Лабораторная работа №1

по курсу «Высокоуровневое программирование» (2 семестр)

«Классы и объекты в C++»

Оглавление

Цель: приобретение практических навыков и основ объектно-ориентированного программирования, средствами языка C++.

Задачи:

1. Изучение основных концепций ООП;
2. Познакомиться с типом данных – «class»;
3. Познакомиться с операторами, предназначенными для работы с классами;
4. Научиться создавать объекты классов;
5. Изучить работу с методами класса;
6. Познакомиться с инициализацией пользовательских объектов.

Содержание отчёта:

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

В начало

Основные теоретические сведения:

Концепция ООП

Объектно-ориентированное программирование (ООП), концепция программирования в которой основной упор делается на данные, а не на алгоритмы. ООП – это попытка связать реальные сущности из мира с бизнес – логикой программы. Свойство настоящих объектов и их поведения переносятся в код, тем самым позволяя удобно организовать работу с ним.

Объектно-ориентированный подход строится на трёх основных концепциях, он не заключён в «магическом» ключевом слове – «класс». ООП – это ещё одна парадигма программирования, коих очень много, какую выбирать для своей задачи решать только вам. Сейчас мы будем учиться работать именно с этим подходом и для этого нам понадобиться разобраться в ***трёх главных принципах ООП***, а именно:

1. ***Инкапсуляция*** – принцип, при котором логически связанные куски алгоритма представляют из себя подобие «чёрного ящика». Пользователь этого алгоритма должен получать только интерфейс взаимодействия с ним, но не саму реализацию. Примеров инкапсуляции в реальном мире можно найти массу, как вариант примитивная поездка в автобусе. Вы, как пассажир, заходите в автобус через двери, садитесь на сиденье и выходите на своей остановке. Для Вас интерфейсом взаимодействия являются двери и кресла в салоне (поручни, окна и т д). При этом Вам совсем не обязательно знать, как именно работает автобус, как он открывает двери и от чего питается. Всё это не нужно для того, чтобы Вы просто переместились из точки А в точку В. Так же и с кодом. Представим у нас есть почтовый клиент, который принимает письма и выводит их на экран. Также есть другая программа, которая читает вслух письма, выведенные на экран. Не зная реализации почтового ящика, но зная, что его интерфейс позволяет задать нужное письмо и положение экрана, Вы можете подключить к нему другую программу, интерфейс которой подразумевает

выбор нужного экрана, с которого производить чтение. Таким образом из двух программ мы получили новую третью, которая реализована на связи первых двух, но, что самое ценное для нас в этом примере это осознать, что мы совсем не читали код первых двух программ, так как нам это не нужно.

Таким образом Ваш код должен делиться на сущности, которые будут «чёрными ящиками» с внешними интерфейсами взаимодействия. А внешние интерфейсы будут связываться между собой образуя бизнес – логику программы.

2. ***Наследование*** – ещё один принцип ООП. Его часто путают с третьим принципом – «Полиморфизм», но в полиморфизм он входит и является его частью и всё же это более узкое понятие. Наследование – это способность одних объектов иметь свойства других объектов расширяя их своими конкретными свойствами. Таким образом новая сущность строится на предыдущей, но добавляет в неё что – то своё, уникальное для новой сущности.

   Примеров, как всегда, масса, ведь всё ООП было взято из реального мира. Представим такой **объект**, как – Студент. У студента есть **свойства** и **поведения**. К свойствам можно отнести: цвет глаз, пол, возраст, оценки за семестр, текущий семестр и т д, к поведению относится: желание учиться, гулять, питаться, что – то видеть, писать и т д. А теперь, возьмём сущность – Человек. Какие у неё есть свойства и поведения? Как минимум – возраст, цвет глаз, пол, а к поведению можно отнести: желание питаться, возможность гулять, видеть, слушать и т д. А теперь сравните эти две сущности. Мы видим, что сущность студента, как бы вмещает в себя все свойства и поведения сущности человека, но при этом дополняя её своими: писать, получать оценки, текущий семестр, оценки и т д. Мы получили определённую **иерархию сущностей** – это и есть наследование. Мы не должны вмешивать в сущность студента свойства человека, а должны

создать две независимые сущности и указать, что одна сущность расширяет другую, что и называется связать их наследованием. Но стоит заметить, наследование – это не великая панацея, его следует использовать с умом, а не пытаться унаследовать автомобиль от холодильника, потому что и тот и тот наследуется от сущности дверная ручка. Более тесно работать с наследованием Вы будете во второй Л/Р.

