Лабораторная работа №4

по курсу «Высокоуровневое программирование» (2 семестр)

«Знакомство с контейнерами»

Оглавление

Цель: приобретение практических навыков и знаний по созданию и обработки классов – контейнеров данных.

Задачи:

1. Изучить понятие контейнера;
2. Научиться описывать простой контейнер класса;
3. Изучить написание элементов для контейнера;
4. Познакомиться с умными указателями;
5. Познакомиться с итераторами и научиться применять их;
6. Реализовать свою структуру контейнера.

Содержание отчёта:

1. Титульный лист;
2. Цель, задачи работы;
3. Формулировка общего задания;
4. UML – диаграмма реализованных классов;
5. Листинги заголовочных файлов реализованных классов (кроме MyString);
6. Результаты работы;
7. Выводы по работе в целом.

В начало

Основные теоретические сведения

Контейнеры

В жизни мы часто встречаемся с контейнерами, например: коробка с вещами, пачка бумаги, стаканчик с офисными принадлежностями, пластиковый контейнер с едой и т д. Контейнер – это какая – то ёмкость в которую можно что – то класть и воспринимать её, как что – то единое и целое. В ООП не могли не позаимствовать из реального мира эту полезную вещь. Таким образом, в программировании также появилось понятие контейнера. Контейнер в языке C++ — это класс, предназначенный для хранения и организации нескольких объектов определенного типа данных (пользовательских или фундаментальных). Существует много разных контейнерных классов, каждый из которых имеет свои преимущества, недостатки или ограничения в использовании. Безусловно, наиболее часто используемым контейнером в программировании является массив, который мы уже использовали во многих программах. Хотя в языке C++ есть стандартные обычные массивы, большинство программистов используют контейнерные классы-массивы: std::array или std::vector из-за преимуществ, которые они предоставляют. В отличие от стандартных массивов, контейнерные классы-массивы имеют возможность динамического изменения своего размера, когда элементы добавляются или удаляются. Это не только делает их более удобными, чем обычные массивы, но и безопаснее.

Обычно, функционал классов-контейнеров языка C++ следующий:

- Создание пустого контейнера (через конструктор);
- Добавление нового объекта в контейнер;
- Удаление объекта из контейнера;

- Просмотр количества объектов, находящихся на данный момент в контейнере;
- Очистка контейнера от всех объектов;
- Доступ к сохраненным объектам;
- Сортировка объектов/элементов (не всегда).

Иногда функционал контейнерных классов может быть не столь обширным, как это указано выше. Например, контейнерные классы-массивы часто не имеют функционала добавления/удаления объектов, так как они и так медленные, и разработчик просто не хочет увеличивать нагрузку.

Типом отношений в классах-контейнерах является «член чего-то». Например, элементы массива «являются членами» массива (принадлежат ему). Обратите внимание, мы здесь используем термин «член чего-то» не в смысле члена класса C++.

В начало

Типы контейнерных классов

Контейнерные классы обычно бывают двух типов:

- Контейнеры значения — это композиции, которые хранят копии объектов (и, следовательно, ответственные за создание/уничтожение этих копий).
- Контейнеры ссылки — это агрегации, которые хранят указатели или ссылки на другие объекты (и, следовательно, не ответственные за создание/уничтожение этих объектов).

В отличие от реальной жизни, когда контейнеры могут хранить любые типы объектов, которые в них помещают, в языке C++ контейнеры обычно содержат только один тип данных. Например, если у вас целочисленный массив, то он может содержать только целочисленные значения. В отличие от некоторых других языков, C++ не позволяет смешивать разные типы данных внутри одного контейнера. Если вам нужны контейнеры для хранения значений типов int и double, то вам придется написать два отдельных контейнера (или использовать шаблоны, о которых мы поговорим на соответствующем уроке). Несмотря на ограничения их использования, контейнеры чрезвычайно полезны, так как делают программирование проще, безопаснее и быстрее.

В начало

Создание и работа с контейнером

Очевидно, Вы слышали про такой класс – контейнер, как – string. Он предоставляет удобные инструменты по работе со строками. Для того, чтобы разобраться в работе контейнеров, мы попробуем сами написать, простой аналог класса string. Для начала начнём с простого класса, который и будет выступать в роле контейнера:

```
class MyString {

};
```

Что будет хранить класс MyString?, т к строка – это массив символов, то соответственно мы и будем хранить динамический массив символов. Также, нам нужно будет знать, сколько

элементов мы храним, поэтому добавим поле счётчик. На первых порах это всё, посмотрим, что получилось:

```
class MyString {
private:
	unsigned int m_len{};
	char* m_chars{};
};
```

Отлично, теперь нам нужно позаботиться о выходе за пределы массива. Если бы мы использовали обычный С – массив, то мы бы в программе постоянно проверяли его границы, чтобы не выйти за последний элемент. Но, по сколько мы пишем класс контейнер, мы на него возложим эту обязанность. Для этого мы добавим ещё одно поле, которое будет хранить указатель за последний элемент. Почему за последний? – всё просто, чтобы мы могли наш контейнер использовать в циклах, перебирая не индексы, а указатели. Тогда в условии мы будем идти по массиву, пока не зайдём за границу.

```
class MyString {
private:
	unsigned int m_len{};
	char* m_chars{};
	char* m_end{};
};
```

Теперь добавим несколько конструкторов для создания объекта нашего класса:

```
class MyString {
public:
	MyString() {
		m_chars = new char[++m_len]{ '\0' };
		m_end = m_chars + 1;
	}

	MyString(const char* str) {
		for (const char* it = str; it; ++it) {
			AddSymbol(*it);
		}
	}

	void AddSymbol(const char symbol) {
		char* m_tmp = m_chars;
		m_chars = new char[++m_len]{};

		for (int i{}; i < (m_len - 2); ++i) {
			m_chars[i] = m_tmp[i];
		}

		m_chars[m_len - 2] = symbol;
		m_chars[m_len - 1] = '\0';
		m_end = m_chars + m_len;

		delete[] m_tmp;
	}

private:
	unsigned int m_len{};
	char* m_chars{};
	char* m_end{};
};
```

Мы добавили два перегруженных конструктора, один из которых не принимает аргументов и создаёт пустую строку, а другой принимает на вход строку типа C и вызывая вспомогательный метод добавления элемента в массив создаёт новую строку.

