МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ОБРАЗОВАНИЯ
«НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»
Кафедра вычислительной техники

ОТЧЕТ
ПО ЛАБОРАТОРНОЙ РАБОТЕ №7
ПО ДИСЦИПЛИНЕ «ПРОГРАММИРОВАНИЕ»

Факультет: Заочное отделение Преподаватель: Дубков Илья Сергеевич

Группа: ДТ-460а

Студент: Дроздов Иван Сергеевич

Вариант: 0

Новосибирск, 2025 г.

Цель работы: Ознакомиться с созданием шаблонов функций и классов. Изучить написание контейнерных шаблонных классов и их применение для построения различных структур данных.

Задание: Задание представляет собой типовую задачу по разработке шаблонов стандартных структур данных. Протестировать структуру данных. В качестве хранимых объектов использовать встроенные типы C++ (int и char).

Исходные коды модулей проекта:

```
//Stack.h
#ifndef STACK_H
#define STACK_H

#include <cstdio>
#include <cstring>

// Класс исключения без STL
class StackException {
private:
    char message[100];
public:
    StackException(const char* msg) {
        strncpy(message, msg, sizeof(message) - 1);
        message[sizeof(message) - 1] = '\0';
    }

    const char* what() const {
        return message;
    }
};

template<typename T, size_t Capacity = 10>
class Stack {
private:
    T* data;          // Динамический массив
    size_t topIndex;   // Индекс верхнего элемента
    size_t capacity;   // Текущая емкость

    // Увеличение емкости
    void resize() {
        size_t newCapacity = capacity * 2;
        T* newData = new T[newCapacity];

        // Копируем элементы
        for (size_t i = 0; i < topIndex; ++i) {
            newData[i] = data[i];
        }

        delete[] data;
        data = newData;
```

```
        capacity = newCapacity;

        printf("Стек увеличен. Новая емкость: %zu\n", capacity);
    }

public:
    // Конструктор
    Stack() : capacity(Capacity), topIndex(0) {
        data = new T[capacity];
        printf("Стек создан с емкостью: %zu\n", capacity);
    }

    // Деструктор
    ~Stack() {
        delete[] data;
        printf("Стек уничтожен\n");
    }

    // Копирующий конструктор
    Stack(const Stack& other) : capacity(other.capacity), topIndex(other.topIndex) {
        data = new T[capacity];
        for (size_t i = 0; i < topIndex; ++i) {
            data[i] = other.data[i];
        }
        printf("Стек скопирован\n");
    }

    // Оператор присваивания
    Stack& operator=(const Stack& other) {
        if (this != &other) {
            delete[] data;

            capacity = other.capacity;
            topIndex = other.topIndex;
            data = new T[capacity];

            for (size_t i = 0; i < topIndex; ++i) {
                data[i] = other.data[i];
            }
        }
        printf("Стек присвоен\n");
        return *this;
    }

    // Добавление элемента в стек (загрузка)
    void push(const T& value) {
        if (topIndex >= capacity) {
            resize();
        }

        data[topIndex] = value;
        topIndex++;
        printf("Элемент добавлен в стек. Текущий размер: %zu\n", topIndex);
    }

    // Извлечение элемента из стека
    T pop() {
        if (isEmpty()) {
            throw StackException("Попытка извлечения из пустого стека");
        }

        topIndex--;
```

```
        T value = data[topIndex];
        printf("Элемент извлечен из стека. Текущий размер: %zu\n", topIndex);

        return value;
    }

    // Просмотр верхнего элемента без извлечения
    T peek() const {
        if (isEmpty()) {
            throw StackException("Стек пуст");
        }
        return data[topIndex - 1];
    }

    // Проверка на пустоту
    bool isEmpty() const {
        return topIndex == 0;
    }

    // Получение текущего размера
    size_t size() const {
        return topIndex;
    }

    // Получение емкости
    size_t getCapacity() const {
        return capacity;
    }

    // Вывод содержимого стека
    void display() const {
        if (isEmpty()) {
            printf("Стек пуст\n");
            return;
        }

        printf("Содержимое стека (размер: %zu, емкость: %zu):\n", topIndex, capacity);
        for (int i = topIndex - 1; i >= 0; --i) {
            printf("[%d] ", i);

            // Специальный вывод для char
            if constexpr (sizeof(T) == sizeof(char)) {
                printf("'%c'\n", data[i]);
            } else {
                printf("%d\n", data[i]);
            }
        }
    }
};

#endif
```