3. ***Полиморфизм*** – это самое сложное понятие из этих трёх принципов. Полиморфизм – это возможность использования сущности – потомка в алгоритмах, которые были написаны для сущности – предка. Возьмём простой пример: у нас есть программа, которая печатает имя и возраст человека. У нас есть сущность человек, которая имеет в свойствах возраст и имя. Также, у нас есть сущность банковского работника, которая наследует от сущности человека свойства возраста и имени. Таким образом в программу для печати имени мы можем подсунуть вместо сущности человека, сущность банковского работника и программа в обоих случаях должна справиться со своей задачей. Из этой концепции выходят ещё две, которые часто применяются для унификации кода, это – **абстракция и интерфейс**. С ними мы также познакомимся в следующей Л/Р.

Разобрав концепции ООП, можно перейти к основным понятиям из этой области. Язык программирования C++ является объектно-ориентированным, но правильнее было бы сказать так: C++ - язык с поддержкой парадигмы ООП разработки. Существуют и другие языки, поддерживающие ООП: GoLang, Java, JS, Python и пр. На них можно **удобно** писать программы по этой концепции, т.к. они предоставляют встроенные средства для подобной реализации. Язык C, к примеру, не является объектно-ориентированным, но на нём тоже можно реализовать ООП, правда выглядеть это будет немного иначе, да и сложность написания программ увеличится в разы. Вы должны понять, что ООП – это всего – лишь стиль программирования, который можно выразить на любом языке.

Рассмотрим основные понятия ООП в контексте конкретного языка – C++.

Подобные классам конструкции мы уже писали с помощью ключевого слова – «struct». Класс не сильно отличается от структуры в языке программирования C++. Поэтому для понятия их схожестей и различий мы постепенно перейдем от написания структур к классам.

Повторим создание структур в C++:

```
struct Human {
	char* name{};
	int age{};
	Gender gender{};
};
```

В структуре Человек заложены три переменные. Первая переменная является указателем на тип char, по имени переменной видно, что в ней будет храниться имя. Следующая переменная типа int для возраста. Третья переменная представляет пользовательский тип данных, который определяет пол человека, это (enum class).

Структуры являются пользовательским типом данных. Соответственно, мы можем создать переменную типа struct.

```
Human Ivan{};
```

Мы создали переменную типа – Human, присвоили ей имя – Ivan, не указав аргументов в линейном инициализаторе (фигурные скобочки {}). При отсутствии аргументов в инициализаторе переменным присваивается значение по умолчанию. При передаче аргументов получаем заполненный объект (заполняется компилятором).

Таким образом будет создан новый объект структуры с пустыми переменными внутри.

Класс – механизм для создания объектов.

Описание структуры или класса не подразумевает под собой выделение памяти для объекта. Память выделяется непосредственно во время создания переменной нового типа данных.

Приведём пример: в мастерской вы хотите собрать машинку на радиоуправлении. Для этого Вы составляете чертеж будущей машинки, определяете её свойства (цвет, форма, габариты и прочее), а также её поведения (движения вперёд-назад, гудок, рулевые повороты). Если вы возьмёте пульт управления и попробуете применить его на чертеже, то ничего не произойдет: чертеж не поедет, не издаст сигнал, не проявит поведение машинки. Так как чертёж - лишь образ будущего объекта, инструкция к его созданию, однако он сам не является объектом. Чертёж определяет (описывает) основные свойства и поведения объекта, а объект инициализирует (реализует) их. По одному чертежу можно собрать множество машинок, разной формы, цвета, звукового сигнала, и их будет объединять общий набор свойств и поведений.

Исходя из этого примера Вы должны сделать вывод, что класс всего лишь описывает, как данные будут организованы в вашей программе и что их будет связывать. А объекты уже будут представлять из себя экземпляры этих классов с инициализированными свойствами. Под понятием класс подразумевается сам механизм создания объектов, т. е. структура и класс в данном контексте одно и тоже. Таким образом создавая новый класс – вы определяете новый тип данных, переменную которого Вы позже сможете создать.

