Лабораторная работа №5

по курсу «Высокоуровневое программирование» (2 семестр)

«Обобщённое программирование и шаблоны»

Оглавление

Цель: приобретение практических навыков и знаний по обобщённому программированию.

Задачи:

1. Изучить основы и принципы обобщённого программирования;
2. Познакомиться с шаблонами функций;
3. Научиться создавать универсальные функции;
4. Познакомиться с шаблонами классов;
5. Получение навыков работы с шаблонами типа и шаблонами значения;
6. Научиться реализовывать обобщённые контейнеры.

Содержание отчёта:

1. Титульный лист;
2. Цель, задачи работы;
3. Формулировка общего задания;
4. Блок – схемы основной программы и созданных подпрограмм;
5. Листинги пользовательских функций, классов и основной программы;
6. Результаты работы;
7. Выводы по работе в целом.

Основные теоретические сведения

Обобщённое программирование

Обобщённое программирование – это парадигма программирования, суть которой заключается в описании алгоритмов, подразумевающих работу с различными типами данных, **одной общей реализацией**. Обобщённое программирование в языке C++ представлено в виде шаблонов.

Шаблоны функций

Шаблон функций — это трафарет функции, который служит образцом для создания подобных ему функций. Главная идея — создание функций без указания точного типа некоторых или всех переменных. Обход передачи точного типа осуществляется созданием образного (несуществующего) типа и подстановкой его в параметры шаблона.

При вызове шаблона функции компилятор использует «трафарет» в качестве образца функции, заменяя тип параметра шаблона на фактический тип переменных, передаваемых в функцию. Использование шаблонов позволяет использовать множество функций, каждую под конкретный тип данных, имея при этом в коде только образ функции. Что значительно сокращает объем кода в программе, за счет исключения дублирования кода, а так же позволяет облегчить контроль за совпадением типов (в сравнении с написанием перегрузок отдельно под каждый тип).

Для того, чтобы создать шаблон функции, нужно над или прямо в заголовке нужной функции прописать template<typename T> (вместо «T», можно подставить любое другое имя типа) Тип «T» будет шаблонным типом, который мы можем писать в любом месте функции, а на этапе компиляции он будет заменён на реальный тип.

```
template<typename T>
T sum(T a, T b) {
	return a + b;
}
```

Чтобы вызвать шаблонную функцию, нужно после её имени в угловых скобках прописать тип, которым будет заменён шаблон T.

```
	int a = 5;
	int b = 6;

	cout << sum<int>(a, b) << endl;
```

Также, помимо шаблона типа, существует **шаблон значения**. Шаблон значения нужен реже, чем шаблон типа, но бывают моменты, когда он необходим. Допустим, необходимо передать в функцию массив. Можно передать массив в функцию как указатель на первый элемент с отдельной передачей размера массива, однако это не позволяет использовать диапазонный for. Можно передать весь массив по ссылке, либо по указателю.

```
void print(int (&arr)[5]) {}
```

Это позволит не передавать явно размер в функцию и использовать диапазонный цикл for. Но размер массива при такой передаче фиксированный.

Добавив к последнему варианту шаблон значения, можно будет работать с массивами разной длины, подставляя вместо V любое значение типа int.

```
template<int V>
void print(int (&arr)[V]) {
	for (int it : arr) {
		cout << it << endl;
	}
}
```

Вызов функции:

```
	const int len = 5;
	int a[len]{ 1, 2, 3, 4, 5 };
	print<len>(a);
```

Возможно использование и нескольких шаблонов одновременно:

```
template<typename T, int V>
void print(T (&arr)[V]) {
	for (T it : arr) {
		cout << it << endl;
	}
}
```

Пример применения:

```
	const int len = 5;
	int a[len]{ 1, 2, 3, 4, 5 };
	print<int, len>(a);
```

Шаблоны классов

Наравне с шаблонами функций, существуют и шаблоны классов. Они применяются, например, при создании списков, которые смогут хранить любые типы данных.

