Министерство высшего образования и науки Российской Федерации

Федеральное государственное автономное образовательное учреждение высшего образования

ПЕРМСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ (ПНИПУ)

Электротехнический факультет

Дисциплина: основы алгоритмизации и программирования, 2 семестр

**ОТЧЁТ**

Тема: «Обработка исключительных ситуаций»

Выполнил

Студент РИС-22-2б

Зубов Р.А.

Проверил

Доцент кафедры ИТАС

Полякова О. А.

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.

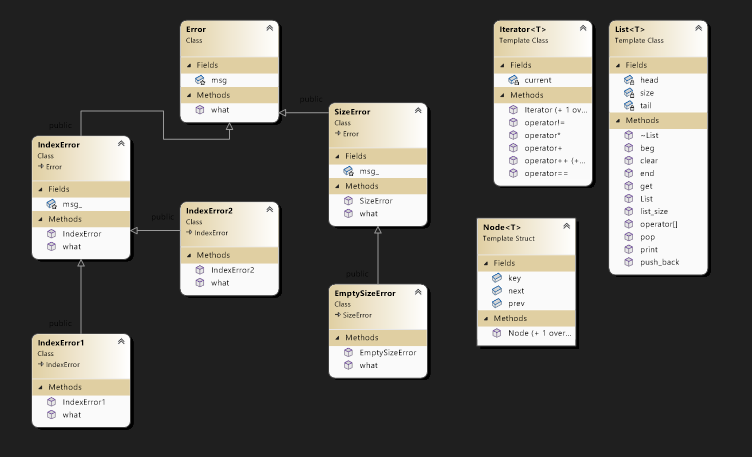
Реализовать операции:

[] – 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).

**UML**

****

**Код программы**

List.h:

#pragma once

#include "Error.h"

#include <cstddef>

#include <iostream>

template <typename T>

struct Node {

T key;

Node<T>\* next = NULL;

Node<T>\* prev = NULL;

Node();

Node(T k, Node<T>\* n, Node<T>\* p);

};

template<typename T>

class Iterator {

template<typename T>

friend class List;

private:

Node<T>\* current = NULL;

public:

Iterator() { current = NULL; }

Iterator(Node<T>\* node) : current(node) {}

bool operator==(const Iterator<T>& other) const { //сравнивают текущий узел

return current == other.current; // и узел другого итератора

} //

bool operator!=(const Iterator<T>& other) const { // на равенство и неравенство.

return current != other.current;

}

T& operator\*() const { // операция разыменования итератора

return current->key;

}

Iterator& operator++() { //++i

current = current->next;

return \*this;

}

Iterator& operator--() { //--i

current = current->prev;

return \*this;

}

Iterator operator++(int) { // i++

Iterator<T> old = \*this;

current = current->next;

return old;

}

Iterator operator+(int n) const { // переход вправо к n элементу

Iterator<T> it(\*this);

while (n > 0 && it.current != NULL) {

it.current = it.current->next;

n--;

}

return it;

}

};

template<typename T>

class List {

Node<T>\* head;

Node<T>\* tail;

int size;

public:

List() {

head = NULL;

tail = NULL;

size = 0;

}

~List() {

clear();

}

T& get(int index);

T& operator[](int index) {

return get(index); // возвращаем ссылку на элемент по индексу

}

// Метод добавления элемента в конец списка

void push\_back(T k);

void pop(T key);

int list\_size() {

return size;

}

void clear();

void print() const;

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

{

list.print();

return out;

}

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

{

int k;

in >> k;

list.push\_back(k);

return in;

}

Iterator<T> beg() {

return Iterator<T>(head); // возвращаем итератор на первый элемент

}

Iterator<T> end() {

return Iterator<T>(tail->next); // возвращаем итератор на последний элемент

}

friend List<T> operator\*(List<T>& left, List<T>& right) {

List<T> result; // создаем новый список

Iterator<T> it1 = left.beg(); // итератор по первому списку

Iterator<T> it2 = right.beg(); // итератор по второму списку

while (it1 != left.end() && it2 != right.end()) {

T res = (\*it1) \* (\*it2);

result.push\_back(res); // добавляем произведение элементов в новый список

++it1; // переходим к следующему элементу первого списка

++it2; // переходим к следующему элементу второго списка

}

return result; // возвращаем новый список

}

};

template<typename T>

Node<T>::Node(T k, Node<T>\* n, Node<T>\* p)

{

key = k;

next = n;

prev = p;

}

template<typename T>

T& List<T>::get(int index)

{

Node<T>\* current = head; // начинаем с головы списка

int i = 0;

while (i < index && current->next != NULL) {

current = current->next; // переходим к следующему узлу

}

return current->key; // возвращаем ссылку на данные узла

}

template<typename T>

void List<T>::push\_back(T k)

