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: «Основы алгоритмизации и программирования»

# Лабораторная работа №9

# «Обработка исключительных ситуаций»

# Семестр 2

Выполнил работу

Студент группы РИС-22-1Б

Юхновец Владимир

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:

[] – доступа по индексу;

int() – определение размера списка;

\* вектор – умножение элементов списков a[i]\*b[i];

+n - переход вправо к элементу с номером n.

Вариант реализации: 1,3

**Ответы на контрольные вопросы**

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

Исключение в С++ - это механизм обработки ошибок, который позволяет программисту обрабатывать неожиданные ситуации, возникающие во время выполнения программы.

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

Исключение в С++ можно разделить на три части: генерация, передача и обработка. Генерация исключения происходит в теле функции, передача исключения осуществляется в стеке вызовов функций, а обработка исключения выполняется в блоке try-catch. Представление исключения в виде таких отдельных частей позволяет более гибко и точно контролировать процесс обработки ошибок в программе.

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

Для генерации исключительной ситуации в С++ используется оператор throw.

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

Контролируемый блок в С++ - это блок кода, в котором возможно возникновение исключения. Для того, чтобы контролировать выполнение этого блока и корректно обрабатывать возможные ошибки, вокруг него оборачивается блок try-catch.

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

Секция-ловушка в С++ - это блок кода внутри блока catch, который выполняется в случае возникновения исключения, соответствующего типу указанному в блоке catch. Секция-ловушка позволяет обработать возникшую ошибку и принять необходимые меры для продолжения выполнения программы.

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

Спецификация исключения в секции ловушке может иметь три формы:

catch (exception\_type e) - ловушка для определенного типа исключения;

catch (...) - ловушка для всех типов исключений;

catch - ловушка для всех типов исключений.

В первом случае используется, когда нужно обработать конкретный тип исключения, во втором и третьем - когда нужно обработать любое исключение.

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

В С++ стандартной библиотекой предоставляется класс std::exception, который является базовым классом для всех стандартных исключений. Этот класс определяет метод what(), который возвращает строку, описывающую исключение.

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

Для создания собственной иерархии исключений, необходимо определить пользовательский класс, который будет наследоваться от std::exception или от другого пользовательского класса, который уже наследуется от std::exception.

Пример определения пользовательского класса исключения, который наследуется от std::exception:

#include <exception>

class MyException : public std::exception {

public:

const char\* what() const noexcept {

return "Custom exception occurred";

}

};

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

Если функция f1() имеет спецификацию исключений throw(int, double), то это означает, что функция может генерировать исключения типа int и double.

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

Если функция f1() имеет спецификацию исключений throw(), то это означает, что функция не может генерировать исключения.

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

Исключение может быть сгенерировано в любой части программы, где необходимо обработать некоторое исключительное состояние. Обычно исключение генерируется в блоке try с помощью оператора throw, когда возникает ошибка или иное исключительное состояние.

**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.**

**без спецификации исключений:**

#include <iostream>

#include <cmath>

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

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

double area = sqrt(s \* (s - a) \* (s - b) \* (s - c));

return area;

}

int main() {

double a, b, c;

std::cout << "Enter the lengths of the sides of the triangle: ";

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

try {

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

std::cout << "The area of the triangle is: " << area << std::endl;

} catch (...) {

std::cout << "An exception occurred while calculating the area of the triangle" << std::endl;

}

return 0;

}

**с использованием спецификации throw():**

#include <iostream>

#include <cmath>

#include <stdexcept>

double calculateTriangleArea(double a, double b, double c) throw(std::runtime\_error) {

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

throw std::runtime\_error("The given lengths do not form a valid triangle");

}

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

double area = sqrt(s \* (s - a) \* (s - b) \* (s - c));

return area;

}

int main() {

double a, b, c;

std::cout << "Enter the lengths of the sides of the triangle: ";

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

try {

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

std::cout << "The area of the triangle is: " << area << std::endl;

} catch (std::runtime\_error& e) {

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

}

return 0;

}

**реализация функции с использованием конкретной спецификации с подходящим стандартным исключением (в данном случае, invalid\_argument):**