Теперь, чтобы с нашим классом можно было бы работать, как с полноценной строкой, давайте перегрузим операторы ввода и вывода:

```
class MyString {
public:
	MyString() {
		m_chars = new char[++m_len]{ '\0' };
		m_end = m_chars + 1;
	}

	MyString(const char* str) {
		for (const char* it = str; *it; ++it) {
			AddSymbol(*it);
		}
	}

	void AddSymbol(const char symbol) {
		char* m_tmp = m_chars;
		m_chars = new char[++m_len]{};

		for (int i{}; i < (m_len - 2); ++i) {
			m_chars[i] = m_tmp[i];
		}

		m_chars[m_len - 2] = symbol;
		m_chars[m_len - 1] = '\0';
		m_end = m_chars + m_len;

		delete[] m_tmp;
	}

	friend ostream& operator<<(ostream& out, const MyString& str);

	friend istream& operator>>(istream& in, MyString& str);

private:
	unsigned int m_len{};
	char* m_chars{};
	char* m_end{};
};

ostream& operator<<(ostream& out, const MyString& str) {
	out << str.m_chars;

	return out;
}

istream& operator>>(istream& in, MyString& str) {
	char ch{};

	while (in.get(ch) && ch != '\n') {
		str.AddSymbol(ch);
	}

	return in;
}
```

В наших перегруженных функциях ввода / вывода, мы реализовали вывод и заполнение строки. Как видите, выглядит всё достаточно неплохо. Теперь давайте реализуем геттеры для возврата указателя на начало строки на конец:

```
	char* getBegin() {
		return m_chars;
	}

	char* getEnd() {
		return m_end;
	}
```

И напоследок, давайте добавим два перегруженных оператора: «+» и «+=», я удобной работы со строкой. Также, добавим функцию, которая будет очищать строку и дублировать её заданное кол – во раз:

```
class MyString {
public:
	MyString() {

		/*
			Конструктор без параметров,
			вызывает функцию инициализации
			строки
		*/

		initStr();
	}

	MyString(const char* str) : MyString() {

		/*
			Делегирующий конструктор, с одним параметром
			В теле вызывает метод добавления строки в объект
		*/

		addStr(str);
	}

	MyString(const char* str, int count) : MyString() {

		/*
			Делегирующий конструктор с двумя параметрами
			В теле добавляет одну и туже строчку
			Заданное кол - во раз в объект
		*/

		for (int i{}; i < count; ++i) {
			addStr(str);
		}
	}

	MyString(const char* str_1, const char* str_2) : MyString(str_1) {

		/*
			Делегирующий конструктор с двумя параметрами
			Создаёт новую строку из двух предидущих
		*/

		addStr(str_2);
	}

	MyString(const char* str, char ch) : MyString(str) {
```

```
		/*
			Делегирующий конструктор с двумя параметрами
			Создаёт новую строку из одной строки и символа
		*/

		addSymbol(ch);
	}

	MyString(const MyString& str) : MyString(str.m_chars) {
		/*
			Пустой делегирующий конструктор для создания новой строки
			из строки
			Также, такой конструктор называется - конструктором копирования
		*/
	}

	MyString(const MyString& str_1, const MyString &str_2) : MyString(str_1.m_chars,
str_2.m_chars) {
		/*
			Пустой делегирующий конструктор для создания новой строки
			из двух строк
		*/
	}

	~MyString() {
		/*
			Деструктор объекта
		*/

		delete[] m_chars;
	}

	void addStr(const char* str) {

		/*
			Метод, добавляющий строку в объект
		*/

		for (const char* it = str; *it; ++it) {
			addSymbol(*it);
		}
	}

	void addSymbol(const char symbol) {

		/*
			Метод, добавляющий символ в строку
		*/

		char* m_tmp = m_chars; // сохранение старой строки
		m_chars = new char[++m_len]{}; // создание новой строки

		// копирование старой строки в новую
		for (size_t i{}; i < (m_len - 2); ++i) {
			m_chars[i] = m_tmp[i];
		}

		// запись нового символа в строку
		m_chars[m_len - 2] = symbol;

		// добавление нуль - терминатора
		m_chars[m_len - 1] = '\0';

		// запись указателя на элемент после последнего
		m_end = m_chars + m_len;
```

```
        // удаление старой строки
        delete[] m_tmp;
}

char& operator[](size_t index) {

        /*
                Перегруженный оператор индекса
                Возвращает "зацикленный" элемент строки
        */

        return m_chars[index % m_len];
}

// возврат указателя на первый элемент
char* getBegin() {
        return m_chars;
}

// возврат указателя на последний элемент
char* getEnd() {
        return m_end;
}

// возврат длины строки
size_t getLen() {
        return m_len;
}

// очистка строки
void erase() {
        delete[] m_chars;

        initStr();
}

// дублирование строки
MyString duplicate(int count) {
        return MyString(m_chars, count);
}

// перегруженный оператор умножения для дублирования строки
MyString operator*(int count) {
        return duplicate(count);
}

// перегруженный оператор сложение строки и С - строки
MyString operator+(const char* str) {
        return MyString(m_chars, str);
}

// перегруженный оператор сложение строки и строки
MyString operator+(const MyString &str) {
        return MyString(*this, str);
}

// перегруженный оператор сложение строки и символа
MyString operator+(const char ch) {
        return MyString(m_chars, ch);
}

// перегруженный оператор добавления к строке С - строки
MyString& operator+=(const char* str) {
        addStr(str);
```

```
	return *this;
}

// перегруженный оператор добавления к строке строки
MyString& operator+=(const MyString &str) {
	addStr(str.m_chars);

	return *this;
}

// перегруженный оператор для дублирования строки в строке
MyString& operator*=(int count) {

	MyString tmp(*this);

	for (int i{}; i < count - 1; ++i) {
		addStr(tmp.m_chars);
	}

	return *this;
}

// перегруженный оператор добавления к строке символа
MyString& operator+=(const char ch) {
	addSymbol(ch);

	return *this;
}

// перегруженный оператор присваивания к строке строки
MyString& operator=(const MyString& str) {

	if (&str == this) {
		return *this;
	}

	erase();
	addStr(str.m_chars);

	return *this;
}

// перегруженный оператор присваивания строке С - строки
MyString& operator=(const char* str) {
	erase();
	addStr(str);

	return *this;
}

// перегруженный оператор присваивания строке символа
MyString& operator=(const char ch) {
	erase();
	addSymbol(ch);

	return *this;
}

// перегруженный оператор вывода строки
friend ostream& operator<<(ostream& out, const MyString& str);

// перегруженный оператор ввода строки
friend istream& operator>>(istream& in, MyString& str);
```

```
private:
	size_t m_len{};
	char* m_chars{};
	char* m_end{};

	// скрытый метод инициализации строки
	void initStr() {
		m_len = 1;
		m_chars = new char[m_len]{ '\0' };
		m_end = m_chars + 1;
	}
};

ostream& operator<<(ostream& out, const MyString& str) {
	out << str.m_chars;

	return out;
}

istream& operator>>(istream& in, MyString& str) {
	char ch{};

	while (in.get(ch) && ch != '\n') {
		str.addSymbol(ch);
	}

	return in;
}
```

