**1. Тема и цель лабораторной работы**

• **Тема**: «Шаблоны классов. Работа с исключениями. Разработка однонаправленного списка» (Вариант № 7).

• **Цель**:

1. Освоить создание собственных **шаблонных** классов (templates) в C++.

2. Научиться корректно обрабатывать **исключения** при работе с динамическими структурами данных.

3. Реализовать **однонаправленный список** (linked list) без использования стандартных контейнеров (STL).

4. Продемонстрировать работу списка с пользовательским типом данных — рациональные числа из ЛР № 1 (при этом класс **Rational** также сделать шаблонным).

Таким образом, лабораторная работа направлена на закрепление навыков:

• использования механизма **шаблонов** (templates),

• организации кода **по модулям** (файлы .h + .inl),

• управления **динамической памятью**,

• и обработки **исключительных ситуаций**.

**2. Задание**

1. **Создать** шаблонный класс SinglyLinkedList<T> без использования контейнеров STL.

2. **Реализовать** методы:

• append(const T&) — добавление элемента в конец списка,

• prepend(const T&) — добавление элемента в начало списка,

• insertAt(size\_t index, const T&) — вставка по индексу,

• removeAt(size\_t index) — удаление по индексу с возвратом удалённого значения,

• getAt(size\_t index) const — доступ к элементу по индексу,

• getSize() const и isEmpty() const.

3. **Обработать** исключения: выбрасывать std::out\_of\_range, std::underflow\_error и т.д. в граничных ситуациях (удаление из пустого списка, индекс вне диапазона).

4. **Разделить код** на модули: заголовочные файлы (и .inl при желании), где объявляются классы и их реализации.

5. **Продемонстрировать** использование второго модуля — шаблонного класса Rational<T> (из ЛР № 1, где реализованы операции +, -, \*, /, сравнения, редукция дробей). Показать корректную работу списка с этим типом.

**3. Теоретическая часть**

**3.1 Что такое шаблоны (Templates) в C++**

• **Шаблоны** позволяют писать обобщённый код, не зависящий от конкретного типа, который будет подставлен при использовании.

• **Компилятор** на этапе трансляции генерирует «конкретные» версии кода для тех типов, которые реально использует программа (например, SinglyLinkedList<int>, SinglyLinkedList<Rational<int>>).

• **Преимущества**:

1. Нет необходимости писать разные классы для int, double, Rational<int> и т.д.

2. Снижается вероятность дублирования кода и связанных с этим ошибок.

3. Обеспечивается безопасность типов (компилятор всё проверяет на этапе компиляции).

**3.2 Однонаправленный связный список**

• **Суть**: Набор узлов (Node), каждый хранит данные (data) и указатель на следующий узел (next).

• **Операции**:

• Добавление элемента в начало (prepend) и конец (append),

• Вставка/удаление по индексу,

• Доступ к элементу (линейный обход),

• Проверка пустоты, получение размера (количества узлов).

• **Плюсы**: эффективное добавление/удаление на произвольных позициях — не нужно «сдвигать» все последующие элементы, как в массиве.

• **Минусы**: нет быстрых случайных обращений (нужно итерироваться по списку от начала до позиции index).

**3.3 Исключения (Exceptions) в C++**

• Позволяют «выбрасывать» (throw) ошибку и «ловить» (catch) её в другом месте, не теряя контекст.

• **Стандартные исключения**:

• std::out\_of\_range — используют при выходе за пределы списка (если index >= size).

• std::underflow\_error — когда операция невозможна при данных условиях (удаление из пустого списка).

• std::invalid\_argument — некорректный аргумент функции (например, нулевой знаменатель в Rational).

**4. Структура программы (модули)**

1. rational.h / rational.inl

• rational.h: объявление шаблонного класса Rational<T> (рациональные числа).

• Определяем поля numerator, denominator типа T, конструкторы, прототипы арифметических операций и сравнений.

• В самом конце делаем #include "rational.inl", чтобы подключить реализации.

• rational.inl: реализация (определения) методов Rational<T>:

• Используем template<typename T> перед каждой функцией.

• Приведение к редуцированной форме (через НОД), операции +, -, \*, /, сравнения <, >, == и т.д.

2. singlylinkedlist.h / singlylinkedlist.inl

• singlylinkedlist.h: объявление шаблонного класса SinglyLinkedList<T> и вложенного Node<T>.

• Содержит поля head, tail (указатели на Node<T>), size\_t size — текущее число элементов.

• Описываем методы append, prepend, insertAt, removeAt, getAt, getSize, isEmpty.