Экземпляр класса (объект) – если сам класс – это трафарет, то экземпляр класса – это реальный объект, в котором инициализированы все свойства класса.

Поля - Переменные, которые находятся в классе представляют его свойства. Для того, чтобы чётко понимать, что эти переменные являются полями класса, существует, общепринятое правило называть эти переменные начиная с

префикса «m», сокращение от member – член, участник класса. Таким образом, объявление переменной в классе будет иметь синтаксис: `int m_age{};`

Методы – помимо свойств класс имеет ряд функций, которые находятся в теле самого класса и описывают поведение объекта. (сам синтаксис описания функций ничем не отличается). Чтобы вызвать такую функцию нужно обратиться к ней через точку, словно это поле объекта: `Ivan.print();` (синтаксис ничем не отличается от вызова обычной функции, за тем исключением, что сперва мы указываем объект, у которого мы вызываем эту функцию) Методы имеют доступ к полям класса. При вызове они используют данные объекта. Это можно увидеть на примере следующего кода:

```
struct Human {
	char* m_name{};
	int m_age{};
	Gender m_gender{};

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

int main() {
	Human Ivan{ new char[] {"Ivan"}, 19, Gender::male }; // передача аргументов
	Human Dima{ new char[] {"Dima"}, 19, Gender::male };

	Ivan.print(); // вывод - Ivan
	Dima.print(); // вывод – Dima
			// выводится заданное значение поля
	return 0;
}
```

Концепция интерфейсов: зная, что у класса человек, есть метод принт, мы можем его вызывать для всех экземпляров этого класса и его дочерних классов не обращая внимания на то, для каких данных мы его вызываем.

Конструктор класса – тема конструкторов очень обширная, и чтобы её понять, нужно много времени. Сейчас мы рассмотрим лишь основные особенности конструкторов классов. **Конструктор класса** – функция, которая вызывается *автоматически* при создании объекта, имеет такое же имя, как и имя класса, в котором она объявлена, а также *не имеет* типа возвращаемого значения. *Конструкторы служат для начальной инициализации полей данных класса.* Не путать с линейными инициализаторами.

```
struct Human {
    char* m_name{};
    int m_age{};
    Gender m_gender{};

    // Создаём конструктор класса Human
    Human() {
          m_age = 25;
    }
};

int main() {

    // Создаём объект класса Human
    // Для этого вызываем конструктор класса, словно это обычная функция
    Human Ivan = Human();

    return 0;
}
```

В данном примере написан конструктор класса Human, который задаёт начальное значение переменной класса m_age, равное 25. Теперь обратите внимание на создание объекта при использовании конструктора. Мы пишем тип, имя будущей переменной и через равно снова имя класса с круглыми скобками, словно мы вызываем функцию. Таким образом компилятор вызывает конструктор класса, который мы написали в структуре, и присваивает начальное значение переменной возраста.

Раз конструктор — это обычная функция, значит в неё можно что – то передать? – совершенно точно, рассмотрим, как это можно сделать ещё на одном примере:

```
struct Human {
	char* m_name{};
	int m_age{};
	Gender m_gender{};

	// Создаём конструктор класса Human
	// который принимает несколько аргументов
	Human(char* const name, const int age, const Gender& gender) {
		m_name = name;
		m_age = age;
		m_gender = gender;
	}
};

int main() {

	// Создаём объект класса Human
	// Для этого вызываем конструктор класса, словно это обычная функция
	// в качестве параметров мы передаём основные данные класса
	Human Ivan = Human(new char[] {"Ivan"}, 20, Gender::male);

	return 0;
}
```

В данном примере мы передаём в конструктор значения, которые необходимо присвоить переменным класса.