Теперь готовый наш класс будет выглядеть таким образом. Как видите мы добавили несколько дополнительных конструкторов для удобного создания объектов. Также, мы вынесли инициализацию объектов в отдельные функции, которые вызываются в конструкторах.

Также, стоит обратить Ваше внимание на подобную запись:

```
MyString(const char* str) : MyString() {}
```

Это так называемый делегирующий конструктор – конструктор, который вызывает другой конструктор. Это сокращает кол – во кода и делает его более аккуратным. Но, будьте внимательны, второй конструктор вызывается не в теле первого, а в заголовке после двоеточия. Только так мы можем вызывать другие конструкторы! Если мы попытаемся вызвать второй конструктор в теле, то получим создание анонимного объекта, результат будет похожим на делегирующий конструктор, за тем исключением, что во – втором случае, мы просто так теряем ресурсы.

В начало

Умный указатель

Умный указатель – это класс – контейнер, который не только предоставляет и хранит адрес объекта, но также имеет конструктор и деструктор его. Такие указатели очень удобно использовать. С ним нам не нужно больше задумываться об освобождении памяти в нужный момент, его деструктор будет вызван автоматически компилятором.

Давайте напишем умный указатель для работы с динамическим, одномерным массивом типа int:

```
class PtrArr {
public:
	PtrArr() = default; // создания конструктора по умолчанию

	// создания явного конструктора
	// явный конструктор - это тот, который
	// нельзя использовать для неявного
	// преобразования
	explicit PtrArr(int* arr) {
		m_arr = arr;
	}

	// удаляем конструктор копирования
	PtrArr(const PtrArr& p_arr) = delete;

	// перегруженный оператор взятия индекса
	int& operator[](size_t index) {
		return m_arr[index];
	}

	// перегруженный оператор разименования объекта
	int& operator*() const {
		return *m_arr;
	}

	// перегруженный оператор возврата указателя
	int* operator->() const {
		return m_arr;
	}

	// деструктор
	~PtrArr() {
		delete[] m_arr;

		m_arr = nullptr;
	}

private:
	int* m_arr{};
};
```

Как видите, здесь нет ничего нового. Однако, стоит обратить внимание на строчку:

```
PtrArr(const PtrArr& p_arr) = delete;
```

Здесь мы удаляем конструктор копий ради безопасности нашего указателя. Если бы мы могли создать два указателя на одно и тоже значение, то получилось бы так, что при удалении одного указателя и обращения его в нулевой, во – втором объекте этого бы не происходило и мы бы обращались к «мусорной» памяти. Конечно, мы бы могли копировать само значение указателей, но это уже будет не умный указатель, а обычный контейнер.

В начало

Итераторы

Итерация / перемещение по элементам массива (или какой-нибудь другой структуры) является довольно распространенным действием в программировании. Мы уже рассматривали множество различных способов выполнения данной задачи, а именно: с использованием циклов и индексов (циклы for и while), с помощью указателей и адресной арифметики, а также с помощью циклов for с явным указанием диапазона.

Использование циклов с индексами в ситуациях, когда мы используем их только для доступа к элементам, требует написания большего количества кода, нежели могло бы быть.

При этом данный способ работает только в том случае, если контейнер (например, массив), содержащий данные, дает возможность прямого доступа к своим элементам (что делают массивы, но не делают некоторые другие типы контейнеров, например, списки).

