Министерство образования и науки Российской Федерации

Федеральное государственное бюджетное образовательное учреждение высшего образования

«**Пермский национальный исследовательский политехнический университет»**

Кафедра «Информационные технологии и автоматизированные системы»

**ОТЧЕТ**

Дисциплина: «Информатика»

Тема: Лабораторная работа №10

"Сохранение данных в файле с использованием потоков"

Семестр 2

Выполнил работу

Студент групп РИС-22-2Б

Барыбин К.Ю.

Проверил

Доцент кафедры ИТАС

Полякова О.А.

Г. Пермь-2023

**Постановка задачи**

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

Вариант 15

Класс-контейнер список с ключевыми значениями типа int.

Реализовать операции:

[] – доступ по индексу;

Int() – определение размера списка;  
\* список – умножение элементов списков a[i] \* b[i];

+n – переход вправо к элементу с номером n (с помощью класса-итератора).

За основы мы возьмем лабораторную работу 6, в которой уже был реализован элемент list.

Для списка: оператор [], исключительная ситуация – выход за границы, метод добавления/удаления элемента по итератору, исключительная ситуация – итератор никуда не указывает, или указывает на неприемлемый элемент, оператор \* - исключительная ситуация, при разных размерах списка.

Были внесены исключения для итератора – оператор +, ++() и ++(int) – если итератор пытается выйти за последний элемент списка, то выкидывается исключение.

Также необходимо реализовать 2 типа генерации исключений. Для второго типа создадим класс Error. Добавим в него поля код ошибки и строку с описанием этих ошибок.

**Работа программы**

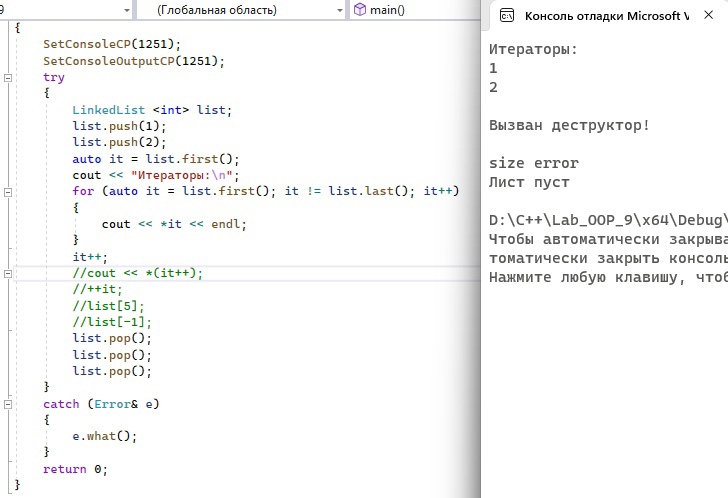


Рис. 1 – исключение size error

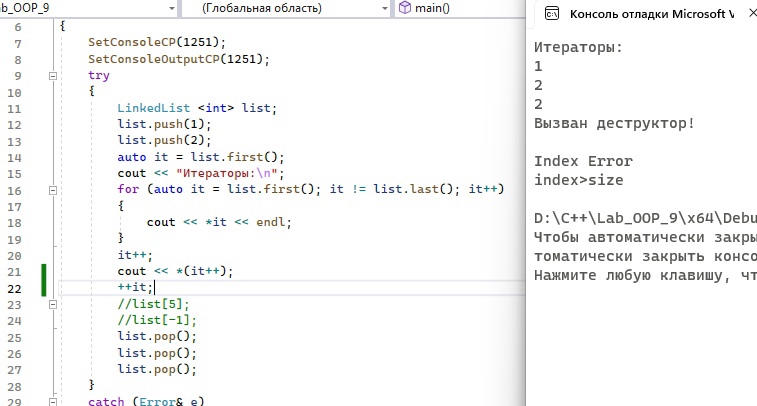


Рис. 2 – исключение index error (index > size)

