# Лабораторная работа №1

# по курсу «Высокоуровневое программирование» (2 семестр)

# «КЛАССЫ И ОБЪЕКТЫ В С++»

Оглавление


Основные теоретические сведения....3

Концепция ООП....3

Правила хорошего тона при написании классов....13

Создание обёрток для типов....14

Указатели на компоненты-функции:....15

Методические указания....15

Задания....18

Вариант №1....18

Вариант №2....18

Вариант №3....18

Вариант №4....18

Вариант №5....18

Вариант №6....18

Вариант №7....18

Вариант №8....19

Вариант №9....19

Вариант №10....19

Вариант №11....19

Вариант №12....19

Вариант №13....19

Вариант №14....19

Вариант №15....19

Контрольные вопросы....20

Список литературы....20

**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.

**Основное содержание работы:**

Написать программу, в которой создаются и разрушаются объекты, определенного пользователем класса. Выполнить исследование вызовов конструкторов и деструкторов.

**Порядок выполнения работы:**

1. Определить пользовательский класс в соответствии с вариантом задания (смотри приложение).
2. Определить в классе следующие конструкторы: без параметров, с параметрами, копирования.
3. Определить в классе деструктор.
4. Определить в классе компоненты-функции для просмотра и установки полей данных.
5. Написать демонстрационную программу, в которой создаются и разрушаются объекты пользовательского класса и каждый вызов конструктора и деструктора сопровождается выдачей соответствующего сообщения (какой объект какой конструктор или деструктор вызвал).
6. Определить указатель на экземпляр класса.
7. Определить указатель на компоненту-функцию.
8. Показать в программе использование указателя на объект и указателя на компоненту-функцию.
9. Создать массив объектов, размещенной как в статической области памяти, так и в динамической.
10. Продемонстрировать содержание массивов.

**Содержание отчёта:**

1. Титульный лист;
2. Цель, задачи работы;
3. Формулировка общего задания;
4. Листинги пользовательских функций, классов и основной программы;
5. Результаты работы программы;
6. 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:

- ***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.
- ***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 строиться на предыдущей, но добавляет в неё что – то своё, уникальное для новой сущности.
  Примером, 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 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 просто 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 в следующей Л/Р.

- ***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 выходя 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, именно с этой целью его и создавали. Более правильнее было бы сказать так: C++ язык с поддержкой парадигмы ООП разработки. Но он не единственный язык, который её поддерживает. Существуют и другие языки по типу: GoLang, Java, JS, Python и пр. Что означает, если язык поддерживает ООП? Это вовсе не значит, что на нём можно писать исключительно в ООП-стиле. Это означает, что на нём можно **удобно** писать программы по этой концепции, т.к. он предоставляет встроенные средства для подобной реализации. Хотя язык C и не является объектно-ориентированным, но на нём тоже можно реализовать ООП, правда выглядеть это будет немного иначе, да и сложность написания такой программы увеличиться в разы, а так – всё возможно. Вы должны понять, что ООП – это всего – лишь стиль программирования, который можно выразить на любом языке.

Начнём с нового для нас слова – «class». На самом деле нового в нём ничего нет, и подобные конструкции мы уже писали с помощью ключевого сова – «struct». И вообще класс от структуры в языке программирования C++ отличается только одним поведением, о котором мы поговорим немного позже. А сейчас вспомним, как писать структуры:

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

Мы написали структуру – Человек, в этой структуре есть три вида переменных. Первая переменная является указателем на тип char, по имени переменной видно, что в ней будет храниться имя человека. Следующая переменная типа int и в ней будет содержаться возраст. Третья переменная представляет пользовательский тип данных, который определяет пол человека, это (enum class).