Деструктор класса – наравне с конструктором класса существует противоположное ему понятие - деструктор. Деструктор вызывает либо компилятор, в конце использования, либо мы, если наш объект, который мы хотим уничтожить, находится в свободной памяти. Деструктор точно также, как и конструктор, *не может возвращать ничего из функции*, поэтому тип возвращаемого значения для него не указывается. *Имеет название самого класса*, но перед ним содержит символ тильда «~». И он *не умеет работать с аргументами*. Деструкторы бывают полезны, когда в объекте идёт динамическое

выделение памяти, либо же открытие потока. Компилятор не сможет автоматически закрыть поток, или освободить память, поэтому мы должны делать это явно в деструкторе. В качестве примера рассмотрим такой код:

```
struct Human {
	char* m_name{};
	int m_age{};
	Gender m_gender{};

	// Создаём конструктор класса Human
	// который принимает несколько аргументов
	Human(const char* name, const int age, const Gender& gender) {

		// в конструкторе мы динамически
		// выделяем память под имя человека
		m_name = new char[256] {};

		strcpy(m_name, name); // копируем переданный аргумент в поле класса
				      // (необходимо при работе с динамической памятью)
		m_age = age;
		m_gender = gender;
	}
	// определяем деструктор объекта и в нём
	// удаляем динамически выделенную память под имя человека
	~Human() {
		delete[] m_name;
	}
};

int main() {

	// обратите внимание, вместо создания нового объекта
	// мы передаём строку в конструктор
	Human Ivan = Human("Ivan", 20, Gender::male);

	// динамически создаём новый объект структуры человека, вызывая конструктор
	Human* Dima = new Human("Dima", 10, Gender::male);

	// перед удалением объекта компилятор вызовет его деструктор
	delete Dima;
```

```
    return 0;
}
```

Как видно, из примера, деструктор для объекта Ivan будет вызван при выходе из функции main, а деструктор для экземпляра класса Dima, будет вызван при вызове команды delete.

Стоит отметить, что по умолчанию для класса уже определены конструктор и деструктор, они пустые и нужны лишь для создания единого интерфейса. Явно указывая две эти сущности, мы переопределяем (обновляем) их.

Модификаторы доступа – одно из отличий классов от структур, поэтому для их применения заменим в объявлении struct на class.

```
class Human {
    char* m_name{};
    int m_age{};
    Gender m_gender{};
    /* реализация класса – конструкторы, деструкторы, методы*/
};

int main() {

    Human Ivan = Human("Ivan", 20, Gender::male);

    Ivan.print();

    return 0;
}
```

Как видите, после изменения ключевого слова struct на class визуально ничего не поменялось, но стоит попробовать скомпилировать данную программу, и компилятор выдаст сообщение об ошибке “невозможно вызвать член класса print у объекта Ivan”. Дело модификаторах доступа - ключевых славах, которые в пределах класса задают уровни доступа к элементам объектов.

Всего ***модификаторов доступа существует три***:

4. ***Public*** – после этого ключевого слова все поля и методы класса будут считаться *доступными извне* в любом файле или функции, которая импортирует наш класс.
5. ***Private*** – после этого ключевого слова все поля и методы класса, будут считаться *закрытыми для доступа извне.* При этом к ним всё ещё можно обратиться в текущем классе, но вне класса вызвать их у экземпляра этого класса не выйдет.
6. ***Protected*** – очень специфичный модификатор доступа. Все методы, поля у класса, которые идут после него, будут доступны только для текущего класса или его потомков. Этот модификатор мы рассмотрим еще раз после наследования.

Возвращаясь к отличию структуры от класса, нужно сказать, что по умолчанию в структуре все поля и методы подписаны модификатором public, поэтому мы с лёгкостью могли вызвать любой член класса у нашего объекта. В классе же, все поля по стандарту имеют модификатор private, поэтому мы не можем получить доступ к методу – print. Для того, чтобы это исправить, нужно прописать явно модификатор свободного доступа, тогда наш пример изменится следующим образом:

```
class Human {

	// модификатор доступа
public:
	char* m_name{};
	int m_age{};
	Gender m_gender{};
	/* реализация класса – конструкторы, деструкторы, методы*/
};

int main() {

	Human Ivan = Human("Ivan", 20, Gender::male);

	Ivan.print();
```

```
	return 0;
}
```