#include <iostream>

#include <cmath>

#include <stdexcept>

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

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

throw std::invalid\_argument("The given lengths do not form a valid triangle");

}

double s = (a + b + c)

**Реализация 1**

**Диаграмма класса**

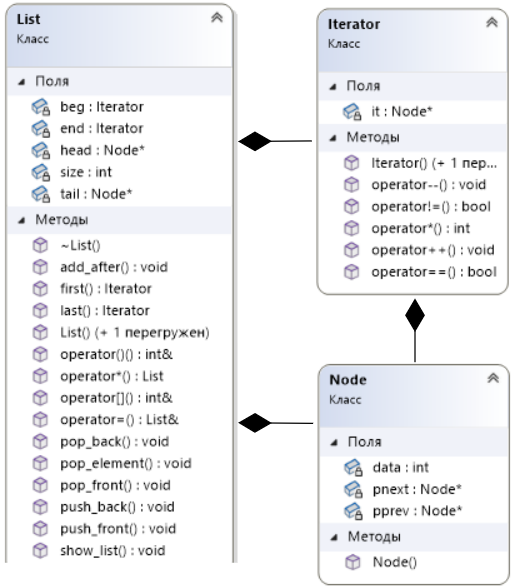


Рисунок 1 – диаграмма классов List, Iterator, Node

**Описание класса**

**Класс List**

#pragma once

#include <iostream>

#include <string>

using namespace std;

class List

{

private:

class Node

{

friend class List;

private:

int data;

Node\* pnext = nullptr;

Node\* pprev = nullptr;

public:

Node(int data);

};

int size = 0;

Node\* head = nullptr;

Node\* tail = nullptr;

public:

List();

List(List& object);

~List();

class Iterator

{

friend class List;

private:

Node\* it = nullptr;

public:

Iterator();

Iterator(const Iterator& it);

bool operator==(const Iterator& it);

bool operator!=(const Iterator& it);

void operator++();

void operator--();

int operator\*();

};

void push\_front(int data);

void pop\_front();

void push\_back(int data);

void pop\_back();

void show\_list();

void add\_after(int number, int data);

void pop\_element(int number);

int& operator[](int index) const;

int& operator()();

List& operator=(const List& object);

List operator\*(List& object);

Iterator first()

{

if (head != nullptr)

beg.it = head;

return beg;

}

Iterator last()

{

if (tail != nullptr)

end.it = tail->pnext;

return end;

}

private:

Iterator beg;

Iterator end;

};

**Определение методов**

**Класс List**

#include "List.h"

List::List()

{

size = 0;

head = nullptr;

tail = nullptr;

}

List::List(List& object)

{

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

(\*this).push\_back(object[i]);

}

List::~List()

{

if (head != nullptr)

{

Node\* current = head;

while (current != nullptr)

{

Node\* next = current->pnext;

delete current;

current = next;

}

head = nullptr;

tail = nullptr;

size = 0;

}

}

List::Node::Node(int data)

{

this->data = data;

}

void List::push\_front(int data)

{

if (head == nullptr)

{

head = new Node(data);

tail = head;

}

else

{

Node\* temp = head;

head = new Node(data);

temp->pprev = head;

head->pnext = temp;

}

++size;

}

void List::pop\_front()

{

if (head != nullptr)

{

Node\* temp = head->pnext;

delete head;

head = temp;

if (head != nullptr)

head->pprev = 0;

else

tail = nullptr;

--size;

}

else

throw 3;

}

void List::push\_back(int data)

{

if (head == nullptr)

{

head = new Node(data);

tail = head;

}

else

{

Node\* current = tail;

current->pnext = new Node(data);

tail = current->pnext;

tail->pprev = current;

tail->pnext = 0;

}

++size;

}

void List::pop\_back()

{

if (head != nullptr)

{

Node\* current = tail->pprev;

delete tail;

tail = current;

if (tail != nullptr)

current->pnext = nullptr;

else

head = nullptr;

--size;

}

else

throw 3;

}

void List::show\_list()

{

if (head != nullptr)

{

int choice;

cout << "1 - Print the list from the beginning\n2 - Print the list from the end\n";

cout << "choice: "; cin >> choice;

switch (choice)

