Министерство образования и науки Российской Федерации

Федеральное государственное образовательное автономное учреждение высшего образования

"Пермский национальный исследовательский политехнический университет"

ОТЧЕТ

ПО ЛАБОРАТОРНОЙ РАБОТЕ №9

Дисциплина: Основы алгоритмизации и программирования

Тема: Обработка исключительных ситуаций

|  |  |
| --- | --- |
| Выполнил работу | |
| Студент группы РИС-22-1б | |
| Деревнин И.В. | |
|  | |
| Проверил работу | |
| Доцент кафедры ИТАС | |
| Полякова О.А. | |
|  | |

Пермь – 2023

**Анализ предметной области**

**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:

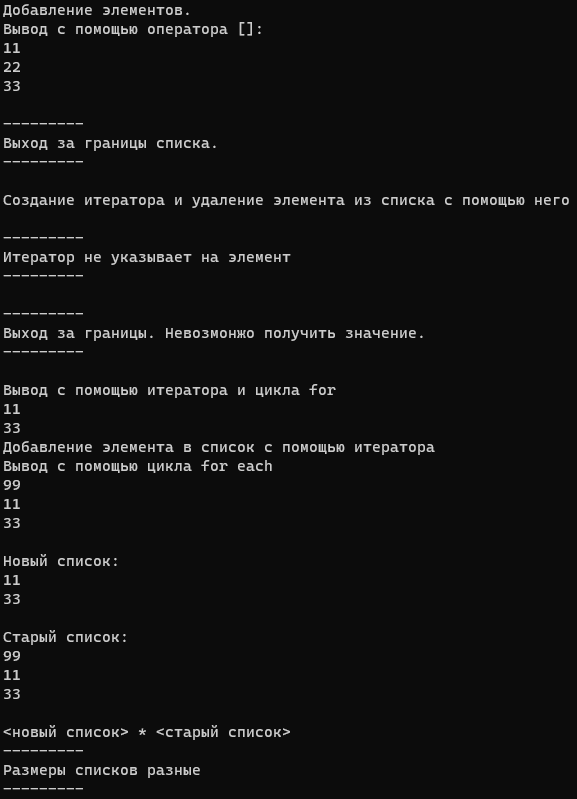
[] – доступ по индексу;

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

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

**Анализ задачи**

Данный класс список уже реализован в 6 лабораторной работе, поэтому она возьмется за основу, и в нее будут добавлены исключительные ситуации. 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. Также добавлены исключения для итератора – оператор +, ++() и ++(int) – если итератор пытается выйти за последний элемент списка, то выкидывается исключение.

****Тестирование программы**

*Рис. 1 – Тестирование программы.*

**Заключение**

Была разработана программа, которая дополняет 6 лабораторную работу. Дополнениями является генерация исключительных ситуаций для методов класса и методов итератора. Данные исключительные ситуации в основном предупреждают выход за границы списка, а следовательно упреждает утечку данных.

# 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.

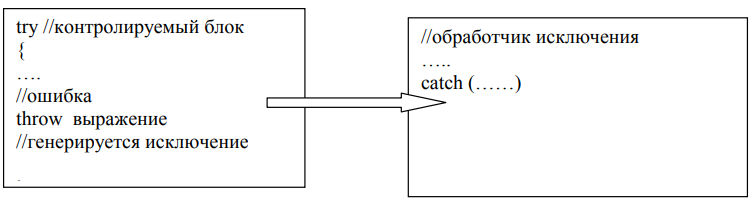
1. На какие части исключения позволяют разделить вычислительный процесс? Достоинства такого подхода?

