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: «Шаблоны классов»

Выполнил

Студент РИС-22-2б

Зубов Р.А.

Проверил

Доцент кафедры ИТАС

Полякова О. А.

Пермь 2023

**Постановка задачи**

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

Класс- контейнер СПИСОК с ключевыми значениями типа int. Реализовать операции:

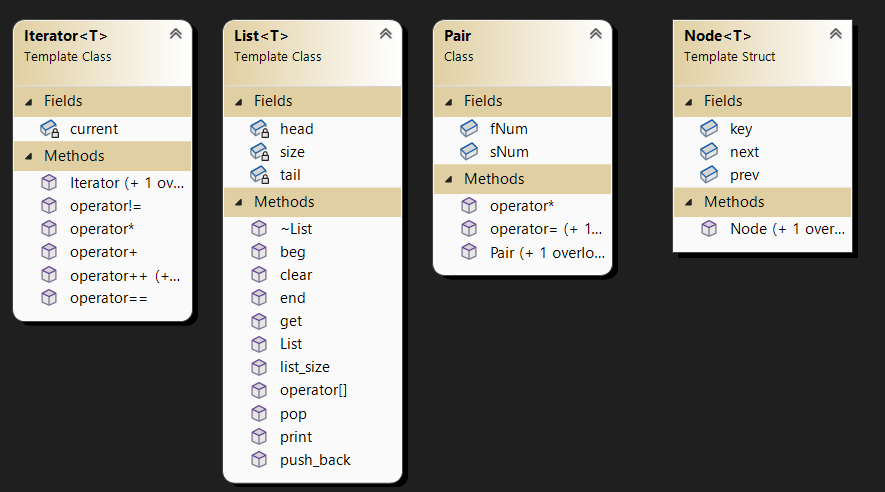
[] – доступа по индексу;

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.

**UML**

****

**Код программы**

**Pair.h:** #pragma once

#include <iostream>

class Pair

{

public:

int fNum = 0;

double sNum = 0;

Pair() {}

Pair(int first, double second) {

fNum = first;

sNum = second;

}

Pair& operator=(const Pair& t);

Pair& operator=(const int& t);

Pair operator\*(Pair& t);

friend std::ostream& operator<<(std::ostream& out, const Pair& t);

friend std::istream& operator>>(std::istream& in, const Pair& t);

};

**Pair.cpp:**

#include "Pair.h"

Pair& Pair::operator=(const Pair& pair) {

this->fNum = pair.fNum;

this->sNum = pair.sNum;

return \*this;

}

Pair& Pair::operator=(const int& t) {

this->fNum = t;

this->sNum = t;

return \*this;

}

Pair Pair::operator\*(Pair& pair) {

Pair new\_pair;

new\_pair.fNum = this->fNum \* pair.fNum;

new\_pair.sNum = this->sNum \* pair.sNum;

return new\_pair;

}

std::ostream& operator<<(std::ostream& out, const Pair& pair) {

out << pair.fNum << " : " << pair.sNum;

return out;

}

std::istream& operator>>(std::istream& in, Pair& pair) {

in >> pair.fNum >> pair.sNum;

return in;

}

**List.h:** #pragma once

#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 { //ñðàâíèâàþò òåêóùèé óçåë

return current == other.current; // è óçåë äðóãîãî èòåðàòîðà

} //

bool operator!=(const Iterator<T>& other) const { // íà ðàâåíñòâî è íåðàâåíñòâî.

return current != other.current;

}

T& operator\*() const { // îïåðàöèÿ ðàçûìåíîâàíèÿ èòåðàòîðà

return current->key;

}

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

current = current->next;

return \*this;

}

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

Iterator<T> old = \*this;

current = current->next;

return old;

}

Iterator operator+(int n) const { // ïåðåõîä âïðàâî ê n ýëåìåíòó

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); // âîçâðàùàåì ññûëêó íà ýëåìåíò ïî èíäåêñó

