Предефиниране на операции

Трифон Трифонов

Обектно-ориентирано програмиране,
спец. Компютърни науки, 1 поток,
спец. Софтуерно инженерство,
2016/17 г.

30 март – 6 април 2017 г.

Операции в C++

- C++ разполага с широк набор от операции

Операции в C++

- C++ разполага с широк набор от операции
- Всяка операция се характеризира с:

Операции в C++

- C++ разполага с широк набор от операции
- Всяка операция се характеризира с:
  - местност (едноместна, двуместна, триместна)

Операции в C++

- C++ разполага с широк набор от операции
- Всяка операция се характеризира с:
  - местност (едноместна, двуместна, триместна)
  - позиция спрямо аргументите (инфиксна, префиксна, постфиксна)

Операции в C++

- C++ разполага с широк набор от операции
- Всяка операция се характеризира с:
  - местност (едноместна, двуместна, триместна)
  - позиция спрямо аргументите (инфиксна, префиксна, постфиксна)
  - приоритет

Операции в C++

- C++ разполага с широк набор от операции
- Всяка операция се характеризира с:
  - местност (едноместна, двуместна, триместна)
  - позиция спрямо аргументите (инфиксна, префиксна, постфиксна)
  - приоритет
  - асоциативност за бинарните операции (лява, дясна)

Операции в C++

- C++ разполага с широк набор от операции
- Всяка операция се характеризира с:
  - местност (едноместна, двуместна, триместна)
  - позиция спрямо аргументите (инфиксна, префиксна, постфиксна)
  - приоритет
  - асоциативност за бинарните операции (лява, дясна)
- **Примери:**

Операции в C++

- C++ разполага с широк набор от операции
- Всяка операция се характеризира с:
  - местност (едноместна, двуместна, триместна)
  - позиция спрямо аргументите (инфиксна, префиксна, постфиксна)
  - приоритет
  - асоциативност за бинарните операции (лява, дясна)
- **Примери:**
  - \- е двуместна инфиксна лявоасоциативна операция

Операции в C++

- C++ разполага с широк набор от операции
- Всяка операция се характеризира с:
  - местност (едноместна, двуместна, триместна)
  - позиция спрямо аргументите (инфиксна, префиксна, постфиксна)
  - приоритет
  - асоциативност за бинарните операции (лява, дясна)
- **Примери:**
  - \- е двуместна инфиксна лявоасоциативна операция
  - също така - е и едноместна префиксна операция

Операции в C++

- C++ разполага с широк набор от операции
- Всяка операция се характеризира с:
  - местност (едноместна, двуместна, триместна)
  - позиция спрямо аргументите (инфиксна, префиксна, постфиксна)
  - приоритет
  - асоциативност за бинарните операции (лява, дясна)
- **Примери:**
  - \- е двуместна инфиксна лявоасоциативна операция
  - също така - е и едноместна префиксна операция
  - = е двуместна инфиксна дясноасоциативна операция

Операции в C++

- C++ разполага с широк набор от операции
- Всяка операция се характеризира с:
  - местност (едноместна, двуместна, триместна)
  - позиция спрямо аргументите (инфиксна, префиксна, постфиксна)
  - приоритет
  - асоциативност за бинарните операции (лява, дясна)
- **Примери:**
  - `-` е двуместна инфиксна лявоасоциативна операция
  - също така `-` е и едноместна префиксна операция
  - `=` е двуместна инфиксна дясноасоциативна операция
  - `!` е едноместна префиксна операция

Операции в C++

- C++ разполага с широк набор от операции
- Всяка операция се характеризира с:
  - местност (едноместна, двуместна, триместна)
  - позиция спрямо аргументите (инфиксна, префиксна, постфиксна)
  - приоритет
  - асоциативност за бинарните операции (лява, дясна)
- **Примери:**
  - `-` е двуместна инфиксна лявоасоциативна операция
  - също така `-` е и едноместна префиксна операция
  - `=` е двуместна инфиксна дясноасоциативна операция
  - `!` е едноместна префиксна операция
  - `++` е едноместна префиксна или постфиксна операция

Операции над обекти

- **Основен принцип в C++**
  Класовете са потребителски типове данни, с които трябва да може се работи както с примитивни типове данни

Операции над обекти

- **Основен принцип в C++**
  Класовете са потребителски типове данни, с които трябва да може се работи както с примитивни типове данни
- **Пример:**

```
Rational p = 2, q = 3 / p, r = 3;
if (p + q <= r)
  p += q;
else
  p *= r;
```

