Классы и объекты

Михаил Марков
C++ - Разработчик

Михаил Марков

О спикере:
C++-разработчик, фрилансер

- Разработка алгоритма для релевантной выдачи объявлений.
- Разработка эмуляторов оборудования

Вспоминаем прошлое занятие

Вопрос: связанный набор именованных констант - это?

Вспоминаем прошлое занятие

Вопрос: связанный набор именованных констант - это?

Ответ: перечисление

Вспоминаем прошлое занятие

Вопрос: какие значения у констант перечисления?

Вспоминаем прошлое занятие

Вопрос: какие значения у констант перечисления?

Ответ: значениями констант перечисления являются целые числа. По умолчанию стартуют с 0 и увеличиваются на 1 для каждой следующей константы, но можно указать свои значения

Вспоминаем прошлое занятие

Вопрос: пользовательский тип данных, объединяющий разнородные данные - это?

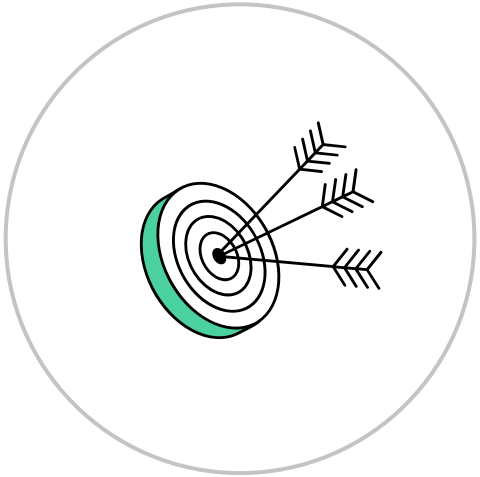
Вспоминаем прошлое занятие

Вопрос: пользовательский тип данных, объединяющий разнородные данные - это?

Ответ: структура

Цели занятия

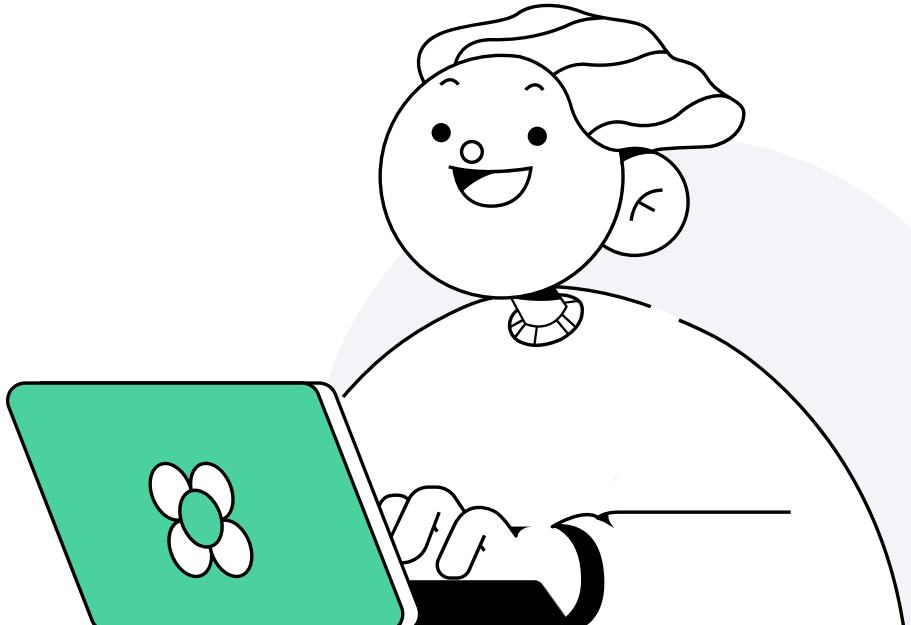
- Разберёмся, что такое классы
- Познакомимся с членами класса
- Узнаем, что такое конструкторы
- Выясним, что такое модификаторы доступа