}

// Ìåòîä äîáàâëåíèÿ ýëåìåíòà â êîíåö ñïèñêà

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); // âîçâðàùàåì èòåðàòîð íà ïåðâûé ýëåìåíò

}

Iterator<T> end() {

return Iterator<T>(tail->next); // âîçâðàùàåì èòåðàòîð íà ïîñëåäíèé ýëåìåíò

}

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

List<T> result; // ñîçäàåì íîâûé ñïèñîê

Iterator<T> it1 = left.beg(); // èòåðàòîð ïî ïåðâîìó ñïèñêó

Iterator<T> it2 = right.beg(); // èòåðàòîð ïî âòîðîìó ñïèñêó

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

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

result.push\_back(res); // äîáàâëÿåì ïðîèçâåäåíèå ýëåìåíòîâ â íîâûé ñïèñîê

++it1; // ïåðåõîäèì ê ñëåäóþùåìó ýëåìåíòó ïåðâîãî ñïèñêà

++it2; // ïåðåõîäèì ê ñëåäóþùåìó ýëåìåíòó âòîðîãî ñïèñêà

}

return result; // âîçâðàùàåì íîâûé ñïèñîê

}

};

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; // íà÷èíàåì ñ ãîëîâû ñïèñêà

int i = 0;

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

current = current->next; // ïåðåõîäèì ê ñëåäóþùåìó óçëó

}

return current->key; // âîçâðàùàåì ññûëêó íà äàííûå óçëà

}

template<typename T>

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

{

Node<T>\* newNode = new Node<T>(k, NULL, tail); // ñîçäàåì íîâûé óçåë ñ êëþ÷îì key, óêàçàòåëåì íà ñëåäóþùèé óçåë ðàâíûì NULL è óêàçàòåëåì íà ïðåäûäóùèé óçåë ðàâíûì tail

if (tail != NULL) {

tail->next = newNode; // åñëè tail íå ðàâåí NULL, òî óñòàíàâëèâàåì óêàçàòåëü íà ñëåäóþùèé óçåë ó tail íà íîâûé óçåë

}

tail = newNode; // óñòàíàâëèâàåì tail íà íîâûé óçåë

if (head == NULL) {

head = newNode; // åñëè ñïèñîê áûë ïóñòîé, òî óñòàíàâëèâàåì head íà íîâûé óçåë

}

size++;

}

template<typename T>

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

Node<T>\* p = head;

while (p != NULL && p->key != key) { // Ïðîõîäèì ïî ñïèñêó äî íóæíîãî êëþ÷à

p = p->next;

}

if (p != NULL) { // Åñëè íàøëè óçåë ïî ïîçèöèè

if (p->prev != NULL) { // Åñëè óçåë íå ÿâëÿåòñÿ ãîëîâíûì

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

}

else { // Åñëè óçåë ÿâëÿåòñÿ ãîëîâíûì

head = p->next;

}

if (p->next != NULL) { // Åñëè óçåë íå ÿâëÿåòñÿ õâîñòîâûì

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

}

else { // Åñëè óçåë ÿâëÿåòñÿ õâîñòîâûì

tail = p->prev;

}

delete p; // Óäàëÿåì òåêóùèé óçåë èç ïàìÿòè

size--;

}

}

template<typename T>

void List<T>::clear()

{

Node<T>\* p = head;

while (p != NULL) {

Node<T>\* q = p;

p = p->next;

delete q; // Óäàëÿåì òåêóùèé óçåë èç ïàìÿòè

}

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;

}

**Source.cpp:**

#include "List.h"

#include <iostream>

#include "Pair.h"

using namespace std;

int main()

{

Pair p(2, 3.4);

List<Pair> l;

cout << "Made a empty list: " << l;

/\*l.print();\*/

cout << "List size: " << l.list\_size() << "\n";

cout << "Add pair to list: ";

l.push\_back(p);

l.print();

cout << "in index 0 elem is: " << l[0];

cout << "\nList size: " << l.list\_size();

cout << "\n";