• В конце файла #include "singlylinkedlist.inl" — чтобы подключить реализации методов.

• singlylinkedlist.inl: реализация всех методов класса SinglyLinkedList<T>.

• Каждая функция с template<typename T>.

• В методах при добавлении узла создаём Node<T>\* newNode = new Node<T>(value, ...).

• Проверяем граничные условия, выбрасываем исключения std::out\_of\_range или std::underflow\_error там, где нужно.

3. main.cpp

• Подключаем заголовки:

#include "singlylinkedlist.h"

#include "rational.h"

• Создаём объекты:

• SinglyLinkedList<int> intList;

• Применяем методы append, removeAt и т.д.

• SinglyLinkedList<Rational<int>> ratList;

• Добавляем рациональные числа, проверяем корректность работы.

• Ловим исключения std::exception (или более конкретные) и выводим сообщения об ошибках.

Таким образом, шаблонный код **автоматически** инстанцируется компилятором для нужных типов (int, Rational<int>, и т.д.), без ручной прописки template class ... в .cpp.

**5. Листинг кода**

**rational.h***#ifndef RATIONAL\_H*

*#define RATIONAL\_H*

*#include <iostream>*

*#include <stdexcept>*

*#include <cmath> // std::abs*

*/\*\**

*\* @brief Шаблонный класс для рациональных чисел*

*\* @tparam T Целочисленный тип (int, long long и т.д.)*

*\*/*

*template<typename T>*

*class Rational {*

*private:*

*T numerator; ///< Числитель*

*T denominator; ///< Знаменатель*

*void reduce(); ///< Приведение к редуцированной форме (деление на НОД)*

*public:*

*// Конструкторы*

*Rational(); // 0/1 по умолчанию*

*Rational(T n, T d); // с параметрами*

*// Вывод на экран*

*void print() const;*

*// Арифметические операции*

*Rational operator+(const Rational& other) const;*

*Rational operator-(const Rational& other) const;*

*Rational operator\*(const Rational& other) const;*

*Rational operator/(const Rational& other) const;*

*// Операции сравнения*

*bool operator==(const Rational& other) const;*

*bool operator!=(const Rational& other) const;*

*bool operator<(const Rational& other) const;*

*bool operator>(const Rational& other) const;*

*bool operator<=(const Rational& other) const;*

*bool operator>=(const Rational& other) const;*

*};*

*// ===== Подключаем реализацию из файла .inl =====*

*#include "rational.inl"*

*#endif // RATIONAL\_H*

**rational.inl***#pragma once*

*// Вспомогательная функция для вычисления НОД:*

*template<typename T>*

*inline T gcd\_template(T a, T b) {*

*while (b != 0) {*

*T temp = b;*

*b = a % b;*

*a = temp;*

*}*

*return a;*

*}*

*// ==== Определения всех методов шаблонного класса Rational<T> ====*

*template<typename T>*

*Rational<T>::Rational() : numerator(0), denominator(1) {}*

*template<typename T>*

*Rational<T>::Rational(T n, T d) : numerator(n), denominator(d) {*

*if (d == 0) {*

*throw std::invalid\_argument("Denominator cannot be zero");*

*}*

*reduce();*

*}*

*template<typename T>*

*void Rational<T>::reduce() {*

*T divisor = gcd\_template(std::abs(numerator), std::abs(denominator));*

*numerator /= divisor;*

*denominator /= divisor;*

*// Если знаменатель < 0, «переносим» знак к числителю*

*if (denominator < 0) {*

*numerator = -numerator;*

*denominator = -denominator;*

*}*

*}*

*template<typename T>*

*void Rational<T>::print() const {*

*std::cout << numerator << "/" << denominator << "\n";*

*}*

*// +*

*template<typename T>*

*Rational<T> Rational<T>::operator+(const Rational& other) const {*

*T newNum = numerator \* other.denominator + other.numerator \* denominator;*

*T newDen = denominator \* other.denominator;*

*return Rational<T>(newNum, newDen);*

*}*

*// -*

*template<typename T>*

*Rational<T> Rational<T>::operator-(const Rational& other) const {*

*T newNum = numerator \* other.denominator - other.numerator \* denominator;*

*T newDen = denominator \* other.denominator;*

*return Rational<T>(newNum, newDen);*

*}*

*// \**

*template<typename T>*

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

*T newNum = numerator \* other.numerator;*

*T newDen = denominator \* other.denominator;*

*return Rational<T>(newNum, newDen);*

*}*

*// /*

*template<typename T>*

*Rational<T> Rational<T>::operator/(const Rational& other) const {*

*if (other.numerator == 0) {*

*throw std::invalid\_argument("Division by zero");*

*}*

*T newNum = numerator \* other.denominator;*

*T newDen = denominator \* other.numerator;*

*return Rational<T>(newNum, newDen);*

*}*

*// ==*

*template<typename T>*

*bool Rational<T>::operator==(const Rational& other) const {*

*return (numerator == other.numerator && denominator == other.denominator);*

*}*

*// !=*

*template<typename T>*

*bool Rational<T>::operator!=(const Rational& other) const {*

*return !(\*this == other);*

*}*

*// <*

*template<typename T>*

*bool Rational<T>::operator<(const Rational& other) const {*

*return (numerator \* other.denominator < other.numerator \* denominator);*

*}*

*// >*

*template<typename T>*

*bool Rational<T>::operator>(const Rational& other) const {*

*return (numerator \* other.denominator > other.numerator \* denominator);*

*}*

*// <=*

*template<typename T>*

*bool Rational<T>::operator<=(const Rational& other) const {*

*return !(\*this > other);*

*}*

*// >=*

*template<typename T>*

*bool Rational<T>::operator>=(const Rational& other) const {*

*return !(\*this < other);*

*}*

**singlylinkedlist.h**

*#ifndef SINGLYLINKEDLIST\_H*

*#define SINGLYLINKEDLIST\_H*

*#include <cstddef> // size\_t*

*#include <stdexcept> // для исключений std::out\_of\_range, std::underflow\_error*