{

case 1:

{

Node\* current = head;

for (size\_t i = 0; i < size; i++)

{

cout << current->data << " ";

current = current->pnext;

}

cout << endl;

break;

}

case 2:

{

cout << "List: ";

Node\* current = tail;

for (size\_t i = size; i > 0; i--)

{

cout << current->data << " ";

current = current->pprev;

}

cout << endl;

break;

}

default:

throw 1;

break;

}

}

else

throw 3;

}

void List::add\_after(int number, int data)

{

if (head != nullptr)

{

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

{

if (number == size)

push\_back(data);

else if (number == 0)

push\_front(data);

else

{

if (number <= size / 2)

{

Node\* current = head;

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

current = current->pnext;

Node\* temp = current->pnext;

current->pnext = new Node(data);

current->pprev = current;

current = current->pnext;

current->pnext = temp;

++size;

}

else

{

Node\* current = tail;

for (int i = size; i > number; i--)

current = current->pprev;

Node\* temp = current->pnext;

Node\* temp2 = current;

current->pnext = new Node(data);

temp->pprev = current->pnext;

current = current->pnext;

current->pprev = temp2;

current->pnext = temp;

++size;

}

}

}

else

throw 2;

}

else

throw 3;

}

void List::pop\_element(int number)

{

if (head != nullptr)

{

if (number >= 1 && number <= size)

{

if (number == size)

pop\_back();

else if (number == 1)

pop\_front();

else

{

if (number <= size / 2)

{

Node\* current = head;

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

current = current->pnext;

Node\* temp = current->pprev;

Node\* temp2 = current->pnext;

delete current;

temp->pnext = temp2;

temp2->pprev = temp;

}

else

{

Node\* current = tail;

for (int i = size; i > number; i--)

current = current->pprev;

Node\* temp = current->pprev;

Node\* temp2 = current->pnext;

delete current;

temp->pnext = temp2;

temp2->pprev = temp;

}

--size;

}

}

else

throw 2;

}

else

throw 3;

}

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

{

if (head == nullptr)

throw 3;

else if (index < 0 or index >= size)

throw 2;

else

{

if (index <= size / 2)

{

Node\* current = head;

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

current = current->pnext;

return current->data;

}

else

{

Node\* current = tail;

for (int i = size - 1; i > index; --i)

current = current->pprev;

return current->data;

}

}

}

int& List::operator()()

{

return size;

}

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

{

if (this != &object)

if (this->head != nullptr)

{

this->~List();

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

this->push\_back(object[i]);

}

else

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

this->push\_back(object[i]);

return \*this;

}

List List::operator\*(List& object)

{

if (object.size > size)

{

List temp;

Node\* current = (\*this).head;

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

{

if (current != nullptr)

temp.push\_back((\*this)[i] \* object[i]);

else

temp.push\_back(0);

if (current != 0)

current = current->pnext;

}

return temp;

}

else

{

List temp;

Node\* current = object.head;

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

{

if (current != nullptr)

temp.push\_back((\*this)[i] \* object[i]);

else

temp.push\_back(0);

if (current != 0)

current = current->pnext;

}

return temp;