Для того, чтобы создать переменную типа этой структуры, нужно написать в какой-нибудь функции:

```
Human Ivan{};
```

Таким образом мы создали переменную типа – Human, назвали её – Ivan и в линейном инициализаторе (фигурные скобочки), не указали аргументы, таким образом будет создан новый объект структуры с пустыми переменными внутри, а точнее у этих переменных будет значение по умолчанию. Как только мы передадим в скобки нужные аргументы, компилятор автоматически догадается куда, что подставлять и мы получим уже заполненный объект. Это мы всё помним ещё с уроков работы со структурами, но что меняется в ООП? Ровным счётом ничего, синтаксис будет всё такой же, просто мы наложим на наш код трафарет основных принципов ООП. Таким образом мы получим нужное нам поведение. Давайте пока для наглядности, не будем менять ключевое слово структуры на класс, а сделаем это под конец.

**Определения:**

- ***Класс (структура)*** – это шаблон, трафарет, макет, чертёж, гайд, который гласит – каким образом строить реальные объекты. Важно понимать, что, когда мы описываем структуру, или класс, память не выделяется под все существующие переменные. Она будет выделена, когда будет создана непосредственно переменная нового типа данных. Приведём такой пример: в мастерской вы хотите собрать машинку на радиоуправлении. Для этого Вы составляете чертёж будущей машинки, определяете её свойства (цвет, форма, габариты и прочее), а также её поведения (движения вперёд назад, победный «би - би» и рулевые повороты). Всё это у Вас отражено на чертеже. Если вы возьмёте пульт управления и попробуете его применить для чертежа, даже если будете очень стараться – он не поедет, не издаст сигнал и прочее. Это означает, что чертёж лишь образ будущей машинки (объекта), который по нему можно построить. Чертёж определяет (декларирует, описывает) основные свойства и поведения объекта, а сам объект их инициализирует (реализовывает). По одном чертежу можно собрать множество машинок, разной формы, цвета, звукового сигнала, но их будет объединять общий набор свойств и поведений. Общий чертёж, по которому они построены. И теперь направив на них пульт управления, они придут в движении и издадут значимое для нас – «би – би»!

  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, уже 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 создаёте новый тип данных, переменную которого Вы теперь можете создать.

- ***Экземпляр класса (объект)*** – если сам класс – это трафарет, то экземпляр класса – это уже реальный объект, созданный по этому трафарету, в котором уже инициализированы все свойства класса.
- ***Поля*** – переменные, которые находятся в классе и которые представляют свойства этого класса, в контексте ООП называются полями класса. Для того, чтобы чётко понимать, что эти переменные являются полями класса, существует, общепринятое парило 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 описывают поведение объекта. Эти функции пишутся в теле самого класса (сам синтаксис описания функций ничем не отличается). Такие функции называются методами класса. Чтобы вызвать такую функцию нужно обратиться к ней через точку, словно это поле объекта, например так: `Ivan.print();`. Как видите, синтаксис ничем не отличается от вызова обычной функции, за тем исключением, что сперва мы указываем через точку объект у которого мы вызываем эту функцию, что логично. Здесь стоит сделать акцент на том, что названия функций у всех объектов будет одинаковое, но функция будет работать только с теми данными, которые даны объекту, у которого она вызывается. Таким образом следующий код будет работать так:

```
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 };
	Human Dima{ new char[] {"Dima"}, 19, Gender::male };

	Ivan.print(); // вывод - Ivan
	Dima.print(); // вывод - Dima

	return 0;
}
```

Такой механизм позволяет нам использовать концепцию интерфейсов, которая уже упоминалась в данной теории. Зная, что у класса человек, есть метод принт, мы можем его вызывать для всех экземпляров этого класса и его дочерних классов не думая о том, 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 – это такая 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 (тип возврата такой функции полностью отсутствует, писать его не нужно, он не void). Конструкторы служат для начальной инициализации полей данных класса. В отличии от линейной инициализации объекта, механизм которой достаточно сложен для понимания и рассмотрения на данном этапе, механизм инициализации полей конструктором достаточно прозрачен.

