**Правительство Российской Федерации**

**Федеральное государственное автономное образовательное учреждение высшего профессионального образования   
"Национальный исследовательский университет   
"Высшая школа экономики"**

Московский институт электроники и математики Национального

исследовательского университета "Высшая школа экономики"

Департамент прикладной математики

**ОТЧЕТ**

**По лабораторной работе № 2**

**По курсу «Алгоритмизация и программирование»**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| |  |  | | --- | --- | |  | ФИО студента | | Номер группы | Дата |
| Рыбаков Владислав Евгеньевич | БПМ-204 | 23.11.2021 |
|  |
|  |
|  |

**Москва – 2021 г.**

**ПОСТАНОВКА ЗАДАЧИ (17 вариант)**

Задача: Написать шаблон класса-контейнера «Односвязный список элементов», который смог бы работать со стандартными типами данных и с пользовательскими типами данных. Реализовать конструкторы, деструкторы, операторы и операции, указанные в варианте, а так же необходимые для корректной работы программы. Написать программу, которая будет тестировать операции контейнера, содержащего стандартные типы данных int, float, double. Реализовать пользовательский класс Time, добавить все необходимые методы и функции, а так же те, которые необходимы для работы контейнера. Написать программу, которая будет тестировать операции контейнера, содержащего элементы пользовательского класса Time.

**ОПИСАНИЕ КЛАССА-КОНТЕЙНЕРА**

Односвязный список элементов является контейнером, в каждом объекте которого хранится значение, а так же указатель на следующий элемент. У последнего элемента значение указателя равно NULL. Элементы списка находятся в разных частях динамической памяти, поэтому доступ к ним можно получить только через указатель. В представленном шаблоне голова списка находится не в динамической, а в статической памяти для удобства работы со списком. Класс List реализован так, чтобы все детали реализации были скрыты от пользователя, то есть он не задумывался о работе с лишними вещами, необходимыми для внутренней работы класса, а делал только то, что ему нужно. Это решение было принято с целью достичь такой же простоты работы с классом-контейнером, как и со стандартными контейнерами. К методам класса List относятся: создание списка, добавление элемента в любую позицию, печать списка, удаление элемента из любой позиции, удаление всего списка, проверка списка на пустоту, взаимообмен двух элементов списка, а так же операторы ввода, вывода и присваивания элементов списка. Все эти методы могут работать с разными типами данных, как и положено шаблонному классу.

**ОПРЕДЕЛЕНИЕ КЛАССА-КОНТЕЙНЕРА**

#pragma once

#include <iostream>

template <typename abstracted>

class List

{

public:

abstracted data; // переменная в которой хранится значение

List\* next; // указатель на следующий элемент списка

public:

List() : next(NULL) {} // при создании элемента всегда указатель на следующий элемент делаем равным NULL

explicit List(abstracted smth) : data(smth), next(NULL) {}

~List() { // деструктор (удаляет весь список)

while (this->next != NULL) this->delete\_tail(); // голова всегда в статической памяти, а остальные элементы в динамической, поэтому при удалении головы чистится сразу вся выделенная память

}

friend std::ostream& operator << (std::ostream& out, List& a) { // оператор вывода

out << a.data << " "; // у типа abstracted должен быть перегружен оператор вывода

return out;

}

friend std::istream& operator >> (std::istream& in, List& a) { // оператор ввода

abstracted smt;

in >> smt; // у типа abstracted должен быть перегружен оператор ввода

a.data = smt;

return in;

}

List& operator=(const List& a) { // оператор присваивания

data = a.data;

next = a.next;

return \*this;

}

void add\_tail(abstracted one) { // добавление в хвост

List\* tmp = this;

for (; tmp->next != NULL; tmp = tmp->next); // проходим указателем до конца списка

tmp->next = new List(one); // добавляем в конец новый элемент

}

void add\_head(abstracted one) { // добавление в голову

if (this->data == NULL && this->next == NULL) { // если список пустой, то просто меняем значение головы

this->data = one;

return;

