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;

Содержание отчёта:

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

В начало

Основные теоретические сведения

Обобщённое программирование

Обобщённое программирование – это такая парадигма программирования, которая заключается в описании алгоритмов таким образом, чтобы они могли работать с различными типами данных, но при этом реализация оставалась одной и той же. Обобщённое программирование в языке C++ представлено в виде шаблонов. Шаблон – это некий трафарет, по которому компилятор сам построит нужный участок кода для обработки данных. Шаблоны, также, как и указатели являются узким местом в языке. Поэтому, их следует использовать с особой осторожностью!

В начало

Шаблоны функций

В языке C++ шаблоны функций — это функции, которые служат образцом для создания других подобных функций. Главная идея — создание функций без указания точного типа(ов) некоторых или всех переменных. Для этого мы определяем функцию, указывая тип параметра шаблона, который используется вместо любого типа данных. После того, как мы создали функцию с типом параметра шаблона, мы фактически создали «трафарет функции».

При вызове шаблона функции компилятор использует «трафарет» в качестве образца функции, заменяя тип параметра шаблона на фактический тип переменных, передаваемых в функцию. Таким образом, мы можем создать 50 «оттенков» функции, используя всего лишь один шаблон.

Давайте напишем функцию, которая складывает два числа:

```
int sum(int a, int b) {
	return a + b;
}
```

Как видно, это совершенно обычная функция, которая складывает два числа типа int и возвращает получившийся целочисленный результат. А, что, если мы захотим сложить два дробных числа? Мы напишем перегрузку для этой функции:

```
double sum(double a, double b) {
	return a + b;
}
```

Всё бы ничего, но посмотрите внимательно на две эти функции. Их тела совсем не отличаются, а это дублирование кода, можно ли его избежать? В таких случаях на помощь и приходит обобщённое программирование, а именно – шаблоны функций. Для того, чтобы создать шаблон функции, нужно над, или прямо в заголовке нудной функции прописать template<typename T>, вместо T, можно подставить любое другое имя. Теперь T и будет нашим шаблонным типом, который мы можем писать и использовать, где угодно в нашей функции. Позже, этот тип будет заменён на реальный тип, который нам нужен:

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

Отлично, мы создали шаблонную функцию, в которую вместо T можно подставлять любой тип данных. Но, как её вызывать? Всё достаточно тривиально:

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

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

Как видно, чтобы вызвать шаблонную функцию, нужно после её имени в угловых скобках прописать тип, которым будет заменён шаблон T. Теперь с помощью данной функции мы можем складывать значения любых доступных типов.

Также, вместо шаблона типа, можно написать шаблон значения. Шаблон значения нужен реже, чем шаблон типа, но тоже бывают моменты, когда он необходим. Давайте рассмотрим

такой пример. Мы хотим передать в функцию массив, но мы помним, что в целях оптимизации массив в функцию передаётся, как указатель на первый элемент. Из этого следует, что нам нужно будет отдельно передавать размер массива, а также мы не сможем использовать диапазонный for. Но, к счастью, это можно исправить, для этого нужно передать весь массив по ссылке, либо по указателю. Например:

```
void print(int (&arr)[5]) {
	for (int it : arr) {
		cout << it << endl;
	}
}
```

Отлично, теперь мы можем не передавать явно размер в функцию и использовать диапазонный цикл for. Но, появилась другая проблема, теперь размер массива, передаваемый в функцию – фиксированный. Как быть? На помощь опять приходят шаблоны! А точнее, шаблон значения, давайте напишем его:

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

Как видно, теперь мы можем вместо V подставлять любое значение типа int и работать с массивами разной длины. Давайте вызовем эту функцию:

```
	const int len = 5;

	int a[len]{ 1, 2, 3, 4, 5 };

	print<len>(a);
```

Кроме того, мы можем создавать больше одного шаблонного параметра, давайте перепишем нашу функцию таким образом, чтобы она могла выводить массив любого типа:

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

Теперь в нашу функцию мы можем передавать массив любого типа и любой длинны, при этом сохраняя все его свойства и не расходуя память, красота! Давайте вызовем:

```
	const int len_1 = 5;
	const int len_2 = 10;

	int a[len_1]{ 1, 2, 3, 4, 5 };
	double b[len_2]{ 1.5, 2.3, 3.4, 4.5, 5.6, 6.7, 7.8, 8.9, 9.1, 10.11 };

	print<int, len_1>(a);

	print<double, len_2>(b);
```

В начало

Шаблоны классов

На ровне с шаблонами функций, существуют и шаблоны классов. И те и те по реализации почти не отличаются, но есть пара моментов, которые мы рассмотрим. Давайте напишем простой шаблонный список, который будет хранить любые типы данных. Для начала напишем класс элемента списка:

```
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;
	}

private:
	T m_value{};
```

```
        ItemList<T>* m_next{};
        ItemList<T>* m_back{};
};
```

В реализации данного класса применяются принципы написания кода, которые мы рассматривали в предыдущей работе, поэтому на них останавливаться мы не будем. Мы 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, происходит штатно:

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

В угловых скобках указывается параметр, на который нужно подменить шаблон. Во всём остальном это обычный класс, который имеет несколько перегруженных конструкторов, а также, поля с самим значением, указатель на предыдущий и следующе элементы списка.

Теперь давайте реализуем контейнер, который будет управлять элементами списка:

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

        // конструктор по умолчанию
        MyList() = default;

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

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

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

Мы реализовали простой контейнер. Здесь Вам нужно обратить внимание на шаблоны классов, как мы прописываем типы параметров методов, а также, как создаём экземпляры шаблонных типов и храним их.

Ещё один момент, на который вам стоит обратить внимание – это на последний элемент списка. Как помните в простом контейнере, где все элементы хранились в массиве (линейном участке памяти) мы ставили последним элементом элемент, который располагался за границей массива, дабы удобно было обходить его циклом for. Но здесь мы имеем структуру данных – списка, элементы которого располагаются не последовательно, какой же элемент считать последним элементом? На самом деле, решений подобной проблемы существует масса, Вы сами можете придумать себе одно из них. Как вариант использовать нулевой указатель, но мы сделали немного сложнее, мы создали нуль – терминатор объекта. Объект, который выступает в качестве указателя на последний элемент списка. Такой подход более безопасный при проверке указателей на неравенство.

Теперь реализуем итератор, с помощью которого будем передвигаться по списку. Итератор у нас будет также 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) {}
};
```

Этот итератор ничем не отличается от того, что мы писали для класса строк, единственное его отличие в том, что здесь мы использовали обобщённое программирование. Таким образом наш итератор стал универсальным. 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;
	}
```

Также в контейнер мы добавили метод очистки списка и перегрузили оператор вывода. Для того, чтобы вывод работал корректно, стоит в класс элемента списка также добавить перегрузку оператора вывода:

```
	// 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;
	}
```

Ещё стоит обратить внимание на реализацию диапазонного цикла:

```
		for (ItemList<typename T::objType> it : lst) {
			cout << it << endl;
		}
```

Слева нам нужно определить тип переменной, которой будет присваиваться значение на каждой итерации цикла, а справа прописать наш список. Т. к. элемент списка – это шаблонный класс, то на момент написания программы мы не знаем каким типом он будет инициализирован. Для этих целей мы создали отдельный typedef в теле класса ItemList, который будет создавать псевдоним для шаблона этого класса. Теперь нам нужно только обратиться к нему через оператор разрешения области видимости и подставить вместо типа ItemList. Зачем писать явно typename? Это ключевое слово явно указывает на то, что T является типом, а не каким ни будь значением переменной. Так компилятору будет проще создавать шаблоны.