Компоненты-функции – проявление одного из принципов ООП – инкапсуляции - специальные функции для изменения и получения закрытых значений. Они позволяют удобным способом предотвратить нежелательное обращение с полями класса. (например, присвоение отрицательного возраста)

```
class Human {

// закрываем поля класса
private:

	char* m_name{};
	unsigned int m_age{};
	Gender m_gender{};

// создаём открытые функции
public:

	// функция для задания возраста - сеттер
	void setAge(unsigned int age) {
		if (age <= 200) {
			m_age = age;
		}
	}

	// функция для получения возраста - геттер
	unsigned int getAge() {
		return m_age;
	}

	Human(const char* name, const int age, const Gender& gender) {
		m_name = new char[256] {};
		strcpy(m_name, name);
		m_age = age;
		m_gender = gender;
	}
```

```
	~Human() {
		delete[] m_name;
	}

	void print() {
		cout << m_name << endl;
		cout << m_age << endl;
	}
};

int main() {

	Human Ivan = Human("Ivan", 20, Gender::male);

	Ivan.setAge(121212); // не сработает
	Ivan.setAge(21); // значение будет изменено

	Ivan.print();

	return 0;
}
```

Сеттер – setAge, функции-сеттеры всегда начинаются со слова set, оно не является ключевым и может быть заменено на любое другое слово, но оно принято всеми программистами и является примером хорошего тона. С помощью этой функции мы можем задать новый возраст, а в самой функции проверить, является ли новый возраст допустимым значением для переменной m_age.

Для получения значения переменной мы используем Геттер – getAge, он служит для получения значения переменной, ведь она закрыта и по-другому мы не сможем обратиться к её значению. get – опять же не является ключевым словом, но написание его общепринято.

Указатели на компоненты-функции: тип_возвр_значения (имя_класса :: *имя_указателя_на_функцию) (специф_параметров_функции);

В начало

Правила хорошего тона при написании классов

В итоге, отличие структуры от класса заключается лишь в том, что первая имеет открытые поля по умолчанию, а класс – закрытые. Во всём остальном это два одинаковых способа реализации ООП. Общепринято использование классов, так как они имеют расширенный потенциал.

Также принято разделять прототип класса от реализации его функций. А именно – прототипы классов, как и прототипы функций следует прописывать в заголовочных файлах, а реализацию методов класса в файлах исходного кода. Организуем написанный код так, как написали выше:

Заголовочный файл .h:

```
#ifndef HEADER_H
#define HEADER_H

class Human {

public:
	enum class Gender {
		male,
		female
	};

	// прототип конструктора
	Human(const char* name, const int age, const Gender& gender);

	// прототип деструктора
	~Human();

	// прототип сеттера для возраста
	void setAge(unsigned int age);

	// прототип геттера для возраста
	unsigned int getAge();

	// прототип функции вывода
	void print();
```

```
private:
	char* m_name{};
	unsigned int m_age{};
	Gender m_gender{};

};
#endif
```

Содержание файла исходного кода .cpp:

```
#include <iostream>

#include "Header.h"


using namespace std;


Human::Human(const char* name, const int age, const Gender& gender) {
	m_name = new char[256]{};

	strcpy(m_name, name);
	m_age = age;
	m_gender = gender;
}

Human::~Human() {
	delete m_name;
}

unsigned int Human::getAge() {
	return m_age;
}

void Human::setAge(unsigned int age) {
	if (age < 200) {
		m_age = age;
	}
}
```

```
void Human::print() {
	cout << m_name << endl;
	cout << m_age << endl;
}

int main() {

	Human Ivan = Human("Ivan", 20, Human::Gender::male);

	Ivan.setAge(121212); // не сработает
	Ivan.setAge(21); // значение будет изменено

	Ivan.print();

	return 0;
}
```

Стоит обратить особое внимание на реализацию методов. Чтобы получить доступ к реализации мы должны написать тип возвращаемого значения, затем имя класса, затем идёт оператор разрешения области видимости, затем имя нужного нам метода, потом в скобках параметры и в фигурных скобках реализация нужного метода.

```
void Human::setAge(unsigned int age) {
	if (age < 200) {
		m_age = age;
	}
}
```

В начало

Создание обёрток для типов

```
// создание псевдонима (обёртки) типа через typedef
typedef unsigned int money;

// создание псевдонима (обёртки) типа через alias (C++ 11)
using age = unsigned int;
```

В первом случае мы пишем ключевое слово – typedef, затем тип, для которого хотим создать псевдоним, а затем имя этого псевдонима.