Итератор — это объект, разработанный специально для перебора элементов контейнера (например, значений массива или символов в строке), обеспечивающий во время перемещения по элементам доступ к каждому из них.

Контейнер может предоставлять различные типы итераторов. Например, контейнер на основе массива может предлагать прямой итератор, который проходится по массиву в прямом порядке, и реверсивный итератор, который проходится по массиву в обратном порядке.

После того, как итератор соответствующего типа создан, программист может использовать интерфейс, предоставляемый данным итератором, для перемещения по элементам контейнера или доступа к его элементам, не беспокоясь при этом о том, какой тип перебора элементов задействован или каким образом в контейнере хранятся данные. И, поскольку итераторы в языке C++ обычно используют один и тот же интерфейс как для перемещения по элементам контейнера (оператор ++ для перехода к следующему элементу), так и для доступа (оператор * для доступа к текущему элементу) к ним, итерации можно выполнять по разнообразным типам контейнеров, используя последовательный метод.

Простейший пример итератора — это указатель, который (используя адресную арифметику) работает с последовательно расположенными элементами данных. Также, в качестве итератора мы можем использовать умный указатель, но, хорошей практикой считается написать отдельный итератор для взаимодействия с контейнерами. В стандартной библиотеке C++ есть уже реализованные итераторы и контейнеры, которые используют «шаблоны» для унификации кода, работать с шаблонами мы будем на следующей Л/Р.

В начало

Итераторы и циклы for с явным указанием диапазона

Все типы данных, которые имеют методы begin() и end() или используются с std::begin() и std::end(), могут быть задействованы в циклах for с явным указанием диапазона:

```
const int len = 5;

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

for (int it : arr) {
	cout << it << endl;
}
```

Эта форма цикла for создана специально для перебора различных коллекций объектов и является аналогом подобной записи:

```
for (int i{}; i < len; ++i) {
	cout << arr[i] << endl;
}
```

Согласитесь, что перва форма намного удобнее. Но в таком виде, как она используется в первом примере, она не работает с присваиванием значений массиву. Что делать? Использовать ссылку:

```
const int len = 5;

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

for (int &it : arr) {
	cin >> it;
}
```

Теперь в нашем цикле мы можем инициализировать массив.

На самом деле, циклы for с явным указанием диапазона для осуществления итерации незаметно обращаются к вызовам функций begin() и end(). Тип данных std::array также имеет в своем арсенале методы begin() и end(), а значит и его мы можем использовать в циклах for с явным указанием диапазона. Массивы C-style с фиксированным размером также можно использовать с функциями std::begin() и std::end(). Однако с динамическими массивами данный способ не работает, так как для них не существует функции std::end() (из-за того, что отсутствует информация о длине массива).

Давайте вспомним про нами созданный класс MyString, мы не сможем использовать его в этом цикле, потому – что он тоже не имеет методов работы с итераторами. Давайте исправим это. Начнём с того, что создадим свой итератор, который будет работать с нашим классом строк. Реализуем его, как подкласс класса MyString:

```
class StrIterator {
	// делаем внешний класс дружественным к подклассу
	// чтобы он имел доступ к нашим закрытым свойствам
	friend class MyString;

public:
	// пишем конструктор копирования
	// который будет производить инициализацию членов класса
	// тело конструктора будет пустым
	StrIterator(const StrIterator& it) : m_symbol(it.m_symbol) {}

	// перегружаем оператор сравнения
	bool operator==(const StrIterator& it) const {
		return m_symbol == it.m_symbol;
	}

	// перегружаем оператор сравнения на не
	bool operator!=(const StrIterator& it) const {
		return m_symbol != it.m_symbol;
	}

	// перегружаем оператор икремента
	StrIterator& operator++() {
		++m_symbol;

		return *this;
	}

	// перегружаем оператор разыменования указателя
	char& operator*() const {
		return *m_symbol;
	}

private:
```

```
        char* m_symbol{};

        // создаём закрытый конструктор инициализации членов класса
        StrIterator(char* p) : m_symbol(p) {}
};
```

Как видите в создании итератора нет ничего сложного, однако несколько моментов заслуживают отдельного внимания. В конструкторах итератора мы использовали инициализацию полей класса. До этого, в других классах мы не выполняли данную инициализацию, а просто в теле присваивали значение полям. Теперь, нужно привыкать к написанию именно такой формы инициализации переменных:

```
StrIterator(char* p) : m_symbol(p)
```

Через двоеточие мы указываем имя поля и вызываем конструктор у этого поля передавая туда параметр, которым его необходимо инициализировать. Полей может быть сколько угодно, все они могут быть инициализированы таким способом, через запятую.

Таким же образом (через двоеточие), мы создавали делегирующий конструктор, который вызывал другой конструктор, здесь нужно сказать, что делегирующий конструктор, не может инициализировать поля класса, также, конструктор, который инициализирует поля класса не может быть делегирующем.

```
friend class MyString;
```

Ещё обратите внимание на эту строчку, мы можем делать дружественными классу не только функции, но и другие классы, тогда все методы дружественного класса будут иметь доступ к сокрытым членам класса.

