Перегрузка операторов

Михаил Смирнов
Разработчик C++

Проверка связи

Поставьте “+”, если меня видно и слышно

Если у вас нет звука:

- убедитесь, что на вашем устройстве и на колонках включен звук
- обновите страницу вебинара (или закройте страницу и заново присоединитесь к вебинару)
- откройте вебинар в другом браузере
- перезагрузите компьютер (ноутбук) и заново попытайтесь зайти

Михаил Смирнов

О спикере:

- В C++ разработке с В C++ разработке с 2010 года
- С 2002 года работаю в Муромском Институте Владимирского Государственного Университета
- Цифровая обработка сигналов в радиолокации и гидролокации
- Траекторная обработка для радиолокаторов ближней зоны
- Создание автоматизированного рабочего места для управления гидролокатором

Вспоминаем прошлое занятие

Вопрос: какие бывают ошибки?

Вспоминаем прошлое занятие

Вопрос: какие бывают ошибки?

Ответ: бывают ошибки времени компиляции и ошибки времени выполнения

Вспоминаем прошлое занятие

Вопрос: как можно искать ошибки?

Вспоминаем прошлое занятие

Вопрос: как можно искать ошибки?

Ответ: с помощью отладки и/или логирования

Вспоминаем прошлое занятие

Вопрос: что такое код возврата?

Вспоминаем прошлое занятие

Вопрос: что такое код возврата?

Ответ: это код, сообщающий о результате выполнения функции

Вспоминаем прошлое занятие

Вопрос: что такое исключение?

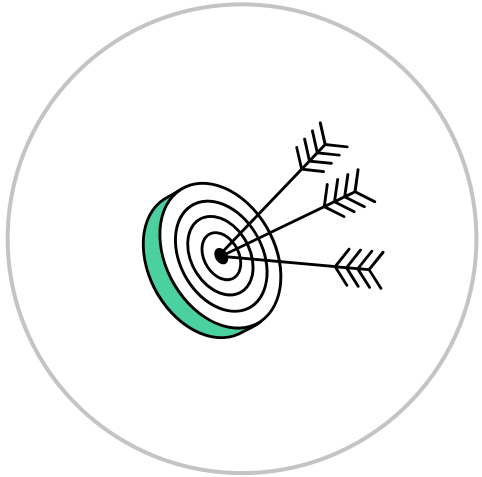
Вспоминаем прошлое занятие

Вопрос: что такое исключение?

Ответ: исключения - это механизм, позволяющий прерывать выполнение программы из-за ошибки и обрабатывать эту ошибку в разных местах программы

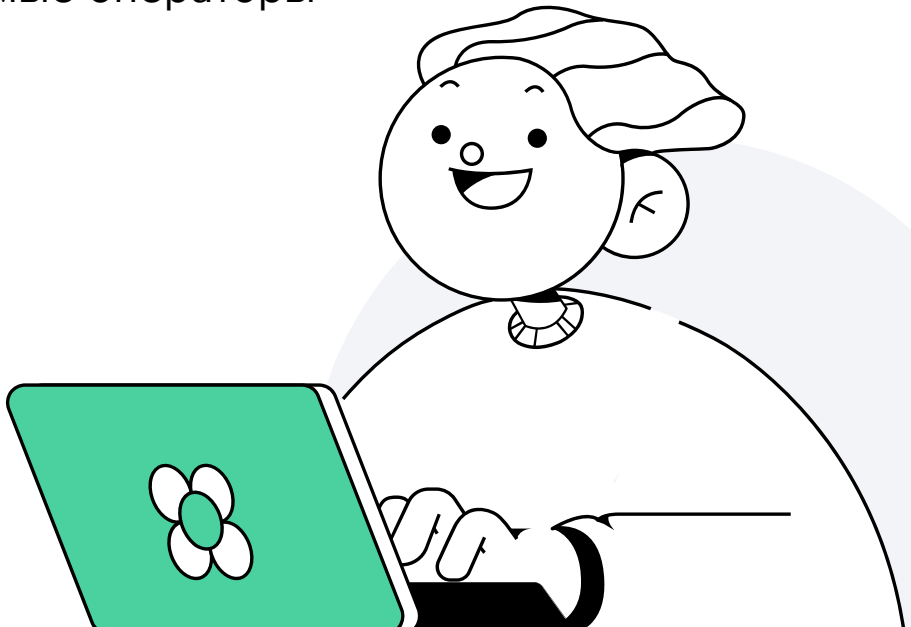
Цели занятия

- Разберёмся, что такое перегрузка операторов
- Познакомимся с тем, какие бинарные операторы можно перегружать
- Узнаем, какие унарные операторы можно перегружать
- Выясним, какие ещё операторы поддаются перегрузке, а какие нет