Предефиниране на операции

C++ позволява повечето вградени операции да бъдат предефинирани, така че да работят за обекти от произволен клас:

- аритметични (+, -, *, /, %)

Предефиниране на операции

C++ позволява повечето вградени операции да бъдат предефинирани, така че да работят за обекти от произволен клас:

- аритметични (+, -, *, /, %)
- логически (!, &&, ||)

Предефиниране на операции

C++ позволява повечето вградени операции да бъдат предефинирани, така че да работят за обекти от произволен клас:

- аритметични (+, -, *, /, %)
- логически (!, &&, ||)
- указателни (&, *, ->, [])

Предефиниране на операции

C++ позволява повечето вградени операции да бъдат предефинирани, така че да работят за обекти от произволен клас:

- аритметични (+, -, *, /, %)
- логически (!, &&, ||)
- указателни (&, *, ->, [])
- за сравнение (==, !=, <, >, <=, >=)

Предефиниране на операции

C++ позволява повечето вградени операции да бъдат предефинирани, така че да работят за обекти от произволен клас:

- аритметични (+, -, *, /, %)
- логически (!, &&, ||)
- указателни (&, *, ->, [])
- за сравнение (==, !=, <, >, <=, >=)
- побитови (&, |, ^, ~, <<, >>)

Предефиниране на операции

C++ позволява повечето вградени операции да бъдат предефинирани, така че да работят за обекти от произволен клас:

- аритметични (+, -, *, /, %)
- логически (!, &&, ||)
- указателни (&, *, ->, [])
- за сравнение (==, !=, <, >, <=, >=)
- побитови (&, |, ^, ~, <<, >>)
- за присвояване (=, +=, -=, *=, /=, %=, &=, |= , <<=, >>=, ++, --)

Предефиниране на операции

C++ позволява повечето вградени операции да бъдат предефинирани, така че да работят за обекти от произволен клас:

- аритметични (+, -, *, /, %)
- логически (!, &&, ||)
- указателни (&, *, ->, [])
- за сравнение (==, !=, <, >, <=, >=)
- побитови (&, |, ^, ~, <<, >>)
- за присвояване (=, +=, -=, *=, /=, %=, &=, |= , <<=, >>=, ++, --)
- за работа с паметта (`new`, `new[]`, `delete`, `delete[]`)

Предефиниране на операции

C++ позволява повечето вградени операции да бъдат предефинирани, така че да работят за обекти от произволен клас:

- аритметични (+, -, *, /, %)
- логически (!, &&, ||)
- указателни (&, *, ->, [])
- за сравнение (==, !=, <, >, <=, >=)
- побитови (&, |, ^, ~, <<, >>)
- за присвояване (=, +=, -=, *=, /=, %=, &=, |= , <<=, >>=, ++, --)
- за работа с паметта (`new`, `new[]`, `delete`, `delete[]`)
- операция за изброяване (,)

Предефиниране на операции

C++ позволява повечето вградени операции да бъдат предефинирани, така че да работят за обекти от произволен клас:

- аритметични (+, -, *, /, %)
- логически (!, &&, ||)
- указателни (&, *, ->, [])
- за сравнение (==, !=, <, >, <=, >=)
- побитови (&, |, ^, ~, <<, >>)
- за присвояване (=, +=, -=, *=, /=, %=, &=, |= , <<=, >>=, ++, --)
- за работа с паметта (`new`, `new[]`, `delete`, `delete[]`)
- операция за изброяване (,)
- операция за извикване на функция (())

Предефиниране на операции

C++ позволява повечето вградени операции да бъдат предефинирани, така че да работят за обекти от произволен клас:

- аритметични (+, -, *, /, %)
- логически (!, &&, ||)
- указателни (&, *, ->, [])
- за сравнение (==, !=, <, >, <=, >=)
- побитови (&, |, ^, ~, <<, >>)
- за присвояване (=, +=, -=, *=, /=, %=, &=, |= , <<=, >>=, ++, --)
- за работа с паметта (`new`, `new[]`, `delete`, `delete[]`)
- операция за изброяване (,)
- операция за извикване на функция (())
- операции за преобразуване на тип

Предефиниране на операции: ограничения

Следните операции не могат да бъдат предефинирани:

- условна операция (`?:`)
- операция за указване на област (`::`)
- операция за избор на член (`.`)
- операция за намиране на големина (`sizeof`)
- препроцесорни операции (#, ##)

Предефиниране на операции чрез член-функции

- <тип> operator<операция>([<тип>]) [const];