L7.cpp

```
#include "Stack.h"
#include <cstdio>

// Функция для демонстрации работы с int
void demonstrateIntStack() {
    printf("=== Демонстрация стека для int ===\n");

    // Создаем стек для int с начальной емкостью 3
    Stack<int, 3> intStack;
```

```
    // Добавляем элементы (будет происходить увеличение размера)
    printf("\n1. Добавление элементов в стек:\n");
    for (int i = 1; i <= 7; ++i) {
        intStack.push(i * 11);
    }
    intStack.display();

    // Просмотр верхнего элемента
    printf("\n2. Верхний элемент: %d\n", intStack.peek());

    // Извлечение элементов
    printf("\n3. Извлечение элементов из стека:\n");
    while (!intStack.isEmpty()) {
        int value = intStack.pop();
        printf("Извлечено: %d\n", value);
    }

    printf("Состояние после извлечения всех элементов: ");
    intStack.display();
}

// Функция для демонстрации работы с char
void demonstrateCharStack() {
    printf("\n=== Демонстрация стека для char ===\n");

    // Создаем стек для char с начальной емкостью 2
    Stack<char, 2> charStack;

    // Добавляем элементы
    printf("\n1. Добавление символов в стек:\n");
    char chars[] = {'A', 'B', 'C', 'D', 'E'};
    for (int i = 0; i < 5; ++i) {
        charStack.push(chars[i]);
    }
    charStack.display();

    // Просмотр верхнего элемента
    printf("\n2. Верхний элемент: '%c'\n", charStack.peek());

    // Извлечение некоторых элементов
    printf("\n3. Извлечение 2 элементов:\n");
    for (int i = 0; i < 2; ++i) {
        char value = charStack.pop();
        printf("Извлечено: '%c'\n", value);
    }

    printf("Состояние после извлечения 2 элементов:\n");
    charStack.display();

    // Добавление еще элементов
    printf("\n4. Добавление новых символов:\n");
    charStack.push('X');
    charStack.push('Y');
    charStack.push('Z');
    charStack.display();
}

// Функция для демонстрации обработки ошибок
void demonstrateErrorHandling() {
    printf("\n=== Демонстрация обработки ошибок ===\n");
```

```
    Stack<int, 2> errorStack;

    // Попытка извлечения из пустого стека
    printf("1. Попытка извлечения из пустого стека:\n");
    try {
        errorStack.pop();
        printf("Ошибка: исключение не было брошено!\n");
    } catch (const StackException& e) {
        printf("Поймано исключение: %s\n", e.what());
    }

    // Попытка просмотра пустого стека
    printf("\n2. Попытка просмотра пустого стека:\n");
    try {
        errorStack.peek();
        printf("Ошибка: исключение не было брошено!\n");
    } catch (const StackException& e) {
        printf("Поймано исключение: %s\n", e.what());
    }

    // Корректные операции
    printf("\n3. Корректные операции:\n");
    errorStack.push(100);
    errorStack.push(200);
    errorStack.display();

    printf("Верхний элемент: %d\n", errorStack.peek());
    printf("Извлечено: %d\n", errorStack.pop());
    printf("Извлечено: %d\n", errorStack.pop());

    // Снова попытка извлечения из пустого стека
    printf("\n4. Снова попытка извлечения из пустого стека:\n");
    try {
        errorStack.pop();
        printf("Ошибка: исключение не было брошено!\n");
    } catch (const StackException& e) {
        printf("Поймано исключение: %s\n", e.what());
    }
}

// Функция для демонстрации копирования
void demonstrateCopying() {
    printf("\n=== Демонстрация копирования ===\n");

    Stack<int, 4> original;
    original.push(10);
    original.push(20);
    original.push(30);

    printf("Оригинальный стек:\n");
    original.display();

    // Копирующий конструктор
    Stack<int, 4> copy1 = original;
    printf("\nСкопированный стек (конструктор копирования):\n");
    copy1.display();

    // Оператор присваивания
    Stack<int, 4> copy2;
    copy2 = original;
    printf("\nСкопированный стек (оператор присваивания):\n");
    copy2.display();
```

```
    // Изменение копии не влияет на оригинал
    copy1.push(40);
    printf("\nПосле изменения копии:\n");
    printf("Оригинал:\n");
    original.display();
    printf("Копия:\n");
    copy1.display();
}

// Функция для демонстрации разных размеров
void demonstrateDifferentSizes() {
    printf("\n=== Демонстрация разных начальных размеров ===\n");

    // Стек с маленькой начальной емкостью
    Stack<int, 2> smallStack;
    printf("Маленький стек (емкость 2):\n");
    for (int i = 1; i <= 5; ++i) {
        smallStack.push(i);
    }
    smallStack.display();

    // Стек с большой начальной емкостью
    Stack<char, 10> bigStack;
    printf("\nБольшой стек (емкость 10):\n");
    for (char c = 'A'; c <= 'D'; ++c) {
        bigStack.push(c);
    }
    bigStack.display();
}

int main() {
    printf("=== Демонстрация шаблонного стека на динамическом массиве ===\n");

    // Демонстрация для int
    demonstrateIntStack();

    // Демонстрация для char
    demonstrateCharStack();

    // Демонстрация обработки ошибок
    demonstrateErrorHandling();

    // Демонстрация копирования
    demonstrateCopying();

    // Демонстрация разных размеров
    demonstrateDifferentSizes();

    printf("\n=== Все тесты завершены успешно ===\n");
    return 0;
}
```

Выводы:

В результате выполнения лабораторной работы были успешно освоены принципы создания и применения шаблонов в языке C++. На практическом примере разработки контейнерной структуры данных были изучены механизмы создания шаблонных функций и классов, позволяющих реализовывать универсальные и типонезависимые решения.

