Перегрузка операций

Проблема

Допустим, мы реализуем функционал для работы с комплексными числами.

```
struct Complex {
  double re;
  double im;
};

Complex Sum(const Complex& x, const Complex& y);

Sum(Complex(1, 2), Complex(-2, 3));
1 + 3;
```

Возникает ассиметричность интерфейсов сложения - в одних ситуациях нужно вызывать специальную функцию, а в других использовать `+`.

Перегрузка операций

В C++ есть механизм, позволяющий определить поведение операций при работе с классами, структурами и перечислениями. Этот механизм называется *перегрузкой операций*.

Перегрузка операций наравне с перегрузкой функций и шаблонами функций является примером проявления *статического полиморфизма*.

Синтакис перегрузки операций

Синтаксис перегрузки операций совпадает с определением функций: сначала идет тип возвращаемого значения, затем имя операции (`operator<op>`, `<op>` - символ операции), далее список аргументов операции и тело операции.

```
Complex operator+(const Complex& x, const Complex& y) {
  return Complex(x.re + y.re, x.im + y.im);
}
Complex(1, 2) + Complex(3, 4);  // Ok: 4 + 6i
// Эквивалентно operator+(Complex(1, 2), Complex(3, 4))
```

Определенная вами операция может выполнять любые действия и возвращать все, что угодно.

```
void operator+(const Complex& x, const Compex& y) {
  std::cout << "Hello, world!";
}
Complex(1, 2) + Complex(3, 4);  // Ok, но зачем?.. : Hello, world!
```

Синтакис перегрузки операций

Альтернативно, можно объявить операцию как член класса, а не как внешнюю функцию. В этом случае текущий объект автоматически является левым операндом, и в качестве аргумента достаточно передать только правый (при необходимости)

```
Complex Complex::operator+(const Complex& y) const {
  return Complex(re + y.re, im + y.im);
}

Complex(1, 2) + Complex(3, 4);  // Ok: 4 + 6i
// Эквивалентно Complex(1, 2).operator+(Complex(3, 4))
```

Перегрузка унарных операций

Некоторые операции принимают ровно 1 аргумент (унарный `+`, унарный `-`, унарная `*` и т.д.).

В этом случае соответствующая перегрузка тоже принимает 1 аргумент.

```
// Реализация в виде внешней функции
Complex operator-(const Complex& x) {
  return {-x.re, -x.im};
}

// Реализация в виде метода класса
Complex Complex::operator-() const {
  return {-re, -im};
}
```

Правила и особенности перегрузки операций

Правила перегрузки операций

1. Нельзя переопределять операции с примитивными типами (хотя бы один из аргументов должен быть пользовательского типа).

```
const char* operator*(const char* str, int n);  // CE
Complex operator+(const Complex& x, int y);     // Ok
```

2. Нельзя вводить новые операции в язык (например, `**` для возведения в степень).

```
Complex operator**(const Complex& x, int n);  // CE
```

3. Нельзя менять арность и приоритет операций.
4. Нельзя переопределять операции `::`, `.`, `?:`, `.*`.
5. Операции `=`, `()`, `[]`, `->` могут быть перегружены только в виде методов.
6. Операции `&&` и `||` теряют свойство "короткого вычисления", (до C++17 теряли строгий порядок вычисления вместе с оператором `,`).

Особенности перегрузки бинарных операций

Какой вариант предпочесть при реализации данных операций для своего класса?

```
Complex::Complex(double re = 0, double im = 0);  // implicit
```

```
Complex operator+(const Complex& x, const Complex& y);

Complex(1, 1) + 2;  // Ok: operator+(Complex(1, 1), 2);
1 + Complex(2, 2);  // Ok: operator+(1, Complex(2, 2));
```

```
Complex Complex::operator+(const Complex& y) const;

Complex(1, 1) + 2;  // Ok: Complex(1, 1).operator+(2);
1 + Complex(2, 2);  // CE
```

```
error: no match for 'operator+' (operand types are 'int' and 'Complex')
```

Внешняя функция позволяет использовать неявные пр-я в обоих аргументах

Перегрузка пре- инкремента и декремента

Операции префиксного инкремента и декремента - унарные операторы, которые (для базовых типов) изменяют значение переменной на +1/-1 и *возвращают ссылку на ту же переменную*.