}

List\* som = new List; // иначе создаем новый элемент

\*som = \*this; // приравниваем его к прошлой голове

this->data = one; // меняем старую голову, делая ее новой

this->next = som;

}

void add\_pos(abstracted one, int pos) { // добавление в нужную позицию

if (pos < 0) { // проверка на отрицательную позицию

std::cout << "\nWrong position!\n";

return;

}

if (pos == 0) { // если позиция равна нулю, то добавляем в голову

this->add\_head(one);

return;

}

List\* tmp = this;

int count = 1;

for (; tmp->next != NULL; tmp = tmp->next) count++; // считаем количество элементов в списке

if (pos > count) { // проверка если указанная позиция больше длины списка

std::cout << "\nError. Position is more than list lenght.\n";

return;

}

else if (pos == count) { // если элемент последний, то добавляем в хвост

this->add\_tail(one);

return;

}

tmp = this;

for (int i = 1; i < pos; tmp = tmp->next, i++); // иначе доходим до нужной позиции

List\* p = new List(one); // выделяем память под новый элемент с заданным значением

p->next = tmp->next; // вставляем его в нужное место

tmp->next = p;

}

void printList() { // функция печати списка

List tmp;

std::cout << "\nYour list:\n";

if (this->data == NULL && this->next == NULL) { // проверка на пустоты списка

std::cout << "\nList is empty!\n";

return;

}

for (tmp = \*this; tmp.next != NULL; tmp = \*tmp.next) std::cout << tmp; // печать списка (без последнего элемена)

std::cout << tmp << std::endl; // печать последнего элемента списка

}

void createList() { // создание списка

int n;

abstracted a;

List\* tmp = this;

std::cout << "Enter number of elements: ";

std::cin >> n; // ввод количества элементов

if (n <= 0) { // проверка на некорректный ввод

std::cout << "\nWrong enter!\n";

return;

}

std::cout << "Enter " << n << " elements:\n";

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

std::cin >> a; // вводим n значений

if (i == 0) data = a; // если голова - просто приравниваем ей значение

else {

tmp->add\_tail(a); // добавляем в хвост

tmp = tmp->next; // смещаемся на элемент вправо

}

}

}

void delete\_head() { // удаление головы

if (this->next == NULL) { // если список из одного элемента (головы), то зануляем ее значение

this->data = NULL;

return;

}

List\* p = this->next; // иначе создаем указатель на первый элемент списка (нулевой - голова)

\*this = \*(this->next); // приравниваем голове первый элемент списка

p->next = NULL; // у прошлого первого элемента удаляем указатель на следующий, чтобы не удалить весь список

delete p; // очищаем p

}

void delete\_tail() { // удаление хвоста

List\* tmp = this;

if (this->next == NULL) { // если голова - зануляем значение

this->data = NULL;

return;

}

for (; (tmp->next)->next != NULL; tmp = tmp->next); // двигаемся до предпоследнего элемента списка

delete (tmp->next); // удаляем последний

tmp->next = NULL; // зануляем указатель у предпоследнего

}

void delete\_pos(int pos) { // удаление конкретной позиции

if (pos < 0) { // проверка на некорректный ввод

std::cout << "\nWrong position!\n";

return;

}

if (pos == 0) { // проверка на удаление головы

this->delete\_head();

return;

}

List\* tmp = this;

int count = 1;

for (; tmp->next != NULL; tmp = tmp->next) count++; // считаем кол-во элементов списка

if (pos >= count) { // проверка на выход за пределы списка

std::cout << "\nError. Position is more than list lenght.\n";

return;

}

else if (pos + 1 == count) { // проверка на удаление хвоста

this->delete\_tail();

return;

}

tmp = this;

for (int i = 1; i < pos; tmp = tmp->next, i++); // двигаемся до элемента с позицией pos-1

List\* p = tmp->next; // в p - удаляемый элемент

tmp->next = tmp->next->next; // пропускаем p

p->next = NULL; // зануляем у р указатель чтобы не стереть весь список

delete p;

