ПНИПУ

Федеральное государственное автономное образовательное учреждение высшего образования «Пермский национальный исследовательский политехнический университет»

Лабораторная работа №7.

Шаблоны классов.

Выполнил студент группы РИС-23-3Б

Буковский Денис Владимирович

Проверила доцент кафедры ИТАС О.А. Полякова

2024

1. Постановка задачи
2. Определить шаблон класса-контейнера (см. лабораторную работу №6).
3. Реализовать конструкторы, деструктор, операции ввода-вывода, операцию присваивания.
4. Перегрузить операции, указанные в варианте.
5. Инстанцировать шаблон для стандартных типов данных (int, float, double).
6. Написать тестирующую программу, иллюстрирующую выполнение операций для контейнера, содержащего элементы стандартных типов данных.
7. Реализовать пользовательский класс (см. лабораторную работу №3).
8. Перегрузить для пользовательского класса операции ввода-вывода.
9. Перегрузить операции необходимые для выполнения операций контейнерного класса.
10. Инстанцировать шаблон для пользовательского класса.
11. Написать тестирующую программу, иллюстрирующую выполнение операций для контейнера, содержащего элементы пользовательского класса.

Класс- контейнер СПИСОК с ключевыми значениями типа int.

Реализовать операции:

[] - доступа по индексу;

int() — определение размера списка;

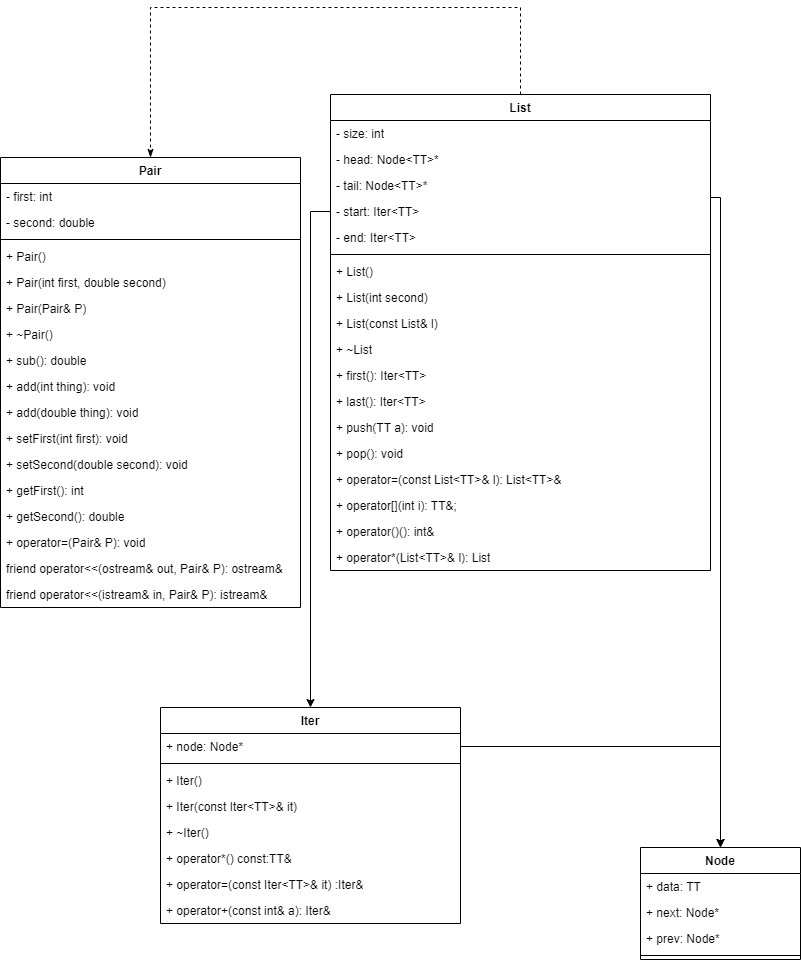
\* вектор — умножение элементов списков a[i]\*b[i];

Пользовательский класс Pair (пара чисел). Пара должна быть представлено 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.

1. Анализ задачи

Создадим классы List и Pair. Напишем реализацию всех стандартных методов. Протестируем программу в функции main().

