ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ

ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«ПЕРМСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ

ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»

Лабораторная работа №9

Обработка исключительных ситуаций

Вариант №15

Выполнил студент группы РИС-23-3Б

Блинов А. Е.

Проверила доцент кафедры ИТАС

О. А. Полякова

2024 г.

**Постановка задачи**

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

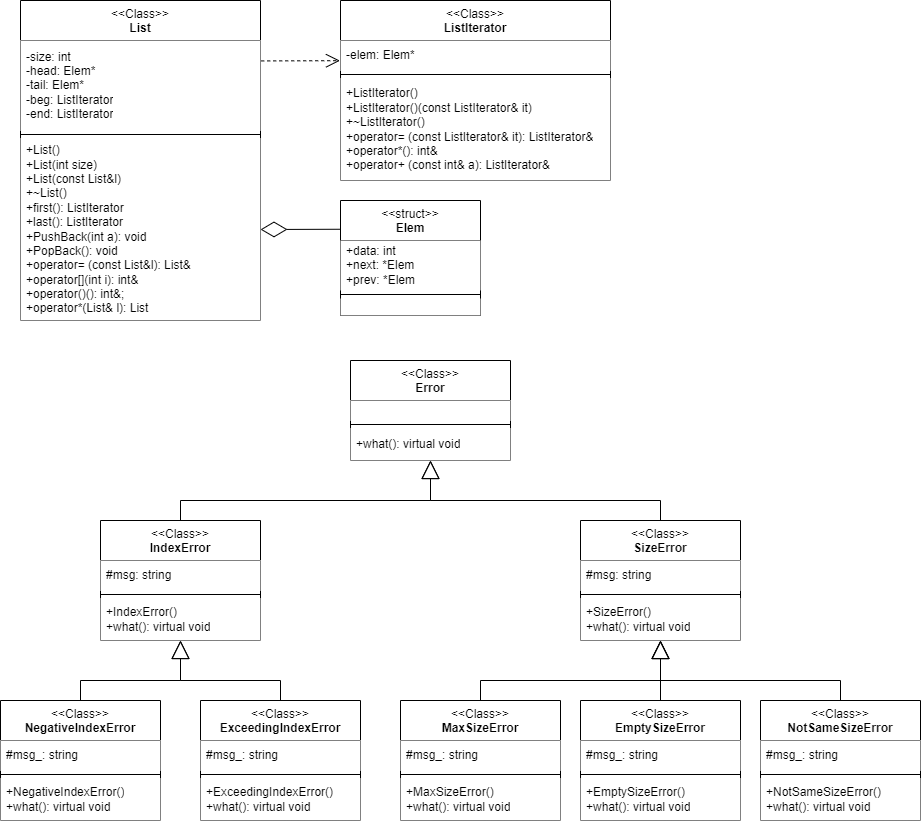
2. Определить исключительные ситуации.

3. Предусмотреть генерацию исключительных ситуаций.

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.

**UML диаграмма**

****

**Код на языке C++**

#include <iostream>

using namespace std;

const int MAX\_SIZE = 20;

struct Node {

int data;

Node\* next\_node;

Node\* prev\_node;

};

class Error {

public:

virtual void what() {};

};

class IndexError : public Error {

protected:

string message;

public:

IndexError() { message = "Ошибка индекса.\n"; }

virtual void what() { cout << message; }

};

class SizeError :public Error {

protected:

string message;

public:

SizeError() { message = "Ошибка размера.\n"; }

virtual void what() { cout << message; }

};

class maxSizeError : public SizeError {

protected:

string newMessagae;

public:

maxSizeError() { SizeError(); newMessagae = "Размер списка больше, чем максимальный размер.\n"; }

virtual void what() { cout << message << newMessagae; }

};

class emptySizeError : public SizeError {

protected:

string newMessage;

public:

emptySizeError() { SizeError(); newMessage = "Список пуст.\n"; }

virtual void what() { cout << message << newMessage; }

};

class notSameSize : public SizeError {

protected:

string newMessage;

public:

notSameSize() { SizeError(); newMessage = "Списки разной длины.\n"; }

virtual void what() { cout << message << newMessage; }

};

class negativeIndex : public IndexError {

protected:

string someMessage;

public:

negativeIndex() { IndexError(); someMessage = "Индекс меньше нуля.\n"; }

virtual void what() { cout << message << someMessage; }

};

class ExceedingIndexError : public IndexError {

protected:

string someMessage;

public:

ExceedingIndexError() { IndexError(); someMessage = "Индекс превышает размер списка.\n"; }

virtual void what() { cout << message << someMessage; }

};

class Iterator {

private:

friend class List;

Node\* elem;

public:

Iterator() { this->elem = nullptr; }

Iterator(const Iterator& iterator) { this->elem = iterator.elem; }

Iterator& operator=(const Iterator& iter) {

this->elem = iter.elem;

return \*this;

}

bool operator==(const Iterator& iter) { return this->elem == iter.elem; }

bool operator!=(const Iterator& iter) { return this->elem != iter.elem; }

Iterator& operator++(int) {

this->elem = this->elem->next\_node;

return \*this;

}

Iterator& operator--() {

this->elem = this->elem->prev\_node;

return \*this;

}

Iterator operator+(const int number) const {

Iterator temp(\*this);

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

temp.elem = temp.elem->next\_node;

}

return temp;

}

Iterator operator-(const int number) const {

Iterator temp(\*this);

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

temp.elem = temp.elem->prev\_node;

}

return temp;

}

int& operator\*() { return this->elem->data; }

};

class List {

private:

int size;

Node\* head;

Node\* tail;

Iterator begin;

Iterator end;

public:

List(int size, int data);

List(const List& list);

List(int size);

List() { size = 0; head = nullptr; Node\* tail = nullptr; }

~List();

Iterator first();

Iterator last();

void push\_back(int data);

void push\_front(int data);

int pop\_back();

int pop\_front();

bool is\_empty();

int operator () ();

List& operator = (const List& list);

int& operator [] (int index);

List operator \* (List& list);

friend ostream& operator << (ostream& out, const List& list);

friend istream& operator >> (istream& in, const List& list);

};

List::List(int size, int data) {

this->size = size;

if (size > 0) {

Node\* node = new Node;

node->data = data;

this->head = node;

this->tail = node;

for (int i = 1; i < size; i++) {

Node\* newNode = new Node;

newNode->data = data;

tail->next\_node = newNode;

newNode->prev\_node = tail;

tail = newNode;

}

tail->next\_node = nullptr;

}

else {

this->head = nullptr;

this->tail = nullptr;

}

this->begin.elem = this->head;

this->end.elem = this->tail;

}

List::List(int size) {

if (size > MAX\_SIZE) throw maxSizeError();

this->size = size;

if (size > 0) {

Node\* node = new Node;

this->head = node;

this->tail = node;

for (int i = 1; i < size; i++) {

Node\* newNode = new Node;

tail->next\_node = newNode;

newNode->prev\_node = tail;

tail = newNode;

}

tail->next\_node = nullptr;

}

else {

this->head = nullptr;

this->tail = nullptr;

}

this->begin.elem = this->head;

this->end.elem = this->tail;

}

List::List(const List& list) {

this->head = nullptr;

this->tail = nullptr;

this->size = 0;

Node\* current\_node = list.head;

while (current\_node != nullptr) {

push\_back(current\_node->data);

current\_node = current\_node->next\_node;

}

this->begin = list.begin;

this->end = list.end;

}

List::~List() {

Node\* current\_node = head;

while (current\_node != nullptr) {

Node\* next\_node = current\_node->next\_node;

delete current\_node;

current\_node = next\_node;

}

head = nullptr;

}

void List::push\_back(int data) {

Node\* new\_node = new Node;

new\_node->data = data;

new\_node->next\_node = nullptr;

if (this->head == nullptr) {

this->head = new\_node;

this->tail = new\_node;

this->begin.elem = this->head;

this->end.elem = this->tail;