Теперь, когда итератор написан, самое время переписать наш класс string, чтобы он работал с итератором. Для начала, добавим несколько typedef для создания контейнера, который будет иметь общий интерфейс совместимости со всеми библиотеками языка C++:

```
typedef StrIterator iterator;
typedef StrIterator const_iterator;
```

Как видим, мы просто создали псевдонимы имён для нашего класса итератора, они нужны для совместимости. Необходимо заметить, что вторая строчка должна создавать псевдоним класса для константного итератора, т е итератора, который будет возвращать не изменяемые объекты. Как правила в таких случаях итераторы являются шаблонными классами, но мы в целях упрощения кода и так, как не разбирали темы шаблонов, пока опустим этот итератор и сделаем его обычным. Теперь, перепишем наши методы получения указателя на первый и за – последний элементы нашей строки. Они будут возвращать итераторы:

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

// 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_chars);
}
```

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

Вот так теперь будут выглядеть наши методы. Наш класс снова готов к работе, давайте попробуем пройтись по строке диапазонным циклом for:

```
	MyString str("Hi!");

	for (char ch : str) {
		cout << ch << endl;
	}
```

Всё работает. Теперь ниже будет приведён весь код нашего класса по работе со строками, чтобы Вы его могли использовать в выполнении Л/Р:

```
class MyString {
public:

	class StrIterator {
		// делаем внешний класс дружественным к подклассу
		// чтобы он имел доступ к нашим закрытым свойствам
		friend class MyString;

	public:
		// пишем конструктор копирования
		// который будет производить инициализацию членов класса
		// тело конструктора будет пустым
		StrIterator(const StrIterator& it) : m_symbol(it.m_symbol) {}

		// перегружаем оператор сравнения
		bool operator==(const StrIterator& it) const {
			return m_symbol == it.m_symbol;
		}

		// перегружаем оператор сравнения на не
		bool operator!=(const StrIterator& it) const {
			return m_symbol != it.m_symbol;
		}

		// перегружаем оператор икремента
		StrIterator& operator++() {
			++m_symbol;

			return *this;
		}

		// перегружаем оператор разыменования указателя
		char& operator*() const {
			return *m_symbol;
		}

	private:
		char* m_symbol{};

		// создаём закрытый конструктор инициализации членов класса
		StrIterator(char* p) : m_symbol(p) {}
	};


	typedef StrIterator iterator;
	typedef StrIterator const_iterator;
```

```
	MyString() {

		/*
			Конструктор без параметров,
			вызывает функцию инициализации
			строки
		*/

		initStr();
	}

	MyString(const char* str) : MyString() {

		/*
			Делегирующий конструктор, с одним параметром
			В теле вызывает метод добавления строки в объект
		*/

		addStr(str);
	}

	MyString(const char* str, int count) : MyString() {

		/*
			Делегирующий конструктор с двумя параметрами
			В теле добавляет одну и туже строчку
			Заданное кол - во раз в объект
		*/

		for (int i{}; i < count; ++i) {
			addStr(str);
		}
	}

	MyString(const char* str_1, const char* str_2) : MyString(str_1) {

		/*
			Делегирующий конструктор с двумя параметрами
			Создаёт новую строку из двух предидущих
		*/

		addStr(str_2);
	}

	MyString(const char* str, char ch) : MyString(str) {

		/*
			Делегирующий конструктор с двумя параметрами
			Создаёт новую строку из одной строки и символа
		*/

		addSymbol(ch);
	}

	MyString(const MyString& str) : MyString(str.m_chars) {
		/*
			Пустой делегирующий конструктор для создания новой строки
			из строки
			Также, такой конструктор называется - конструктором копирования
		*/
	}

	MyString(const MyString& str_1, const MyString &str_2) : MyString(str_1.m_chars,
str_2.m_chars) {
		/*
```

```
                Пустой делегирующий конструктор для создания новой строки
                из двух строк
        */
}

~MyString() {
        /*
                Деструктор объекта
        */

        delete[] m_chars;
}

void addStr(const char* str) {

        /*
                Метод, добавляющий строку в объект
        */

        for (const char* it = str; *it; ++it) {
                addSymbol(*it);
        }
}

void addSymbol(const char symbol) {

        /*
                Метод, добавляющий символ в строку
        */

        char* m_tmp = m_chars; // сохранение старой строки
        m_chars = new char[++m_len]{}; // создание новой строки

        // копирование старой строки в новую
        for (size_t i{}; i < (m_len - 2); ++i) {
                m_chars[i] = m_tmp[i];
        }

        // запись нового символа в строку
        m_chars[m_len - 2] = symbol;

        // добавление нуль - терминатора
        m_chars[m_len - 1] = '\0';

        // запись указателя на элемент после последнего
        m_end = m_chars + m_len;

        // удаление старой строки
        delete[] m_tmp;
}

char& operator[](size_t index) {

        /*
                Перегруженный оператор индекса
                Возвращает "зацикленный" элемент строки
        */