*/\*\**

*\* @brief Узел однонаправленного списка*

*\* @tparam U Тип данных, хранящихся в узле*

*\*/*

*template<typename U>*

*class Node {*

*public:*

*U data;*

*Node\* next;*

*Node(const U& value, Node\* nextNode = nullptr)*

*: data(value), next(nextNode)*

*{}*

*};*

*/\*\**

*\* @brief Шаблонный класс однонаправленного списка*

*\* @tparam T Тип элементов списка*

*\*/*

*template<typename T>*

*class SinglyLinkedList {*

*private:*

*Node<T>\* head; ///< Указатель на первый узел*

*Node<T>\* tail; ///< Указатель на последний узел*

*size\_t size; ///< Текущее число элементов*

*public:*

*// Конструктор и деструктор*

*SinglyLinkedList();*

*~SinglyLinkedList();*

*// Методы добавления*

*void append(const T& value);*

*void prepend(const T& value);*

*// Вставка и удаление по индексу*

*void insertAt(size\_t index, const T& value);*

*T removeAt(size\_t index);*

*// Доступ к элементу*

*T getAt(size\_t index) const;*

*// Служебные методы*

*bool isEmpty() const { return size == 0; }*

*size\_t getSize() const { return size; }*

*};*

*// ===== Подключаем реализацию (inl) =====*

*#include "singlylinkedlist.inl"*

*#endif // SINGLYLINKEDLIST\_H*

**singlylinkedlist.inl**

*#pragma once*

*// ==== Реализация методов шаблонного класса SinglyLinkedList<T> ====*

*template<typename T>*

*SinglyLinkedList<T>::SinglyLinkedList()*

*: head(nullptr), tail(nullptr), size(0)*

*{ }*

*template<typename T>*

*SinglyLinkedList<T>::~SinglyLinkedList() {*

*Node<T>\* current = head;*

*while (current) {*

*Node<T>\* temp = current;*

*current = current->next;*

*delete temp;*

*}*

*}*

*template<typename T>*

*void SinglyLinkedList<T>::append(const T& value) {*

*Node<T>\* newNode = new Node<T>(value);*

*if (!head) {*

*head = tail = newNode;*

*} else {*

*tail->next = newNode;*

*tail = newNode;*

*}*

*++size;*

*}*

*template<typename T>*

*void SinglyLinkedList<T>::prepend(const T& value) {*

*Node<T>\* newNode = new Node<T>(value, head);*

*head = newNode;*

*if (!tail) {*

*tail = head;*

*}*

*++size;*

*}*

*template<typename T>*

*void SinglyLinkedList<T>::insertAt(size\_t index, const T& value) {*

*if (index > size) {*

*throw std::out\_of\_range("Index out of range");*

*}*

*if (index == 0) {*

*prepend(value);*

*return;*

*}*

*if (index == size) {*

*append(value);*

*return;*

*}*

*Node<T>\* current = head;*

*for (size\_t i = 0; i < index - 1; ++i) {*

*current = current->next;*

*}*

*Node<T>\* newNode = new Node<T>(value, current->next);*

*current->next = newNode;*

*++size;*

*}*

*template<typename T>*

*T SinglyLinkedList<T>::removeAt(size\_t index) {*

*if (isEmpty()) {*

*throw std::underflow\_error("Cannot remove from an empty list");*

*}*

*if (index >= size) {*

*throw std::out\_of\_range("Index out of range");*

*}*

*Node<T>\* current = head;*

*T value;*

*if (index == 0) {*

*value = current->data;*

*head = head->next;*

*delete current;*

*if (!head) {*

*tail = nullptr;*

*}*

*} else {*

*for (size\_t i = 0; i < index - 1; ++i) {*

*current = current->next;*

*}*

*Node<T>\* temp = current->next;*

*value = temp->data;*

*current->next = temp->next;*