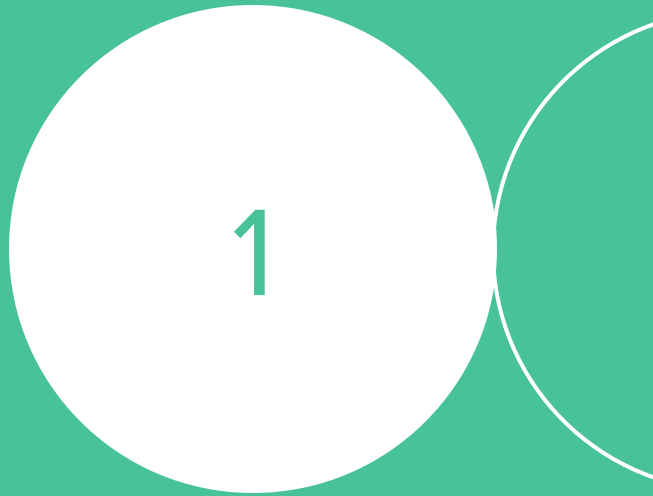
План занятия

1. Классы
2. Члены класса
3. Модификаторы доступа
4. Итоги
5. Домашнее задание

*Нажми на нужный раздел для перехода



Классы



Что такое класс?

Понятие класса является основой парадигмы ООП (объектно-ориентированное программирование), поэтому класс в том или ином виде присутствует во всех языках программирования, поддерживающих ООП

В общем и целом класс - это сущность, объединяющая данные и операции над этими данными

Что такое класс в C++?

В C++ класс - это пользовательский тип данных, который объединяет в себе **данные** и **поведение** - то есть работу с этими данными

Классы в C++ - это практически то же самое, что и структуры, за одним маленьким исключением, с которым мы познакомимся позже

Синтаксис

Класс объявляется с помощью ключевого слова **class**. Синтаксис объявления выглядит следующим образом:

```
class <название класса>
{
public:
    <члены класса>
};
```

Члены класса - это то, из чего класс состоит и то, что он умеет делать.
Познакомимся с каждым членом отдельно

Чтобы иметь возможность обращаться к членам класса за его пределами, нужно в самом начале класса указать “public:”

Синтаксис. Пример

Создадим класс Person, в котором будет храниться имя, фамилия и возраст человека:

```
class Person
{
public:
    std::string first_name;
    std::string last_name;
    int age;
};
```

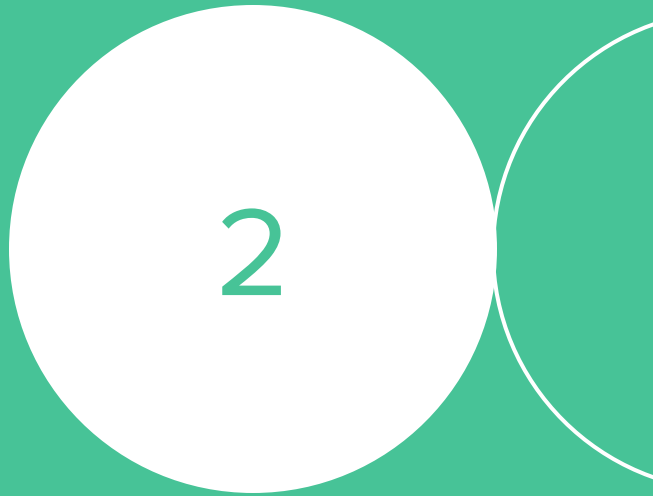
Ещё про класс

Класс, как и структура - это новый, пользовательский тип данных. В объявлении класса описывается его устройство - своеобразная “схема”

Конкретные экземпляры, построенные по этой схеме (то есть содержащие в себе члены, указанные в объявлении класса), называются **экземпляры класса**, или **объекты**

Объектов может существовать много, тогда как объявление класса - всего одно

Члены класса



Поля

Поля нам знакомы из прошлой лекции про структуры. Поля - это “переменные” внутри одного экземпляра класса

У каждого поля должен быть указан тип данных этого поля и имя поля

Например, у объявленного нами ранее класса Person три поля - два поля типа std::string (first_name и last_name) и одно поле типа int (age)

```
class Person
{
public:
    std::string first_name;
    std::string last_name;
    int age;
};
```

Зачем нужны поля?

Поля используются для того, чтобы хранить данные, специфичные для одного экземпляра класса

В дальнейшем данные, хранящиеся в этих полях, используются внешним по отношению к классу кодом (любым, кто использует этот класс) или внутренним (методами - с ними сейчас познакомимся)

Методы - это функции, “принадлежащие” экземпляру класса (объекту)

Имея на руках экземпляр класса, можно вызвать его метод с помощью оператора . (точка)

Методы. Синтаксис

Синтаксис объявления методов ничем не отличается от синтаксиса объявления обычной функции - за исключением того, что метод объявляется **внутри** класса

```
<тип возвращаемого значения> <имя метода>([<аргументы>])
{
    <тело метода>
}
```