}

}

**Реализация 2**

**Диаграмма класса**

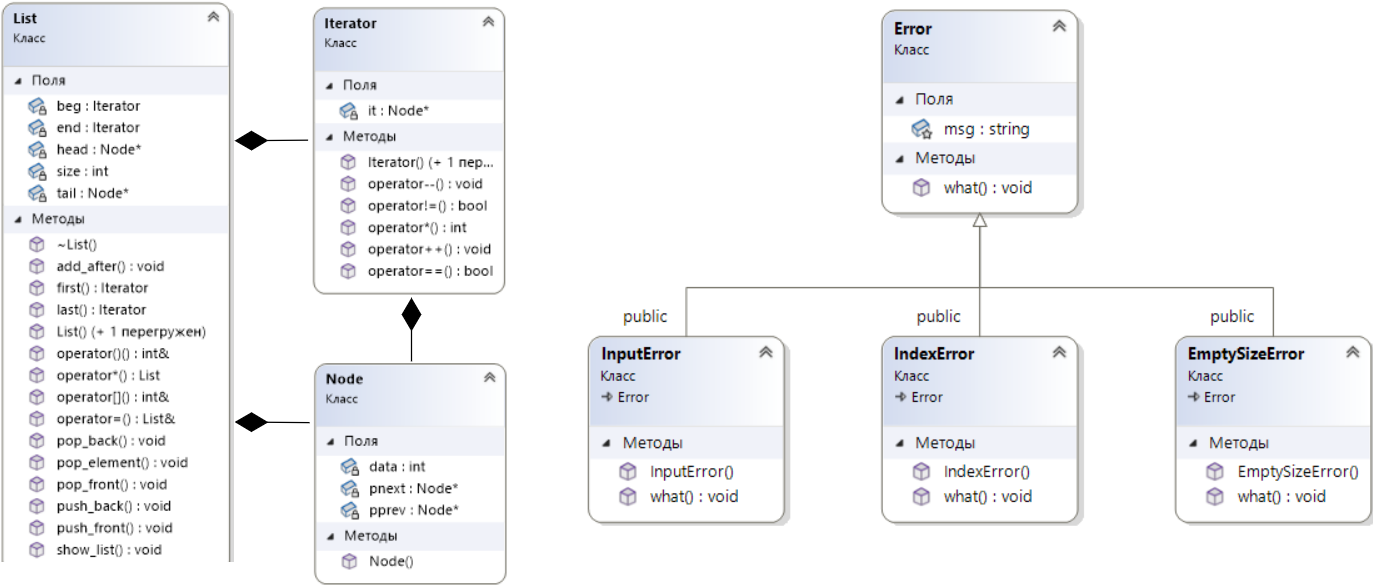


Рисунок 1 – диаграмма классов List, Iterator, Node, Error, InputError, IndexError, EmptySizeError

**Описание класса**

**Класс List**

#pragma once

#include <iostream>

#include <string>

using namespace std;

class List

{

private:

class Node

{

friend class List;

private:

int data;

Node\* pnext = nullptr;

Node\* pprev = nullptr;

public:

Node(int data);

};

int size = 0;

Node\* head = nullptr;

Node\* tail = nullptr;

public:

List();

List(List& object);

~List();

class Iterator

{

friend class List;

private:

Node\* it = nullptr;

public:

Iterator();

Iterator(const Iterator& it);

bool operator==(const Iterator& it);

bool operator!=(const Iterator& it);

void operator++();

void operator--();

int operator\*();

};

void push\_front(int data);

void pop\_front();

void push\_back(int data);

void pop\_back();

void show\_list();

void add\_after(int number, int data);

void pop\_element(int number);

int& operator[](int index) const;

int& operator()();

List& operator=(const List& object);

List operator\*(List& object);

Iterator first()

{

if (head != nullptr)

beg.it = head;

return beg;

}

Iterator last()

{

if (tail != nullptr)

end.it = tail->pnext;

return end;

}

private:

Iterator beg;

Iterator end;

};

**Класс Error**

#pragma once

#include <string>

#include <iostream>

using namespace std;

class Error

{

protected:

string msg;

public:

virtual void what() {};

};

**Класс IndexError**

class IndexError : public Error

{

public:

IndexError() { msg = "Out of range!\n"; };

void what() override { cout << msg; }

};

**Класс InputError**

class InputError : public Error

{

public:

InputError() { msg = "Input Error!"; };

void what() override { cout << msg; }

};

**Класс EmptySizeError**

class EmptySizeError : public Error

{

public:

EmptySizeError() { msg = "List is empty!\n"; };

void what() override { cout << msg; }

};

**Определение методов**

**Класс List**

#include "List.h"

List::List()

{

size = 0;

head = nullptr;

tail = nullptr;

}

List::List(List& object)

{

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

(\*this).push\_back(object[i]);

}

List::~List()

{

if (head != nullptr)

{

Node\* current = head;

while (current != nullptr)

{

Node\* next = current->pnext;

delete current;

current = next;

}

head = nullptr;

tail = nullptr;

size = 0;

}

}

List::Node::Node(int data)

{

this->data = data;