В ходе работы была разработана и протестирована шаблонная структура данных, демонстрирующая фундаментальный принцип generic programming. Особое внимание уделялось обеспечению корректной работы шаблона с различными типами данных, для чего проводилось тестирование на встроенных типах языка (int и char). Это позволило убедиться в универсальности и надежности реализованного решения.

Практическая значимость работы заключается в создании переиспользуемого компонента, который может быть применен для работы с различными типами данных без модификации исходного кода. Разработанная структура демонстрирует все преимущества шаблонного подхода: типобезопасность, эффективность и гибкость использования, что соответствует современным стандартам разработки на языке C++.

1. В чем смысл использования шаблонов?

Шаблоны позволяют создавать **обобщенный код**, который работает с разными типами данных без их явного указания. Основные преимущества:

- **Повторное использование кода** — одна реализация для многих типов
- **Type safety** — проверка типов на этапе компиляции
- **Исключение дублирования кода** — не нужно писать одинаковые функции/классы для разных типов

2. Каковы синтаксис/семантика шаблонов функций?

Синтаксис:

```
template <typename T>
T functionName(T parameter) {
    // тело функции
}
```

Семантика: компилятор генерирует конкретные версии функции для каждого используемого типа.

Пример:

```
template <typename T>
T max(T a, T b) {
    return (a > b) ? a : b;
}
```

3. Каковы синтаксис/семантика шаблонов классов?

Синтаксис:

```
template <typename T>
class ClassName {
```

```
    // члены класса, использующие T
};
```

Семантика: компилятор создает отдельный класс для каждого конкретного типа.

Пример:

```
template <typename T>
class Stack {
private:
    T* data;
public:
    void push(T value);
    T pop();
};
```

4. Что такое параметры шаблона функции?

Параметры шаблона — это типы или значения, которые передаются в шаблон. Бывают:

- **Параметры-типы** (`typename T`, `class T`)
- **Параметры-не типы** (целые числа, указатели, ссылки)

```
template <typename T, int size>  // T - параметр-тип, size - параметр-не тип
class Array { ... };
```

5. Перечислите основные свойства параметров шаблона функции.

- Могут быть **типами** или **значениями**
- Поддерживают **значения по умолчанию**
- Может быть **несколько параметров**

- Поддерживают **шаблонные параметры шаблонов**

6. Как записывать параметр шаблона?

```
template <typename T>          // параметр-тип
template <class T>             // альтернатива typename
template <int N>               // параметр-не тип (целое)
template <typename T = int>    // параметр со значением по умолчанию
```

7. Можно ли перегружать параметризованные функции?

Да, шаблонные функции можно перегружать:

- Шаблонную функцию с обычной функцией
- Несколько шаблонных функций между собой

```
template <typename T> void func(T a) {}
template <typename T> void func(T* a) {}  // перегрузка для указателей
void func(int a) {}                       // перегрузка обычной функцией
```

8. Перечислите основные свойства параметризованных классов.

- Могут иметь **статические члены** (для каждой специализации свои)
- Поддерживают **наследование**
- Могут содержать **вложенные классы**
- Поддерживают **специализацию**
- Могут иметь **дружественные функции**

9. Может ли быть пустым список параметров шаблона? Объясните.

Нет, список параметров шаблона не может быть пустым. Должен быть указан хотя бы один параметр, иначе компилятор выдаст ошибку.

10. Как вызвать параметризованную функцию без параметров?

Если тип не выводится из аргументов, нужно явно указать тип:

```
template <typename T>
T create() {
   return T();
}

// Вызов:
auto obj = create<int>();     // явное указание типа
```

11. Все ли компонентные функции параметризованного класса являются параметризованными?

Да, все функции-члены шаблонного класса автоматически становятся шаблонными с теми же параметрами, что и класс.

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

Зависит от объявления:

- Если объявлены внутри класса — **да**, становятся шаблонными
- Если объявлены снаружи — могут быть как шаблонными, так и обычными

```
template <typename T>
class MyClass {
   friend void friendFunc(MyClass<T> obj);  // шаблонная функция
};
```

13. Могут ли шаблоны классов содержать виртуальные компонентные функции?

Да, могут. Однако сама виртуальная функция не может быть шаблонной.

```
template <typename T>
class Base {
public:
   virtual void func() { ... }  // допустимо
   // template <typename U> virtual void wrong() {}  // ОШИБКА!
};
```

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

С использованием полной квалификации с параметрами шаблона:

```
template <typename T>
class MyClass {
public:
   void memberFunc();
};

// Определение вне класса:
template <typename T>
void MyClass<T>::memberFunc() {
   // реализация
}
```

Для специализированных версий:

```
template <>
void MyClass<int>::memberFunc() {
   // специализированная реализация для int
```

```
}
```