        return m_chars[index % m_len];
}

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

```
// 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_chars);
}

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

// возврат длины строки
size_t getLen() {
	return m_len;
}

// очистка строки
void erase() {
	delete[] m_chars;

	initStr();
}

// дублирование строки
MyString duplicate(int count) {
	return MyString(m_chars, count);
}

// перегруженный оператор умножения для дублирования строки
MyString operator*(int count) {
	return duplicate(count);
}

// перегруженный оператор сложение строки и С - строки
MyString operator+(const char* str) {
	return MyString(m_chars, str);
}

// перегруженный оператор сложение строки и строки
MyString operator+(const MyString &str) {
	return MyString(*this, str);
}

// перегруженный оператор сложение строки и символа
MyString operator+(const char ch) {
	return MyString(m_chars, ch);
}

// перегруженный оператор добавления к строке С - строки
MyString& operator+=(const char* str) {
	addStr(str);

	return *this;
}

// перегруженный оператор добавления к строке строки
MyString& operator+=(const MyString &str) {
	addStr(str.m_chars);

	return *this;
```

```
	}

	// перегруженный оператор для дублирования строки в строке
	MyString& operator*=(int count) {

		MyString tmp(*this);

		for (int i{}; i < count - 1; ++i) {
			addStr(tmp.m_chars);
		}

		return *this;
	}

	// перегруженный оператор добавления к строке символа
	MyString& operator+=(const char ch) {
		addSymbol(ch);

		return *this;
	}

	// перегруженный оператор присваивания к строке строки
	MyString& operator=(const MyString& str) {

		if (&str == this) {
			return *this;
		}

		erase();
		addStr(str.m_chars);

		return *this;
	}

	// перегруженный оператор присваивания строке С - строки
	MyString& operator=(const char* str) {
		erase();
		addStr(str);

		return *this;
	}

	// перегруженный оператор присваивания строке символа
	MyString& operator=(const char ch) {
		erase();
		addSymbol(ch);

		return *this;
	}

	// перегруженный оператор вывода строки
	friend ostream& operator<<(ostream& out, const MyString& str);

	// перегруженный оператор ввода строки
	friend istream& operator>>(istream& in, MyString& str);

private:
	size_t m_len{};
	char* m_chars{};
	char* m_end{};

	// скрытый метод инициализации строки
	void initStr() {
		m_len = 1;
		m_chars = new char[m_len]{ '\0' };
```

```
            m_end = m_chars + 1;
        }
};

ostream& operator<<(ostream& out, const MyString& str) {
        out << str.m_chars;

        return out;
}

istream& operator>>(istream& in, MyString& str) {
        char ch{};

        while (in.get(ch) && ch != '\n') {
            str.addSymbol(ch);
        }

        return in;
}
```

В начало

Список инициализации

До этого вам приходилось инициализировать поля класса в конструкторе через присваивание. Но такая форма инициализации является не всегда приемлемой. Рассмотрим простой пример:

```
class Book {
private:
        const size_t m_count_sheet{};
        const string m_title{};
        const string m_author{};
};
```

У нас есть класс книги. В этом классе определено три поля: кол-во страниц книги, её название и автор. Все эти поля объявлены, как константные значения, что логично. Давайте попробуем написать конструктор для этого класса:

```
Book(string title, string author, size_t count) {
            m_title = title;
            m_author = author;
            m_count_sheet = count;
        }
```

Всё бы хорошо и логично, но данный пример обречён не скомпилироваться. Почему? Всё очень просто. Мы объявили поля константами, эти константы были неявно инициализированы значениями по умолчанию. А в конструкторе мы пытаемся выполнить присваивание этим константам нового значения, что невозможно по определению. Как быть? На помощь приходит список инициализации конструктора. Что это такое? Список инициализации – это единственный верный способ инициализации полей класса. Синтаксис выглядит таким образом:

```
Book(string title, string author, size_t count)
            : m_title(title), m_author(author), m_count_sheet(count) {}
```

Что мы здесь накодили? Прямо в заголовке конструктора мы пишем двоеточие и перечисляем через запятую наши поля класса. В каждом поле мы вызываем его инициализатор и передаём туда значение, которое пришло к нам в параметрах конструктора. Синтаксис похож на наследование классов, только здесь отсутствует

модификатор доступа. Впредь, рекомендуется использовать именно инициализацию в конструкторе, а не присваивание.

Порядок выполнения в списке инициализации

Удивительно, но переменные в списке инициализации не инициализируются в том порядке, в котором они указаны. Вместо этого они инициализируются в том порядке, в котором объявлены в классе, поэтому следует соблюдать следующие рекомендации:

- Не инициализируйте переменные-члены таким образом, чтобы они зависели от других переменных-членов, которые инициализируются первыми (другими словами, убедитесь, что все ваши переменные-члены правильно инициализируются, даже если порядок в списке инициализации отличается).
- Инициализируйте переменные в списке инициализации в том порядке, в котором они объявлены в классе.
- Нельзя использовать список инициализации и вызов делегирующего конструктора в одном месте. Если конструктор вызывает другой конструктор, то он не может инициализировать члены класса.

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 реализовать такое меню, а также создать файл с функциями – затычками для тестирования меню. В каждом варианте задания будет описана функциональность программы, которую вы должны реализовать в меню. Продуктивной и интересной работы!

Задание для всех вариантов

Сейчас у вас должны быть следующие наработки, которые были реализованы на прошлых Л/Р:

- Интерактивное меню
- Сущности данных
- Функции обработки этих данных
  - Добавление
  - Удаление
  - Редактирование
  - Сортировка (по одному логически главному полю)
  - Фильтрация (по одному логически главному полю)

Начиная с этой работы вы начнёте связывать эти наработки в цельное приложение. Для начала вам нужно придумать, где и как будут храниться данные. Курс Л/Р не ограничивает вас в собственной реализации работы с ними, но даёт рекомендации, как должно это выглядеть.

Задача 1

Создайте библиотеку (папку в корне вашего проекта, а в ней файлы), которая будет являться неким глобальным хранилищем данных вашей программы, назовите её Store. В ней создайте структуру данных State (напишите два файла State.h и State.cpp), в заголовочном файле этой структуры пропишите поля, в которых будут храниться наборы ваших сущностей (по одному полю для каждого типа набора) для реализации наборов можете использовать библиотеку STL, а конкретно тип: vector. Причём данные, которые будет хранить вектор должны быть указателями на объекты сущностей.