Это стоит учитывать при перегрузке операций для своего класса (хотя, конечно, компилятор этого проверять не будет).

```
// Так
Complex& operator++(Complex& value) {
  ++value.re;
  return value;
}

// Или так
Complex& Complex::operator++() {
  ++re;
  return *this;
}
```

Перегрузка пост- инкремента и декремента

Существуют постфиксные версии инкремента и декремента. Их отличие от префиксных в том, что они возвращают копию старого значения переменной. Чтобы перегрузить постфиксную операцию надо (внезапно!) добавить фиктивный аргумент типа `int`.

```
// Так
Complex operator++(Complex& value, int) {
  auto old_value = value;
  ++value.re;
  return old_value;
}

// Или так
Complex Complex::operator++(int) {
  auto old_value = *this;
  ++re;
  return old_value;
}
```

Перегрузка операции индексирования

Операция индексирования (aka операция доступа к элементу массива) может быть перегружена только *в форме метода класса* и может иметь *только один аргумент*.

```
int IntArray::operator[](size_t i) const {  // только для чтения
  return buffer_[i];
}

int& IntArray::operator[](size_t i) {  // чтение и запись
  return buffer_[i];
}

// А так нельзя
// int& Matrix::operator[](size_t i, size_t j);
```

Перегрузка круглых скобок

Операция "круглые скобки" (aka операция функционального вызова) может быть перегружена только *в форме метода класса* и может иметь *сколько угодно аргументов*.

```
struct Printer {
  void operator()(int i, const char* str, char c) const {
    std::cout << i << ' ' << str << ' ' << c;
  }
};

Printer print;
print(1, "Hello", '+');  // 1 Hello +
```

Перегрузка операции присваивания

- Классическая операция присваивания присваивает значение правого аргумента левому аргументу и *возвращает ссылку на левый операнд*.
- Ваша реализация не обязана удовлетворять этому соглашению, но мир станет лучше, если вы будете следовать этим правилам.
- Операция присваивания одна из четырех операций, которая может быть реализована только как метод класса.
- Операция присваивания - часть "правила трех".

```
Complex& Complex::operator=(const Complex& other) {
  re = other.re;
  im = other.im;
  return *this
}
```

Перегрузка операции присваивания

```
Complex& Complex::operator=(const Complex& other) {
  re = other.re;
  im = other.im;
  return *this
}
```

Если вы не реализуете свой оператор присваивания компилятор создаст за вас свой, который присваивает каждое поле поотдельности и возвращает ссылку.

То есть код выше излишний - без него оператор присваивания был бы создан автоматически и делал бы то же самое.

Как и ранее, можно явно попросить компилятор создать свою версию:

```
Complex& Complex::operator=(const Complex& other) = default;
```

Перегрузка операции присваивания

Вспомним класс стека на динамическом массиве

```
Stack& Stack::operator=(Stack& other) = default;

Stack a;
Stack b = a;  // это не присваивание!
// ...

a = b;        // присваивание, но в чем проблема?
```

Если в классе есть нетривиальное управление ресурсами, то необходимо самостоятельно переопределять присваивание (правило трех).

Проблема самоприсваивания

Рассмотрим следующую реализацию присваивания стека

```
Stack& Stack::operator=(const Stack& other) {
  delete[] buffer_;
  buffer_ = new T[kCapacity];
  size_ = other.size_;
  for (size_t i = 0; i < size_; ++i) {
    buffer_[i] = other.buffer_[i];
  }
  return *this;
}

Stack a;
// ...
a = a;  // что может пойти не так?
```

Проблема самоприсваивания

Рассмотрим следующую реализацию присваивания стека

```
Stack& Stack::operator=(const Stack& other) {
    delete[] buffer_;
    buffer_ = new T[kCapacity];
    size_ = other.size_;
    for (size_t i = 0; i < size_; ++i) {
        buffer_[i] = other.buffer_[i];
    }
    return *this;
}

Stack a;
// ...
a = a;  // что может пойти не так?
```

Чтобы избежать этой проблемы, нужно добавить проверку в начале

```
if (this != &other) { ... }
```

Составные операции присваивания

Составные операции присваивания имеют вид `+=`, `-=`, `*=` и т.д. В языке C++ они изменяют левый операнд и *возвращают ссылку на левый операнд*.
В отличие от простого присваивания на перегрузку этих операторов не накладывается ограничений (это обычный бинарный оператор).
Однако, как правило, его реализуют как член класса и возвращают ссылку на левый операнд.

```
// Можно так
Complex& Complex::operator+=(const Complex& other) {
    re += other.re; im += other.im;
    return *this;
}

// А можно так (в отличие от operator=)
Complex& operator+=(Complex& lhs, const Complex& rhs) {
    lhs.re += rhs.re; lhs.im += rhs.im;
    return lhs;
}
```