Во – втором случае мы пишем using, затем имя псевдонима, а после знака равно тип, для которого создаём обёртку.

А дальше используем как обычный тип данных:

```
int main() {

	money mny = 125;

	age my_age = 50;

	cout << mny << endl;

	cout << my_age << endl;

	return 0;
}
```

Обратите внимание, что cout автоматически преобразует наш псевдоним к изначальному типу, это означает, что мы всего – лишь создали другое имя для типа данных.

В начало

Задание первой лабораторной работы

Задание для первой работы одинаково для всех вариантов:

1. Реализовать самостоятельно все примеры из методического пособия.
2. Разобраться с кодом проекта «MyMenu», высланный преподавателем.
3. В пункте 1 добавить по желанию свою реализацию (например, посчитать корень 25, как в примере на стр. 21, вывести псевдографику и т.д.).
4. Ответить на контрольные вопросы.

Некоторые пояснения для создания меню:

В отдельной директории MyMenu Вашего проекта (проект один на весь семестр) создайте два файла: .cpp и .h. и пространство имён, название которого будет состоять из ваших инициалов: ФИО. В пространстве имён напишите класс меню. Прототип класса должен быть вынесен в заголовочный файл, а его реализация в файл исходного кода.

Поля класса MyMenu должны включать:

- название меню
- кол-во пунктов в меню
- последний выбор пользователя
- массив указателей на пункты меню

Основные методы:

- Метод печати меню на консоль.
- Метод с циклом запроса ввода у пользователя.
- Метод запуска функции в зависимости от ввода пользователя.
- Конструктор, принимающий в себя массив нужных функций меню и название меню.
- Деструктор, который будет удалять пункты меню, освобождая память.
- Геттер- и Сеттер-методы для получения доступа к полям класса.

По такому же принципу в отдельном файле написать класс MenuItem для пункта меню, состоящий из названия пункта и функции (реализовать через typedef внутри класса), которая ему соответствует. Класс должен иметь конструктор, деструктор, сеттеры, геттеры, методы запуска и печати. Пространство имен использовать то же, что и в первом классе.

Перед написанием основной программы main, добавьте в реализации методов информационные сообщения, по которым вы будете отслеживать работу программы. Создайте объект класса меню используя конструктор. Передайте в качестве аргумента массив пунктов с произвольными заданиями прошлого семестра (или функциями-заглушками). Проконтролируйте работу методов при помощи информационных сообщений. Пример работы меню:

```
Конструктор с параметрами для объекта вызван 0x7ffc70921f58 класса MyMenu
Меню входа в систему
1. Вычислить корень из 25
2. Поздороваться с пользователем
3. Изобразить картинку псевдографикой
0. Exit
   Select >> 1

Запуск пункта номер 1:
Корень из 25 - 5.

Меню входа в систему
1. Вычислить корень из 25
2. Поздороваться с пользователем
3. Изобразить картинку псевдографикой
0. Exit
   Select >> 2

Запуск пункта номер 1:
Введите Ваше имя: Петр
Здравствуйте, Петр!

Меню входа в систему
1. Вычислить корень из 25
2. Поздороваться с пользователем
```

```
3. Изобразить картинку псевдографикой
0. Exit
   Select >> 0
Деструктор вызван для объекта 0x7ffc70921f58 класса MyMenu
D:\repos_labs_course1\VP2_LR\x64\Debug\main.exe (процесс 17352) завершил
работу с кодом 0.
```

В начало

Контрольные вопросы

1. Что такое ООП?
2. Какие ещё существуют парадигмы программирования?
3. Назовите основные концепции ООП.
4. Что такое класс? Отличие класса от структуры.
5. Что такое экземпляр класса?
6. Что такое поле класса? Как к нему обратиться?
7. Что такое метод класса? Как его вызвать?
8. Что такое конструктор и деструктор? Для чего они нужны?
9. Что такое модификаторы доступа? Для чего они используются?
10. Какими способами можно создать класс?
11. Какими способами можно создать объект класса?
12. Что такое псевдоним типа, как его создать?
13. Как правильно реализовать класс на языке программирования C++?
14. Приведите пример программы, где нужно использовать ООП.

В начало