

---

# Лекция 9

---

## Перегрузка операций

# Операции со встроенными типами

```
int i, j, k;  
float x, y;  
  
k = i + j;  
if (x > y)  
    i++;
```

---

# Способы перегрузки операторов

2 способа объявления операторной функции:

- 1) как глобальной функции,
  - 2) как метода класса.
-

# Оператор как глобальная функция

Синтаксис объявления глобальной функции

```
тип operator# ( список_аргументов );
```

# - символ оператора (+, -, <, ++ и т.д.).

# Пример: класс КОМПЛЕКСНЫХ ЧИСЕЛ

1) Объявление класса и операторных функций (заголовочный файл модуля complex).

```
...  
class complex  
{  
    public:  
        float re, im;  
};  
  
complex operator+(complex a, complex b);  
complex operator-(complex a, complex b);  
complex operator*(complex a, complex b);  
complex operator/(complex a, complex b);  
...
```

## 2) Использование перегруженных операторов (файл драйвера).

```
#include <iostream>
#include "complex.h"

using namespace std;

void main()
{
    complex a(0., 1.), b(-1., 0.), c, d;
    c = a + b; // неявный вызов
    d = operator+(b, c); // явный вызов
    cout << "c=(" << c.re << ", " << c.im << ");";
    cout << "d=(" << d.re << ", " << d.im << ");";
    cin.get();
}
```

### 3) Реализация перегруженного оператора сложения (файл реализации модуля complex).

```
#include "complex.h"

complex operator+ (complex c1, complex c2)
{
    complex tmp;
    tmp.re = c1.re + c2.re;
    tmp.im = c1.im + c2.im;
    return tmp;
}
```

---

# Правила перегрузки операторов

1. Не допускается определять новые операторы (например, `**`).
  2. Не допускается перегружать операторы встроенных типов.
  3. Перегруженный оператор может быть методом класса или глобальной функцией.
  4. Операторы подчиняются правилам приоритета, группирования и числа операндов.
  5. Если имеется и унарная, и бинарная версия оператора (например, `&`, `*`, `+`, `-`), то они перегружаются отдельно.
  6. Перегруженные операторы не имеют аргументов по умолчанию.
  7. Все перегруженные операторы, кроме оператора присваивания, наследуются
-



---

# Дружественные функции

Если оператор определен как глобальная (внешняя) функция, то доступ к закрытым (`private`) и защищенным (`protected`) полям и методам для нее запрещен.

Однако это ограничение не касается **дружественных (`friend`)** функций, которые имеют доступ ко всем полям класса.


Дружественная функция объявляется внутри класса с ключевым словом `friend`, однако не является методом класса.

---

# Пример: класс complex и дружественные функции-операторы

```
...  
class complex  
{  
    private:  
        float r;  
        friend  
};  
...
```

```
complex operator~ (const complex c)  
{  
    complex tmp;  
    tmp.re = c.re;  
    tmp.im = -c.im;  
    return tmp;  
}
```



ДОСТУП К  
ЗАКРЫТЫМ  
ПОЛЯМ

---

# Оператор как метод класса

Операторная функция может быть объявлена не только как глобальная и дружественная, но и **как метод класса**.

В этом случае функция объявляется в открытой (public) части определения класса. Количество аргументов функции при этом на единицу меньше, чем арность соответствующего оператора.

Таким образом, унарные операторы (~, !, \*, &, ++,...) объявляются как методы без аргументов, а бинарные операторы (<,>, ==, &, ||, >>, <<, ...) - с одним аргументом.

---

# Пример: класс complex

Объявим операторные функции сложения (бинарный оператор +) комплексных чисел и операцию комплексного сопряжения (унарный оператор ~) как методы класса complex.

```
...  
class complex  
{  
    float re, im;  
public:  
    complex operator+(const complex a);  
    complex operator~();  
};  
...
```

# Реализация операторных методов

```
complex complex::operator+(const complex c)
```

```
{
```

```
    complex tmp;
```

```
    tmp.re = re + c.re;
```

```
    tmp.im = im + c.im;
```

```
    return tmp;
```

```
}
```

```
tmp.re = this->re + c.re;
```

```
tmp.im = this->im + c.im;
```

```
complex complex::operator~()
```

```
{
```

```
    complex tmp;
```

```
    tmp.re = re;
```

```
    tmp.im = -im;
```

```
    return tmp;
```

```
}
```

```
tmp.re = this->re;
```

```
tmp.im = -this->im;
```

## Вызов операторных функций:

### 1) явный (с указанием имени функции)

```
complex a(1.,0), b(0.,1.), c, d;  
c = a.operator+(b);  
d = c.operator~();
```

### 2) неявный (с помощью символа операции)

```
complex a(1.,0), b(0.,1.), c, d;  
c = a + b;  
d = ~c;
```

# Выбор способа перегрузки - I.

При выборе способа перегрузки оператора (глобальная функция или метод класса) необходимо принимать во внимание, что

1) некоторые операторы могут перегружаться только как глобальные функции (пример ниже);

2) некоторые операторы могут перегружаться только как методы класса (=, (), [], ->);

3) некоторые операторы не могут быть перегружены вообще (., .\* , ::, ?: , #, ##).

# Выбор способа перегрузки - II.

Несмотря на то, что выбор способа перегрузки оператора остается за программистом, существуют некоторые общепринятые рекомендации относительно этого выбора.

Оператор	Форма
Все унарные операторы	метод
= () [] -> ->*	только метод
+= -= /= *= ^= &=  = %= >>= <<=	метод
Остальные бинарные операторы	глоб. функция



---

# Особенности реализации некоторых операторов

1) Оператор присваивания =.

Реализуется компилятором **неявно** с использованием "поверхностного копирования". Должен быть **переопределен явно** в случаях, когда объект класса содержит неразделяемую ссылку (например, динамический указатель).

Для объектов с неразделяемыми ссылками необходима реализация "глубокого копирования". Кроме того, вместе с оператором = требуется также явным образом определить конструктор копирования и деструктор (**Правило Трех**).

---

## 1) объявление

```
complex& operator=(const complex& c);
```

## 2) определение (реализация)

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

---

## 2) Операторы потокового ввода/вывода (>>, <<)

При использовании операторов сдвига >> и << левым операндом всегда является потоковый объект (например, `cin >> x`, `cout << y`). В этом случае операторная функция может быть объявлена только как глобальная и дружественная.

---

## 1) объявление

```
friend ostream& operator<<(ostream& os, const complex c);  
friend istream& operator>>(istream& is, const complex c);
```

## 2) определение (реализация)

```
ostream& operator<<(ostream& os, const complex c)  
{  
    os << "(" << setprecision(2) << c.re;  
    os << "," << setprecision(2) << c.im << ")";  
    return os;  
}
```