}

else {

tail->next\_node = new\_node;

new\_node->prev\_node = tail;

tail = new\_node;

}

this->size++;

}

void List::push\_front(int data) {

Node\* new\_node = new Node;

new\_node->data = data;

if (this->head == nullptr) {

this->head = new\_node;

this->tail = new\_node;

this->size++;

this->begin.elem = this->head;

this->end.elem = this->tail;

}

else {

head->prev\_node = new\_node;

new\_node->next\_node = head;

head = new\_node;

this->size++;

this->begin.elem = this->head;

}

}

Iterator List::first() { return this->begin; }

Iterator List::last() { return this->end; }

bool List::is\_empty() { return this->size == 0; }

int List::pop\_back() {

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

int temp;

if (this->tail != nullptr) {

Node\* current\_node = this->tail;

tail = current\_node->prev\_node;

temp = current\_node->data;

tail->next\_node = nullptr;

this->size--;

this->end.elem = tail;

}

return temp;

}

int List::pop\_front() {

int temp;

if (this->head != nullptr) {

Node\* current\_node = this->head;

head = current\_node->next\_node;

temp = current\_node->data;

head->prev\_node = nullptr;

this->size--;

this->begin.elem = head;

}

return temp;

}

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

cout << "Оператор присваивания\n";

if (this == &list) {

return \*this;

}

while (head != nullptr) {

Node\* temp = head;

head = head->next\_node;

delete temp;

}

size = 0;

Node\* current\_node = list.head;

while (current\_node != nullptr) {

push\_back(current\_node->data);

current\_node = current\_node->next\_node;

}

this->begin = list.begin;

this->end = list.end;

return \*this;

}

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

if (index < 0) throw negativeIndex();

if (this->size == 0) throw emptySizeError();

if (index > this->size) throw ExceedingIndexError();

if (index < this->size && index >= 0) {

Node\* current\_node = this->head;

for (int i = 0; i != index; i++) {

current\_node = current\_node->next\_node;

}

return current\_node->data;

}

else {

cerr << "\nИндекс вышел за границы списка!\n";

exit(0);

}

}

int List:: operator () () { return this->size; }

List List::operator\* (List& list) {

if (this->size != list.size) throw notSameSize();

int temp\_size;

if (this->size > list.size) {

temp\_size = list.size;

}

else {

temp\_size = this->size;

}

List temp(temp\_size, 0);

for (int i = 0; i < temp\_size; i++) {

temp[i] = (\*this)[i] \* list[i];

}

return temp;

}

ostream& operator << (ostream& out, const List& list) {

out << "\nВывод элементов списка\n";

Node\* current\_node = list.head;

while (current\_node != nullptr)

{

out << current\_node->data << " ";

current\_node = current\_node->next\_node;

}

return out;

}

istream& operator >> (istream& in, const List& list) {

cout << "\nВвод элементов списка\n";

Node\* current\_node = list.head;

while (current\_node != nullptr) {

in >> current\_node->data;

current\_node = current\_node->next\_node;

}

return in;

}

int main() {

system("chcp 1251 > Null");

try {

int listSize, value, index, amountToDelete;

List list1;

Iterator iter;

cout << "Введите кол-во элементов списка: "; cin >> listSize;

cout << endl;

if (listSize < 1) throw "Размер списка должен быть больше, чем ноль.\n";

if (listSize > MAX\_SIZE) throw maxSizeError();

cout << "Введите элементы списка\n";

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

cout << i + 1 << " элемент: "; cin >> value;

list1.push\_back(value);

}

iter = list1.first();

cout << "\nВведите индекс элемента для поиска: "; cin >> index;

if (index < 0) throw negativeIndex();

if (index > MAX\_SIZE) throw ExceedingIndexError();

cout << "\nЭлемент списка с индексом " << index << ": " << list1[index] << endl;

cout << "\nВведите кол-во элементов для удаления с начала: "; cin >> amountToDelete;

iter + amountToDelete;

cout << \*iter << endl << endl;