*if (!current->next) {*

*tail = current;*

*}*

*delete temp;*

*}*

*--size;*

*return value;*

*}*

*template<typename T>*

*T SinglyLinkedList<T>::getAt(size\_t index) const {*

*if (index >= size) {*

*throw std::out\_of\_range("Index out of range");*

*}*

*Node<T>\* current = head;*

*for (size\_t i = 0; i < index; ++i) {*

*current = current->next;*

*}*

*return current->data;*

*}*

**singlylinkedlist.inl**

*#pragma once*

*// ==== Реализация методов шаблонного класса SinglyLinkedList<T> ====*

*template<typename T>*

*SinglyLinkedList<T>::SinglyLinkedList()*

*: head(nullptr), tail(nullptr), size(0)*

*{ }*

*template<typename T>*

*SinglyLinkedList<T>::~SinglyLinkedList() {*

*Node<T>\* current = head;*

*while (current) {*

*Node<T>\* temp = current;*

*current = current->next;*

*delete temp;*

*}*

*}*

*template<typename T>*

*void SinglyLinkedList<T>::append(const T& value) {*

*Node<T>\* newNode = new Node<T>(value);*

*if (!head) {*

*head = tail = newNode;*

*} else {*

*tail->next = newNode;*

*tail = newNode;*

*}*

*++size;*

*}*

*template<typename T>*

*void SinglyLinkedList<T>::prepend(const T& value) {*

*Node<T>\* newNode = new Node<T>(value, head);*

*head = newNode;*

*if (!tail) {*

*tail = head;*

*}*

*++size;*

*}*

*template<typename T>*

*void SinglyLinkedList<T>::insertAt(size\_t index, const T& value) {*

*if (index > size) {*

*throw std::out\_of\_range("Index out of range");*

*}*

*if (index == 0) {*

*prepend(value);*

*return;*

*}*

*if (index == size) {*

*append(value);*

*return;*

*}*

*Node<T>\* current = head;*

*for (size\_t i = 0; i < index - 1; ++i) {*

*current = current->next;*

*}*

*Node<T>\* newNode = new Node<T>(value, current->next);*

*current->next = newNode;*

*++size;*

*}*

*template<typename T>*

*T SinglyLinkedList<T>::removeAt(size\_t index) {*

*if (isEmpty()) {*

*throw std::underflow\_error("Cannot remove from an empty list");*

*}*

*if (index >= size) {*

*throw std::out\_of\_range("Index out of range");*

*}*

*Node<T>\* current = head;*

*T value;*

*if (index == 0) {*

*value = current->data;*

*head = head->next;*

*delete current;*

*if (!head) {*

*tail = nullptr;*

*}*

*} else {*

*for (size\_t i = 0; i < index - 1; ++i) {*

*current = current->next;*

*}*

*Node<T>\* temp = current->next;*

*value = temp->data;*

*current->next = temp->next;*

*if (!current->next) {*

*tail = current;*

*}*

*delete temp;*

*}*

*--size;*

*return value;*

*}*

*template<typename T>*

*T SinglyLinkedList<T>::getAt(size\_t index) const {*

*if (index >= size) {*

*throw std::out\_of\_range("Index out of range");*

*}*

*Node<T>\* current = head;*

*for (size\_t i = 0; i < index; ++i) {*

*current = current->next;*

*}*

*return current->data;*

*}*

**main.cpp**

*#include <iostream>*

*#include "singlylinkedlist.h"*

*#include "rational.h"*

*int main() {*

*try {*

*// Пример: список целых чисел*

*SinglyLinkedList<int> intList;*

*intList.append(10);*

*intList.append(20);*

*intList.prepend(5); // [5, 10, 20]*

*intList.insertAt(1, 15); // [5, 15, 10, 20]*

*std::cout << "intList elements:\n";*

*for (size\_t i = 0; i < intList.getSize(); ++i) {*