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

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

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

int main() {

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

    Ivan.print();

    return 0;
}
```

Как видим, в данном примере создан конструктор класса Human, который задаёт начальное значение переменной класса m_age равным 25. Теперь обратите внимание на создание объекта при использовании конструктора. Мы пишем тип, имя будущей переменной и через равно снова имея класса с круглыми скобками, словно мы вызываем функцию. Таким образом компилятор вызывает конструктор класса, который мы написали в структуре, и он присваивает начальное значение переменной возраста. Стоит заметить, что показанная инициализация не является эффективной. Более подробно про инициализацию будет написано во – второй работе.

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 конструткор 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;
    }

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

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);

    Ivan.print();

    return 0;
}
```

В данном примере мы передаём в конструктор значения, которые необходимо присвоить переменным класса. Значения передаются и обрабатываются, словно это обычные функции. Также, стоит обратить внимание на то, как передаются эти значения для более оптимизированной работы с памятью.

- ***Деструктор класса*** – на ровне с конструктором класса существует такое понятие, как деструктор. Как видно из названия это функция, противоположна конструктору по назначению. Деструктор вызывает компилятор, когда объект больше не нужен, либо мы, когда 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. Также, деструктор не умеет работать с аргументами, переданными в круглых скобках вызова функции, так что передавать их туда бесполезно. Деструктор имеет название такое же, как название самого класса, только перед ним ставится символ – тильда: «~». Деструкторы бывают полезны, когда в объекте идёт динамическое выделение памяти, либо же открытие потока. Т. к. компилятор не сможет автоматически закрыть поток, или освободить память, мы должны это делать 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 папять под имя человека
		m_name = new char[256] {};

		// здесь, мы копируем переданное
		// имя в поле класса
		strcpy(m_name, name);

		m_age = age;
		m_gender = gender;
	}

	// здесь мы определяем деструктор
	// объекта и в нём удаляем динамически
	// выделенную память под имя человека
	~Human() {
		delete[] m_name;
	}

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

int main() {

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

```
        // здесь мы создаём динамически новый объект
        // структуры человека, вызывая сразу конструктор
        // этого объекта
        Human* Dima = new Human("Dima", 10, Gender::male);

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

        return 0;
}
```

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

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

- ***Модификаторы доступа*** – вот и настал тот самый момент, когда мы подошли к главному различию ключевого слова struct и class. Давайте попробуем переписать наш предыдущий пример с новым ключевым словом:

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

    /* реализация класса */

};

int main() {

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

    Ivan.print();

    return 0;
}
```

Как видите после изменения ключевого слова struct на class – ничего не поменялось, но стоит попробовать скомпилировать данную программу. Компилятор выдаст некое сообщение об ошибки, в чём может быть проблема? В этой ошибки будет сказано, что невозможно вызвать член класса print у объекта Ivan. Но почему? Он же там есть. Дело всё в том, что существует такое понятие, как модификатор доступа. Это некие ключевые слава, которые задают уровни доступа к объектам в пределах класса. Всего ***модификаторов доступа существует три***:

- ***Public*** – после этого ключевого слова все поля и методы класса будут считаться доступными из вне, в любом файле, функции, которая импортирует наш класс.
- ***Private*** – после этого ключевого слова все поля и методы класса, будут считаться закрытыми для доступа из вне. При этом к ним всё ещё можно обратиться в текущем классе, но вне класса вызвать их у экземпляра этого класса не выйдет.
- ***Protected*** – очень специфичный модификатор доступа, который встречается не очень часто. Его основная задача сделать все методы, поля у класса, которые идут после

него доступными для текущего класса, либо для класса потомка. Этот модификатор мы рассмотрим во – второй Л/Р.

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, всё наоборот, 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 структуры */

};

int main() {

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

	Ivan.print();

	return 0;
}
```

Всё, что прописано ниже модификатора доступа public может быть получено у любого объекта класса в любом файле.