}

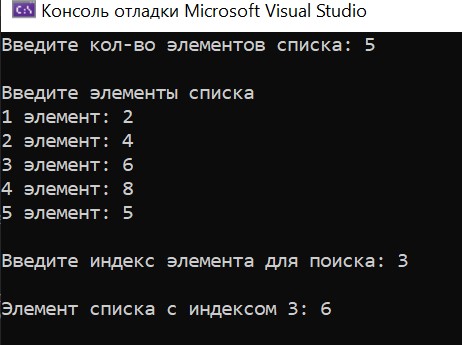
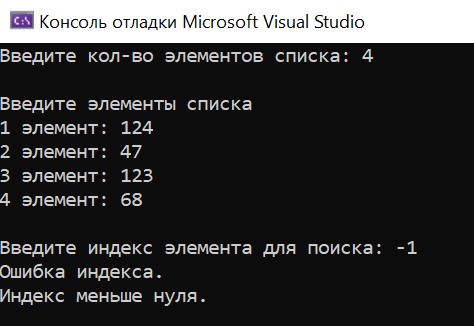
catch (Error& e) {

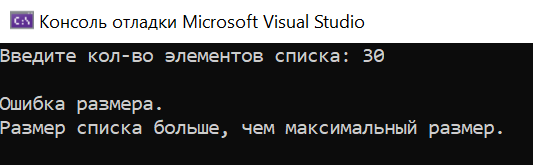
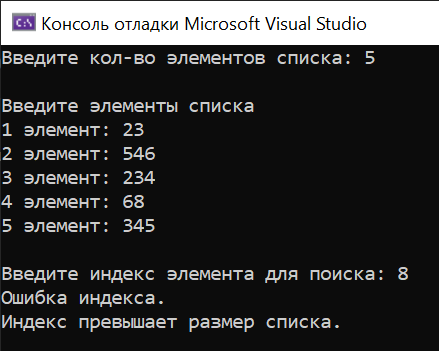
e.what();

}

}

**Результаты работы программы**

**Выводы**

Программа работает успешно

GitHub:

**Контрольные вопросы**

1. Что представляет собой исключение в С++?

Исключение (исключительная ситуация) – это непредвиденное событие, делающее невозможным дальнейшее выполнение программы по базовому алгоритму

2. На какие части исключения позволяют разделить вычислительный процесс? Обнаружение исключения

1. Обнаружение исключения
2. Передача управления обработчику исключений
3. Выполнение обработчика

3. Какой оператор используется для генерации исключительной ситуации?

throw() - генерация объекта или передача значения

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

Блок tryиспользуется для проверки возникновения исключения, соответственно в блок try помещается та часть кода, в которой возможно возникновение исключения; try – определяет блок кода, в котором может произойти исключение;

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

**Секция-ловушка** представляет собой блок кода, который обрабатывает исключения. Она используется для перехвата и обработки исключений, которые могут возникнуть внутри блока. Секция-ловушка позволяет программе продолжить выполнение, даже если произошла ошибка.

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

Спецификация исключения в секции-ловушке может иметь несколько форм:

* throw() — это пустая спецификация, которая означает, что функция не генерирует исключения. Это используется, когда вы хотите явно указать, что функция не может вызвать исключение.
* throw(T1, T2, ...) — здесь T1, T2 и так далее представляют типы исключений, которые функция может генерировать

Например, throw(int, double) означает, что функция может генерировать исключения типа int или double.

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

Чаще используют class Error; также **Стандартный класс** для создания собственной иерархии исключений в C++ — это std::exception.

8. Каким образом можно создать собственную иерархию исключений?

Для **создания собственной иерархии исключений нужно**:

* Определите свой пользовательский класс исключения
* Переопределите метод what(), чтобы предоставить описание ошибки.
* Генерировать экземпляры своего пользовательского класса исключения в нужных местах вашей программы.

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

функция f1() может генерировать исключения типа int или double.

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

Если спецификация исключений имеет вид void f1() throw(), то функция f1() не может генерировать никаких исключений.

11. В какой части программы может генерироваться исключение?

В любой части программы