Используем готовый класс элемента списка:

```
template <class T>
class ItemList {
public:
	// сохраним имеющийся тип рабочего объекта
	typedef T objType;

	ItemList() = default; // конструктор по умолчанию

	// явный конструктор, принимающий значение
	explicit ItemList(T v) : m_value(v) {}

	// перегруженный конструктор, принимающий
	// значение и указатель на предыдущий элемент
	ItemList(T v, ItemList<T>* p_b) : m_value(v), m_back(p_b) {}

	// перегруженный конструктор, принимающий
	// значение и указатели на предыдущий и следующий элементы
	ItemList(T v, ItemList<T>* p_b, ItemList<T>* p_n) :
				m_value(v), m_back(p_b), m_next(p_n) {}

	// делегирующий конструктор копий, принимающий ссылку на константный объект
	ItemList(const ItemList<T>& it) : ItemList<T>(it.m_value, it.m_back,
it.m_next) {}

	// сеттеры и геттеры
	void set(T v) {
		m_value = v;
	}

	T get() {
		return m_value;
	}
```

```
        void setNext(ItemList<T>* p_n) {
                m_next = p_n;
        }

        ItemList<T>* getNext() {
                return m_next;
        }

        void setBack(ItemList<T>* p_b) {
                m_back = p_b;
        }

        ItemList<T>* getBack() {
                return m_back;
        }

private:
        T m_value{};
        ItemList<T>* m_next{};
        ItemList<T>* m_back{};
};
```

Рассмотрим создание шаблона класса и создание объекта этого класса. Для создания шаблона класса применяется конструкция типа: `template <class T>`

Она ничем не отличается от создания шаблона функций, за исключением смены слова typename на class, но это не существенно. Как и с шаблонами функций теперь T является шаблоном типа, который мы можем подставлять во все переменные, которые нам нужны. Точно также, мы можем создавать несколько шаблонов типа через запятую. Создание объекта класса:

```
ItemList<T> obj();
```

Реализация контейнера, который будет управлять элементами списка:

```
template <class T>
class MyList {
public:
        MyList() = default; // конструктор по умолчанию
```

```
explicit MyList(T *it) {  // явный конструктор
	initList(*it);
}

MyList(const MyList<T>& lst) = delete; // уберем конструктор копий

bool isEmpty() { // метод проверки на пустоту
	return m_start;
}

size_t len() { // метод получения размера списка
	return m_len;
}

// метод добавления элемента в конец списка
void add(T* it) {
	if (isEmpty()) {
		initList(*it);
	}
	else {
		m_end->getBack()->setNext(it);
		it->setNext(m_end);
		m_end->setBack(it);
		++m_len;
	}
}

// метод добавления элемента в начало списка
void pushStart(T* it) {
	if (isEmpty()) {
		initList(*it);
	}
	else {
		it->setNext(m_start);
		m_start = it;
		++m_len;
	}
}

// метод удаления элемента с конца
void delEnd() {
```

```
		if (isEmpty()) {
			return;
		}
		else if (len() == 1) {
			delete m_start;
			m_start = nullptr;
			delete m_end;
			m_end = nullptr;

			m_len = 0;
		}
		else {
			m_end->setBack(m_end->getBack()->getBack());
			delete m_end->getBack()->getNext();
			m_end->getBack()->setNext(m_end);
			--m_len;
		}
	}

	// метод удаления элемента с начала
	void delStart() {
		if (isEmpty()) {
			return;
		}
		else if (len() == 1) {
			delete m_start;
			m_start = nullptr;
			delete m_end;
			m_end = nullptr;

			m_len = 0;
		}
		else {
			m_start = m_start->getNext();
			delete m_start->getBack();
			m_start->setBack(nullptr);
			--m_len;
		}
	}

private:
```

```
        T* m_start{};
        T* m_end{};
        size_t m_len{};

        // закрытый метод инициализации списка
        void initList(T& it) {
                m_start = &it;
                m_end = new T(0, m_start);
                it.setNext(m_end);
                m_len = 1;
        }
};
```

Как помните в простом контейнере, где все элементы хранились в массиве (линейном участке памяти), последним элементом обозначается элемент, который располагался за границей массива, чтобы удобно было обходить циклом for. Но в списках, элементы располагаются не последовательно, поэтому последним элементом помещается нулевой указатель или нуль – терминатор объекта (объект, который выступает в качестве указателя на последний элемент списка). Такой подход более безопасный при проверке указателей на неравенство.

Итератор также может быть шаблонным классом, благодаря чему он станет универсальным и сможет принимать любые типы объектов.

```
template <class T>
class MyList {
public:

        template <class V>
        class IteratorList {
                // делаем внешний класс дружественным к подклассу
                // чтобы он имел доступ к нашим закрытым свойствам
                friend class MyList<V>;

        public:
                // пишем конструктор копирования
                // который будет производить инициализацию членов класса
                // тело конструктора будет пустым
```

```
        IteratorList<V>(const IteratorList<V>& it) : m_item(it.m_item) {}

        // перегружаем оператор сравнения
        bool operator==(const IteratorList<V>& it) const {
                return m_item == it.m_item;
        }

        // перегружаем оператор сравнения на не
        bool operator!=(const IteratorList<V>& it) const {
                return m_item != it.m_item;
        }

        // перегружаем оператор икремента
        IteratorList<V>& operator++() {
                m_item = m_item->getNext();

                return *this;
        }

        // перегружаем оператор разыменования указателя
        V& operator*() const {
                return *m_item;
        }

private:
        V* m_item{};

        // создаём закрытый конструктор инициализации членов класса
        explicit IteratorList(V* p) : m_item(p) {}
};
```