Методы. Синтаксис. Пример

Создадим в нашем классе Person метод, который выводит на консоль строку, описывающую этого человека - она будет сообщать имя, фамилию и возраст

```
class Person
{
public:
     std::string first_name;
     std::string last_name;
     int age;

     void print_person()
     {
          std::cout << "Имя: " << first_name << " Фамилия: " << last_name << "
Возраст: " << age << std::endl;
     }
};
```

Методы. Синтаксис. Пример

Вызовем наш метод

```
int main(int argc, char** argv)
{
    Person person;
    person.print_person(); // Имя: Фамилия: Возраст: -858993460
}
```

Как вы думаете, почему так получилось? **Напишите в чат**

Методы. Синтаксис. Пример

Правильно - мы не инициализировали поля нашего нового объекта. Теперь вызовем наш метод, предварительно присвоив значения полям

```
int main(int argc, char** argv)
{
    Person person;
    person.first_name = "Иван";
    person.last_name = "Иванов";
    person.age = 40;
    person.print_person(); // Имя: Иван Фамилия: Иванов Возраст: 40
}
```

Вспоминаем

Вопрос: мы уже сталкивались с методами. Где и с какими?

Вспоминаем

Вопрос: мы уже сталкивались с методами. Где и с какими?

Ответ:

- В классах (std) ifstream, ofstream и fstream - методы *is_open*, *close* и другие
- В классе std::string методы *length*, *find* и другие

Методы. Ключевое слово this

Есть ещё одна особенность методов - в них доступно ключевое слово **this**

С помощью этого ключевого слова можно получить доступ к **объекту, у которого был вызван этот метод**

Обычно в этом нет необходимости, но иногда оно бывает полезным - например, если имя аргумента метода совпадает с именем поля класса

Ключевое слово this имеет тип указателя на тот класс, в рамках которого оно используется. Поэтому для получения доступа к членам объекта, на который указывает this, нужно использовать оператор -> (стрелочка)

Методы. Ключевое слово this

Например, напишем метод, который будет изменять возраст человека

```
class Person
{
public:
    std::string first_name;
    std::string last_name;
    int age;

    void change_age(int age)
    {
        age = age; // такой код будет работать неправильно
    }
};
```

Методы. Ключевое слово this

Например, напишем метод, который будет изменять возраст человека

```
class Person
{
public:
    std::string first_name;
    std::string last_name;
    int age;

    void change_age(int age)
    {
        this->age = age; // тут всё правильно
    }
};
```

Конструктор

Помните, как мы инициализировали поля после создания экземпляра класса Person? А если мы этого не делали, то получали очень странный результат после вызова метода *print_person*

При этом если вы забудете присвоить значение какому-то важному полю после создания объекта, вы можете получить неправильное поведение целого объекта

Для того, чтобы избежать таких ситуаций, существуют **конструкторы**

Конструктор

это специальный метод, который вызывается при создании объекта

В нём программист пишет код, который должен быть выполнен во время создания экземпляра класса

Конструктор

Конструктор **обязан** называться точно так же, как класс - так компилятор (и программист) отличает конструктор от обычных методов

Конструктор **не имеет** возвращаемого значения, и указывать тип возвращаемого значения для конструктора **нельзя**

Класс обязан иметь **хотя бы один** конструктор. При этом конструкторов у класса может быть несколько

Конструкторы одного и того же класса различаются между собой **количеством** и **типом** параметров

Конструктор. Синтаксис

Обобщённый синтаксис конструктора представлен ниже

```
class <имя класса>
{
    <имя класса>([<аргументы>])
    {
        <тело конструктора>
    }
}
```

Конструктор. Пример

Создадим пустой конструктор для нашего класса Person

```
class Person
{
public:
    std::string first_name;
    std::string last_name;
    int age;
    Person() { } // это пустой конструктор
};
```

Вопрос

Если мы добавили классу конструктор только сейчас, и при этом класс обязан иметь хотя бы один конструктор - то как наш код работал раньше?