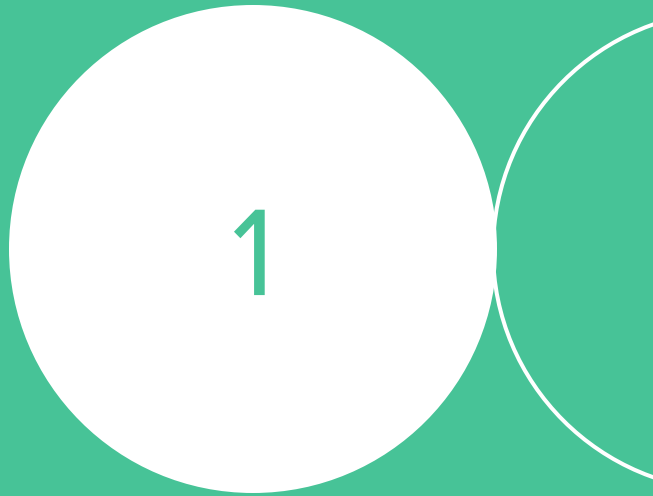
План занятия

1. Перегрузка операторов
2. Бинарные операторы
3. Унарные, остальные и неперегружаемые операторы
4. Итоги
5. Домашнее задание



*Нажми на нужный раздел для перехода

Перегрузка операторов



Задача

Представим, что нам нужно написать класс ComplexNumber - класс для комплексных чисел - и реализовать простую арифметику для него (сложение и вычитание)

Как мы помним, комплексное число состоит из двух частей - действительной (re) и мнимой (im)

При сложении и вычитании двух комплексных чисел их действительные и мнимые части складываются и вычитаются по отдельности

Создадим класс для решения нашей задачи и продемонстрируем его работу

Решение

```
class ComplexNumber
{
    double re_;
    double im_;
public:
    ComplexNumber(double re, double im) { re_ = re; im_ = im; }
    ComplexNumber Add(ComplexNumber other)
    { return ComplexNumber(re_ + other.re_, im_ + other.im_); }
    ComplexNumber Sub(ComplexNumber other)
    { return ComplexNumber(re_ - other.re_, im_ - other.im_); }
    void Print() { std::cout << re_ << (im_ < 0 ? "" : "+") << im_ << "i\n"; }
};
int main(int argc, char** argv)
{
    ComplexNumber c1(1, 1), c2(4, 2);
    ComplexNumber sum = c1.Add(c2);
    ComplexNumber sub = c1.Sub(c2);
    sum.Print(); // 5+3i
    sub.Print(); // -3-1i
}
```

Решение

Вроде бы всё хорошо, класс полностью решает поставленную задачу

Однако всё-таки выглядит не очень красиво. Было бы гораздо лучше, если бы мы могли складывать комплексные числа как обычные - с помощью знака “+”, а вычитать с помощью знака “-”

Было бы гораздо нагляднее, лучше бы читалось, код выглядел бы чище

Перегрузка операторов

Хорошая новость - C++ даёт нам такую возможность

Мы можем **перегрузить** операторы “+” и “-” (а также много других) для класса – то есть создать **свою версию** этих операторов, которая применительно к этому классу будет делать то, что нужно нам

Перегрузка операторов

Свои операторы, к сожалению, создавать нельзя - только перегружать уже существующие

Для каждого перегружаемого оператора есть **специальный метод**, который является **полной формой** для этого оператора - сам оператор, по сути, является “синтаксическим сахаром”, то есть более короткой и удобной для использования формой. Когда мы перегружаем оператор - мы перегружаем тот самый метод

Перегрузка операторов. Синтаксис

Для большинства операторов писать код, перегружающий эти операторы, можно как внутри класса, так и вне его

Сигнатура специального метода у каждого оператора своя. Тем не менее, можно выделить общую синтаксическую часть, справедливую для всех операторов

<возвращаемый тип> operator<символ оператора>([<аргументы>])
{ <реализация> }

Какие бывают операторы

В целом все операторы можно классифицировать следующим образом:

- перегружаемые
  - унарные
  - бинарные
  - остальные
- неперегружаемые

Унарные и бинарные операторы

Унарные операторы - это операторы, которым требуется всего один аргумент (или операнд)