- ***Геттеры и сеттеры*** – Вы наверняка задались вопросом: а зачем вообще нужны модификаторы доступа, неужели нельзя просто сделать все свойства и методы открытыми? – на этот вопрос, лучше всего ответит один из принципов ООП – инкапсуляция, а также следующий пример:

```
class Human {

public:
	char* m_name{};
	int m_age{};
	Gender m_gender{};

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

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

		// здесь, мы копируем переданное
		// имя в поле класса
		strcpy(m_name, name);

		m_age = age;
		m_gender = gender;
	}
```

```
        // здесь мы определяем деструктор
        // объекта и в нём удаляем динамически
        // выделенную память под имя человека
        ~Human() {
                delete[] m_name;
        }

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

    int main() {

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

        Ivan.print();

        Dima.print();

        return 0;
    }
```

Здесь создаётся два объекта класса человек. В первый объект передаётся возраст 20 лет, а в другой -20. Вы скажите, что второй объект создан неправильно, это противоречит бизнес логики, но не логике программы. Она скомпилируется и выведет отрицательный возраст. Как с этим бороться? Самый простой ответ – создать переменную int с префиксом: `unsigned`. Таким образом мы вроде как избавились от проблемы, но, что если пользователь введёт очень большое число в возраст? Конечно, если люди живут долго – это очень хорошо, но увы, люди не могут жить более 1000 лет. Это опять же является противоречием в бизнес – логике. Второй способ решить проблему, это явно в конструкторе проверять диапазон значений возраста и если заданный параметр под него не подходит оставлять значение по умолчанию, но такой способ тоже не подходит. Ведь, если мы изменим саму переменную m_age вне конструктора, мы опять сможем положить туда некорректное значение. Как быть? Нужно, каким – то образом, перехватывать присваивание переменной нового значения и сравнивать с нужным диапазоном. Здесь и приходит на помощь концепция инкапсуляции и использования геттеров и сеттеров. Посмотрим, что это такое на примере:

```
class Human {

// делаем поля класса закрытыми
private:
        char* m_name{};
        unsigned int m_age{};
        Gender m_gender{};

// создаём открытое поле
public:

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

        // функция для получения возраста
        unsigned int getAge() {
```

```
		return m_age;
	}

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

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

		// здесь, мы копируем переданное
		// имя в поле класса
		strcpy(m_name, name);

		m_age = age;
		m_gender = gender;
	}

	// здесь мы определяем деструктор
	// объекта и в нём удаляем динамически
	// выделенную память под имя человека
	~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;
}
```

В примере мы закрыли доступ к полям класса из объекта. Но, как же установить новый возраст? Для этого мы написали функцию 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 метод, getAge, как понятно из названия 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.

В начало

**Правила хорошего тона при написании классов**

Как мы убедились 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. Но, правилами хорошего тона определено написание классов через одноимённое ключевое слово, и мы будем этого придерживаться. Также, стоит заметить, что общепринято 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. Давайте рассмотрим, как это делать:

**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;
}
```

Создание прототипов методов ничем не отличается от созданий таковых функций. Стоит обратить особое внимание на реализацию методов. Чтобы получить доступ к реализации мы должны указать у какого класса и какой метод будем реализовывать и тут синтаксис такой:

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

Пишем 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.

В начало

## Создание обёрток для типов

Последнее, что мы рассмотрим в этой Л/Р это создание обёрток для типов данных и не важно какие они: пользовательские, или встроенные, их все можно переопределить. Для чего это нужно? Например: мы пишем банковское приложение, и в одну из его функций должна передаваться некоторая сумма денежных средств. Какой тип поля мы должны указать? – правильно unsigned int, но что, если мы хотим получить полную безопасность

типов, или же добавить дополнительные операции с суммами, ну, или же нам просто надоест писать постоянно unsigned int. Для таких целей существуют сразу два оператора, один был в плюсах всегда, другой появился только в 11-м стандарте. Давайте рассмотрим их:

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

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

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

Во – втором случаи 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.

**Указатели на компоненты-функции:**

Можно определить указатель на компоненты-функции.

***тип_возвр_значения (имя_класса :: *имя_указателя_на_функцию) (специф_параметров_функции);***

**Методические указания**

1. Пример определения класса.