Теперь давайте добавим методы работы с итератором в наш контейнер:

```
typedef IteratorList<T> iterator;
typedef IteratorList<T> const_iterator;

// конструктор по умолчанию
MyList() = default;

// явный конструктор
explicit MyList(T *it) {
        initList(*it);
```

```
}

// возврат итератора на первый элемент
iterator begin() {
	return iterator(m_start);
}

// возврат итератора на за-последний элемент
iterator end() {
	return iterator(m_end);
}

// возврат константного итератора на первый элемент
const_iterator begin() const {
	return const_iterator(m_start);
}

// возврат константного итератора на за-последний элемент
const_iterator end() const {
	return const_iterator(m_end);
}

// метод очистки списка
void erase() {
	for (int i{}; i < m_len; ++i) {
		m_start = m_start->getNext();
		delete m_start->getBack();
	}

	delete m_end;

	m_start = m_end = nullptr;
}

~MyList() {
	erase();
}

// добавим дружественную функцию перегрузки оператора вывода списка на консоль
friend ostream& operator<<(ostream& out, const MyList<T>& lst) {
	for (ItemList<typename T::objType> it : lst) {
```

```
            cout << it << endl;
        }

        return out;
    }
```

Для работы диапазонного цикла, нужно создать переменную, в которую каждую итерацию будет помещаться значение из списка. Тип такой переменной будет: ItemList<typename T::objType> Ключевое слово auto автоматически на стадии компиляции определяет нужный тип переменной и подставляет его вместо auto.

Задание

Используя принципы обобщённого программирования создайте шаблонный класс для хранения данных на основе класса-контейнера из предыдущей лабораторной работы.

Замените все массивы сущностей в программе на пользовательский шаблонный класс-контейнер.

Для корректной работы шаблонного класса с разными пользовательскими типами необходимо, чтобы подставляемые при вызове типы поддерживали операции, используемые в шаблонном классе. Гарантировать это можно при помощи абстрактных классов.

Создание абстрактного класса с виртуальными функциями, которые необходимы классу-шаблону, и наследование от него пользовательских классов позволит однозначно определить, что пользовательский класс можно использовать с данным шаблоном.

