Министерство образования и науки Российской Федерации

Федеральное государственное образовательное автономное учреждение высшего образования

"Пермский национальный исследовательский политехнический университет"

ОТЧЕТ

ПО ЛАБОРАТОРНОЙ РАБОТЕ №7

Дисциплина: Основы алгоритмизации и программирования

Тема: Шаблоны классов.

|  |  |
| --- | --- |
| 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. Определить шаблон класса контейнера.
2. Реализовать конструкторы, деструктор, операции ввода-вывода, операцию присваивания.
3. Перегрузить операции, указанные в варианте.
4. Реализовать пользовательский класс.
5. Перегрузить для пользовательского класса операции ввода вывода.
6. Перегрузить операции необходимые для выполнения операций контейнерного класса.
7. Инстанцировать шаблон для пользовательского класса.
8. Реализовать итератор. Реализовать с его помощью операции последовательного доступа.
9. Написать тестирующую программу, иллюстрирующую выполнение операций для контейнера, содержащего элементы пользовательского класса.

Вариант 15: Класс-контейнер список с ключевыми значениями типа int.

Реализовать операции:

[] – доступ по индексу;

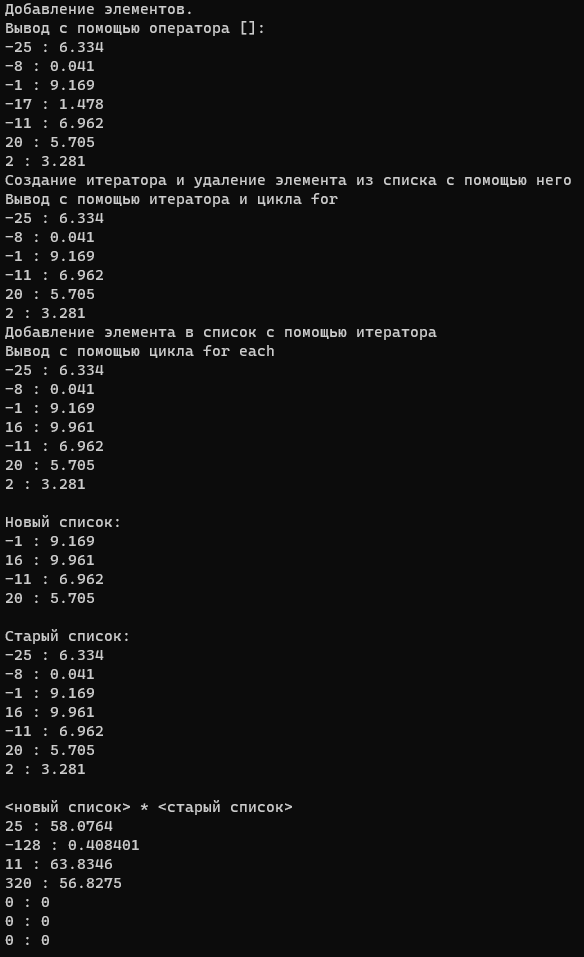
Int() – определение размера списка;  
\* список – умножение элементов списков a[i] \* b[i];

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

Пользовательский класс Pair (пара чисел). Пара должна быть представлена двумя полями: типа int для первого числа и типа double для второго. Первое число при выводе на экран должно быть отделено от второго числа двоеточием.

**Анализ задачи**

Необходимо реализовать шаблонный класс односвязный список и итератор. Для этого необходимо: реализовать вложенный класс Node – который будет являться минимальной ячейкой памяти списка, в Node будет хранится data шаблонного класса и указатель на следующий элемент. Далее необходимо вести подсчет элементов в списке, для этого необходимо создать поле size и в данное поле подсчитывать изменения, вносимые в список. Также для списка необходимо перегрузить операторы: [] – для доступа по индексу и «\*» для умножения одного списка на другой. Для быстродействия данной структуры данных необходимо реализовать итератор, который также будет являться вложенным классом, так как данный итератор никуда больше не сможет подойти. Он будет шаблонным, так как вложен в шаблонный класс список и также должен работать с любым типом данных. Итератор должен указывать на какой-либо элемент, следовательно поле его класса – минимальная ячейка памяти списка – указатель на node. Для итератора необходимо реализовать операторы: +n, префиксный и постфиксный инкремент, оператор разыменования, для того, чтобы можно было получать данные, оператор == и != для возможности перебирать данные с помощью цикла и итератора. А также необходимо для списка реализовать методы begin и end, которые будут возвращать итераторы, которые указывают на первый и последний элемент соответственно. Пользовательский класс Pair возьмем из 3 лабораторной работы, только для полноценной работы нашего списка перегрузим у него оператор «\*» для того, чтобы выполнять умножение элемента на элемент.