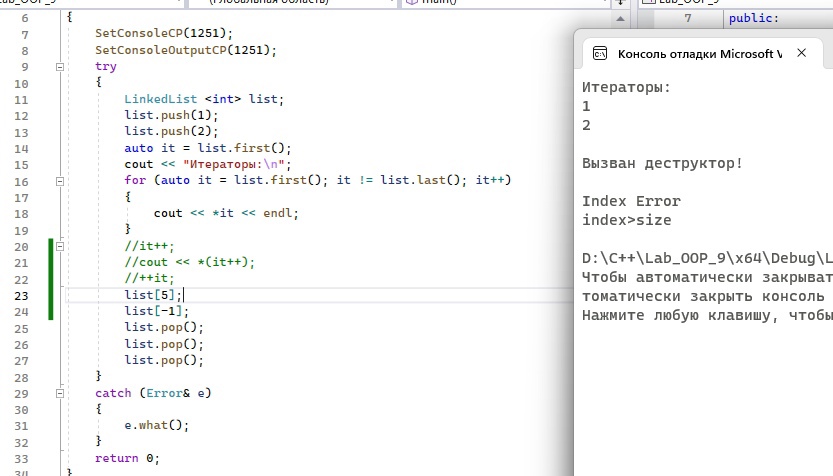


Рис. 3 – исключение index error (index > size)

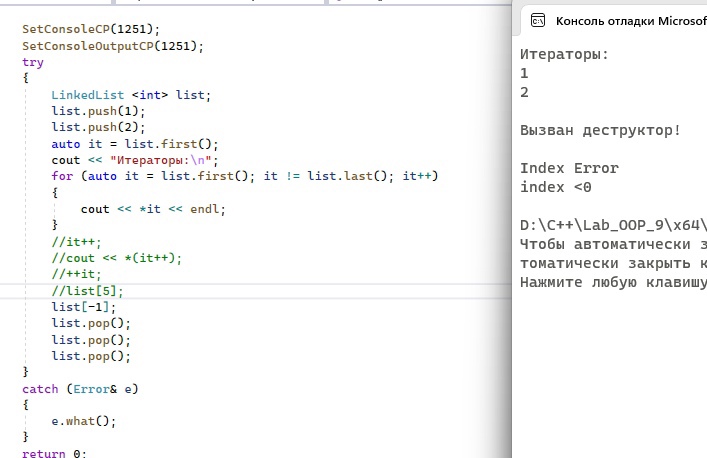


Рис. 4 – исключение index error (index < 0)

**Заключение**

Была разработана программа, которая расширяет 6 лабораторную работу. Дополнениями представляется генерация исключительных ситуаций для методов класса и методов итератора. Данные исключительные ситуации в основном предостерегают выход за границы списка, а следовательно, упреждает утечку данных. В добавок был разработан свой класс Error. Для этого добавлены поля код ошибки и строка с описанием ошибок, а также методы возвращающие код ошибки и метод возвращающий документацию.

**Контрольные вопросы**

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

В С++ исключение – это объект, который система должна генерировать при возникновении исключительной ситуации. Генерация такого объекта и создает исключительную ситуацию.

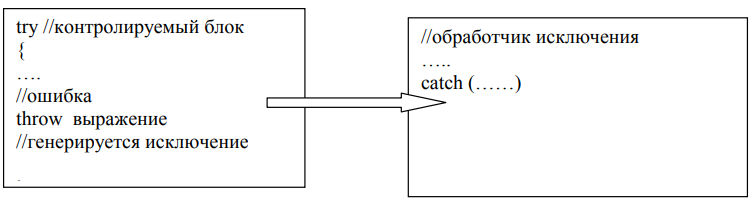
1. На какие части исключения позволяют разделить вычислительный процесс? Достоинства такого подхода?

Исключения позволяют разделить вычислительный процесс на 2 части:

* Обнаружение аварийной ситуации (неизвестно как обрабатывать);
* Обработка аварийной ситуации (неизвестно, где она возникла).

Достоинства такого подхода:

* Удобно использовать в программе, которая состоит из нескольких модулей;
* Не требуется возвращать значение в вызывающую функцию.

Общая схема:  


1. Какой оператор используется для генерации исключительной ситуации?

Исключение генерируется оператором throw <выражение>, где <выражение> -

* либо константа,
* либо переменная некоторого типа,
* либо выражение некоторого типа.

Тип объекта-исключения может быть как встроенным, так и определяемым пользователем. Для представления исключений часто используют пустой класс:

class ZeroDevide{};

Генерация исключения будет выглядеть:

throw ZeroDevide();//вызывается конструктор без параметров

или

throw new ZeroDevide();

1. Что представляет собой контролируемый блок? Для чего он нужен?

Проверка возникновения исключения делается с помощью оператора try, с которым неразрывно связаны одна или несколько блоков обработки исключений — catch. Оператор try объявляет в любом месте программы контролируемый блок, который имеет следующий вид:

try { */\* контролируемый блок \*/* }

Контролируемый блок, помимо функции контроля, обладает функциями обычного блока: все переменные, объявленные внутри него, являются локальными в этом блоке и не видны вне его.