Бинарные операторы - это операторы, которым требуется два операнда

Приведите пример

унарных и бинарных операторов,
которые вы знаете

Напишите в чат

Унарные и бинарные операторы

Бинарные операторы - это, например, сложение, вычитание, умножение и многие другие - им всем требуется два операнда, из которых получается третий

Унарные операторы - это, например, постфиксный и префиксный инкремент и декремент (++ и --). Также можно вспомнить логическое отрицание (!), но есть ещё и другие - мы сегодня с ними познакомимся

Бинарные операторы

2

Список перегружаемых бинарных операторов

В C++ можно перегрузить следующие бинарные операторы:

+	сложение
+=	сложение с присвоением
-	вычитание
-=	вычитание с присвоением
*	умножение
*=	умножение с присвоением
/	деление
/=	деление с присвоением
%	остаток от деления
%=	остаток с присвоением
->	доступ к члену
->*	доступ по указателю на член
=	присвоение
,	запятая
&&	логическое И
||	логическое ИЛИ

<	меньше
<=	меньше или равно
>	больше
<=	больше или равно
==	равенство
!=	неравенство
<<	сдвиг налево
<<=	сдвиг налево с присвоением
>>	сдвиг направо
>>=	сдвиг направо с присвоением
&	битовое И
&=	битовое И с присвоением
|	битовое ИЛИ
|=	битовое ИЛИ с присвоением
^	исключающее ИЛИ
^=	исключающее ИЛИ с присвоением

Бинарные операторы. Синтаксис внутри класса

При определении бинарных операторов **внутри** класса используется следующий синтаксис:

<возвращаемый тип> operator<символ оператора>(<аргумент 1>)

На **<возвращаемый тип>** не накладывается никаких ограничений, но чаще всего это будет класс или ссылка на класс, в котором перегружается оператор

<символ оператора> – это один из символов с предыдущего слайда

Вы также можете просто объявить оператор, а определить его позднее - это делается как у обычной функции или метода

Бинарные операторы. Синтаксис внутри класса

При определении бинарных операторов **внутри** класса используется следующий синтаксис:

<возвращаемый тип> operator<символ оператора>(<аргумент 1>)

Вопрос: Внутри класса бинарный оператор имеет всего один **<аргумент>**
Как вы думаете, почему? Напишите в чат

Бинарные операторы. Синтаксис внутри класса

При определении бинарных операторов **внутри** класса используется следующий синтаксис:

<возвращаемый тип> operator<символ оператора>(<аргумент 1>)

Вопрос: Внутри класса бинарный оператор имеет всего один **<аргумент>**
Как вы думаете, почему? Напишите в чат

Ответ: внутри класса у бинарного оператора всего один аргумент, потому что второй (на самом деле он первый) - это сам экземпляр класса, в котором вызывается оператор

Бинарные операторы. Синтаксис снаружи класса

При определении бинарных операторов **снаружи** класса оператор является глобальной функцией, используется следующий синтаксис:

<возвращаемый тип> operator<символ оператора>(<аргумент 1>, <аргумент 2>)

Определение (и объявление) оператора снаружи класса отличается явным наличием первого операнда и кое-чем ещё

Вопрос: как вы думаете, какое ещё отличие между перегрузкой операторов внутри и вне класса? Напишите в чат

Бинарные операторы. Синтаксис снаружи класса

При определении бинарных операторов **снаружи** класса оператор является глобальной функцией, используется следующий синтаксис:

<возвращаемый тип> operator<символ оператора>(<аргумент 1>, <аргумент 2>)

Ответ: так как оператор, объявленный и определённый вне класса - это глобальная функция, то по умолчанию она не имеет доступа к приватным и защищённым членам класса

А оператор, объявленный и определённый внутри класса - имеет

Кроме этого, реализация оператора снаружи класса позволяет вам менять операнды местами, если они разного типа. В реализации оператора внутри класса первый операнд всегда типа класса

Бинарные операторы. Пример

Вооружившись новым знанием, улучшим наш пример с комплексными числами

Вместо метода Add мы перегрузим оператор +

Вместо метода Sub мы перегрузим оператор -

А ещё добавим перегрузку операторов + и -, в которые будем принимать обычное целое число - такая операция изменит только действительную часть комплексного числа

