size) {

throw std::out\_of\_range("Index out of range");

}

return data[index];

}

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

if (index < 0 || index >= size) {

throw std::out\_of\_range("Index out of range");

}

return data[index];

}

int List::operator\*(const List &other) const {

if (size != other.size) {

throw std::invalid\_argument("Lists have different sizes");

}

int result = 0;

for (int i = 0; i < size; ++i) {

result += data[i] \* other.data[i];

}

return result;

}

List List::operator+(int n) const {

if (n < 0 || n >= size) {

throw std::out\_of\_range("Index out of range");

}

List result(size - n);

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

result.data[i] = data[i + n];

}

return result;

}

int List::length() const {

return size;

}

std::ostream &operator<<(std::ostream &os, const List &list) {

os << "[";

for (int i = 0; i < list.size; ++i) {

os << list.data[i];

if (i != list.size - 1) {

os << ", ";

}

}

os << "]";

return os;

}

**main.cpp**

#include <iostream>

#include "list.h"

int main() {

List a(5);

a[0] = 1;

a[1] = 2;

a[2] = 3;

a[3] = 4;

a[4] = 5;

std::cout << "a = " << a << std::endl;

try {

std::cout << "a[2] = " << a[2] << std::endl;

std::cout << "a[10] = " << a[10] << std::endl;

}

catch (const std::out\_of\_range& e) {

std::cerr << "Error: " << e.what() << std::endl;

}

try {

List b(3);

b[0] = 2;

b[1] = 3;

b[2] = 4;

int product = a \* b;

std::cout << "a \* b = " << product << std::endl;

List c = a + 2;

std::cout << "c = " << c << std::endl;

List d = a + 10;

std::cout << "d = " << d << std::endl;

}

catch (const std::exception& e) {

std::cerr << "Error: " << e.what() << std::endl;

}

return 0;

}

9.3.

**list.h**

#pragma once

#ifndef LIST\_H

#define LIST\_H

#include <iostream>

#include <vector>

#include "error.h"

class List

{

public:

List() = default;

explicit List(const std::vector<int>& data);

List(const List& other);

List(List&& other) noexcept;

~List();

List& operator=(const List& other);

List& operator=(List&& other) noexcept;

int& operator[](int index);

const int& operator[](int index) const;

int size() const;

std::vector<int> operator\*(const std::vector<int>& other) const;

void jump(int n);

private:

struct Node

{

int data;

Node\* next = nullptr;

};

Node\* m\_head = nullptr;

int m\_size = 0;

void deleteList();

};

#endif // LIST\_H

**list.cpp**

#include "list.h"

List::List(const std::vector<int>& data)

{

for (auto it = data.rbegin(); it != data.rend(); ++it)

{

auto node = new Node;

node->data = \*it;

node->next = m\_head;

m\_head = node;

++m\_size;

}

}

List::List(const List& other)

{

for (auto it = other.m\_head; it != nullptr; it = it->next)

{

auto node = new Node;

node->data = it->data;

node->next = m\_head;

m\_head = node;

++m\_size;

}

}

List::List(List&& other) noexcept : m\_head(other.m\_head), m\_size(other.m\_size)

{

other.m\_head = nullptr;

other.m\_size = 0;

}

List::~List()

{

deleteList();

}

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

{

if (this != &other)

{

deleteList();

for (auto it = other.m\_head; it != nullptr; it = it->next)

{

auto node = new Node;

node->data = it->data;

node->next = m\_head;

m\_head = node;

++m\_size;

}

}

return \*this;

}

List& List::operator=(List&& other) noexcept

{

if (this != &other)

{

deleteList();

m\_head = other.m\_head;

m\_size = other.m\_size;

other.m\_head = nullptr;

other.m\_size = 0;

}

return \*this;

}

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

{

if (index < 0 || index >= m\_size)

{

throw IndexError();

}

auto node = m\_head;

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

{

node = node->next;

}

return node->data;

}

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

{

if (index < 0 || index >= m\_size)

{

throw IndexError();

}

auto node = m\_head;

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

{

node = node->next;

}

return node->data;

}

int List::size() const

{

return m\_size;

}

std::vector<int> List::operator\*(const std::vector<int>& other) const

{

if (m\_size != other.size())

{

throw SizeError();

}

std::vector<int> result;

result.reserve(m\_size);

auto node = m\_head;

for (const auto& value : other)

{

result.push\_back(node->data \* value);

node = node->next;

}

return result;

}

void List::jump(int n)

{

if (n < 0 || n >= m\_size)

{

throw IndexError();

}

auto node = m\_head;

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

{

node = node->next;

}

m\_head = node;

m\_size -= n;

}

void List::deleteList()

{

while (m\_head != nullptr)

{

auto node = m\_head;

m\_head = m\_head->next;

delete node;

}

m\_size = 0;

}

**error.h**

#pragma once

#ifndef ERROR\_H

#define ERROR\_H

#include <exception>

#include <string>

class ListException : public std::exception

{

public:

ListException(const std::string& message) : m\_message(message) {}

virtual const char\* what() const noexcept override { return m\_message.c\_str(); }

private:

std::string m\_message;

};

class IndexError : public ListException

{

public:

IndexError() : ListException("Index out of range") {}

};

class SizeError : public ListException

{

public:

SizeError() : ListException("Lists have different sizes") {}

};

#endif // ERROR\_H

**main.cpp**

#include <iostream>

#include "list.h"

int main()

{

try

{

std::vector<int> data = { 1, 2, 3, 4, 5 };

List list(data);

std::cout << "list[2]: " << list[2] << std::endl;

std::cout << "list size: " << list.size() << std::endl;

std::vector<int> other = { 2, 3, 4, 5, 6 };

auto result = list \* other;

std::cout << "list \* other: ";

for (const auto& value : result)

{

std::cout << value << " ";

}

std::cout << std::endl;

list.jump(3);

std::cout << "list[0]: " << list[0] << std::endl;

std::cout << "list size: " << list.size() << std::endl;

// Testing exceptions

// IndexError

try

{

std::cout << "list[10]: " << list[10] << std::endl;

}

catch (const IndexError& e)

{

std::cerr << "IndexError: " << e.what() << std::endl;

}

// SizeError

try

{

std::vector<int> invalid = { 1, 2, 3 };

auto invalidResult = list \* invalid;

}

catch (const SizeError& e)

{

std::cerr << "SizeError: " << e.what() << std::endl;

}

// Testing copy constructor

List copy = list;

std::cout << "copy[0]: " << copy[0] << std::endl;

std::cout << "copy size: " << copy.size() << std::endl;

// Testing assignment operator

List assigned;

assigned = list;

std::cout << "assigned[0]: " << assigned[0] << std::endl;

std::cout << "assigned size: " << assigned.size() << std::endl;

// Testing self-assignment

assigned = assigned;

std::cout << "assigned[0]: " << assigned[0] << std::endl;

std::cout << "assigned size: " << assigned.size() << std::endl;

}

catch (const std::exception& e)

{

std::cerr << "Exception: " << e.what() << std::endl;

return 1;

}

return 0;

}