}

void deleteList() { // удаление листа (то же самое что и деструктор)

while (this->next != NULL) this->delete\_tail();

this->data = NULL;

}

bool IsEmpty() { // проверка на пустоту

if (this->next == NULL && this->data == NULL) return true; // если значение равно нулю и указатель на следующий отсутствует, то список считается пустым

return false;

}

void change(const int& first, const int& second) { // обмен двух элементов списка

if (first < 0 || second < 0) { // проверка на корректность ввода

std::cout << "\nWrong position!\n";

return;

}

List\* tmp = this;

int count = 1;

for (; tmp->next != NULL; tmp = tmp->next) count++; // считаем кол-во лементов в списке

if (first >= count || second >= count) { // проверка на выход за пределы списка

std::cout << "\nError. Position is more than list lenght.\n";

return;

}

List\* tmp1 = this; // создаем два указателя, которые дойдут до нужных элементов

tmp = this;

for (int i = 1; i <= first; tmp = tmp->next, i++); // доходим до первого элемента

for (int i = 1; i <= second; tmp1 = tmp1->next, i++); // доходим до второго элемента

abstracted chng;

chng = tmp->data;

tmp->data = tmp1->data; // меняем значения этих двух элементов местами

tmp1->data = chng;

}

};

**ОПИСАНИЕ ПОЛЬЗОВАТЕЛЬСКОГО КЛАССА TIME**

Класс Time для работы со временем (минутами и секундами). В классе реализованы необходимые методы и операторы для работы с классом как напрямую, так и в классе-контейнере. Минуты и секунды вводятся и выводятся в обычном формате времени, то есть через двоеточие. При неправильном вводе чисел (минут или секунд больше 59), то правильные значения времени автоматически пересчитываются и лишние секунды прибавляются к минутам. Особые тонкости отмечены в комментариях в коде класса Time.

**ОПРЕДЕЛЕНИЕ КЛАССА TIME**

#pragma once

#include <iostream>

class Time

{ //Time class

public:

Time() = default; // конструктор по умолчанию

Time(int, int); // конструктор со значениями минут и секунд

void setTime(int, int); // сеттер

void getTime(int&, int&) const; // геттер

friend std::ostream& operator << (std::ostream& out, Time& a);

friend std::istream& operator >> (std::istream& in, Time& a);

friend bool operator==(const Time& a, const int& i);

Time& operator=(const Time& a);

Time& operator=(const int& a);

private:

int min, sec; // минуты и секунды соответственно

};

Time::Time(int mi, int se) {

min = (mi + se / 60) % 60; // проверка если секунд ввелось больше 60 и минут получается больше 60, то высчитывается правильное время

sec = se % 60; // высчитывается правильное кол-во секунд

}

void Time::setTime(int mi, int se) {

min = (mi + se / 60) % 60;

sec = se % 60;

}

void Time::getTime(int& one, int& two) const { // в две ссылки кладутся значения полей объекта

one = min;

two = sec;

}

std::ostream& operator << (std::ostream& out, Time& a) { // оператор вывода

if (a.sec > 9) { // если число секунд двузначно

out << a.min << ":" << a.sec;

}

else {

out << a.min << ":" << "0" << a.sec; // если число секунд однозначно, то добавляем после двоеточия 0

}

return out;

}

std::istream& operator >> (std::istream& in, Time& a) { // оператор ввода

char o; // в переменную о должнен ввестись символ ':'

int mi, se;

in >> mi >> o >> se; // подразумевается, что ввод будет в виде mm:ss

a.setTime(mi, se);

return in;

}

Time& Time::operator=(const Time& a) { // оператор присваивания такого же типа

min = a.min;

sec = a.sec;

return \*this;

}

Time& Time::operator=(const int& a) { // оператор присваивания переменной типа Time значения int (для работы с NULL)

min = (a / 60) % 60;

sec = a % 60;

return \*this;

}

bool operator==(const Time& a, const int& i) { // оператор сравнения переменной типа Time и типа int (для сравнения с NULL)

if (a.min == (i / 60) % 60 && a.sec == i % 60) return true;

return false;

}

**ТЕСТИРУЮЩАЯ ПРОГРАММА ДЛЯ ТИПА INT**

В программе представлены основные методы, а так же неправильные вводы данных и работа программы в этих случаях.