Улучшенное решение

```
class ComplexNumber
{
    double re_, im_;
public:
    ComplexNumber(double re, double im) { re_ = re; im_ = im; }
    ComplexNumber operator+(ComplexNumber other)
    { return ComplexNumber(re_ + other.re_, im_ + other.im_); }
    ComplexNumber operator+(int other) { return ComplexNumber(re_ + other, im_); }
    ComplexNumber operator-(ComplexNumber other)
    { return ComplexNumber(re_ - other.re_, im_ - other.im_); }
    ComplexNumber operator-(int other) { return ComplexNumber(re_ - other, im_); }
    void Print() { std::cout << re_ << (im_ < 0 ? "" : "+") << im_ << "i\n"; }
};
int main(int argc, char** argv)
{
    ComplexNumber c1(1, 1), c2(4, 2);
    ComplexNumber sum_c_c = c1 + c2;
    ComplexNumber sum_c_i = c1 + 3;
    ComplexNumber sub_c_c = c1 - c2;
    ComplexNumber sub_c_i = c1 - 5;
    sum_c_c.Print(); // 5+3i
    sum_c_i.Print(); // 4+1i
    sub_c_c.Print(); // -3-1i
    sub_c_i.Print(); // -4+1i
}
```

Перегрузка операторов сравнения

Перегрузим для наших комплексных чисел операторы **сравнения**

Вспомним, какие они бывают: >, >=, <, <=, ==, !=

Сравнивать комплексные числа будем по модулю - одно число больше другого, если модуль этого комплексного числа больше модуля другого

Модуль вычисляется как квадратный корень из суммы квадратов вещественной и мнимой частей - определим функцию Abs, которая будет его вычислять

Как вы понимаете, нет смысла писать практически один и тот же код во всех 6 операциях - ведь, например, операция > является обратной по отношению к операции <=

Как вы думаете,

в скольких операторах сравнения
нам надо написать *уникальный* код,
чтобы покрыть все 6 операторов?

Напишите в чат

Перегрузка операторов сравнения

Уникальный код нужно написать всего в **двух** операторах сравнения - в одном из операторов != и ==, и в одном из операторов <, <=, >, >=

Операторы == и != обратны друг другу

Оператор < - это то же самое, что оператор >, но с аргументами, поменянными местами

Операторы < и >=, а также > и <= - обратны друг другу

Перегружаем операторы сравнения

```
class ComplexNumber
{
    double re_, im_;
public:
    // предыдущий код
    double Abs() { return std::sqrt(re_ * re_ + im_ * im_); }
    bool operator==(ComplexNumber other) { return Abs() == other.Abs(); }
    bool operator!=(ComplexNumber other) { return !(*this == other); }
    bool operator>(ComplexNumber other) { return Abs() > other.Abs(); }
    bool operator<(ComplexNumber other) { return other > *this; }
    bool operator>=(ComplexNumber other) { return !(*this < other); }
    bool operator<=(ComplexNumber other) { return !(*this > other); }
};
int main(int argc, char** argv)
{
    ComplexNumber c1(1, 1), c2(4, 2);
    std::cout << (c1 < c2) << std::endl;  // 1
    std::cout << (c1 <= c2) << std::endl; // 1
    std::cout << (c1 > c2) << std::endl;  // 0
    std::cout << (c1 >= c2) << std::endl; // 0
    std::cout << (c1 == c2) << std::endl; // 0
    std::cout << (c1 != c2) << std::endl; // 1
}
```

Перегрузка оператора присвоения

Оператор присвоения тоже можно перегрузить. На него накладываются ограничения:

- он может быть только членом класса, глобальной функцией быть не может
- не наследуется
- если оператор явно не перегружен - компилятор автоматически генерирует свою реализацию (как с конструктором по умолчанию)

Чтобы можно было организовывать **цепочки присвоений**, оператор присвоения должен возвращать **ссылку на самого себя**

Перегружаем оператор присвоения

```
class ComplexNumber
{
    double re_, im_;
public:
    // предыдущий код
    ComplexNumber& operator=(const ComplexNumber& other)
    {
        re_ = other.re_;
        im_ = other.im_;
        return *this;
    }
};
int main(int argc, char** argv)
{
    ComplexNumber c1(1, 1), c2(4, 2), c3(5, 6);
    c1 = c3 = c2; // цепочка присвоений
    c3.Print(); // 4+2i
    c1.Print(); // 4+2i
}
```

Перегрузка операторов доступа к члену

Один из самых хитрых бинарных операторов - оператор **доступа к члену** (->). Похожий оператор ->* работает практически так же за одним небольшим исключением, которое мы пока не можем рассмотреть за недостатком знаний