{

Node<T>\* newNode = new Node<T>(k, NULL, tail); // создаем новый узел с ключом key, указателем на следующий узел равным NULL и указателем на предыдущий узел равным tail

if (tail != NULL) {

tail->next = newNode; // если tail не равен NULL, то устанавливаем указатель на следующий узел у tail на новый узел

}

tail = newNode; // устанавливаем tail на новый узел

if (head == NULL) {

head = newNode; // если список был пустой, то устанавливаем head на новый узел

}

size++;

}

template<typename T>

void List<T>::pop(T key) {

Node<T>\* p = head;

while (p != NULL && p->key != key) { // Проходим по списку до нужного ключа

p = p->next;

}

if (p != NULL) { // Если нашли узел по позиции

if (p->prev != NULL) { // Если узел не является головным

p->prev->next = p->next;

}

else { // Если узел является головным

head = p->next;

}

if (p->next != NULL) { // Если узел не является хвостовым

p->next->prev = p->prev;

}

else { // Если узел является хвостовым

tail = p->prev;

}

delete p; // Удаляем текущий узел из памяти

size--;

}

}

template<typename T>

void List<T>::clear()

{

Node<T>\* p = head;

while (p != NULL) {

Node<T>\* q = p;

p = p->next;

delete q; // Удаляем текущий узел из памяти

}

head = NULL;

tail = NULL;

size = 0;

}

template<typename T>

void List<T>::print() const

{

Node<T>\* node = head;

while (node != NULL) {

std::cout << node->key << "->";

node = node->next;

}

std::cout << std::endl;

}

Error.h:

#pragma once

#include <string>

#include <iostream>

class Error

{

public:

virtual void what() {};

protected:

std::string msg;

};

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"; }

virtual void what() { std::cout << msg; }

};

class EmptySizeError :public SizeError

{

public:

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

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

};

class IndexError1 :public IndexError

{

public:

IndexError1() { IndexError(); msg\_ = "Index < 0\n"; }

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

};

class IndexError2 :public IndexError

{

public:

IndexError2() { IndexError(); msg\_ = "Index > size\n"; }

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

};

**Контрольные вопросы**

1. Что представляет собой исключение в С++?

Исключение в C++ представляет собой событие, которое возникает во время выполнения программы и приводит к ее некорректному поведению.

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

Исключение позволяет разделить вычислительный процесс на две части: основную и обработку ошибок. Это позволяет улучшить структуру программы, сделать ее более надежной и легче поддерживаемой.

3. Какой оператор используется для генерации исключительной ситуации?

Для генерации исключительной ситуации используется оператор throw.

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

Контролируемый блок представляет собой участок кода, который может генерировать исключения. Он нужен для того, чтобы обеспечить правильную обработку исключений в программе.

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

Секция-ловушка представляет собой участок кода, который обрабатывает исключения. Она нужна для того, чтобы обеспечить корректную обработку исключений и предотвратить прерывание работы программы.

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

Спецификация исключения может иметь форму throw(), throw(type1), throw(type1, type2), где type1 и type2 - типы исключений. Форма throw() означает, что функция не генерирует исключений, форма throw(type1) - что функция может генерировать исключение типа type1, форма throw(type1, type2) - что функция может генерировать исключения типов type1 и type2. Эти формы используются для указания того, какие исключения может генерировать функция.

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

Стандартный класс exception можно использовать для создания собственной иерархии исключений.

8. Каким образом можно создать собственную иерархию исключений?

Для создания собственной иерархии исключений необходимо создать новый класс, который будет наследоваться от класса exception.

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

Функция f1() может прождать исключения типа int и double.

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

Функция f1() не может прождать никаких исключений.

11. В какой части программы может генерироваться исключение?

Исключение может генерироваться в любой части программы, включая функции, методы классов, конструкторы и деструкторы.

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.

1. double area(double a, double b, double c){

double p = (a + b + c) / 2.0;

double S = sqrt(p \* (p - a) \* (p - b) \* (p - c));

return S;

}

2)

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

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

throw std::domain\_error("Invalid triangle sides");

}

double p = (a + b + c) / 2.0;

double S = sqrt(p \* (p - a) \* (p - b) \* (p - c));

return S;

}

3) double area(double a, double b, double c){

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) / 2.0;

double S = sqrt(p \* (p - a) \* (p - b) \* (p - c));

return S;

}

4) #include <cmath>

#include <stdexcept>

class TriangleError : public std::runtime\_error {

public:

TriangleError(const char\* message)

: std::runtime\_error(message) {}

};

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

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

throw TriangleError("Invalid triangle sides");

}

double p = (a + b + c) / 2.0;

double S = sqrt(p \* (p - a) \* (p - b) \* (p - c));

return S;

}