Предефиниране на операции чрез член-функции

- <тип> `operator`<операция>`(`[<тип>]`)` [`const`];
- <тип> <клас>`::operator`<операция>`(`[<тип> <име>]`)` [`const`]
  `{` <тяло> `}`

Предефиниране на операции чрез член-функции

- <тип> `operator`<операция>`(`[<тип>]`)` [`const`];
- <тип> <клас>`::operator`<операция>`(`[<тип> <име>]`)` [`const`] `{` <тяло> `}`
- **Примери:**

Предефиниране на операции чрез член-функции

- <тип> operator<операция>([<тип>]) [const];
- <тип> <клас>::operator<операция>([<тип> <име>]) [const] { <тяло> }
- **Примери:**
  - ```
    Rational operator-() const {
      return Rational(-numer, denom);
    }
    ```
    ```

Предефиниране на операции чрез член-функции

- <тип> operator<операция>([<тип>]) [const];
- <тип> <клас>::operator<операция>([<тип> <име>]) [const] { <тяло> }
- **Примери:**
  - ```
    Rational operator-() const {
      return Rational(-numer, denom);
    }

    Rational operator*(Rational const& r) const {
      return Rational(numer * r.numer, denom * r.denom);
    }
    ```
    ```

Предефиниране на операции чрез обикновени функции

- <тип> operator<операция>(<тип$_1$> <име$_1$>) { <тяло> }

Предефиниране на операции чрез обикновени функции

- <тип> operator<операция>(<тип$_1$> <име$_1$>) { <тяло> }
- <тип> operator<операция>(<тип$_1$> <име$_1$>,
  <тип$_2$> <име$_2$>) { <тяло> }

Предефиниране на операции чрез обикновени функции

- <тип> operator<операция>(<тип$_1$> <име$_1$>) { <тяло> }
- <тип> operator<операция>(<тип$_1$> <име$_1$>,
  <тип$_2$> <име$_2$>) { <тяло> }
- Поне един от <тип$_1$> и <тип$_2$> трябва да е (псевдоним към) потребителски дефиниран тип!

Предефиниране на операции чрез обикновени функции

- <тип> operator<операция>(<тип$_1$> <име$_1$>) { <тяло> }
- <тип> operator<операция>(<тип$_1$> <име$_1$>,
  <тип$_2$> <име$_2$>) { <тяло> }
- Поне един от <тип$_1$> и <тип$_2$> трябва да е (псевдоним към) потребителски дефиниран тип!
  - не може да се предефинират операциите върху примитивните типове

Предефиниране на операции чрез обикновени функции

- <тип> operator<операция>(<тип$_1$> <име$_1$>) { <тяло> }
- <тип> operator<операция>(<тип$_1$> <име$_1$>,
  <тип$_2$> <име$_2$>) { <тяло> }
- Поне един от <тип$_1$> и <тип$_2$> трябва да е (псевдоним към) потребителски дефиниран тип!
  - не може да се предефинират операциите върху примитивните типове
- **Пример:**

```
bool operator==(Rational const& r1, Rational const& r2) {
  return r1.getNumerator()   == r2.getNumerator() &&
         r1.getDenominator() == r2.getDenominator();
}
```

Прилагане на операции към обекти

Изразите с операции приложени върху обекти автоматично се преобразуват до извиквания на съответните предефиниращи функции или член-функции

- `r1 * r2` $\Longleftrightarrow$ `r1.operator*(r2)`
- `-r1` $\Longleftrightarrow$ `r1.operator-()`
- `r1 == r2` $\Longleftrightarrow$ `operator==(r1, r2)`

Предефиниране чрез обикновени или член-функции?

Кога се налага да предефинираме операции чрез обикновени функции?

- когато искаме да предефинираме операция за съществуващ клас без да променяме дефиницията му

Предефиниране чрез обикновени или член-функции?

Кога се налага да предефинираме операции чрез обикновени функции?

- когато искаме да предефинираме операция за съществуващ клас без да променяме дефиницията му
- когато искаме да предефинираме бинарна операция, чиито първи аргумент е от примитивен тип

Предефиниране чрез обикновени или член-функции?

Кога се налага да предефинираме операции чрез обикновени функции?

- когато искаме да предефинираме операция за съществуващ клас без да променяме дефиницията му
- когато искаме да предефинираме бинарна операция, чиито първи аргумент е от примитивен тип
- **Пример:**

Предефиниране чрез обикновени или член-функции?

Кога се налага да предефинираме операции чрез обикновени функции?