#include "List.h"

int main()

{

List<int> head;

head.createList();

head.printList();

std::cout << "\nAdd head 100:";

head.add\_head(100);

head.printList();

std::cout << "\nDelete head:";

head.delete\_head();

head.printList();

std::cout << "\nDelete tail:";

head.delete\_tail();

head.printList();

std::cout << "\nAdd 100 in 10 position:";

head.add\_pos(100, 10);

head.printList();

std::cout << "\nAdd 10 in 2 position:";

head.add\_pos(10, 2);

head.printList();

std::cout << "\nDelete -1 position:";

head.delete\_pos(-1);

head.printList();

std::cout << "\nDelete 3 position:";

head.delete\_pos(3);

head.printList();

std::cout << "\nChange 2 and 4 positions:";

head.change(2, 4);

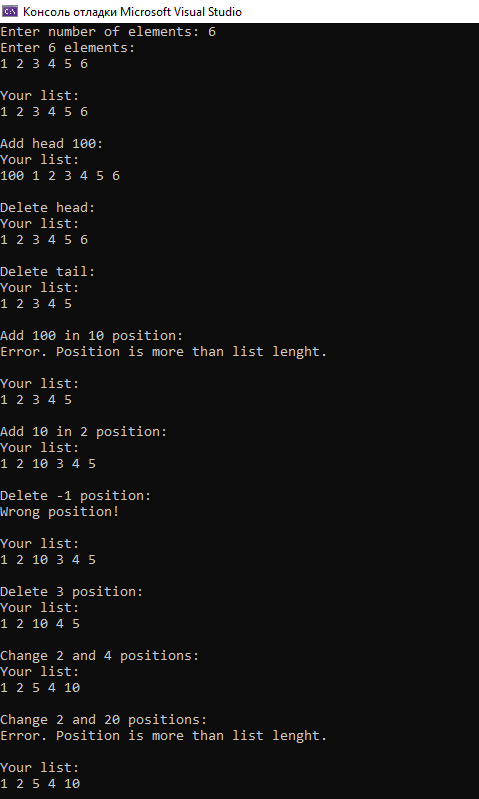
head.printList();

std::cout << "\nChange 2 and 20 positions:";

head.change(2, 20);

head.printList();

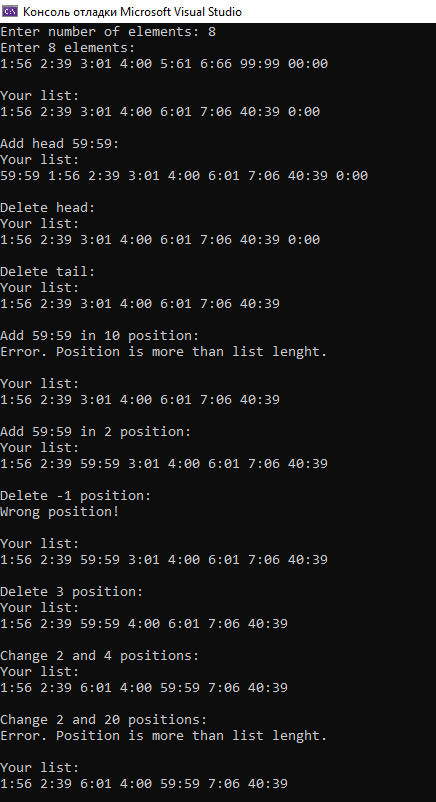
}



Как мы видим, все функции работают исправно и обрабатывают неправильные вводы данных, выводя на экран соответствующие ошибки. Что происходит при вызове каждого метода и результат выводятся на консоли.

**ТЕСТИРУЮЩАЯ ПРОГРАММА ДЛЯ ТИПА TIME**

В программе представлены основные методы, а так же неправильные вводы данных и работа программы в этих случаях с классом Time.