1. Блок-схема



1. Код

Pair.h

#pragma once

#include <iostream>

using namespace std;

class Pair

{

private:

int first;

double second;

public:

Pair();

Pair(int first, double second);

Pair(const Pair& P);

~Pair();

void set\_first(int first);

void set\_second(double second);

int get\_first();

double get\_second();

void print();

Pair operator+(const int& first) const;

Pair operator+(const double& second) const;

Pair operator-(const Pair& P) const;

friend Pair operator+(const int& first, const Pair& P);

friend Pair operator+(const double& second, const Pair& P);

friend ostream& operator<<(ostream& out, const Pair& P);

friend istream& operator>>(istream& in, Pair& P);

};

Pair.cpp

#include "Pair.h"

Pair::Pair() {

first = 0;

second = 0;

}

Pair::Pair(int first = 0, double second = 0) {

this->first = first;

this->second = second;

}

Pair::Pair(const Pair& P) {

this->first = P.first;

this->second = P.second;

}

Pair::~Pair() {

}

int Pair::get\_first() { return this->first; }

void Pair::set\_first(int first) { this->first = first; }

double Pair::get\_second() { return this->second; }

void Pair::set\_second(double second) { this->second = second; }

void Pair::print() {

cout << this->first << " : " << this->second << endl;

}

Pair Pair::operator+(const int& first) const {

return Pair(this->first + first, this->second);

}

Pair Pair::operator+(const double& second) const {

return Pair(this->first, this->second + second);

}

Pair Pair::operator-(const Pair& P) const {

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

}

Pair operator+(const int& first, const Pair& P) {

return P + first;

}

Pair operator+(const double& second, const Pair& P) {

return P + second;

}

ostream& operator<<(ostream& out, const Pair& P) {

out << " " << P.first << " : " << P.second << endl;

return out;

}

istream& operator>>(istream& in, Pair& P) {

cout << "Enter int value: ";

in >> P.first;

cout << "Enter double value: ";

in >> P.second;

return in;

}

List.h

#pragma once

const int MAX\_SIZE = 25;

template <typename TT>

class Node {

public:

TT data;

Node\* next = nullptr;

Node\* prev = nullptr;

};

template <typename TT>

class Iter

{

public:

Node<TT>\* node;

Iter();

Iter(const Iter<TT>& it);

Iter& operator=(const Iter<TT>& a);

TT& operator\*() const;

Iter& operator+ (const int& a);

};

template <typename TT>

class List

{

int size;

Node<TT>\* head = nullptr;

Node<TT>\* tail = nullptr;

Iter<TT> start;

Iter<TT> end;

public:

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

List(int second);

List(const List<TT>& l);

~List();

Iter<TT> first();

Iter<TT> last();

void push(TT a);

void pop();

List& operator= (const List<TT>& l);

TT& operator[] (int i);

int& operator()();

List operator\*(List<TT>& l);

};

template <typename TT>

List<TT>::List(int second)

{

size = second;

Node<TT>\* new\_elem = new Node<TT>;

head = new\_elem;

tail = new\_elem;

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

{

Node<TT>\* new\_elem = new Node<TT>;

tail->next = new\_elem;

new\_elem->prev = tail;

tail = new\_elem;

}

start.node = head;

end.node = tail;

}

template <typename TT>

List<TT>::List(const List<TT>& l)

{

Node<TT>\* node = l.head;

while (node != nullptr) {

push(node->data);

node = node->next;

}

start.node = head;

end.node = tail;

}

template <typename TT>

List<TT>::~List()

{

Node<TT>\* curr = head;

while (curr != nullptr)

{

head = curr->next;

delete curr;

curr = head;

}

}

template <typename TT>

List<TT>& List<TT>::operator= (const List<TT>& l)

{

if (this == &l)return \*this;

if (head != nullptr)

{

Node<TT>\* curr = head;

while (curr != nullptr)

{

head = curr->next;

delete curr;

curr = head;

}

}

Node<TT>\* node = l.head;

while (node != nullptr) {

push(node->data);

node = node->next;

}

start.node = head;

end.node = tail;

return \*this;

}

template <typename TT>

TT& List<TT>::operator[] (int i)

{

Node<TT>\* curr = head;

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

curr = curr->next;

}

return curr->data;

}

template <typename TT>

List<TT> List<TT>::operator\*(List<TT>& l)

{

List base = \*this;

int second = this->size;

List temp;

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

{

temp.push(base[i] \* l[i]);

}

return temp;

}

template <typename TT>

int& List<TT>::operator()() {

return size;

}

template <typename TT>

Iter<TT> List<TT>::first()

{

return start;

}

template <typename TT>

Iter<TT> List<TT>::last()

{

return end;

}

template <typename TT>

void List<TT>::push(TT a)

{

Node<TT>\* new\_elem = new Node<TT>;

new\_elem->data = a;

if (size == 0)

{

head = new\_elem;

tail = new\_elem;

size++;

start.node = head;

end.node = tail;

}

else

{

tail->next = new\_elem;

new\_elem->prev = tail;

tail = new\_elem;

size++;

end.node = tail;

}

}

template <typename TT>

void List<TT>::pop()

{

Node<TT>\* curr = tail;

tail = curr->prev;

delete curr;

tail->next = nullptr;

size--;

end.node = tail;