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

После блока try обязательно прописывается один или несколько блоков catch. Блок кода после catch предложения является обработчиком исключений. Это обработчик, который перехватывает исключение, которое возникает, если типы в throw выражениях и catch совместимы. Форма записи секции-ловушки следующая:

catch (спецификация\_параметра\_исключения) { */\* блок обработки \*/*}

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

Спецификация исключения может иметь три формы:

1) (тип имя)

2) (тип)

3) (…)

Тип – это встроенный тип или тип, определенный программистом.

Формы 1 и 2 обрабатывают конкретные исключения, а форма 3 перехватывает все исключения, такую ловушку надо помещать последней, тогда она будет обрабатывать все исключения, которые еще не были обработаны.

Форма 1 означает, что объект передается в блок обработки, чтобы его каким-то образом там использовать, например, для вывода информации в сообщении об ошибке. Примеры:

* catch( exception e) // по значению
* catch( exception &e) // по ссылке
* catch( const exception &e) // по константной ссылке
* catch( exception \*e) //по указателю

Лучше всего передавать объект по ссылке, т. к. при этом не создается временный объект-исключение.

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

В составе стандартной библиотеки С++ реализован ряд стандартных исключений, которые организованы в иерархию классов.

Эта иерархия может служить основой для создания собственных классов исключений и иерархии исключений. Можно определять собственные исключения, унаследовав их от класса *exception*.

1. Каким образом можно создать собственную иерархию исключений?

Для создания собственной иерархии исключений надо объявить свой базовый класс-исключение, например:

class BaseException{};

Остальные классы будут наследниками этого класса, аналогично тому, как это сделано в иерархии стандартных исключений:

class Child\_Exception1:public BaseException{};

class Child\_Exception2:public BaseException{};

Класс BaseException можно унаследовать от стандартного класса exception class BaseException: public exception{};

Наследование от стандартных классов позволит использовать метод what для вывода сообщений об ошибках.

Иерархия классов-исключений позволяет вместо нескольких разных блоков-ловушек написать единственный блок с типом аргумента базового класса.

1. Если спецификация исключений имеет вид:

void f1()throw(int,double);

то какие исключения может прождать функция f1()?

Для каждой функции, метода, конструктора или деструктора можно в заголовке указать спецификацию исключений. Если в заголовке спецификация исключений не указана, считается, что функция может порождать любое исключение, если указана, то считается, что функция генерирует те исключения, которые явно указаны в этом списке.

Следовательно, функция f1() может генерировать исключения типа int и double.

1. Если спецификация исключений имеет вид: void f1()throw(); то какие исключения может порождать функция f1()?

Если спецификация имеет вид такой вид, то считается, что функция исключений не генерирует.

1. В какой части программы может генерироваться исключение?

Исключение могут генерируется внутри оператора try { }.

1. Написать функцию, которая вычисляет площадь треугольника по трем сторонам (формула Герона).

Функцию реализовать в 4 вариантах:

* без спецификации исключений;

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

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

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

}

* с конкретной спецификацией с подходящим стандартным исключением;

#include <stdexcept>

double triangleArea(double a, double b, double c) throw(std::invalid\_argument) {

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

throw std::invalid\_argument("Invalid triangle sides");

}

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

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

}

* спецификация с собственным реализованным исключением

#include <exception>

class InvalidTriangleException : public std::exception {

public:

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

return "Invalid triangle sides";

}

};

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

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

throw InvalidTriangleException();

}

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

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

}

**Приложения**

Класс main:

#include <iostream>

#include "LinkedList.h"

#include "Pair.h"

int main() {

setlocale(0, ".1251");

Pair pair(1, 1.5);

LinkedList<Pair> list1(5, pair);

list1.push(pair);

std::cout << list1 << std::endl;

Pair pair2(2, 10);

LinkedList<Pair> list2(5, pair2);

std::cout << list2 << std::endl;

LinkedList<Pair> list = list1 \* list2;

std::cout << list << std::endl;

Iterator<Pair> it = list.first();

for (; it != list.last(); it++) {

std::cout << \*it << " ";

}

std::cout << \*it << std::endl;

std::cout << "\n\n\n\n\n" << std::endl;

LinkedList<double> list3(5, 4.5);

list3.push(6);

std::cout << list3 << std::endl;