```
const int LNAME=25;

class STUDENT{

 char name[LNAME]; // имя

 int age;          // возраст

 float grade;      // рейтинг

 public:

  STUDENT();         // конструктор без параметров

  STUDENT(char*,int,float); // конструктор с параметрами

  STUDENT(const STUDENT&);  // конструктор копирования

  ~STUDENT();

  char * GetName() ;
```

```
  int GetAge() const;
  float GetGrade() const;
  void SetName(char*);
  void SetAge(int);
  void SetGrade(float);
  void Set(char*,int,float);
  void Show();
};
```

Более профессионально определение поля name типа указатель: char* name. Однако в этом случае реализация компонентов-функций усложняется.

2. Пример реализации конструктора с выдачей сообщения.

```
STUDENT::STUDENT(char*NAME,int AGE,float GRADE)
{
strcpy(name,NAME); age=AGE; grade=GRADE;
cout<< \nКонструктор с параметрами вызван для объекта  <<this<<endl;
}
```

3. Следует предусмотреть в программе все возможные способы вызова конструктора копирования. Напоминаем, что конструктор копирования вызывается:

а) при использовании объекта для инициализации другого объекта

Пример.

```
STUDENT a("Иванов",19,50), b=a;
```

б) когда объект передается функции по значению

Пример.

```
void View(STUDENT a){a.Show;}
```

в) при построении временного объекта как возвращаемого значения функции

Пример.

```
STUDENT NoName(STUDENT & student)
{STUDENT temp(student);
temp.SetName("NoName");
return temp;}
```

```
STUDENT c=NoName(a);
```

4. В программе необходимо предусмотреть размещение объектов как в статической, так и в динамической памяти, а также создание массивов объектов.

Примеры.

а) массив студентов размещается в статической памяти

```
STUDENT gruppa[3];
gruppa[0].Set("Иванов",19,50);
и т.д.
или
STUDENT gruppa[3]={STUDENT("Иванов",19,50),
STUDENT("Петрова",18,25.5),
STUDENT("Сидоров",18,45.5)};
```

б) массив студентов размещается в динамической памяти

```
STUDENT *p;
p=new STUDENT [3];
p->Set("Иванов",19,50);
           и т.д.
```

5. Пример использования указателя на компонентную функцию

```
void (STUDENT::*pf)();
pf=&STUDENT::Show;
(p[1].*pf)();
```

6. Программа использует три файла:

- заголовочный h-файл с определением класса,
- cpp-файл с реализацией класса,
- cpp-файл демонстрационной программы.

Для предотвращения многократного включения файла-заголовка следует использовать директивы препроцессора

```
#ifndef STUDENTH
#define STUDENTH
// модуль STUDENT.H
```

```
...
#endif
```

или `#pragma once`.

В начало

**Задания**

**Вариант №1**

СТУДЕНТ
имя – char*
курс – int
пол – int (bool)

**Вариант №2**

ИЗДЕЛИЕ
имя – char*
шифр – char*
количество – int

**Вариант №3**

АДРЕС
имя – char*
улица – char*
номер дома – int

**Вариант №4**

ЦЕХ
имя – char*
начальник – char*
количество
работающих – int

**Вариант №5**

СТРАНА
имя – char*
форма
правления – char*
площадь – float

**Вариант №6**

СЛУЖАЩИЙ
имя – char*
возраст – int
рабочий стаж – int

**Вариант №7**

БИБЛИОТЕКА
имя – char*
автор – char*

стоимость – float

**Вариант №8**

ТОВАР
имя – char*
количество – int
стоимость – float

**Вариант №9**

ПЕРСОНА
имя – char*
возраст – int
пол – int(bool)

**Вариант №10**

ЖИВОТНОЕ
имя – char*
класс – char*
средний вес – int

**Вариант №11**

КАДРЫ
имя – char*
номер цеха – int
разряд – int

**Вариант №12**

ЭКЗАМЕН
имя студента – char*
дата – int
оценка – int

**Вариант №13**

КВИТАНЦИЯ
номер – int
дата – int
сумма – float

**Вариант №14**

АВТОМОБИЛЬ
марка – char*
мощность – int
стоимость – float

**Вариант №15**

КОРАБЛЬ
имя – char*
водоизмещение – int
тип – char*

**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.

**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