Перегрузка побитового сдвига

Так сложилось, что побитовый сдвиг переопределяют не по прямому назначению, а для целей ввода из потока и вывода в поток.

```
std::cin >> x;
std::cout << x;
```

Потоки ввода-вывода

Объект `std::cin` - является объектом класса `std::istream`, который осуществляет считывание потока из консоли (cin = Console INput).

Объект `std::cout` - является объектом класса `std::ostream`, который осуществляет запись потока в консоль (cout = Console OUTput).

Заголовочный файл `<iostream>` включает множество других классов потоков, в том числе `std::ifstream` и `std::ofstream` (ввод/вывод в файл), `std::istringstream` и `std::ostringstream` (ввод/вывод в строку).

```
std::istringstream sin("1 2 3");
sin >> x >> y >> z;  // x = 1, y = 2, z = 3

std::ostringstream sout;
sout << x << ", " << y << ", " << z;
sout.str();           // returns "1, 2, 3"
```

Операторы побитового сдвига (ввода/вывода)

Как это работает?

```
int x;
std::cin >> x;
```

Где-то в `<iostream>` перегружена операция `>>`, где в качестве левого операнда принимается объект типа `std::istream`, а в качестве правого - `int`.

Аналогично для оператора вывода а поток.

А что он возвращает?

Операторы побитового сдвига (ввода/вывода)

Так как хотим, чтобы работал следующий код

```
int x;
double y;
std::cin >> x >> y; // То же, что (std::cin >> x) >> y;
```

Необходимо, чтобы `>>` возвращал левый операнд (поток). Тогда после ввода `x` вернется `std::cin` и вызовется следующая операция с `y`.

```
std::istream& operator>>(std::istream& is, int& value);
std::ostream& operator<<(std::ostream& os, int value);
```

Операторы побитового сдвига (ввода/вывода)

Определим ввод-вывод для класса Complex

```
std::ostream& operator<<(std::ostream& os, const Complex& value) {
  os << value.re << " + " << value.im << 'i';
  return os;
}

std::istream& operator>>(std::istream& is, Complex& value) {
  is >> value.re >> value.im;
  return is;
}
```

При реализации `operator>>` может возникнуть проблема - как считать и заполнить данные, если поля приватные?

Поговорим о дружбе

Допустим, сделали поля `Complex` приватными.

```
std::istream& operator>>(std::istream& is, Complex& value) {
    is >> value.re_ >> value.im_;  // CE
    return is;
}
```

Есть ли способ разрешить данной функции (а может и некоторым другим) обращаться к приватным полям класса?

Да! Для этого класс должен объявить эту функцию своим другом

Ключевое слово `friend`

Друзья класса - внешние по отношению к классу сущности, которые имеют доступ к приватной и защищенной частям класса.

- Можно объявлять другие функции другом класса:

```
class Stack {
  // ...
  friend void Print(const Stack& stack);
};

void Print(const Stack& stack) {
  for (size_t i = 0; i < stack.size; ++i) { cout << stack.buffer[i] << ' '; }
}
```

- Друга можно сразу объявить и определить (создать):

```
class Stack {
    // ...
  friend void Print(const Stack& stack) { ... }
};
```

Ключевое слово `friend`

Друзья класса - внешние по отношению к классу сущности, которые имеют доступ к приватной и защищенной частям класса.

- Другом может быть метод другого класса, а также другой класс целиком:

```
class A {
  void f() { B b; b.g(); b.j(); }  // Ok
  void h() { B b; b.g(); b.j(); }  // CE: h() не друг для B
  friend class B;
};

class B {
  void g() { A a; a.f(); a.h(); }  // Ok
  void j() { A a; a.f(); a.h(); }  // Ok
  friend A::f();
};
```

Ключевое слово `friend`

```
class Complex {
  double re_;
  double im_;

  // ...
  friend std::istream& operator>>(std::istream& is, Complex& complex);
};

std::istream& operator>>(std::istream& is, Complex& complex) {
  is >> complex.re >> complex.im;
  return is;
}
```