LinkedList<double> list4(5, 10);

std::cout << list4 << std::endl;

LinkedList<double> list34 = list3 \* list4;

std::cout << list34 << std::endl;

Iterator<double> it2 = list34.first();

for (; it2 != list34.last(); it2++) {

std::cout << \*it2 << " ";

}

std::cout << \*it2;

return 0;

}

Класс LinkedList.h

#ifndef TASKSEVEN\_LINKEDLIST\_H

#define TASKSEVEN\_LINKEDLIST\_H

#include <iostream>

#include <algorithm>

#include "Iterator.h"

#include "Error.h"

template<typename T>

class LinkedList {

private:

size\_t size = 0;

Point<T> \*head = nullptr;

Point<T> \*top = nullptr;

void output(Point<T> \*obj) const

void show() const;

public:

friend std::ostream &operator<<(std::ostream &out, const LinkedList<T> &list);

friend std::istream &operator>>(std::istream &in, LinkedList<T> &list);

LinkedList();

LinkedList(size\_t s, T k);

~LinkedList();

LinkedList(const LinkedList<T> &list);

T &operator[](int index) const;

LinkedList &operator=(const LinkedList<T> &list);

LinkedList operator\*(LinkedList<T> &list);

int operator()();

void push(T k);

T pop();

Iterator<T> first();

Iterator<T> last();

};

#endif

Класс LinkedList.cpp

#include "LinkedList.h"

template<typename T>

LinkedList<T>::LinkedList() {

head = nullptr;

top = nullptr;

size = 0;

}

template<typename T>

LinkedList<T>::LinkedList(size\_t s, T k) {

size = s;

if (size != 0) {

Point<T> \*obj = new Point<T>;

obj->num = k;

head = obj;

Point<T> \*temp = head;

for (int i = size - 1; i != 0; i--) {

obj = new Point<T>;

temp->next = obj;

obj->num = k;

temp = obj;

}

top = obj;

}

}

template<typename T>

void LinkedList<T>::push(T k) {

if (size == 0) {

head = new Point<T>;

head->num = k;

top = head;

} else {

Point<T> \*obj = new Point<T>;

top->next = obj;

obj->num = k;

top = obj;

}

size++;

}

template<typename T>

T LinkedList<T>::pop() {

if (size == 0) throw EmptySizeError();

T k = head->num;

Point<T> \*temp = head;

head = head->next;

size--;

delete temp;

return k;

}

template<typename T>

LinkedList<T>::~LinkedList() {

while (head != nullptr) {

pop();

}

}

template<typename T>

void LinkedList<T>::output(Point<T> \*obj) const {

std::cout << obj->num << " ";

if (obj == top) return;

output(obj->next);

}

template<typename T>

void LinkedList<T>::show() const {

if (size == 0) std::cout << "List is empty" << std::endl;

else output(head);

template<typename T>

int LinkedList<T>::operator()() {

return size;

}

template<typename T>

T &LinkedList<T>::operator[](int index) const {

if (index == 0 && size == 0) throw IndexError();

if (index < 0) throw IndexError1();

if (index >= size) throw IndexError2();

Point<T> \*temp = head;

int count = 0;

while (temp != nullptr) {

if (count == index)

return temp->num;

temp = temp->next;

++count;

}

}

template<typename T>

LinkedList<T>::LinkedList(const LinkedList<T> &list) {

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

this->push(list[i]);

}

}

template<typename T>

std::ostream &operator<<(std::ostream &out, const LinkedList<T> &list) {

list.show();

return out;

}

template<typename T>

std::istream &operator>>(std::istream &in, LinkedList<T> &list) {

T k;

in >> k;

list.push(k);

return in;

}

template<typename T>

LinkedList<T> &LinkedList<T>::operator=(const LinkedList<T> &list) {

int size\_temp = size;

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

this->pop();

}

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

this->push(list[i]);

}

return \*this;

}

template<typename T>

LinkedList<T> LinkedList<T>::operator\*(LinkedList<T> &list) {

T nulik;

nulik = 0;

LinkedList<T> new\_list;

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

new\_list.push((\*this)[i] \* list[i]);

}

for (int i = 0; i < max((int) size, list()) - min((int) size, list()); i++) {

new\_list.push(nulik);

}

return new\_list;

}

template<typename T>

Iterator<T> LinkedList<T>::first() {

return Iterator<T>(head);