Ведь не было никакого конструктора до этого

Как вы думаете? Напишите в чат

Конструктор с параметрами. Пример

Создадим конструктор с параметрами для нашего класса Person

```
class Person
{
public:
    std::string first_name;
    std::string last_name;
    int age;
    Person(std::string first_name, std::string last_name, int age)
    {
        this->first_name = first_name;  // инициализация
        this->last_name = last_name;// инициализация
        this->age = age;            // инициализация
    }                                   // это конструктор с параметрами
};
```

Конструктор по умолчанию

Если в классе программист не написал явно ни одного конструктора, то компилятор сам сгенерирует **конструктор по умолчанию (default constructor)** - пустой, без параметров

А вот если программист написал сам хотя бы один конструктор, то конструктор по умолчанию сгенерирован уже не будет

Как использовать конструктор

Конструктор используется при создании объекта - то есть когда вы пишете **“Person person;”**

Компилятор разрешает для конструкторов без параметров не писать пустые скобочки

Посмотрим, что случится, если мы объявим в классе Person **только** конструктор с параметрами и попробуем создать его экземпляр так, как мы делали это раньше

Использование конструктора

```
class Person
{
public:
    std::string first_name;
    std::string last_name;
    int age;
    Person(std::string first_name, std::string last_name, int age)
    {
        ...
    } // это конструктор с параметрами
};
int main(int argc, char** argv)
{
    Person person; // Ошибка!
}
```

Использование конструктора

Теперь у нас не получается создать новый экземпляр класса Person

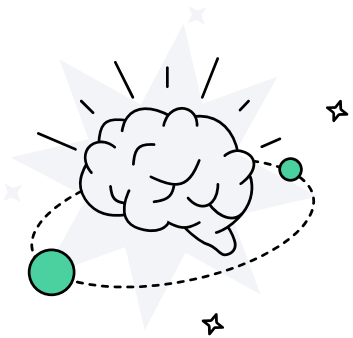
Как вы думаете, как можно исправить эту ситуацию, не удаляя конструктор с параметрами?

Использование конструктора

Есть два способа

Первый:
Удовлетворить условия конструктора с параметрами, то есть при создании объекта предоставить ему те значения, которые ожидает конструктор (в примере - две строки и целое число)

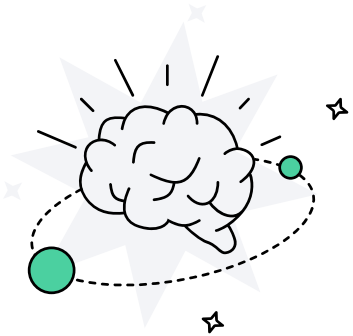
Это - наиболее правильный подход, так как мы добивались именно такого поведения - чтобы нельзя было создать экземпляр класса Person без инициализированных полей



Использование конструктора

Есть два способа

Второй:
Создать второй конструктор, но без параметров - и тогда в нём, чтобы выполнить нашу изначальную задачу, нужно будет присвоить полям какие-то конкретные значения, которые придумать придётся самостоятельно



Решение проблемы. Первый способ

```
class Person
{
public:
     std::string first_name;
     std::string last_name;
     int age;
     Person(std::string first_name, std::string last_name, int age) { ... }
                                          // это конструктор с параметрами
};

int main(int argc, char** argv)
{
     Person person("Пётр", "Петров", 30); // Теперь собирается
}
```

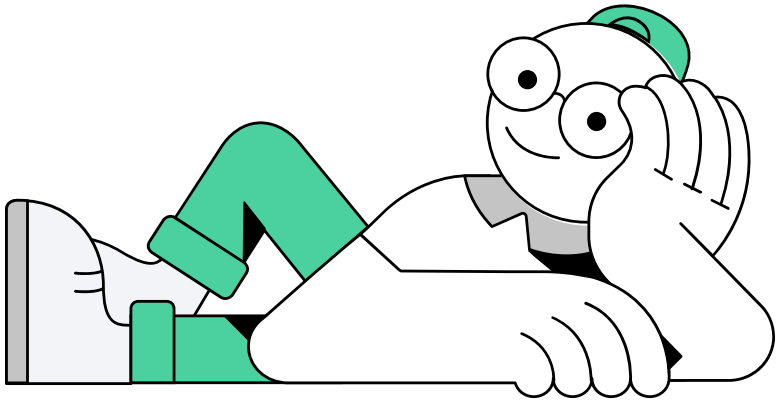
Решение проблемы. Второй способ

```
class Person
{
public:
     ...
     Person(std::string first_name, std::string last_name, int age) { ... }
          // это конструктор с параметрами
     Person()
     {
          first_name = "Неизвестно";
          last_name = "Неизвестна";
          age = 18;
     }    // это конструктор без параметров
};
int main(int argc, char** argv)
{
     Person person; // Неизвестно Неизвестна 18
}
```

Конструкторы. Итоги

Код, который программист пишет в конструкторе, вызывается при создании нового экземпляра класса. Если в классе несколько конструкторов, то выбор будет произведен на основе того, каким способом создаётся новый объект - то есть какие параметры передаются при создании