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();

}

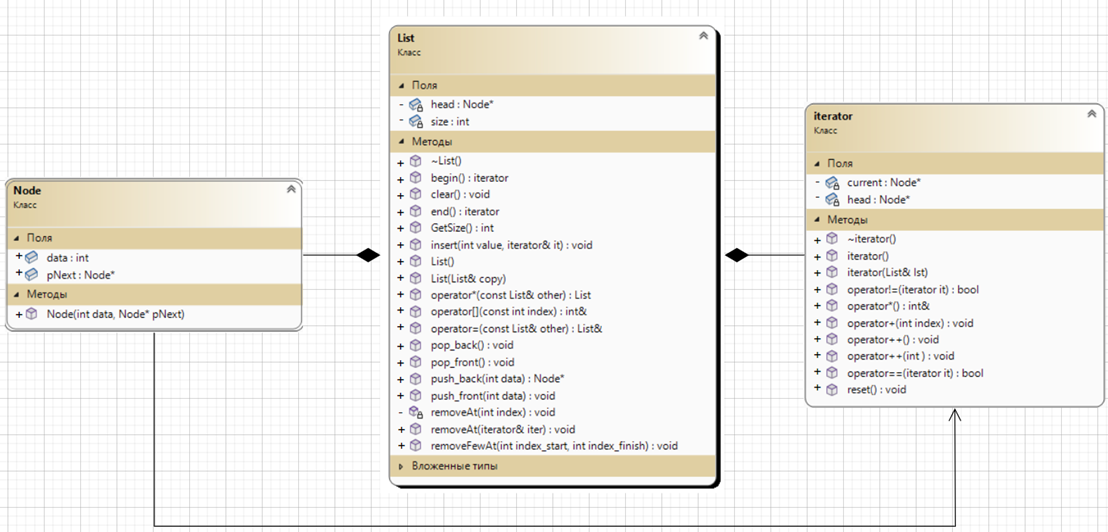
double p = (a + b + c) / 2;

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

}

**Приложения**

UML-диаграмма:



Приложение Б – код программы

Main.cpp:

#include <Windows.h>

#include <fstream>

#include "List.h"

using namespace std;

int main()

{

system("chcp 1251 >> null");

//system("color F0");

List lst;

cout << "Добавление элементов." << endl;

lst.push\_back(22);

lst.push\_front(11);

lst.push\_back(33);

cout << "Вывод с помощью оператора []:" << endl;

try //блок попытки

{

for (int i = 0; i < lst.GetSize() + 2; i++)

{

cout << lst[i] << endl;

}

}

catch (const std::exception& ex) //если выброшено исключение, то отрабатывает этот блок

{

cout << endl<< "---------" << endl << ex.what() << endl << "---------" << endl << endl;

}

cout << "Создание итератора и удаление элемента из списка с помощью него" << endl;

try //блок попытки

{

List::iterator i;

lst.removeAt(i);

}

catch (const std::exception& ex) //если выброшено исключение, то отрабатывает этот блок

{

cout<< endl << "---------" << endl << ex.what() << endl << "---------" << endl ;

}

List::iterator i = lst.begin(); i++;

lst.removeAt(i);

try //блок попытки

{

i++; i++;

}

catch (const std::exception& ex) //если выброшено исключение, то отрабатывает этот блок

{

cout << endl << "---------" << endl << ex.what() << endl << "---------" << endl << endl;

i.reset();

}

cout << "Вывод с помощью итератора и цикла for" << endl;

for (auto it = lst.begin(); it != lst.end(); it++)

{

cout << \*it << endl;

}

cout << "Добавление элемента в список с помощью итератора" << endl;

lst.insert(99, i);

cout << "Вывод с помощью цикла for each\n";

for (auto& it : lst)

{

cout << it << endl;

}

List list2;

list2 = lst;

list2.pop\_front();

cout << endl << "Новый список:" << endl;

for (auto& it : list2)

{

cout << it << endl;

}

cout << endl << "Старый список:" << endl;

for (auto& it : lst)

{

cout << it << endl;

}

cout << "\n<новый список> \* <старый список>" << endl;

try //блок попытки

{

list2 = (list2 \* lst);

}

catch (const std::exception& ex) //если выброшено исключение, то отрабатывает этот блок

{

cout << "---------" << endl << ex.what() << endl << "---------" << endl;

list2 = lst;

}

return 0;

}

list.h:

#pragma once

#include <iostream>

class List

{

private:

void removeAt(int index);

//минимальная ячейка памяти

class Node //вложенный класс - ячейка памяти

{

public:

Node(int data = int(), Node\* pNext = nullptr) //используется параметр по умолчанию, то есть если мы не передаем указатель, то он автоматически присваивается nullptr

{ //для поля data также используется параметр по умолчанию, выглядит он как вызов конструктора по умолчанию

this->data = data;

this->pNext = pNext;

}

Node\* pNext; //хранит указатель на следующий элемент

int data; //хранит наши данные

};

Node\* head; //указатель на первый элемент в спискеint

int size; //количество элементов в списке.

public:

class iterator //вложнный итератор

{

private:

friend class List;

Node\* head; //указатель на node

Node\* current;

public:

iterator(); //конструкторы итератора / деструкторы

~iterator();

iterator(List& lst);

void reset();

void operator +(int index); //операторы итератора

void operator ++(int);

void operator ++();

int& operator \*();

bool operator ==(iterator it);

bool operator !=(iterator it);

};

List(); //Конструктор

List(List& copy);//!!!!!!

~List(); //Деструктор

List& operator=(const List& other);//!!!!!!!!!!

Node\* push\_back(int data); //добавляет элемент в конец списка

int GetSize() { return size; } //так как инкапсуляция не дает нам доступ к количеству элементов в списке, то нам необходимо происать геттер

int& operator[](const int index); //перегрузка оператора [] для того, чтобы мы могли взаимодействовать со списком как с массивом

void pop\_front(); //удаляет элемент из начала списка

void clear(); //удаляет все элементы списка (логика как у pop\_front, деструктор основан на этой штуке)

void push\_front(int data); //добавляет элемент в начало списка

void removeAt(List::iterator& iter); //удаление элемента по указаному индексу

void pop\_back(); //удаление из конца списка

iterator begin();

iterator end();

void removeFewAt(int index\_start, int index\_finish); //удаление нескольких элементов

friend class iterator;

void insert(int value, List::iterator& it);

List operator\*(const List& other);

};

list.cpp:

#include"List.h"

#include <iostream>

List::List() //мы создаем пустой список, а следствено, в нем лежит 0 элементов, а следственно указывать некуда.

{

size = 0;

head = nullptr;

}

List::List(List& copy)

{

this->size = 0;

this->head = nullptr;

Node\* temp = copy.head;

while (temp != nullptr) {

push\_back(temp->data);

temp = temp->pNext;

}

}

List::~List() //деструктор освобождает динамическую память по приципу метода clear,

{ //мы можем делегировать ему эти обязаности

Node\* temp; //но здесь прописана логика удаления

while (size)

{

temp = head;

head = head->pNext;

delete temp;

size--;

}

}

List::Node\* List::push\_back(int data)

{

if (head == nullptr) //создание первого элемента

{

head = new Node(data); //создаем элемент и помещаем его в голову

size++;

return head;

}

else //создание последующих элементов, создаем указатель на первый элемент

{ //и идем по элементам, пока не наткнемся на тот, что указывает на nullptr

Node\* current = this->head;

while (current->pNext != nullptr)

{

current = current->pNext;

}

current->pNext = new Node(data);

++size;

return current->pNext;

}

//количество элементов увеличиваем на 1

}

int& List::operator[](const int index) //мы не можем просто вернуть данные из списка, для этого сначала необходимо перейти в нужную ячейку

{

if (index < 0 || index >= this->size)

{

throw std::exception("Выход за границы списка."); //если индекс выходит за список, то выбрасывается исключение

}

int counter = 0; //счетчик

Node\* current = this->head; //указатель, для того, чтобы итерироваться по нашему списку

while (current != nullptr) //цикл с условием, что мы не вылетим за границы нашего списка

{

if (counter == index)

{

return current->data; //если мы находимся в нужной ячейке то получаем оттуда данные

}

current = current->pNext; //меняем наш временный указатель на следующую ячейку

counter++;

}

}

void List::pop\_front() //удаление из начала

{

Node\* temp = head; //делай временный указатель на объект head

head = head->pNext; //Указательь head переносим на следующую ячейку

delete temp; //удаляем первый объект

size--; //уменьшаем длину на 1

}

void List::clear()

{

//Node<T>\* temp;

while (size)

