C++

Fall 2020

compscicenter.ru

Башарин Егор

eaniconer@gmail.com
https://t.me/egorbasharin

Лекция VI

Friends & Operators

friend specifier

Позволяет получить доступ к приватным членам класса из:

- функций, не являющихся членами этого класса
- других классов

friend functions

Syntax:

```
friend function-declaration
```

```
struct X {
    friend int getI(const X&);
private:
    int i_ = 1;
};

int getI(const X& x) { return x.i_; }
```

friend functions

Syntax:

```
friend function-definition
```

```
struct X {
    friend void setI(X& x, int i) { x.i_ = i; }
private:
    int i_ = 1;
};

int main() {
    X x;
    setI(x, 3);
}
```

- `setI` — не член класса, `inline` функция, external linkage

friend functions

```
struct X {
    friend int f() { return 32; };
    friend int g(const X& x) { return 32; }
private:
    int i_ = 1;
};

int main() {
    f(); // Error
    X x;
    g(x); // OK
}
```

Компилятор находит функцию g, так как он использует знания о типах аргументов для поиска (ADL)

friend class

Syntax:

```
friend elaborated-class-specifier ;
```

```
struct X {
    friend class Y;
private:
    int i_ = 1;
};

class Y {
public:
    int getI(const X& x) { return x.i_; }
    void setI(X& x, int i) { x.i_ = i; }
};
```

friends

- дружественность не транзитивна: из (`A` друг `B`, `B` друг `C`) не следует (`A` друг `C`)
- часто применяется при перегрузке операторов

operator overloading

Позволяет использовать операторы с пользовательскими типами

operator overloading

overloaded operator

Function name syntax:

```
operator op
```

- возможно переопределить **почти все** операторы, за исключением:
  - `::` — scope resolution
  - `.` — доступ к члену класса
  - `.*` — доступ к члену класса по указателю
  - ?: — тернарный условный оператор

overloaded operator

Function name syntax:

```
operator op
```

Нельзя:

- ввести свой оператор: <>, **, etc.
- изменить приоритет и количество операндов
- переопределить опрератор если операнды имеют фундаментальные типы

overloaded operator

Prefix operator

Form:

```
@a
a.operator@() // as member
operator@(a)  // as non-member
```

Пример:

```
class IntHolder {
public:
  IntHolder& operator++() { ++x_; return *this; }
  int value() const { return x_; }
private:
  int x_ = 0;
};

int main() {
    IntHolder holder;
    std::cout << (++holder).value();
    std::cout << (holder.operator++().value()); // alter
}
```

overloaded operator

Postfix operator

Form:

```
a@
a.operator@(0)   // as member
operator@(a, 0)  // as not member
```

Пример:

```
class IntHolder {
public:
  IntHolder operator++(int) {
      IntHolder res = *this; ++x_; return res;
  }
  int value() const { return x_; }
private:
  int x_ = 0;
};
int main() {
    IntHolder holder;
    std::cout << (holder++).value();
    std::cout << (holder.operator++(0).value());
}
```

overloaded operator

Assignment operator

Syntax:

```
a = b
a.operator=(b) // !only member allowed!
```

overloaded operator

Assignment operator

Пример:

```
class IntHolder {
public:
  IntHolder(int x) : x_(x) {}
  IntHolder& operator=(const IntHolder& other) {
      if (this == &other) return *this;
      x_ = other.x_;
      return *this;
  }
private:
  int x_ = 0;
};
int main() {
    IntHolder a{1}, b{2};
    b = a;
    b.operator=(a);
}
```

move-assignment пока не рассматриваем

overloaded operator

Function call operator

Syntax:

```
a(args...)
a.operator()(args...) // !only member allowed!
```

overloaded operator

Function call operator

```
struct FunctionObject {
  int operator()(int arg) {
      return 2*arg;
  }
};

int main() {
    FunctionObject f;
    f(10); // call like function
    f.operator()(10);
}
```

overloaded operator

Array subscript operator

Syntax:

```
a[b]
a.operator[](b) // !only member allowed!
```

overloaded operator

Array subscript operator

```
struct Vector10t {
  T array[10] = {};

  T& operator[](size_t idx) { return array[idx]; }

  const T& operator[](size_t idx) const { return array[i
};

int main() {
    Vector10t v;
    v[0] = T{1};
    v[9] = T{2};
}
```