Приложение 1

```
#include <iostream>
using namespace std;

template <class T>
class ItemList {
public:

	// сохраним имеющийся тип рабочего объекта
	typedef T objType;

	// конструктор по умолчанию
	ItemList() = default;

	// явный конструктор принимающий значение
	explicit ItemList(T v) : m_value(v) {}

	// перегруженный конструктор принимающий
	// значение и указатель на предидущий элемент
	ItemList(T v, ItemList<T>* p_b) : m_value(v), m_back(p_b) {}

	// перегруженный конструктор принимающий
	// значение и указатели на предидущий и следующий элементы
	ItemList(T v, ItemList<T>* p_b, ItemList<T>* p_n) : m_value(v), m_back(p_b),
m_next(p_n) {}

	// делегирующий конструктор копий
	// принимающий ссылку на константный объект
	ItemList(const ItemList<T>& it) : ItemList<T>(it.m_value, it.m_back,
it.m_next) {}

	// метод установки значения
	void set(T v) {
		m_value = v;
	}

	// метод получения значения
	T get() {
		return m_value;
	}
```

```
        // метод установки указателя на след. эл.
        void setNext(ItemList<T>* p_n) {
                m_next = p_n;
        }

        // метод получения указателя на след. эл.
        ItemList<T>* getNext() {
                return m_next;
        }

        // метод установки указателя на пред. эл.
        void setBack(ItemList<T>* p_b) {
                m_back = p_b;
        }

        // метод получения указателя на пред. эл.
        ItemList<T>* getBack() {
                return m_back;
        }

        // добавим дружественную функцию перегрузки
        // оператора вывода элемента на консоль
        friend ostream& operator<<(ostream& out, const ItemList<T>& it) {
                cout << it.m_value << endl;

                return out;
        }

private:
        T m_value{};
        ItemList<T>* m_next{};
        ItemList<T>* m_back{};
};


template <class T>
class MyList {
public:

        template <class V>
        class IteratorList {
```

```
        // делаем внешний класс дружественным к подклассу
        // чтобы он имел доступ к нашим закрытым свойствам
        friend class MyList<V>;

public:
        // пишем конструктор копирования
        // который будет производить инициализацию членов класса
        // тело конструктора будет пустым
        IteratorList<V>(const IteratorList<V>& it) : m_item(it.m_item) {}

        // перегружаем оператор сравнения
        bool operator==(const IteratorList<V>& it) const {
                return m_item == it.m_item;
        }

        // перегружаем оператор сравнения на не
        bool operator!=(const IteratorList<V>& it) const {
                return m_item != it.m_item;
        }

        // перегружаем оператор икремента
        IteratorList<V>& operator++() {
                m_item = m_item->getNext();

                return *this;
        }

        // перегружаем оператор разыменования указателя
        V& operator*() const {
                return *m_item;
        }

private:
        V* m_item{};

        // создаём закрытый конструктор инициализации членов класса
        explicit IteratorList(V* p) : m_item(p) {}
};

typedef IteratorList<T> iterator;
typedef IteratorList<T> const_iterator;
```

```
// конструктор по умолчанию
MyList() = default;

// явный конструктор
explicit MyList(T* it) {
	initList(*it);
}

// возврат итератора на первый элемент
iterator begin() {
	return iterator(m_start);
}

// возврат итератора на за-последний элемент
iterator end() {
	return iterator(m_end);
}

// возврат константного итератора на первый элемент
const_iterator begin() const {
	return const_iterator(m_start);
}

// возврат константного итератора на за-последний элемент
const_iterator end() const {
	return const_iterator(m_end);
}

// метод очистки списка
void erase() {
	for (int i{}; i < m_len; ++i) {
		m_start = m_start->getNext();
		delete m_start->getBack();
	}

	delete m_end;

	m_start = m_end = nullptr;

	m_len = 0;
```

```
}

~MyList() {
	erase();
}

// добавим дружественную функцию перегрузки оператора вывода списка на консоль
friend ostream& operator<<(ostream& out, const MyList<T>& lst) {

	if (lst.isEmpty()) {
		out << "List is empty" << endl;
		return out;
	}

	for (const ItemList<typename T::objType>& it : lst) {
		out << it << endl;
	}

	return out;
}

// не будем реализовывать конструктор копий
MyList(const MyList<T>& lst) = delete;

// метод проверки на пустоту
bool isEmpty() const {
	return !m_start;
}

// метод получения размера списка
size_t len() {
	return m_len;
}

// метод добавления элемента в конец списка
void add(T* it) {

	if (isEmpty()) {
		initList(*it);
	}
	else {
```

```
            m_end->getBack()->setNext(it);
            it->setNext(m_end);
            m_end->setBack(it);
            ++m_len;
        }
}

// метод добавления элемента в начало списка
void pushStart(T* it) {

        if (isEmpty()) {
            initList(*it);
        }
        else {
            it->setNext(m_start);
            m_start = it;
            ++m_len;
        }
}

// метод удаления эл. с конца
void delEnd() {

        if (isEmpty()) {
            return;
        }
        else if (len() == 1) {
            delete m_start;
            m_start = nullptr;

            delete m_end;
            m_end = nullptr;

            m_len = 0;
        }
        else {
            m_end->setBack(m_end->getBack()->getBack());

            delete m_end->getBack()->getNext();

            m_end->getBack()->setNext(m_end);
```

```
                --m_len;
        }
    }

    // метод удаления эл. с начала
    void delStart() {

        if (isEmpty()) {
            return;
        }
        else if (len() == 1) {
            delete m_start;
            m_start = nullptr;

            delete m_end;
            m_end = nullptr;

            m_len = 0;
        }
        else {
            m_start = m_start->getNext();

            delete m_start->getBack();

            m_start->setBack(nullptr);

            --m_len;
        }
    }

private:
    T* m_start{};
    T* m_end{};
    size_t m_len{};

    // приватный метод инициализации списка
    void initList(T& it) {
        m_start = &it;
        m_end = new T(0, m_start);
        it.setNext(m_end);
```

```
            m_len = 1;
        }
};

int main() {

        MyList<ItemList<int>> lst;

        lst.add(new ItemList<int>(25));
        lst.add(new ItemList<int>(26));
        lst.add(new ItemList<int>(27));
        lst.add(new ItemList<int>(28));

        cout << lst << endl;

        for (const auto& it : lst) {
            cout << it << endl;
        }

        return 0;
}
```