****Тестирование программы**

*Рис. 1 – Тестирование программы.*

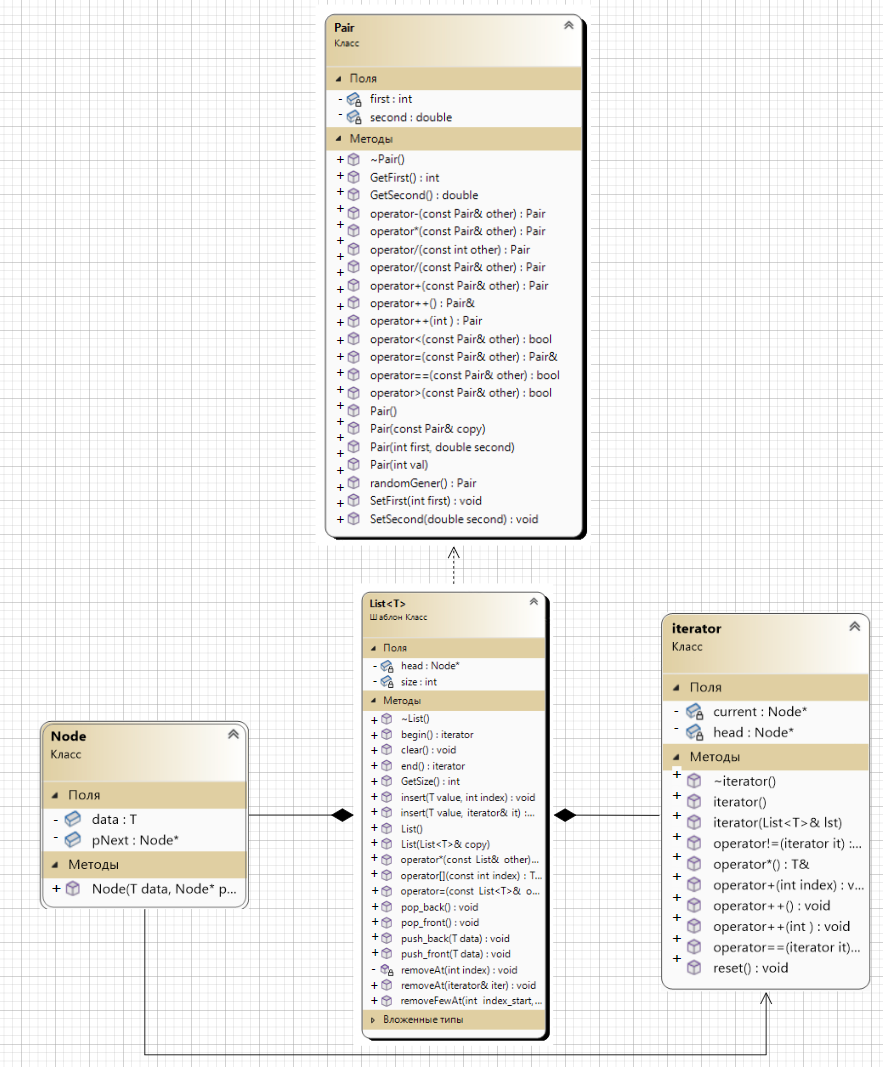
**Заключение**

Была разработана программа, в которой реализована динамическая структура данных – шаблонный односвязный список. В данном односвязном списке реализовано 2 вложенных класса для его нужд: класс node – минимальная ячейка данных и класс iterator – для более быстрого и прямого доступа к данным. Также для односвязного списка и итератора были реализованы методы, благодаря которым список может работать с range-based циклами.

Также из 3 лабораторной работы был взят собственный тип данных Pair и дополнен оператором «\*».