*std::cout << intList.getAt(i) << " ";*

*}*

*std::cout << "\n";*

*int removedInt = intList.removeAt(1); // удаляем элемент с индексом 1 (15)*

*std::cout << "Removed element: " << removedInt << "\n\n";*

*// Пример: список Rational<int>*

*SinglyLinkedList<Rational<int>> ratList;*

*ratList.append(Rational<int>(1, 2)); // 1/2*

*ratList.append(Rational<int>(3, 4)); // 3/4*

*ratList.prepend(Rational<int>(5, 6)); // [5/6, 1/2, 3/4]*

*// Вставим 7/8 на позицию 1 => [5/6, 7/8, 1/2, 3/4]*

*ratList.insertAt(1, Rational<int>(7, 8));*

*std::cout << "ratList elements:\n";*

*for (size\_t i = 0; i < ratList.getSize(); ++i) {*

*ratList.getAt(i).print();*

*}*

*// Удалим элемент с индексом 2 (где сейчас 1/2)*

*Rational<int> removedRat = ratList.removeAt(2);*

*std::cout << "Removed (Rational<int>): ";*

*removedRat.print();*

*std::cout << "\n";*

*// Выведем элементы после удаления*

*std::cout << "ratList after removal:\n";*

*for (size\_t i = 0; i < ratList.getSize(); ++i) {*

*ratList.getAt(i).print();*

*}*

*} catch (const std::exception& e) {*

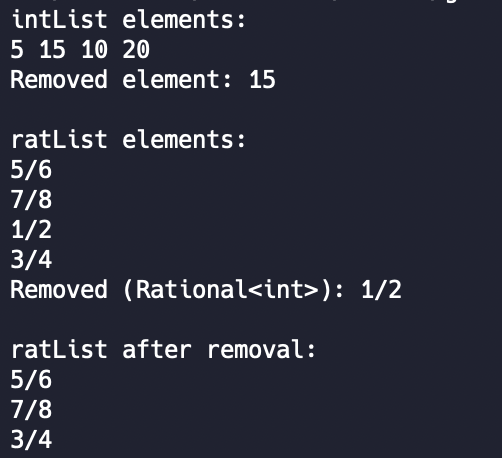
*std::cerr << "Error: " << e.what() << "\n";*

*}*

*return 0;*

*}*

**6. Тестирование и результаты**

****

**6.1 Список int**

1. append(10), append(20) даёт список [10, 20].

2. prepend(5) → [5, 10, 20].

3. insertAt(1, 15) → [5, 15, 10, 20].

4. removeAt(1) удаляет 15, остаётся [5, 10, 20].

5. Если вызвать removeAt(10), программа выбросит std::out\_of\_range, так как 10 >= size.

**6.2 Список Rational<int>**

1. append(Rational<int>(1,2)), append(Rational<int>(3,4)) → [1/2, 3/4].

2. prepend(Rational<int>(5,6)) → [5/6, 1/2, 3/4].

3. insertAt(1, Rational<int>(7,8)) → [5/6, 7/8, 1/2, 3/4].

4. removeAt(2) → удаляем 1/2, остаётся [5/6, 7/8, 3/4].

5. Аналогично при removeAt(10) или удалении из пустого выбрасываются соответствующие исключения.

**6.3 Итог**

• Все базовые операции списка работают корректно.

• Исключения обрабатываются в блоках try-catch и не вызывают краха программы.

• Утечек памяти нет (при желании можно проверить Valgrind или аналогичные утилиты).

**7. Выводы**

1. **Шаблонный класс** однонаправленного списка (SinglyLinkedList<T>) успешно реализован без контейнеров STL.

2. Методы **добавления**, **удаления**, **вставки** и **доступа** к элементам функционируют корректно, протестированы на int и Rational<int>.

3. Код **разделён** на несколько модулей (заголовочные + .inl) — что повышает удобство сопровождения.

4. **Исключения** (std::out\_of\_range, std::underflow\_error, std::invalid\_argument) обеспечивают устойчивость программы в ошибочных сценариях.

5. Rational<int> интегрирован в список без каких-либо дополнительных сложностей, подтверждая универсальность шаблонного решения.

Таким образом, **цель** лабораторной работы — освоение шаблонов, исключений и динамических структур данных — **достигнута**.

**8. Приложения**

1. **Исходный код**:

• rational.h / rational.inl (объявление и реализация Rational<T>).

• singlylinkedlist.h / singlylinkedlist.inl (объявление и реализация SinglyLinkedList<T>).

• main.cpp (демонстрация работы).

2. **Скриншоты** вывода программы при запуске