Если программист не указал ни одного конструктора при создании класса, то компилятор сгенерирует конструктор по умолчанию - без параметров и пустой

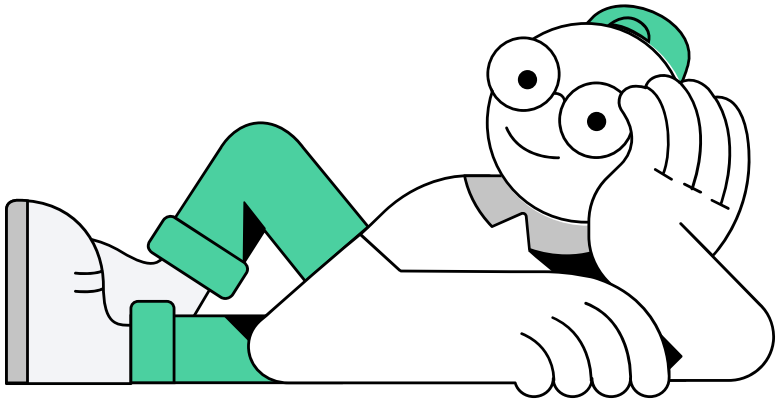


Члены класса. Итоги

Мы с вами рассмотрели 3 разновидности членов класса:

- Поля
- Методы
- Конструкторы

Это далеко не исчерпывающий список возможных членов класса, но это - наиболее нужные, важные и употребительные члены, с которыми обязательно надо хорошо разобраться прежде, чем переходить к более сложным вещам



Перерыв

Модификаторы доступа

3

Модификаторы доступа

Пришло время понять, что значит “public:”, который мы пишем в самом начале класса

В C++ существует такое понятие, как **модификатор доступа** - он определяет, **откуда можно использовать то, что помечено таким модификатором**

Модификаторы доступа применяются к членам класса (разберёмся сейчас) и при наследовании (разберёмся в следующих лекциях)

Модификаторы доступа

Всего в C++ существует три модификатора доступа:

- public
- protected
- private

Модификаторы доступа: члены класса. Синтаксис

С синтаксисом применения модификаторов доступа мы отчасти познакомились на практике. Модификаторы доступа для членов класса указываются в рамках класса, после них ставится двоеточие

Указанный модификатор применяется ко всем членам класса, расположенным ниже него и выше следующего модификатора доступа (если таковой имеется)

```
class MyClass
{
public:
     <член 1>
     <член 2>
private:
     <член 3>
     <член 4>
public:
     <член 5>
}
```

Модификатор доступа private

Модификатор доступа **private** означает, что помеченные им члены доступны **только коду внутри класса**

То есть если вы пометили поле класса как private - обратиться к этому полю (то есть считать из него значение или записать в него новое значение) можно будет только изнутри самого класса (а именно из методов или конструкторов)

```
class TestClass
{
private:
     int priv_field;
     void priv_method() { std::cout << priv_field; } // можно
};

int main(int argc, char** argv)
{
     TestClass test;
     std::cout << test.priv_field; // ошибка доступа!
     test.priv_method(); // ошибка доступа!
}
```

Модификатор доступа protected

Модификатор доступа **protected** означает, что помеченные им члены доступны **коду внутри класса и внутри его наследников**. Что такое наследники - мы узнаем в следующих лекциях, поэтому *пока что* protected для нас работает так же, как private

```
class TestClass
{
protected:
    int prot_field;
    void prot_method() { std::cout << prot_field; } // можно
};

int main(int argc, char** argv)
{
    TestClass test;
    std::cout << test.prot_field; // ошибка доступа!
    test.prot_method(); // ошибка доступа!
}
```

Модификатор доступа public

Модификатор доступа **public** означает, что помеченные им члены доступны **любому коду**. Члены, помеченные как public, могут использоваться внешним кодом (который является пользователем нашего класса)

```
class TestClass
{
public:
    int pub_field;
    void pub_method() { std::cout << pub_field; } // можно
};

int main(int argc, char** argv)
{
    TestClass test;
    std::cout << test.pub_field; // можно
    test.pub_method(); // можно
}
```