overloaded operator

infix operator

Syntax:

```
a @ b
a.operator@(b)   // as member
operator@(a, b)  // as non-member
```

overloaded infix operator

Ввод/вывод

```
struct Complex { int i; int j; };
std::ostream& operator<<(std::ostream& os, const Complex
    os << c.i << " " << c.j;
    return os;
}
std::istream& operator>>(std::istream& is, Complex& c) {
    is >> c.i >> c.j;
    return is;
}
int main() {
    std::stringstream ss{"1 2"};
    Complex c{};
    ss >> c; // operator>>(ss, c);
    std::cout << c; // operator<<(std::cout, c)
}
```

overloaded infix operator

Ввод/вывод

```
...
std::ostream& operator<<(std::ostream& os, const Complex
    os << c.i << " " << c.j;
    return os;
}
std::istream& operator>>(std::istream& is, Complex& c) {
    is >> c.i >> c.j;
    return is;
}
...
```

- non-member функция
- добавить friend объявление, если нужен доступ к приватным полям
- возвращаемый тип позволяет стоит цепочки последовательных операций ввода/вывода

overloaded infix operator

Арифметика

```
Complex operator+(const Complex& lhs, int rhs) {
    return {lhs.i + rhs, lhs.j};
}
Complex operator+(int lhs, const Complex& rhs) {
    return {lhs + rhs.i, rhs.j};
}

int main() {
    Complex c{1, 2};
    Complex c1 = c + 1; // operator+(c, 1)
    Complex c2 = 1 + c; // operator+(1, c)
}
```

- Обычно non-member фукнции, чтобы достигнуть симметричности `(1 + c)` и `(c + 1)`

overloaded infix operator

Операции сравнения

```
struct Complex { int i; int j; };

bool operator<(const Complex& lhs, const Complex& rhs) {
    return std::tie(lhs.i, lhs.j) < std::tie(rhs.i, lhs.
}

bool operator==(const Complex& lhs, const Complex& rhs)
    return lhs.i == rhs.i && lhs.j == rhs.j;
}
```

- Алгоритмы стандартной библиотеки ожидают `operator<` и `operator==`
- $>$, $\geqslant$, $\leqslant$ реализуются через `operator<`
- $\neq$ реализуется через `operator==`

user-defined conversion function

Function name syntax:

```
operator type          // implicit and explicit conversion
explicit operator type // explicit conversion
```

conversion to bool

Использование объектов нашего класса в условном выражении

```
struct Complex {
  int i;
  int j;
};

Complex randComplex() { return {rand(), rand()}; };

int main() {
    Complex c = randComplex();
    if (c) { // Error
        std::cout << "not null complex";
    }
}
```

conversion to bool

Использование объектов нашего класса в условном выражении

Попытка 1.

```
struct Complex {
  int i;
  int j;

  operator bool() const { return i == 0 && j == 0; }
};
```

conversion to bool

Проблемы текущей реализации

```
struct Complex {
  int i;
  int j;

  operator bool() const { return i == 0 && j == 0; }
};

Complex randComplex() { return {rand(), rand()}; };

int main() {
    Complex c = randComplex();
    if (c == 0) { // OK, why?
        // ...
    }
}
```

conversion to bool

Использование объектов нашего класса в условном выражении

Попытка 2.

```
struct Complex {
  int i;
  int j;

  explicit operator bool() const { return i == 0 && j ==
};

Complex randComplex() { return {rand(), rand()}; };

int main() {
    Complex c = randComplex();
    if (c == 0) { // Error
        // ...
    }
}
```

user-defined conversion function

Function name syntax:

```
operator type          // implicit and explicit conversion
explicit operator type // explicit conversion
```

Ограничения:

- В `type` не могут встречаться `()` и `[]`
- `type` не может быть фукнцией или массивом

user-defined conversion function

```
struct T {
  operator int(*)[3]() const { /*...*/ } // Error

  using arr_type = int[10];
  operator arr_type() const { /*...*/ }    // Error
  operator arr_type*() const { /*...*/ }   // OK

  using func = void(int);
  operator func() const { /* ... */ }      // Error
  operator func*() const { /*...*/ }       // OK
};
```