- когато искаме да предефинираме операция за съществуващ клас без да променяме дефиницията му
- когато искаме да предефинираме бинарна операция, чиито първи аргумент е от примитивен тип
- **Пример:**
  - Как да позволим изрази от вида 3 + r?

Предефиниране чрез обикновени или член-функции?

Кога се налага да предефинираме операции чрез обикновени функции?

- когато искаме да предефинираме операция за съществуващ клас без да променяме дефиницията му
- когато искаме да предефинираме бинарна операция, чиито първи аргумент е от примитивен тип
- **Пример:**
  - Как да позволим изрази от вида 3 + r?
  - ```
    Rational operator+(int x, Rational const& r) {
      return Rational(x * r.getDenominator()
                        + r.getNumerator(),
                      r.getDenominator());
    }
    ```
    ```

Приятелски функции

- **Проблем:** ако дефинираме операторна функция външна за класа, тя ще има само външен достъп (няма да вижда `private` компонентите)

Приятелски функции

- **Проблем:** ако дефинираме операторна функция външна за класа, тя ще има само външен достъп (няма да вижда `private` компонентите)
- **Решение:** Приятелски функции: функции, на които се позволява вътрешен достъп до компонентите на класа

Приятелски функции

- **Проблем:** ако дефинираме операторна функция външна за класа, тя ще има само външен достъп (няма да вижда `private` компонентите)
- **Решение:** Приятелски функции: функции, на които се позволява вътрешен достъп до компонентите на класа
- `friend` <тип> <име>`(`<параметри>`);`
- `friend` <тип> <име>`(`<параметри>`) { `<тяло>` }`

Приятелски функции

- **Проблем:** ако дефинираме операторна функция външна за класа, тя ще има само външен достъп (няма да вижда `private` компонентите)
- **Решение:** Приятелски функции: функции, на които се позволява вътрешен достъп до компонентите на класа
- `friend` <тип> <име>`(`<параметри>`);`
- `friend` <тип> <име>`(`<параметри>`) {` <тяло> `}`
- **Пример:**

```
friend Rational operator+(int x, Rational const& r) {
  return Rational(x * r.denom + r.numer, r.denom);
}
```

Приятелски класове

- Приятелски клас е клас, чиито член-функции имат право на вътрешен достъп

Приятелски класове

- Приятелски клас е клас, чиито член-функции имат право на вътрешен достъп
- `friend class` <име>`;`

Приятелски класове

- Приятелски клас е клас, чиито член-функции имат право на вътрешен достъп
- `friend class <име>;`
- **Пример:**

```
class Rational { ... friend class RationalVector;  ... };
class RationalVector {
  Rational x, y; ...
public:
  ...
  void flip() {
    x.numer = -x.numer; y.numer = -y.numer;
  }
};
```

Препоръки за предефинирането на операции

- Избирайте операции, които подходящо описват действието над вашия клас

Препоръки за предефинирането на операции

- Избирайте операции, които подходящо описват действието над вашия клас
- Стремете се операциите, които предефинирате да се използват по същия начин както за примитивните типове

Препоръки за предефинирането на операции

- Избирайте операции, които подходящо описват действието над вашия клас
- Стремете се операциите, които предефинирате да се използват по същия начин както за примитивните типове
- Използвайте приятелството разумно, само когато наистина е необходимо

Препоръки за предефинирането на операции

- Избирайте операции, които подходящо описват действието над вашия клас
- Стремете се операциите, които предефинирате да се използват по същия начин както за примитивните типове
- Използвайте приятелството разумно, само когато наистина е необходимо
- ```
  Rational& Rational::operator*=(Rational const& r) {
    numer *= r.numer; denom *= r.denom;
    return *this;
  }
  ```
  ```

Препоръки за предефинирането на операции

- Избирайте операции, които подходящо описват действието над вашия клас
- Стремете се операциите, които предефинирате да се използват по същия начин както за примитивните типове
- Използвайте приятелството разумно, само когато наистина е необходимо
- ```
  Rational& Rational::operator*=(Rational const& r) {
    numer *= r.numer; denom *= r.denom;
    return *this;
  }
  ```
- Какво не е наред с долния пример?

  ```
  double operator==(Rational& r1, Rational* p2) {
    return r1.numer == p2->numer && r1.denom == p2->denom;
  }
  ```
  ```

Предефиниране на някои операции

Операция за индексиране []

```
long& Rational::operator[](int x) {
  if (x == 0) return numer;
  if (x == 1) return denom;
  cerr << "Грешка!";
  return numer;
}
```

Операция за индексиране []