Геттеры и сеттеры

Поля класса обычно приватные (в следующей лекции узнаем, почему). Для того, чтобы можно было получить доступ к полям снаружи класса (прочитать значение или записать его), существуют методы, которые называются **геттеры** и **сеттеры**. Геттеры позволяют прочитать значение поля, сеттеры позволяют установить значение (и, возможно, проверить значение перед установкой)

```
class TestClass
{
private:
     int priv_field;
public:
     int get_priv_field() { return priv_field; } // геттер
     void set_priv_field(int value) { if (value > 0) priv_field = value; } // сеттер
};
int main(int argc, char** argv)
{
     TestClass test;
     test.set_priv_field(5);
     std::cout << test.get_priv_field; // 5
}
```

Таблица сравнения модификаторов

Здесь представлена сводная таблица соответствия модификаторов доступа и возможности использования членов, помеченных этими модификаторами для кода, находящегося в разных местах

	Члены, помеченные модификатором доступа		
Где находится вызывающий код	**private**	**protected**	**public**
Внутри класса	Можно использовать	Можно использовать	Можно использовать
Внутри наследников класса	Нельзя использовать	Можно использовать	Можно использовать
Вне класса	Нельзя использовать	Нельзя использовать	Можно использовать

Модификаторы для конструкторов

Модификаторами доступа могут помечаться любые члены класса - в том числе и конструкторы

Обратите внимание, что если вы определите один или несколько конструкторов и все их сделаете приватными (private) или защищёнными (protected) - вы не сможете создать экземпляр такого класса во внешнем коде

```
class TestClass
{
private:
    TestClass() { } // приватный конструктор
};

int main(int argc, char** argv)
{
    TestClass test; // ошибка доступа!
}
```

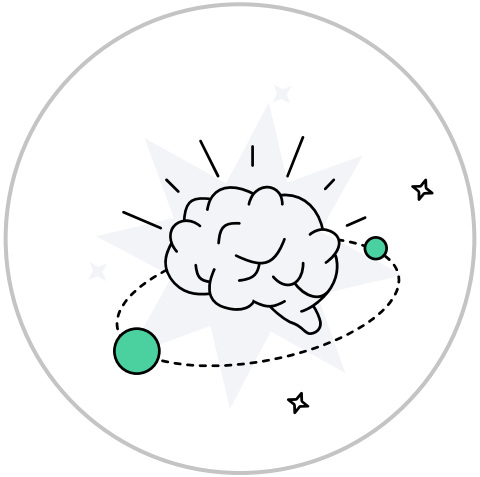
Модификаторы доступа по умолчанию

Что будет, если мы вообще не будем указывать модификатор доступа при создании класса?

Как вы думаете?

Модификаторы доступа по умолчанию

Будет применён модификатор доступа по умолчанию. Для классов (class) модификатор доступа по умолчанию - private, поэтому при создании классов мы пишем в самом начале public - чтобы члены класса были публичными и мы могли их использовать во внешнем коде



Различие между классом и структурой

В противоположность этому, модификатор доступа по умолчанию для **структур** (struct) - public. И именно в этом заключается различие между классами и структурами, в остальном они ведут себя одинаково

Это значит, что у структур тоже можно создавать методы и конструкторы. И всё то, что мы узнаем в дальнейшем о классах, в равной степени относится и ко структурам

Так сложилось исторически. В целом, для сложных структур данных программисты обычно используют классы, но это чисто идеологическое предпочтение

Итоги

Итоги занятия

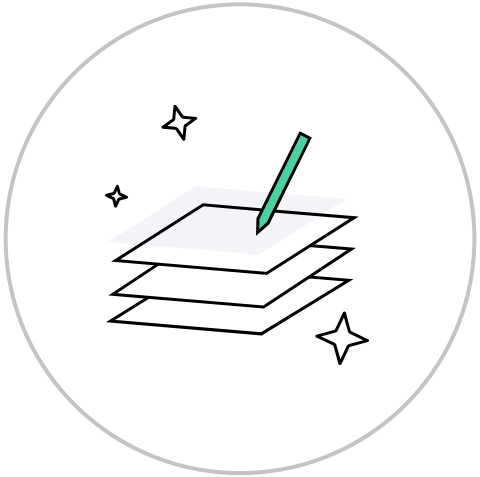
Сегодня мы

1. Разобрались, что такое классы
2. Познакомились с членами класса
3. Узнали, что такое конструкторы
4. Выяснили, что такое модификаторы доступа

Домашнее задание

Давайте посмотрим ваше домашнее задание.

1. Вопросы по домашней работе задавайте в чате группы
2. Задачи можно сдавать по частям
3. Зачёт по домашней работе ставят после того, как приняты все задачи



Дополнительные материалы

- Классы и модификаторы доступа

Задавайте вопросы и пишите отзыв о лекции

Михаил Марков
C++ - разработчик