}

void List::push\_front(int data)

{

if (head == nullptr)

{

head = new Node(data);

tail = head;

}

else

{

Node\* temp = head;

head = new Node(data);

temp->pprev = head;

head->pnext = temp;

}

++size;

}

void List::pop\_front()

{

if (head != nullptr)

{

Node\* temp = head->pnext;

delete head;

head = temp;

if (head != nullptr)

head->pprev = 0;

else

tail = nullptr;

--size;

}

else

throw 3;

}

void List::push\_back(int data)

{

if (head == nullptr)

{

head = new Node(data);

tail = head;

}

else

{

Node\* current = tail;

current->pnext = new Node(data);

tail = current->pnext;

tail->pprev = current;

tail->pnext = 0;

}

++size;

}

void List::pop\_back()

{

if (head != nullptr)

{

Node\* current = tail->pprev;

delete tail;

tail = current;

if (tail != nullptr)

current->pnext = nullptr;

else

head = nullptr;

--size;

}

else

throw 3;

}

void List::show\_list()

{

if (head != nullptr)

{

int choice;

cout << "1 - Print the list from the beginning\n2 - Print the list from the end\n";

cout << "choice: "; cin >> choice;

switch (choice)

{

case 1:

{

Node\* current = head;

for (size\_t i = 0; i < size; i++)

{

cout << current->data << " ";

current = current->pnext;

}

cout << endl;

break;

}

case 2:

{

cout << "List: ";

Node\* current = tail;

for (size\_t i = size; i > 0; i--)

{

cout << current->data << " ";

current = current->pprev;

}

cout << endl;

break;

}

default:

throw 1;

break;

}

}

else

throw 3;

}

void List::add\_after(int number, int data)

{

if (head != nullptr)

{

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

{

if (number == size)

push\_back(data);

else if (number == 0)

push\_front(data);

else

{

if (number <= size / 2)

{

Node\* current = head;

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

current = current->pnext;

Node\* temp = current->pnext;

current->pnext = new Node(data);

current->pprev = current;

current = current->pnext;

current->pnext = temp;

++size;

}

else

{

Node\* current = tail;

for (int i = size; i > number; i--)

current = current->pprev;

Node\* temp = current->pnext;

Node\* temp2 = current;

current->pnext = new Node(data);

temp->pprev = current->pnext;

current = current->pnext;

current->pprev = temp2;

current->pnext = temp;

++size;

}

}

}

else

throw 2;

}

else

throw 3;

}

void List::pop\_element(int number)

{

if (head != nullptr)

{

if (number >= 1 && number <= size)

{

if (number == size)

pop\_back();

else if (number == 1)

pop\_front();

else

{

if (number <= size / 2)

{

Node\* current = head;

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

current = current->pnext;

Node\* temp = current->pprev;

Node\* temp2 = current->pnext;

delete current;

temp->pnext = temp2;

temp2->pprev = temp;

}

else

{

Node\* current = tail;

for (int i = size; i > number; i--)

current = current->pprev;

Node\* temp = current->pprev;

Node\* temp2 = current->pnext;

delete current;

temp->pnext = temp2;

temp2->pprev = temp;

}

--size;

}

}

else

throw 2;

}

else

throw 3;

}

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

{

if (head == nullptr)

throw 3;

else if (index < 0 or index >= size)

throw 2;

else

{

if (index <= size / 2)

{

Node\* current = head;

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

current = current->pnext;

return current->data;

}

else

{

Node\* current = tail;

for (int i = size - 1; i > index; --i)

current = current->pprev;

return current->data;

}

}

}

int& List::operator()()

{

return size;

}

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

{

if (this != &object)

if (this->head != nullptr)

{

this->~List();

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

this->push\_back(object[i]);

}

else

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

this->push\_back(object[i]);

return \*this;

}

List List::operator\*(List& object)

{

if (object.size > size)

{

List temp;

Node\* current = (\*this).head;

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

{

if (current != nullptr)

temp.push\_back((\*this)[i] \* object[i]);

else

temp.push\_back(0);

if (current != 0)

current = current->pnext;

}

return temp;

}

else

{

List temp;