- ```
  long& Rational::operator[](int x) {
    if (x == 0) return numer;
    if (x == 1) return denom;
    cerr << "Грешка!";
    return numer;
  }
  ```
- `Rational r(2, 3);`
- `cout << r[0] << '/' << r[1]; // 2/3`
  ```

Операция за индексиране []

- ```
  long& Rational::operator[](int x) {
    if (x == 0) return numer;
    if (x == 1) return denom;
    cerr << "Грешка!";
    return numer;
  }
  ```
- `Rational r(2, 3);`
- `cout << r[0] << '/' << r[1]; // 2/3`
- `r[0] = 5; r[1] = 7; r.print(); // 5/7`
  ```

Операция за индексиране []

- ```
  long& Rational::operator[](int x) {
    if (x == 0) return numer;
    if (x == 1) return denom;
    cerr << "Грешка!";
    return numer;
  }
  ```
- `Rational r(2, 3);`
- `cout << r[0] << '/' << r[1]; // 2/3`
- `r[0] = 5; r[1] = 7; r.print(); // 5/7`
- Проблем: нарушава се капсулацията на класа!
  ```

Операции за вход (>>) и изход (<<)

- Искаме да позволим `cin >> r` и `cout << r`

Операции за вход (>>) и изход (<<)

- Искаме да позволим `cin >> r` и `cout << r`
- `cin` е обект от клас `istream`, а `cout` е обект от клас `ostream`

Операции за вход (>>) и изход (<<)

- Искаме да позволим `cin >> r` и `cout << r`
- `cin` е обект от клас `istream`, а `cout` е обект от клас `ostream`
- Тъй като `cin` и `cout` са първи аргументи, трябва да предефинираме чрез външна функция

Операции за вход (>>) и изход (<<)

- Искаме да позволим `cin >> r` и `cout << r`
- `cin` е обект от клас `istream`, а `cout` е обект от клас `ostream`
- Тъй като `cin` и `cout` са първи аргументи, трябва да предефинираме чрез външна функция
- **Примери:**

```
friend ostream& operator<<(ostream& o, Rational const& r){
  return o << r.numer << '/' << r.denom << endl;
}

friend istream& operator>>(istream& i, Rational& r) {
  char c;
  return i >> r.numer >> c >> r.denom;
}
```

Операции за вход (>>) и изход (<<)

- Искаме да позволим `cin >> r` и `cout << r`
- `cin` е обект от клас `istream`, а `cout` е обект от клас `ostream`
- Тъй като `cin` и `cout` са първи аргументи, трябва да предефинираме чрез външна функция
- **Примери:**

```
friend ostream& operator<<(ostream& o, Rational const& r){
  return o << r.numer << '/' << r.denom << endl;
}

friend istream& operator>>(istream& i, Rational& r) {
  char c;
  return i >> r.numer >> c >> r.denom;
}
```

- **Проблем:** нарушава се капсулацията на класа!

Операция за присвояване =

- Извиква се при присвояване на обект в друг обект

Операция за присвояване =

- Извиква се при присвояване на обект в друг обект
- Обикновено се предефинира при работа с динамична памет

Операция за присвояване =

- Извиква се при присвояване на обект в друг обект
- Обикновено се предефинира при работа с динамична памет
- **Идея:** разрушава старата памет, заделя нова и копира новите данни

Операция за присвояване =

- Извиква се при присвояване на обект в друг обект
- Обикновено се предефинира при работа с динамична памет
- **Идея:** разрушава старата памет, заделя нова и копира новите данни
- Ако не бъде дефинирана, се дефинира системна, която присвоява сляпо всички полета от единия обект на другия

Операция за присвояване = при динамична памет

Пример:

```
Player& operator=(Player const& p) {
  delete[] name;
  name = new char[strlen(p.name)+1];
  strcpy(name, p.name); score = p.score;
  return *this;
}
```

Операция за присвояване = при динамична памет

Пример:

- ```
  Player& operator=(Player const& p) {
    delete[] name;
    name = new char[strlen(p.name)+1];
    strcpy(name, p.name); score = p.score;
    return *this;
  }
  ```
- Защо връщаме `Player&`, а не просто `Player`?
  ```

Операция за присвояване = при динамична памет

Пример:

- ```
  Player& operator=(Player const& p) {
    delete[] name;
    name = new char[strlen(p.name)+1];
    strcpy(name, p.name); score = p.score;
    return *this;
  }
  ```
- Защо връщаме `Player&`, а не просто `Player`?
  - за да можем да използваме резултата като lvalue
  ```