{

//temp = head; //вызываем pop\_front столько раз, пока size != 0

//head = head->pNext; //логика основывается на pop\_front поэтому мы делегируем ей эти обязанности

//delete temp;

//size--;

pop\_front();

}

this->head = nullptr;

}

void List::push\_front(int data) //добавление элемента в начало

{

head = new Node(data, head); //вместо головного элемента создаем новый элемент, указатель в котором будет на старый головной элемент

size++; //увеличиваем длину на 1

}

void List::insert(int value, List::iterator& it)

{

if (it.current == nullptr) { it.current = push\_back(value); }

else if (it.head == it.current) { push\_front(value); it.current = it.head = this->head; }

else

{

Node\* temp = it.head;

while (temp->pNext != it.current)

temp = temp->pNext;

temp->pNext = new Node(value, temp->pNext);

++size;

}

}

void List::removeAt(List::iterator& iter)

{

if (iter.head == nullptr) { throw std::exception("Итератор не указывает на элемент"); } //проходит проверка указывает ли итератор, на какой-либо элемент списка

if (iter.current == nullptr) { throw std::exception("Итератор не указывает на элемент"); } //если итератор никуда не указывает, то выбрасывается исключение

Node\* temp = iter.head;

while (temp->pNext != iter.current)

temp = temp->pNext;

temp->pNext = iter.current->pNext;

delete iter.current;

iter.current = temp->pNext;

--size;

}

void List::removeAt(int index)

{

if (index == 0)

{

pop\_front();

}

else

{

Node\* previous = head;

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

{

previous = previous->pNext;

}

Node\* toDelete = previous->pNext;

previous->pNext = toDelete->pNext;

delete toDelete;

size--;

}

}

void List::pop\_back()

{

removeAt(size - 1);

}

void List::removeFewAt(int index\_start, int index\_finish) //удаление нескольких элементов

{

if (index\_start + (index\_finish - index\_start) >= size) return; //условие невыхода за границы

for (int i = 0; i < index\_finish - index\_start + 1; i++)

{

removeAt(index\_start);

}

}

List::iterator List::begin()

{

iterator it;

it.head = List::head;

it.current = List::head;

return it;

}

List::iterator List::end()

{

iterator temp;

return temp;

}

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

{

if (this->size != other.size)

{

throw std::exception("Размеры списков разные");

}

List temp;

Node\* l1 = this->head;

Node\* l2 = other.head;

while (l1 != nullptr && l2 != nullptr)

{

temp.push\_back(l1->data \* l2->data);

l1 = l1->pNext;

l2 = l2->pNext;

}

int sz = abs(this->size - other.size);

while (sz)

{

temp.push\_back(0);

--sz;

}

return temp;

}

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

{

this->clear();

Node\* temp = other.head;

while (temp != nullptr)

{

this->push\_back(temp->data);

temp = temp->pNext;

}

return \*this;

}

iterator.cpp:

#include "List.h"

List::iterator::iterator()

{

this->head = nullptr;

this->current = nullptr;

}

List::iterator::~iterator()

{

this->head = nullptr;

this->current = nullptr;

}

List::iterator::iterator(List& lst)

{

this->head = lst.head;

this->current = lst.head;

}

void List::iterator::operator+(int index)

{

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

{

this->current = this->current->pNext;

}

}

void List::iterator::operator++(int)

{ //если итератор пытается выйти за границу списка, то выбрасывается исключение

if (current == nullptr) { throw std::exception("Выход за границы. Невозмонжо получить значение."); }

this->current = this->current->pNext;

}

void List::iterator::operator++()

{ //если итератор пытается выйти за границу списка, то выбрасывается исключение

if (current == nullptr) { throw std::exception("Выход за границы. Невозмонжо получить значение."); }

this->current = this->current->pNext;

}

void List::iterator::reset()

{

this->current = this->head;

}

int& List::iterator::operator\*()

{

return this->current->data;

}

bool List::iterator::operator==(iterator it)

{

return this->current == it.current;

}

bool List::iterator::operator!=(iterator it)

{

return !(\*this == it);

}