**Приложения**

Приложение UML-диаграмма



Приложение Б – код программы

Main.cpp:

#include <iostream>

#include <Windows.h>

#include <fstream>

#include "List.h"

#include "Pair.h"

using namespace std;

int main()

{

system("chcp 1251 >> null");

//system("color F0");

List<Pair>lst;

Pair p;

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

lst.push\_back(p.randomGener());

lst.push\_front(p.randomGener());

lst.push\_back(p.randomGener());

lst.push\_back(p.randomGener());

lst.push\_back(p.randomGener());

lst.push\_back(p.randomGener());

lst.push\_back(p.randomGener());

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

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

{

cout << lst[i] << endl;

}

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

auto i = lst.begin();

i + 3;

lst.removeAt(i);

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(p.randomGener(), i);

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

for (auto& it : lst)

{

cout << it << endl;

}

List<Pair> list2;

list2 = lst;

list2.pop\_front();

list2.pop\_back();

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;

list2 = (list2 \* lst);

for (auto& it : list2)

{

cout << it << endl;

}

return 0;

}

list.h:

#pragma once

template <typename T>

class List

{

private:

void removeAt(int index); //удаление элемента по указаному индексу

//минимальная ячейка памяти

class Node //вложенный класс - ячейка памяти

{

public:

Node(T data = T(), 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

T data; //хранит наши данные

};

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

int size; //количество элементов в списке.

public:

class iterator //итератор для списка

{

private:

friend class List;

Node\* head; //указатель на node

Node\* current;

public:

iterator(); //конструкторы и деструктор

~iterator();

iterator(List<T>& lst);

void reset();

void operator +(int index); //операторы для взаимодействия

void operator ++(int);

void operator ++();

T& operator \*();

bool operator ==(iterator it); //операторы сравнения итераторов

bool operator !=(iterator it);

};

public:

List(); //Конструктор

~List(); //Деструктор

List(List<T>& copy);

void push\_back(T 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

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

List<T>& operator=(const List<T>& other);

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(T data); //добавляет элемент в начало списка

void removeAt(List::iterator& iter);

void insert(T value, int index); //добавление элемента по индексу

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

iterator begin();

iterator end();

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

void insert(T value, List::iterator& it); //добавление элементов по итератору

List<T> operator\*(const List& other); //оператор умнжения списка на список

};

template<typename T>

List<T> List<T>::operator\*(const List<T>& other)

{

List<T> 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;

}

template<typename T>

typename List<T>::iterator List<T>::begin()

{

iterator it;

it.current = this->head;

it.head = this->head;

return it;

}

template<typename T>

typename List<T>::iterator List<T>::end()

{

iterator it(\*this);

it.current = nullptr;

return iterator();

}

template<typename T>

void List<T>::insert(T value, List::iterator& it)

{

if (it.current == nullptr) { return; }

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;

}

}

template <typename T>

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

{

size = 0;

head = nullptr;

}

template <typename T>

List<T>::~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--;

}

}

template <typename T>

List<T>::List(List<T>& copy)

{

this->size = 0;

this->head = nullptr;

Node\* temp = copy.head;

while (temp != nullptr) {

push\_back(temp->data);

temp = temp->pNext;

}

}

template<typename T>

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

{

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

{

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

}

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++; //количество элементов увеличиваем на 1

}

template<typename T>

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

{

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++;

}

}

template<typename T>

List<T>& List<T>::operator=(const List<T>& other)

{

this->clear();

Node\* temp = other.head;

while (temp != nullptr)

{

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

temp = temp->pNext;

}

return \*this;

}

template<typename T>

void List<T>::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

}

template<typename T>

void List<T>::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();

}

}

template<typename T>

void List<T>::push\_front(T 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

}

template<typename T>

void List<T>::insert(T value, int index) //вставка элемента по индексу

{

if (index > size) return; //если индекс больше size, то вставить невозможно

if (index == 0) //если индекс 0 то вызываем вставку в начало списка

{

push\_front(value);

}

else if (index == size) //если индекс равен длине, вызываем вставку в конец списка

{

push\_back(value);

}

else

{

Node\* previous = this->head; //создаем указатель на головной элемент для поиска нужного элемента

for (int i = 0; i < index - 1; i++) //поиск предыдущего элемента, перед индексом, чтобы поменять в нем указатель

{

previous = previous->pNext;

}

previous->pNext = new Node(value, previous->pNext); //создание нового элемента с переданным значением и установка указателя на элемент с переданным индексом

size++; //Предыдущая строка "вклинивает" новый элемент между старыми

}

}

template<typename T>

void List<T>::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--;

}

}

template<typename T>

void List<T>::pop\_back()

{

removeAt(size - 1);

}

template<typename T>

void List<T>::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);

}

}

template<typename T>

void List<T>::removeAt(List::iterator& iter)

{

if (iter.current == nullptr) return;

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;

}

template<typename T>

List<T>::iterator::iterator()

{

this->head = nullptr;

this->current = nullptr;

}

template<typename T>

List<T>::iterator::~iterator()

{

this->head = nullptr;

this->current = nullptr;