Теперь наш список полностью готов и давайте посмотрим, как с ним работать:

```
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;
}
```

Как видим, всё прекрасно работает. Теперь пару слов про ключевое слово auto. Чтобы сработал наш диапазонный цикл, нужно создать переменную, в которую каждую итерацию будет помещаться значение из списка. Тип нашей переменной будет примерно такой: ItemList<typename T::objType>

Писать это весьма утомительно и не всегда наверняка понятно, правильно ли мы вывели тип. Для таких случаев в стандарте языка C++ появилось ключевое слово auto, которое автоматически на стадии компиляции выводит нужный тип переменной и подставляет его вместо auto. Поэтому крайне рекомендуется использовать его.

Также, стоит отметить, что копирование объектов таких структур данных, как списки и прочее, весьма затратная операция. В таких случаях нужно использовать константную ссылку. Это будет выглядеть как – то так:

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

Для ключевого слова auto, эта конструкция записывается следующим образом:

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

Ниже будет приведён весь код списка, который вы можете использовать, как шаблон для решения задач в этой Л/Р:

```
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;
    }
};
```

В начало

Общее задание

В следующем цикле Л/Р по высокоуровневому программированию вам будет предложено реализовать полноценную программу по управлению, каталогизации и обработки информации. Каждая программа будет представлять из себя логически законченную платформу по типу: «Автоматизированная система управления» и содержать в себе следующий функционал:

- Обработка базы данных
  - Создание / удаление / редактирования записей
  - Сортировка и фильтрация записей
- Управление пользователями
  - Создание
  - Удаление
  - Авторизация

Каждая программа будет снабжена интерактивным меню пользователя. В этой Л/Р вам будет предложено внедрить в свою программу обобщённое программирование, а именно – шаблоны. Переписав, тем самым, повторяющиеся куски кода, например – сортировка и фильтрация.

Задание для всех вариантов

На данном этапе ваша Л/Р должна представлять из себя следующий набор каталогов и файлов:

- Screens
  - Auth
    - Auth.h
    - Auth.cpp
  - *… - прочие папки с вашими экранами по типу Auth*
  - InterfaceScreen.h
  - InterfaceScreen.cpp
- Menu
  - AbstractItemMenu.h
  - AbstractItemMenu.cpp
  - ItemMenu.h
  - ItemMenu.cpp
  - Menu.h
  - Menu.cpp
- Models
  - User
    - User.h
    - User.cpp
  - *… - прочие папки с вашими сущностями по типу User*
- State
  - State.h
  - State.cpp
  - Store.h
  - Store.cpp
- main.cpp

Задача

Просмотрите свой код программы и найдите места, где функции отличаются только типами параметров, с которыми они работают. Перепишите эти функции через шаблоны. Как правило, эти функции являются функциями сортировки и фильтрации. Но, в вашей работе могут найтись и другие функции, которые отличаются только типом.

В начало

Контрольные вопросы

1. Что такое обобщённое программирование?
2. Для чего оно используется?
3. Как оно реализовано в языке C++?
4. Что такое шаблон функции?
5. Как строится шаблон функции?
6. Чем отличается шаблон типа от шаблона значения?
7. Приведите пример универсальной функции.
8. Что такое шаблон класса?
9. Как он реализуется?
10. Что такое обобщённые контейнеры?
11. Способы передачи статического массива в функцию.
12. Передача всего массива по указателю.
13. Передача всего массива по ссылке.
14. Как реализовать универсальный список?
15. Для чего используется ключевое слово: typename?
16. Как работает ключевое слово – auto?
17. Когда и где его можно использовать?

Список литературы

1. Курс лекций доцента кафедры ФН1-КФ Пчелинцевой Н.И.
2. Программирование на языке высокого уровня C/C++ [Электронный ресурс]: конспект лекций / – Электрон. текстовые данные. – М.: Московский государственный строительный университет, Ай Пи Эр Медиа, ЭБС АСВ, 2016. – 140 с. – Режим доступа: http://www.iprbookshop.ru/48037.

В начало