Оператор -> обязан быть членом класса (т.е. глобальной функцией он быть не может). Несмотря на то, что он относится к классу бинарных - у него нет аргумента

Перегрузка операторов доступа к члену

Один из самых хитрых бинарных операторов - оператор **доступа к члену** (->). Похожий оператор ->* работает практически так же за одним небольшим исключением, которое мы пока не можем рассмотреть за недостатком знаний

Оператор -> обязан быть членом класса (т.е. глобальной функцией он быть не может). Несмотря на то, что он относится к классу бинарных - у него нет аргумента

Возвращаемое значение оператора используется для поиска члена, указанного после оператора в вызывающем коде - то есть происходит поиск этого члена в объекте, возвращённом перегруженным оператором ->

В случае, если возвращённое значение **не** является указателем - у него также вызывается оператор ->. Это происходит до тех пор, пока не будет возвращён указатель, где уже и будет произведён поиск указанного члена

Перегружаем оператор доступа к члену

```
struct RealStruct
{ int a; };

struct Wrapper1Struct
{
    RealStruct* wrappee;
    RealStruct* operator->() { return wrappee; }
};

struct Wrapper2Struct
{
    Wrapper1Struct* wrappee;
    Wrapper1Struct& operator->() { return *wrappee; }
};
int main(int argc, char** argv)
{
    RealStruct rs = { 5 };
    Wrapper1Struct w1 = { &rs };
    Wrapper2Struct w2 = { &w1 };
    std::cout << rs.a << " " << w1->a << " " << w2->a << std::endl; // 5 5 5
}
```

Перерыв

Унарные, остальные и неперегружаемые операторы

2

Список перегружаемых унарных операторов

В C++ можно перегрузить следующие унарные операторы:

`!`	логическое НЕТ
`&`	взятие адреса
`~`	битовое дополнение
`*`	разыменование указателя
`+`	унарный плюс
`-`	унарный минус
`++`	инкремент - постфиксный и префиксный
`--`	декремент - постфиксный и префиксный
	операторы приведения типа

Унарные операторы. Синтаксис внутри класса

При определении унарных операторов внутри класса используется следующий синтаксис:

<возвращаемый тип> operator<символ оператора>()

На **<возвращаемый тип>** не накладывается никаких ограничений, но чаще всего это будет класс или ссылка на класс, в котором перегружается оператор

<символ оператора> – это один из символов с предыдущего слайда

Вы также можете просто объявить оператор, а определить его позднее - это делается как у обычной функции или метода

Унарные операторы. Синтаксис снаружи класса

При определении унарных операторов снаружи класса оператор является глобальной функцией, используется следующий синтаксис:

<возвращаемый тип> operator<символ оператора>(<аргумент 1>)

Как вы думаете,

в каких случаях может потребоваться перегружать операторы & и * ?

Напишите в чат

Унарные операторы

Правильно! Операторы & и * может потребоваться перегрузить в том же случае, что и оператор -> - в случае использования так называемых классов-обёрток (wrapper). Иногда такие классы называют прокси (proxy)

Идея в том, что когда вы работаете с прокси классом, вы на самом деле хотите работать с тем, что он оборачивает. В таком случае оператор & прокси класса может возвращать адрес значения, которое он оборачивает

Точно так же указатель на прокси класс разыменовывается в значение, которое он оборачивает

Операторы инкремента и декремента

Отдельного внимания заслуживает оператор инкремента, а точнее, разные его версии - префиксная (++a) и постфиксная (a++)

Чтобы различать их между собой, для них были сделаны разные сигнатуры:

- **префиксная** форма подчиняется правилам синтаксиса унарных операторов - внутри класса у него 0 аргументов, снаружи 1
  <возвращаемый тип> operator++(); (внутри)
  <возвращаемый тип> operator++(<аргумент 1>); (снаружи)

- **постфиксная** форма принимает **дополнительный** аргумент типа int, который не используется в реализации оператора
  <возвращаемый тип> operator++(int); (внутри)
  <возвращаемый тип> operator++(<аргумент 1>, int); (снаружи)