Задача 2

После того, как вы стали уверенно чувствовать себя при работе с классами, мы можем упростить нашу архитектуру программы. По факту наши сущности – это просто объекты с данными, которые не хранят в себе никаких методов по работе с ними. В таких случаях не применяется ООП, а используется обычная структура. Поэтому перепишите те сущности, в которых не используется ООП на обычные структуры, удалив геттеры/сеттеры, а также убрав модификаторы доступа. (если вы использовали для полей префикс _m, в структурах его следует убрать). Стоит отметить, что если в сущностях используется наследование, то это уже ООП, такие сущности следует реализовывать только через классы. Также, если сеттер отвечает за обработку корректного значения ввода, то также оставляем сущность классом.

Задача 3

Перепишите все поля, которые использовали с – строки, на использование класса string (можно из STL библиотеки, а можно из методических указаний).

Задача 4

Создайте библиотеку экранов (папку в корне своего проекта под названием Screens, в которой будут лежать файлы). В этой библиотеки создайте под директории, в которых уже будут находиться сами экраны. Каждый экран будет представлять из себя класс, который наследуется от интерфейса InterfaceScreen (который будет лежать в корне папки с экранами). В интерфейсе будет две виртуальные чистые функции: int start(int) и void renderMain() const. Первая будет являться точкой входа в экран, а вторая будет отвечать за базовую его отрисовку в консоли. Каждый экран будет переопределять эти методы по своему усмотрению. Помимо переопределённых методов в экранах, будут и частные методы, которые относятся непосредственно к самому функционалу экрана, например: показ списка пользователей, сортировка, добавление, удаление и т д. Экран – это по сути логическая область вашей программы, которая будет отвечать за тот, или иной раздел функциональности. Разбейте вашу программу на логические блоки (по сути, она уже почти разбита в индивидуальном задании) – экраны и реализуйте в них частную функциональность. Пока, в качестве заглушки, включите в интерфейсный экран библиотеку Store и используйте объекты State, чтобы управлять данными.

В следующей Л/Р мы заставим нашу программу полностью работать. На данном этапе структура вашего проекта должна быть примерно следующей:

Projects

- 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

Примечание по поводу листинга кода

В листинг следует добавить заголовочные файлы: State.h, InterfaceScreen.h, а также заголовочные файлы ваших индивидуальных экранов. Библиотеки, саму реализацию, а также изменённые участки кода, следует добавлять по-своему усмотрение. При добавлении изменённых участков следует делать ссылку по типу:

Изменения в файле: Models/User/User.cpp

```
class User {

	...

	void print() {
		cout << "print" << endl;
	}

	...

};
```

Вариант №1
Основаня задача:

Вам будет предложено написать программу – «Автоматизированная система института». Которая будет включать следующий функционал:

- Ведение базы студентов и преподавателей (разные базы)
    - Создание / удаление / редактирование записей
    - Сортировка / фильтрация
- Ведение базы предметов
- Возможность авторизации
- Создание файлов – отчётов и сохранения состояния

В начало

Вариант №2
Основная задача:

Вам будет предложено написать программу – «Автоматизированная система школы». Которая будет включать следующий функционал:

- Ведение базы школьников и учителей (разные базы)
  - Создание / удаление / редактирование записей
  - Сортировка / фильтрация
- Ведение базы предметов
- Возможность авторизации
- Создание файлов – отчётов и сохранения состояния

В начало

Вариант №3
Основная задача:

Вам будет предложено написать программу – «Автоматизированная система парикмахерской». Которая будет включать следующий функционал:

- Ведение базы сотрудников и клиентов (разные базы)
  - Создание / удаление / редактирование записей
  - Сортировка / фильтрация
- Ведение базы услуг
- Возможность авторизации
- Создание файлов – отчётов и сохранения состояния

В начало

Вариант №4
Основная задача:

Вам будет предложено написать программу – «Автоматизированная система авто – салона». Которая будет включать следующий функционал:

- Ведение базы сотрудников и клиентов (разные базы)
  - Создание / удаление / редактирование записей
  - Сортировка / фильтрация
- Ведение базы автомобилей и услуг
- Возможность авторизации
- Создание файлов – отчётов и сохранения состояния

В начало

Вариант №5
Основная задача:

Вам будет предложено написать программу – «Автоматизированная система банка». Которая будет включать следующий функционал:

- Ведение базы сотрудников и клиентов (разные базы)
  - Создание / удаление / редактирование записей
  - Сортировка / фильтрация
- Ведение базы счетов
- Возможность авторизации
- Создание файлов – отчётов и сохранения состояния

В начало

Вариант №6
Основная задача:

Вам будет предложено написать программу – «Автоматизированная система мастерской». Которая будет включать следующий функционал:

- Ведение базы сотрудников и клиентов (разные базы)
  - Создание / удаление / редактирование записей
  - Сортировка / фильтрация
- Ведение базы услуг
- Возможность авторизации
- Создание файлов – отчётов и сохранения состояния

В начало

Вариант №7
Основная задача:

Вам будет предложено написать программу – «Автоматизированная система кинотеатра». Которая будет включать следующий функционал:

- Ведение базы сотрудников и клиентов
  - Создание / удаление / редактирование записей
  - Сортировка / фильтрация
- Ведение базы сеансов, фильмов, залов (разные базы)
- Возможность авторизации
- Создание файлов – отчётов и сохранения состояния

В начало

Вариант №8
Основная задача:

Вам будет предложено написать программу – «Автоматизированная система диалога (чат бот)». Которая будет включать следующий функционал:

- Ведение базы пользователей
  - Создание / удаление / редактирование записей
  - Сортировка / фильтрация
- Ведение базы диалогов, тем, интересов и напоминаний
- Возможность авторизации
- Создание файлов – отчётов и сохранения состояния

В начало

Вариант №9
Основная задача:

Вам будет предложено написать программу – «Автоматизированная система поликлиники». Которая будет включать следующий функционал:

- Ведение базы врачей и пациентов (разные базы)
  - Создание / удаление / редактирование записей
  - Сортировка / фильтрация
- Ведение базы приёмов и записей в амбулаторных картах
- Возможность авторизации
- Создание файлов – отчётов и сохранения состояния

В начало

Вариант №10
Основная задача:

Вам будет предложено написать программу – «Автоматизированная система почты». Которая будет включать следующий функционал:

- Ведение базы сотрудников клиентов (разные базы)
  - Создание / удаление / редактирование записей
  - Сортировка / фильтрация
- Ведение базы отправлений
- Возможность авторизации
- Создание файлов – отчётов и сохранения состояния

В начало

Вариант №11
Основная задача:

Вам будет предложено написать программу – «Автоматизированная система гостиницы». Которая будет включать следующий функционал:

- Ведение базы сотрудников и посетителей
  - Создание / удаление / редактирование записей
  - Сортировка / фильтрация
- Ведение базы номеров и услуг (предусмотреть напоминание о выезде)
- Возможность авторизации
- Создание файлов – отчётов и сохранения состояния

В начало

Вариант №12
Основная задача:

Вам будет предложено написать программу – «Автоматизированная система завода». Которая будет включать следующий функционал:

- Ведение базы сотрудников и поставщиков (разные базы)
  - Создание / удаление / редактирование записей
  - Сортировка / фильтрация
- Ведение базы производства и продаж
- Возможность авторизации
- Создание файлов – отчётов и сохранения состояния

В начало

Вариант №13
Основная задача:

Вам будет предложено написать программу – «Автоматизированная система управления музыкальной коллекцией». Которая будет включать следующий функционал:

- Ведение базы исполнителей и пользователей
  - Создание / удаление / редактирование записей
  - Сортировка / фильтрация
- Ведение базы треков, жанров (реализовать подборку)
- Возможность авторизации
- Создание файлов – отчётов и сохранения состояния

В начало

Вариант №14
Основная задача:

Вам будет предложено написать программу – «Автоматизированная система электронной почты». Которая будет включать следующий функционал:

- Ведение базы администраторов и пользователей
  - Создание / удаление / редактирование записей
  - Сортировка / фильтрация
- Ведение базы отправлений (реализовать возможность отправки и получения писем)
- Возможность авторизации
- Создание файлов – отчётов и сохранения состояния

В начало

Вариант №15
Основная задача:

Вам будет предложено написать программу – «Автоматизированная система управления фотографиями». Которая будет включать следующий функционал:

- Ведение базы фотографий и альбомов
  - Создание / удаление / редактирование записей
  - Сортировка / фильтрация
- Ведение базы отмеченных людей на фото
- Возможность авторизации
- Создание файлов – отчётов и сохранения состояния

В начало

Вариант №16
Основная задача:

Вам будет предложено написать программу – «Автоматизированная система вокзала». Которая будет включать следующий функционал:

- Ведение базы сотрудников, машинистов и пассажиров (разные базы)
  - Создание / удаление / редактирование записей
  - Сортировка / фильтрация
- Ведение базы направлений поездов и самих составов (возможность продажи билетов)
- Возможность авторизации
- Создание файлов – отчётов и сохранения состояния

В начало

Вариант №17
Основная задача:

Вам будет предложено написать программу – «Автоматизированная система магазина». Которая будет включать следующий функционал:

- Ведение базы сотрудников и покупателей (разные базы)
  - Создание / удаление / редактирование записей
  - Сортировка / фильтрация
- Ведение базы товаров и продаж (продажа товара)
- Возможность авторизации
- Создание файлов – отчётов и сохранения состояния

В начало

Вариант №18
Основная задача:

Вам будет предложено написать программу – «Автоматизированная система космодрома». Которая будет включать следующий функционал:

- Ведение базы сотрудников и астронавтов
  - Создание / удаление / редактирование записей
  - Сортировка / фильтрация
- Ведение базы космических судов и запланированных запусков (реализовать возможность запуска – виртуально!)
- Возможность авторизации
- Создание файлов – отчётов и сохранения состояния

В начало

Вариант №19
Основная задача:

Вам будет предложено написать программу – «Автоматизированная система библиотеки». Которая будет включать следующий функционал:

- Ведение базы сотрудников и читателей
  - Создание / удаление / редактирование записей
  - Сортировка / фильтрация
- Ведение базы книг, журналов, авторов
- Возможность авторизации
- Создание файлов – отчётов и сохранения состояния

В начало

Вариант №20
Основная задача:

Вам будет предложено написать программу – «Автоматизированная система аэропорта». Которая будет включать следующий функционал:

- Ведение базы сотрудников, пилотов и пассажиров (разные базы)
  - Создание / удаление / редактирование записей
  - Сортировка / фильтрация
- Ведение базы рейсов, самолётов и продаж билетов
- Возможность авторизации
- Создание файлов – отчётов и сохранения состояния

В начало

Контрольные вопросы

1. Что такое контейнер?
2. Приведите примеры из реальной жизни.
3. Для чего используется контейнер в программировании?
4. Как реализуется класс контейнер?
5. Какие существуют типы контейнерных классов?
6. Что представляет из себя элемент контейнера?
7. Какие элементы могут быть у контейнера?
8. Что такое делегирующий конструктор?
9. Для чего он используется?
10. Что такое умный указатель?
11. Как реализовать умный указатель?
12. Что такое конструктор копирования?
13. Что такое итераторы?
14. Как реализовать итератор?
15. Как работает диапазонный цикл for?
16. Какие операции можно с его помощью выполнять?
17. К каким данным можно применить этот вид цикла?

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