}

template<typename T>

List<T>::iterator::iterator(List<T>& lst)

{

this->head = lst.head;

this->current = lst.head;

}

template<typename T>

void List<T>::iterator::operator+(int index)

{

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

{

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

}

}

template<typename T>

void List<T>::iterator::operator++(int)

{

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

}

template<typename T>

void List<T>::iterator::operator++()

{

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

}

template<typename T>

void List<T>::iterator::reset()

{

this->current = this->head;

}

template<typename T>

T& List<T>::iterator::operator\*()

{

return this->current->data;

}

template<typename T>

bool List<T>::iterator::operator==(iterator it)

{

if (this->current == it.current) return true;

return false;

}

template<typename T>

bool List<T>::iterator::operator!=(iterator it)

{

return !(\*this == it);

}

Pair.h:

#pragma once

#include <iostream>

using namespace std;

class Pair

{

public:

Pair(); //конструкторы и деструктор класса pair

Pair(int first, double second);

Pair(int val);

Pair(const Pair& copy);

~Pair() {}

int GetFirst() { return this->first; } //геттеры и сеттеры

double GetSecond() { return this->second; }

void SetFirst(int first) { this->first = first; }

void SetSecond(double second) { this->second = second; }

Pair& operator =(const Pair& other); //оператор присваивания

Pair operator +(const Pair& other); //оператор сложения с типом данных Pair

Pair operator \*(const Pair& other); //умножение pair на pair

Pair& operator ++(); //остальные математические операции

Pair operator ++(int);

Pair operator -(const Pair& other); //операции сравнения

bool operator <(const Pair& other) const;

bool operator >(const Pair& other);

bool operator ==(const Pair& other);

friend const Pair operator +(const Pair& left, const int& right); //Математические операции

friend const Pair operator +(const int& left, const Pair& right);

friend const Pair operator +(const Pair& left, const double& right);

friend const Pair operator +(const double& left, const Pair& right);

Pair operator /(const Pair& other); //операция деления

Pair operator /(const int other);

friend ostream& operator <<(ostream& os,const Pair& str); //операция ввода/вывода

friend istream& operator >> (istream& is, Pair& str);

Pair randomGener(); //метод рандомной генерации

private:

int first; //поля класса

double second;

};

Pair.cpp:

#include "Pair.h"

ostream& operator <<(ostream& os, const Pair& val)

{

os << val.first << " : " << val.second;

return os;

}

istream& operator>>(istream& is, Pair& val)

{

is >> val.first >> val.second;

return is;

}

Pair::Pair()

{

first = 0;

second = 0;

}

Pair::Pair(const Pair& copy)

{

this->first = copy.first;

this->second = copy.second;

}

Pair::Pair(int first, double second)

{

this->first = first;

this->second = second;

}

Pair::Pair(int val)

{

this->first = val;

this->second = val;

}

Pair Pair::randomGener()

{

return Pair(rand() % 50 - 25, (double((rand() % 10000)) / 1000));

}

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

{

this->first = other.first;

this->second = other.second;

return \*this;

}

Pair Pair::operator+(const Pair& other)

{

Pair temp(this->first + other.first, this->second + other.second);

return temp;

}

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

{

return Pair(this->first \* other.first, this->second \* other.second);

}

Pair& Pair::operator++()

{

++this->first;

++this->second;

return \*this;

}

Pair Pair::operator++(int)

{

Pair temp(\*this);

++this->first;

++this->second;

return temp;

}

Pair Pair::operator-(const Pair& other)

{

return Pair(this->first - other.first, this->second - other.second);

}

const Pair operator+(const Pair& left, const int& right)

{

return Pair(left.first + right, left.second);

}

const Pair operator+(const int& left, const Pair& right)

{

return right + left;

}

const Pair operator+(const Pair& left, const double& right)

{

return Pair(left.first, left.second + right);

}

const Pair operator+(const double& left, const Pair& right)

{

return right + left;

}

Pair Pair::operator/(const Pair& other)

{

return Pair(this->first/other.first, this->second/other.second);

}

Pair Pair::operator/(const int other)

{

return Pair(this->first / other, this->second / other);

}

bool Pair::operator<(const Pair& other) const

{

double thiss = this->first + this->second, otherr = other.first + other.second;

if (thiss < otherr) return true;

else return false;

}

bool Pair::operator>(const Pair& other)

{

return !(\*this < other);

}

bool Pair::operator==(const Pair& other)

{

if (this->first == other.first && this->second == other.second)

return true;

else

return false;

}