}

template<typename T>

Iterator<T> LinkedList<T>::last() {

return Iterator<T>(top);

}

Класс Iterator.h

#ifndef TASKSEVEN\_ITERATOR\_H

#define TASKSEVEN\_ITERATOR\_H

#include "Error.h"

template<typename T>

struct Point {

T num;

Point \*next = nullptr;

};

template<typename T>

class Iterator {

private:

Point<T> \*elem = nullptr;

public:

Iterator();

Iterator(const Iterator &it);

bool operator==(const Iterator &it);

bool operator!=(const Iterator &it)

void operator++();

void operator++(int);

[[noreturn]] Iterator &operator+(int n) {

for (int i = 0; i < n && elem->next != nullptr; i++)

elem = elem->next;

return \*this;

}

T operator\*() const;

}

#endif

Класс Iterator.cpp

#include "Iterator.h"

template<typename T>

Iterator<T>::Iterator() {

elem = nullptr;

}

template<typename T>

Iterator<T>::Iterator(const Iterator &it) {

elem = it.elem;

}

template<typename T>

bool Iterator<T>::operator==(const Iterator &it) {

return elem == it.elem;

}

template<typename T>

bool Iterator<T>::operator!=(const Iterator &it) {

return elem != it.elem;

}

template<typename T>

void Iterator<T>::operator++() {

if (elem == nullptr) throw IndexError2();

elem = elem->next;

return \*this;

}

template<typename T>

void Iterator<T>::operator++(int) {

if (elem == nullptr) throw IndexError2();

Iterator<T> temp(elem);

++(\*this);

return temp;

}

template<typename T>

T Iterator<T>::operator\*() const {

if (elem)

return elem->num;

else

throw IndexError2();

}

Класс Pair.h

#ifndef TASKSEVEN\_PAIR\_H

#define TASKSEVEN\_PAIR\_H

#include <ostream>

#include <istream>

struct Pair {

Pair();

Pair(int a, double b);

int numINT = 0;

double numDOUBLE = 0;

Pair &operator=(const Pair &pair);

Pair &operator=(const int &k);

friend std::ostream &operator<<(std::ostream &out, const Pair &pair);

friend std::istream &operator>>(std::istream &in, Pair &pair);

Pair operator\*(Pair &pair);

};

#endif

Класс Pait.cpp

#include "Pair.h"

Pair::Pair() {

}

Pair::Pair(int a, double b) {

numINT = a;

numDOUBLE = b;

}

Pair &Pair::operator=(const Pair &pair) {

this->numINT = pair.numINT;

this->numDOUBLE = pair.numDOUBLE;

return \*this;

}

Pair &Pair::operator=(const int &k) {

this->numINT = k;

this->numDOUBLE = k;

return \*this;

}

Pair Pair::operator\*(Pair &pair) {

Pair new\_pair;

new\_pair.numINT = this->numINT \* pair.numINT;

new\_pair.numDOUBLE = this->numDOUBLE \* pair.numDOUBLE;

return new\_pair;

}

std::ostream &operator<<(std::ostream &out, const Pair &pair) {

out << pair.numINT << " : " << pair.numDOUBLE << " |";

return out;

}

std::istream &operator>>(std::istream &in, Pair &pair) {

in >> pair.numINT >> pair.numDOUBLE;

return in;

}

Класс Error

#ifndef TASKNINE\_ERROR\_H

#define TASKNINE\_ERROR\_H

#include <iostream>

class Error {

public:

virtual void what() {};

};

class IndexError : public Error {

protected:

std::string msg;

public:

IndexError() {

msg = "Index Error\n";

}

virtual void what() {

std::cout << msg;

}

}

class SizeError : public Error {

protected:

std::string msg;

public:

SizeError() {

msg = "size error\n";

}

void what() override {

std::cout << msg;

}

};

class EmptySizeError : public SizeError {

protected:

std::string msg\_;

public:

EmptySizeError() {

SizeError();

msg\_ = "List is empty\n";

void what() override {

std::cout << msg << msg\_;

}

};

class IndexError1 : public IndexError {

protected:

std::string msg\_;

public:

IndexError1() {

IndexError();

msg\_ = "index <0\n";

}

void what() override {

std::cout << msg << msg\_;

}

};

class IndexError2 : public IndexError {

protected:

std::string msg\_;

public:

IndexError2() {

IndexError();

msg\_ = "index>size\n";

}

void what() override {

std::cout << msg << msg\_;

}

};

#endif