Как мы видим, все функции работают исправно и обрабатывают неправильные вводы данных, выводя на экран соответствующие ошибки. Некорректный ввод времени обрабатывается и заменяет значения на правильные. Что происходит при вызове каждого метода и результат выводятся на консоли.

**ОТВЕТЫ НА КОНТРОЛЬНЫЕ ВОПРОСЫ**

1. **В чем смысл использования шаблонов?**

Шаблоны позволяют адаптировать функцию или класс для работы с разными типами данных. При использовании шаблонов значительно сокращается объем кода, потому что функции или классы, выполняющие одинаковую работу, но работающие с разными типами данных, можно заменить на один шаблон. Но это работает только с тем, что пишет программист, поскольку после компиляции и инстанцирования шаблонов полученные классы и функции вставляются в код программы. Иными словами, разница в объеме кода при использовании шаблонов заметна только программисту, но не компьютеру.

1. **Что такое параметры шаблона функции?**

Это перемнная(-ые), которая(-ые) используется(-ются) в классе или функции, и ее(их) тип однозначно не определен. Определение типа происходит, когда программист в основной программе задает конкретный тип для шаблона.

1. **Перечислите основные свойства параметров шаблона функции.**

* Список параметров не может быть пустым
* Одно и то же имя параметров шаблона может быть использовано в разных шаблонах
* В списке параметров шаблона может быть один или несколько параметров
* Каждый параметр функции должен появиться в списке параметров хотя бы один раз
* Имя параметра имеет все права имени типа

1. **Как записывать параметр шаблона?**

Сначала перед функцией или классом пишется ключевое слово template – оно указывает, что после него идет шаблон. После template в угловых скобках <> указываются параметры шаблона. Если их несколько, то они разделяются запятыми. Для задания типов параметров используются ключевые слова typename или class (различий между ними никакого нет) и после них названия этих параметров. Пример:

template<typename abstracted, class my\_type>

1. **Можно ли перегружать параметризованные функции?**

Да, можно. По сути, компилятор уже перегружает шаблонные функции, так как все родственные функции, полученные из шаблона, имеют одно и то же имя, но разные наборы параметров. Однако шаблон функции может быть перегружен, причем несколькими способами:

* Другой шаблон функции, который имеет дополнительные параметры
* Нешаблонная функция с таким же именем, но другим набором параметров

1. **Перечислите основные свойства параметризованных классов.**

* Шаблоны методов или функций не могут быть виртуальными
* Шаблоны классов могут так же как и обычные классы содержать статические поля и методы, дружественные функции и классы
* Шаблон класса может быть производным как от шаблонного, так и от нешаблонного класса
* Шаблонный класс может быть производным от шаблона класса
* Не шаблонный класс может быть производным от шаблона класса
* Дружественные функции не являются автоматически параметризованными функциями
* Компонентные функции параметризованного класса автоматически являются параметризованными
* В рамках параметризованного класса нельзя определить дружественные шаблоны

1. **Все ли компонентные функции параметризованного класса являются параметризованными?**

Да, все компонентные функции параметризованного класса автоматически являются параметризованными.

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

Нет, не являются.

1. **Могут ли шаблоны классов содержать виртуальные компонентные функции?**

Нет, не могут.

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

Сначала пишется ключевое слово template и в угловых скобках <> записываются те же параметры, что и у шаблона класса. Потом как обычно пишется функция, но только еще указывается название шаблона класса и его параметры. Пример:

template<class abstracted>

void MyClass<abstracted>::MyFunc(…){…}

1. **Что такое инстанцирование шаблона?**

Это процесс генерации компилятором функций и классов из шаблонов.

1. **На каком этапе происходит генерирование определения класса по шаблону?**

При компиляции. Если в коде было задание конкретного типа для шаблона, то при компиляции идет инстанцирование шаблона для тех типов данных, которые были в коде программы, и код шаблона с заданными параметрами встраивается в исполняемый код.