Операция за присвояване = при динамична памет

Пример:

- ```
  Player& operator=(Player const& p) {
    delete[] name;
    name = new char[strlen(p.name)+1];
    strcpy(name, p.name); score = p.score;
    return *this;
  }
  ```
- Защо връщаме `Player&`, а не просто `Player`?
  - за да можем да използваме резултата като lvalue
  - `(p1 = p2).setName("Катнис Евърдийн");`
  ```

Операция за присвояване = при динамична памет

Пример:

- ```
  Player& operator=(Player const& p) {
    delete[] name;
    name = new char[strlen(p.name)+1];
    strcpy(name, p.name); score = p.score;
    return *this;
  }
  ```
- Защо връщаме `Player&`, а не просто `Player`?
  - за да можем да използваме резултата като lvalue
  - `(p1 = p2).setName("Катнис Евърдийн");`
- Какво се случва, ако напишем `p = p`?
  ```

Операция за присвояване =: защита от самоприсвояване

- При `p = p` се получава разрушаване на обекта!

Операция за присвояване =: защита от самоприсвояване

- При `p = p` се получава разрушаване на обекта!
- **Решение:** игнорираме самоприсвоявания

Операция за присвояване =: защита от самоприсвояване

- При `p = p` се получава разрушаване на обекта!
- **Решение:** игнорираме самоприсвоявания
- ```
  Player& operator=(Player const& p) {
    if (this != &p) {
      delete[] name;
      name = new char[strlen(p.name)+1];
      strcpy(name, p.name); score = p.score;
    }
    return *this;
  }
  ```
  ```

Операция за присвояване =: защита от самоприсвояване

- При `p = p` се получава разрушаване на обекта!
- **Решение:** игнорираме самоприсвоявания

```
Player& operator=(Player const& p) {
  if (this != &p) {
    delete[] name;
    name = new char[strlen(p.name)+1];
    strcpy(name, p.name); score = p.score;
  }
  return *this;
}
```

- А защо не `(*this != p)`?

Операции за съкратено присвояване □=

- Операциите от вида □= трябва да връщат lvalue, както =

Операции за съкратено присвояване □=

- Операциите от вида □= трябва да връщат lvalue, както =
- Можем да използваме операция □= за дефиниране на □

Операции за съкратено присвояване □=

- Операциите от вида □= трябва да връщат lvalue, както =
- Можем да използваме операция □= за дефиниране на □
- **Пример:**

```
Rational& Rational::operator*=(Rational const& r) {
  numer *= r.numer; denom *= r.denom;
  return *this;
}
Rational Rational::operator*(Rational const& r) const {
  Rational temp = *this;
  return temp *= r;
}
```

Операции за инкрементиране ++ и декрементиране --

- Операциите ++ и -- съществуват в два варианта

Операции за инкрементиране ++ и декрементиране --

- Операциите ++ и -- съществуват в два варианта
  - префиксна (++r): новата стойност след промяната (lvalue)

Операции за инкрементиране ++ и декрементиране --

- Операциите ++ и -- съществуват в два варианта
  - префиксна (++r): новата стойност след промяната (lvalue)
  - постфиксна (r++): старата стойност преди промяната (rvalue)

Операции за инкрементиране ++ и декрементиране --

- Операциите ++ и -- съществуват в два варианта
  - префиксна (++r): новата стойност след промяната (lvalue)
  - постфиксна (r++): старата стойност преди промяната (rvalue)
- **Проблем:** Как да укажем коя от двете операции предефинираме?

Операции за инкрементиране ++ и декрементиране --

- Операциите ++ и -- съществуват в два варианта
  - префиксна (++r): новата стойност след промяната (lvalue)
  - постфиксна (r++): старата стойност преди промяната (rvalue)
- **Проблем:** Как да укажем коя от двете операции предефинираме?
- **Решение:** Постфиксният вариант има фиктивен `int` аргумент

Операции за инкрементиране ++ и декрементиране --

- Операциите ++ и -- съществуват в два варианта
  - префиксна (++r): новата стойност след промяната (lvalue)
  - постфиксна (r++): старата стойност преди промяната (rvalue)
- **Проблем:** Как да укажем коя от двете операции предефинираме?
- **Решение:** Постфиксният вариант има фиктивен `int` аргумент