Node\* current = object.head;

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

{

if (current != nullptr)

temp.push\_back((\*this)[i] \* object[i]);

else

temp.push\_back(0);

if (current != 0)

current = current->pnext;

}

return temp;

}

}

**Результаты работы программы**

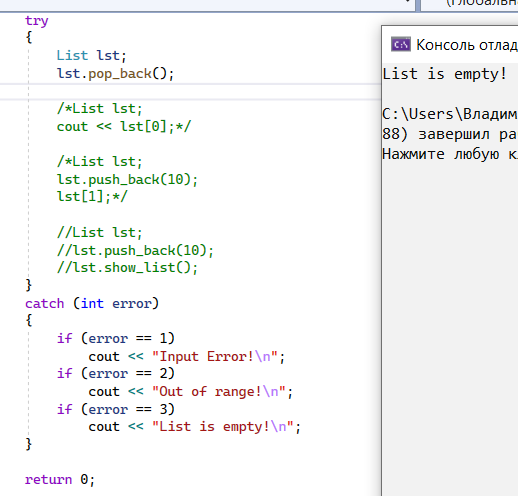


Рисунок 2 – обработка удаления элемента из пустого списка

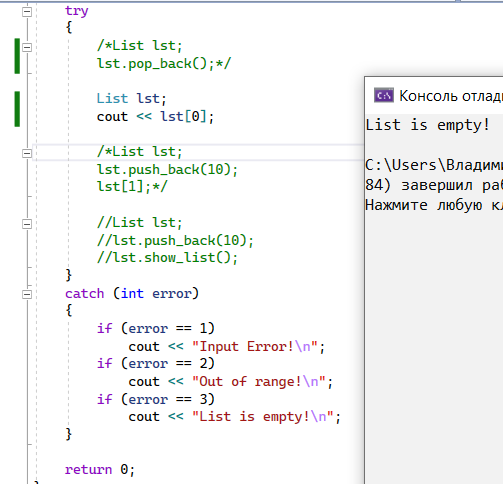


Рисунок 3 – обработка обращения к индексу пустого списка

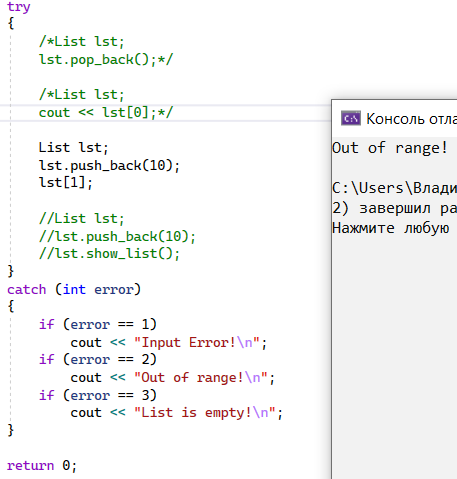


Рисунок 4 – обработка обращения к несуществующему индексу списка

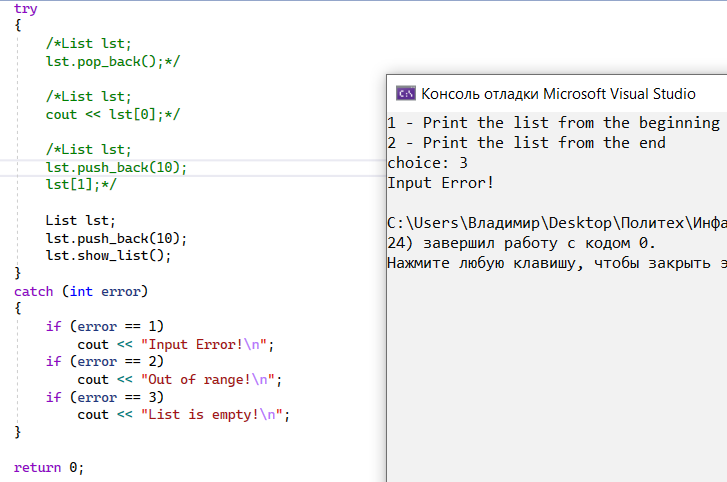


Рисунок 5 – обработка некорректного ввода