Pair p1(3, 2.3);

List<Pair> l1;

l1.push\_back(p1);

cout << "Made a new list l1: "; l1.print();

List<Pair> l3 = l \* l1;

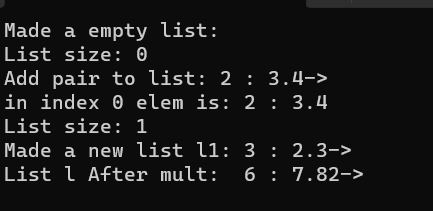
cout << "List l After mult: ";

l3.print();

return 0;

}

**Результаты программ**



**Контрольные вопросы**

1. В чем смысл использования шаблонов?

Использование шаблонов позволяет создавать обобщенный код, который может работать с различными типами данных без необходимости написания отдельного кода для каждого типа. Это упрощает разработку и поддержку кода, а также улучшает его переносимость и гибкость.

1. Каковы синтаксис/семантика шаблонов функций?

Синтаксис шаблонов функций выглядит следующим образом: template <typename T>

void swap(T& a, T& b) {

T temp = a;

a = b;

b = temp;

}

1. Каковы синтаксис/семантика шаблонов классов?

Синтаксис шаблонов классов выглядит следующим образом: template <typename T>

class Stack {

};

1. Что такое параметры шаблона функции?

Параметры шаблона функции - это значения, которые передаются в шаблон функции при его вызове и заменяют параметры шаблона на конкретные типы данных.

1. Перечислите основные свойства параметров шаблона функции.

Основные свойства параметров шаблона функции:

- они могут быть любого типа данных;

- они могут быть константами или ссылками на объекты;

- они могут иметь значение по умолчанию.

6. Как записывать параметр шаблона?

Параметр шаблона записывается в угловых скобках после ключевого слова `template`, например:

template <typename T>

1. Можно ли перегружать параметризованные функции?

Да, можно перегружать параметризованные функции, при этом каждая перегруженная функция должна иметь уникальный список параметров.

1. Перечислите основные свойства параметризованных классов.

Основные свойства параметризованных классов:

- они могут иметь любое количество параметров;

- параметры могут быть любого типа данных;

- параметры могут быть константами или ссылками на объекты;

- параметры могут иметь значение по умолчанию.

9. Все ли компонентные функции параметризованного класса являются параметризованными?

Нет, не все компонентные функции параметризованного класса являются параметризованными. Некоторые функции могут использовать только общие члены класса, которые не зависят от параметров шаблона.

10. Являются ли дружественные функции, описанные в параметризованном классе, параметризованными?

Дружественные функции, описанные в параметризованном классе, могут быть параметризованными или не параметризованными, в зависимости от того, нужно ли им использовать параметры шаблона.

11. Могут ли шаблоны классов содержать виртуальные компонентные функции?

Да, шаблоны классов могут содержать виртуальные компонентные функции.

12. Как определяются компонентные функции параметризованных классов вне определения шаблона класса?

Компонентные функции параметризованных классов могут быть определены вне определения шаблона класса следующим образом:

1. Определение функции должно начинаться со слова-ограничителя template<typename T>, где T - это типовой параметр класса.
2. Затем необходимо указать в угловых скобках имя класса и типовые параметры функции: Stack<T>::.
3. После этого указывается тип возвращаемого значения, имя функции и параметры функции, которые также могут быть параметризованы типом T.

13. Что такое инстанцирование шаблона?

Инстанцирование шаблона - это процесс создания конкретной версии шаблона с заменой параметров шаблона на конкретные типы данных.

14. На каком этапе происходит генерирование определения класса по шаблону?

Генерирование определения класса по шаблону происходит на этапе компиляции, когда вызывается шаблонная функция или создается объект класса с конкретными параметрами шаблона. На этом этапе компилятор заменяет параметры шаблона на конкретные типы данных и генерирует определение класса.