}

template <typename TT>

Iter<TT>::Iter() {

node = nullptr;

}

template <typename TT>

Iter<TT>::Iter(const Iter& it) {

node = it.node;

}

template <typename TT>

Iter<TT>& Iter<TT>::operator=(const Iter<TT>& a)

{

node = a.node;

return \*this;

}

template <typename TT>

TT& Iter<TT>::operator\*() const

{

return node->data;

}

template <typename TT>

Iter<TT>& Iter<TT>::operator+ (const int& a)

{

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

node = node->next;

}

return \*this;

}

Main.cpp

#include "List.h"

#include "Pair.h"

#include <iostream>

using namespace std;

int main()

{

int count, index;

double in\_value;

List<double> list1;

Iter <double> iter;

cout << "Enter number of elements: ";

cin >> count;

cout << endl;

cout << "Enter elements of list:" << endl;

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

{

cin >> in\_value;

list1.push(in\_value);

}

cout << endl;

iter = list1.first();

cout << "Enter index of element in list to call: ";

cin >> index;

cout << list1[index] << endl;

cout << endl;

cout << "Enter number of elements to move for from first: ";

cin >> index;

iter + index;

cout << \*iter << endl;

cout << endl;

List<Pair> list2;

Pair new\_in\_value;

cout << "Enter number of elements of new list: ";

cin >> count;

cout << endl;

cout << "Enter elements of list:" << endl;

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

{

cin >> new\_in\_value;

list2.push(new\_in\_value);

}

cout << "Enter index of element in list to call: ";

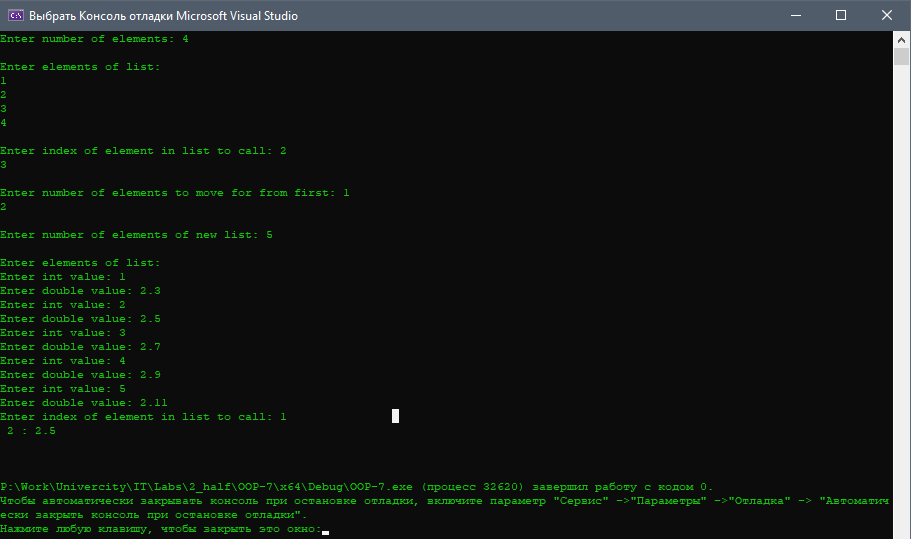
cin >> index;

cout << list2[index] << endl;

cout << endl;

}

1. Вывод программы



1. Ответы на контрольные вопросы

1. Шаблоны нужны для автоматической перегрузки функций под любой тип данных.

2. Перед функцией применяется конструкция template <typename T>, где T используется как переменная любого типа.

3. Перед функцией применяется конструкция template <class T>, где T используется как переменная любого типа.

4. Параметры шаблона функции – это переменные, которые будут использоваться для перегрузки.

5. Свойства параметра шаблона функции: имя параметра должно быть уникальным, список параметров не может быть пустым, в списке может быть несколько параметров.

6. Параметр шаблона записывается в треугольных скобках в формате “class имя” либо “typename имя”.

7. Параметризованные функции можно перегружать.

8. На основе параметризованного класса можно создавать перегруженные объекты. Каждый параметризованный класс должен иметь уникальное имя.

9. Все компонентные функции параметризованного класса являются параметризованными.

10. Дружественные функции, описанные в параметризованном классе, являются параметризованными.

11. Шаблоны классов могут содержать виртуальные компонентные функции.

12. template <class Type>

class MyClass {

Type value\_;

public:

MyClass(const T &value) : value\_(value) {};

void print();

};

template <class T>

void MyClass<T>::print() { std::cout << this->value\_; }

13. Инстанцирование шаблона – создание объекта параметризованного класса.

14. Генерирование определения класса по шаблону происходит на этапе вызова функции.