```
Rational& Rational::operator++() {  // ++r, префиксна
  numer += denom; return *this;
}
Rational Rational::operator++(int) { // r++, постфиксна
  Rational old = *this; numer += denom; return old;
}
```

Операция за извикване на функция ()

- Позволява да разглеждаме обект като функция

Операция за извикване на функция ()

- Позволява да разглеждаме обект като функция
- Може да бъде предефинирана с произволен брой параметри

Операция за извикване на функция ()

- Позволява да разглеждаме обект като функция
- Може да бъде предефинирана с произволен брой параметри
- Трябва да бъде дефинирана като член-функция!

Операция за извикване на функция ()

- Позволява да разглеждаме обект като функция
- Може да бъде предефинирана с произволен брой параметри
- Трябва да бъде дефинирана като член-функция!
- **Примери:**

Операция за извикване на функция ()

- Позволява да разглеждаме обект като функция
- Може да бъде предефинирана с произволен брой параметри
- Трябва да бъде дефинирана като член-функция!
- **Примери:**
  - ```
    double Rational::operator()() const {
      return (double)numer / (double)denom;
    }
    ```
    ```

Операция за извикване на функция ()

- Позволява да разглеждаме обект като функция
- Може да бъде предефинирана с произволен брой параметри
- Трябва да бъде дефинирана като член-функция!
- **Примери:**
  - ```
    double Rational::operator()() const {
      return (double)numer / (double)denom;
    }
    ```
  - `Rational r(3, 5); cout << r(); // 0.6`
    ```

Операция за извикване на функция ()

- Позволява да разглеждаме обект като функция
- Може да бъде предефинирана с произволен брой параметри
- Трябва да бъде дефинирана като член-функция!
- **Примери:**
  - ```
    double Rational::operator()() const {
      return (double)numer / (double)denom;
    }
    ```
  - ```
    Rational r(3, 5); cout << r(); // 0.6

    Rational Rational::operator()(int x, int y) const {
      return Rational(numer + x, denom + y);
    }
    ```

Операция за извикване на функция ()

- Позволява да разглеждаме обект като функция
- Може да бъде предефинирана с произволен брой параметри
- Трябва да бъде дефинирана като член-функция!
- **Примери:**
  - ```
    double Rational::operator()() const {
      return (double)numer / (double)denom;
    }
    ```
  - ```
    Rational r(3, 5); cout << r(); // 0.6
    ```

    ```
    Rational Rational::operator()(int x, int y) const {
      return Rational(numer + x, denom + y);
    }
    ```
  - ```
    r(1, 2).print(); // 4/7
    ```
    ```

Операции за преобразуване на тип

- За всеки <тип> можем да предефинираме специална операция:
- `operator`<тип>`() {` <тяло> `}`

Операции за преобразуване на тип

- За всеки <тип> можем да предефинираме специална операция:
- `operator`<тип>`() {` <тяло> `}`
- Дефинират правило за преобразуване <клас> $\rightarrow$ <тип>

Операции за преобразуване на тип

- За всеки <тип> можем да предефинираме специална операция:
- `operator`<тип>`() {` <тяло> `}`
- Дефинират правило за преобразуване <клас> $\rightarrow$ <тип>
- Обекти от <клас> могат да се използват навсякъде, където се очаква <тип>

Операции за преобразуване на тип

- За всеки <тип> можем да предефинираме специална операция:
- `operator`<тип>`() {` <тяло> `}`
- Дефинират правило за преобразуване <клас> $\rightarrow$ <тип>
- Обекти от <клас> могат да се използват навсякъде, където се очаква <тип>
- Типът на резултата винаги е <тип> и затова се пропуска

Операции за преобразуване на тип

- За всеки <тип> можем да предефинираме специална операция:
- `operator`<тип>`() {` <тяло> `}`
- Дефинират правило за преобразуване <клас> $\rightarrow$ <тип>
- Обекти от <клас> могат да се използват навсякъде, където се очаква <тип>
- Типът на резултата винаги е <тип> и затова се пропуска
- Обратни по действие на конструкторите за преобразуване

Операции за преобразуване на тип

- За всеки <тип> можем да предефинираме специална операция:
- `operator`<тип>`() {` <тяло> `}`
- Дефинират правило за преобразуване <клас> $\rightarrow$ <тип>
- Обекти от <клас> могат да се използват навсякъде, където се очаква <тип>
- Типът на резултата винаги е <тип> и затова се пропуска
- Обратни по действие на конструкторите за преобразуване
  - те дефинират правило <тип> $\rightarrow$ <клас>

Операции за преобразуване на тип

- За всеки <тип> можем да предефинираме специална операция:
- `operator`<тип>`() {` <тяло> `}`
- Дефинират правило за преобразуване <клас> $\rightarrow$ <тип>
- Обекти от <клас> могат да се използват навсякъде, където се очаква <тип>
- Типът на резултата винаги е <тип> и затова се пропуска
- Обратни по действие на конструкторите за преобразуване
  - те дефинират правило <тип> $\rightarrow$ <клас>
- **Примери:**