Всё вышесказанное относится и к оператору декремента

Перегружаем операторы инкремента

```
class ComplexNumber
{
    double re_, im_;
public:
    // предыдущий код
    ComplexNumber& operator++() { re_++; im_++; return *this;}
    ComplexNumber operator++(int)
    {
        ComplexNumber temp = *this;
        ++(*this);
        return temp;
    }
};
int main(int argc, char** argv)
{
    ComplexNumber c1(1, 1), c2(4, 2), c3(1, 1);
    ComplexNumber sum_pre = ++c1 + c2;
    ComplexNumber sum_post = c3++ + c2;
    sum_pre.Print(); // 6+4i
    sum_post.Print(); // 5+3i
    c1.Print(); // 2+2i
    c3.Print(); // 2+2i
}
```

Операторы приведения типа

Операторы приведения типа позволяют нам приводить значение типа нашего класса к другим типам

Операторы приведения бывают **явными** (**explicit**) и **неявными** (implicit - таковы они по умолчанию)

Явные операторы приведения вызываются **только** при явном приведении - то есть если вы пишете перед значением в скобках тип, к которому хотите привести

Операторы приведения типа

Неявные операторы вызываются, например, если вы пытаетесь передать в функцию, ожидающую в качестве аргумента значение какого-то одного типа, – значение другого типа

Приводить значения другого типа к типу нашего класса нам позволяют конструкторы

Операторы приведения типа. Синтаксис

Синтаксис операторов приведения типа немного отличается от уже привычного нам:

[explicit] operator <тип>()

Чтобы сделать оператор приведения явным, нужно указать ключевое слово **explicit**

<тип> приведения определяет, для какого типа этот оператор приведения и одновременно определяет тип возвращаемого значения. **<тип>** должен быть объявлен **до** оператора и не может быть классом, структурой, перечислением или typedef

Операторы приведения могут быть виртуальными

Не принимают аргументов

Перегружаем оператор приведения

```
class ComplexNumber
{
    double re_, im_;
public:
    // предыдущий код
    operator double()
    {
        return Abs();
    }
};
int main(int argc, char** argv)
{
    ComplexNumber c1(1, 1);
    double c1_abs = c1; // 2.82843
}
```

Остальные операторы

4 перегружаемых оператора, которые нельзя отнести ни к унарным, ни к бинарным:

`( )`	вызов функции
`[ ]`	доступ по индексу
`new`	создание объекта в куче
`delete`	освобождение памяти объекта из кучи

Последние два перегружаются особенно редко в специфических случаях

Остальные операторы

Оператор ( ) очень похож на оператор приведения типа, но имеет возвращаемый тип и может иметь аргументы. Он позволяет **вызывать ваш объект как функцию**

<возвращаемый тип> operator()(<аргументы>)

Такой оператор обязан быть членом класса

Остальные операторы

Оператор [ ] позволяет вам использовать ваши объекты как массивы (в том числе и ассоциативные) - перегрузив его, вы сможете обращаться к вашему объекту по индексу

<возвращаемый тип> operator[](<аргумент>)

Такой оператор обязан быть членом класса. Он принимает всего один **<аргумент>**, который может иметь любой тип

С помощью перегрузки этого оператора работают контейнеры из стандартной библиотеки - например, vector

Неперегружаемые операторы

В C++ также есть операторы, перегрузить которые нельзя:

`.`	выбор члена
`.*`	выбор члена по указателю на него
`::`	разрешение области видимости
`? :`	условный тернарный оператор
`#`	препроцессорное преобразование в строку
`##`	препроцессорная конкатенация

Итоги

Итоги занятия

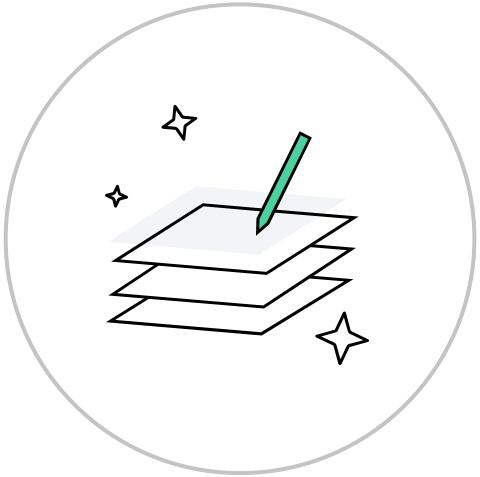
Сегодня мы

1. Разобрались, что такое перегрузка операторов
2. Познакомились с тем, какие бинарные операторы можно перегружать
3. Узнали, какие унарные операторы можно перегружать
4. Выяснили, какие ещё операторы поддаются перегрузке, а какие нет

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

- Перегрузка операторов

Задавайте вопросы и пишите отзыв о лекции