Операции за преобразуване на тип

- За всеки <тип> можем да предефинираме специална операция:
- `operator`<тип>`() {` <тяло> `}`
- Дефинират правило за преобразуване <клас> $\rightarrow$ <тип>
- Обекти от <клас> могат да се използват навсякъде, където се очаква <тип>
- Типът на резултата винаги е <тип> и затова се пропуска
- Обратни по действие на конструкторите за преобразуване
  - те дефинират правило <тип> $\rightarrow$ <клас>
- **Примери:**
  - `Rational::operator int() { return numer/denom; }`

Операции за преобразуване на тип

- За всеки <тип> можем да предефинираме специална операция:
- `operator`<тип>`() {` <тяло> `}`
- Дефинират правило за преобразуване <клас> $\rightarrow$ <тип>
- Обекти от <клас> могат да се използват навсякъде, където се очаква <тип>
- Типът на резултата винаги е <тип> и затова се пропуска
- Обратни по действие на конструкторите за преобразуване
  - те дефинират правило <тип> $\rightarrow$ <клас>
- **Примери:**
  - `Rational::operator int() { return numer/denom; }`
  - `Rational r(5, 3); int x = r; // x = 1`

Операции за преобразуване на тип

- За всеки <тип> можем да предефинираме специална операция:
- `operator`<тип>`() {` <тяло> `}`
- Дефинират правило за преобразуване <клас> $\rightarrow$ <тип>
- Обекти от <клас> могат да се използват навсякъде, където се очаква <тип>
- Типът на резултата винаги е <тип> и затова се пропуска
- Обратни по действие на конструкторите за преобразуване
  - те дефинират правило <тип> $\rightarrow$ <клас>
- **Примери:**
  - `Rational::operator int() { return numer/denom; }`
  - `Rational r(5, 3); int x = r; // x = 1`
  - `Rational::operator double() { return (double) numer / denom; }`

Операции за преобразуване на тип

- За всеки <тип> можем да предефинираме специална операция:
- `operator`<тип>`() {` <тяло> `}`
- Дефинират правило за преобразуване <клас> $\rightarrow$ <тип>
- Обекти от <клас> могат да се използват навсякъде, където се очаква <тип>
- Типът на резултата винаги е <тип> и затова се пропуска
- Обратни по действие на конструкторите за преобразуване
  - те дефинират правило <тип> $\rightarrow$ <клас>
- **Примери:**
  - `Rational::operator int() { return numer/denom; }`
  - `Rational r(5, 3); int x = r; // x = 1`
  - `Rational::operator double() { return (double) numer / denom; }`
  - `Rational r(9, 4); cout << sqrt(r); // 1.5`

Операции за преобразуване на тип

- За всеки <тип> можем да предефинираме специална операция:
- `operator`<тип>`() {` <тяло> `}`
- Дефинират правило за преобразуване <клас> $\rightarrow$ <тип>
- Обекти от <клас> могат да се използват навсякъде, където се очаква <тип>
- Типът на резултата винаги е <тип> и затова се пропуска
- Обратни по действие на конструкторите за преобразуване
  - те дефинират правило <тип> $\rightarrow$ <клас>
- **Примери:**
  - `Rational::operator int() { return numer/denom; }`
  - `Rational r(5, 3); int x = r; // x = 1`
  - `Rational::operator double() { return (double) numer / denom; }`
  - `Rational r(9, 4); cout << sqrt(r); // 1.5`
  - `Player::operator char const*() { return name; }`

Операции за преобразуване на тип

- За всеки <тип> можем да предефинираме специална операция:
- `operator`<тип>`() {` <тяло> `}`
- Дефинират правило за преобразуване <клас> $\rightarrow$ <тип>
- Обекти от <клас> могат да се използват навсякъде, където се очаква <тип>
- Типът на резултата винаги е <тип> и затова се пропуска
- Обратни по действие на конструкторите за преобразуване
  - те дефинират правило <тип> $\rightarrow$ <клас>
- **Примери:**
  - `Rational::operator int() { return numer/denom; }`
  - `Rational r(5, 3); int x = r; // x = 1`
  - `Rational::operator double() { return (double) numer / denom; }`
  - `Rational r(9, 4); cout << sqrt(r); // 1.5`
  - `Player::operator char const*() { return name; }`
  - `Player s("Гандалф Сивия", 45); cout << strlen(s);`