Операторни функции

Операции в C++

- C++ разполага с много операции
- Всяка операция се характеризира с:
  - местност (унарна, бинарна, тернарна)
  - позиция спрямо аргументите (инфиксна, префиксна, постфиксна)
  - приоритет
  - асоциативност за бинарните операции (лява, дясна)
- Примери
  - – е бинарна инфиксна лявоасоциативна операция
    - също така – е и унарна префискна операция
  - = е бинарна инфиксна дясноасоциативна операция
  - ! е унарна префиксна операция
  - ++ е унарна 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
- 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 (+, –, *, /, %)
  - 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

- Следните операции не могат да бъдат предефинирани
  - условна операция (?:)
  - операция за указване на област (::)
  - операция за избор на член (.)
  - операция за избор на член по указател (.*)
  - операция за намиране на големина (sizeof)
  - препроцесорни операции (#, ##)

Предефиниране на операции като 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 като функции

- <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 променят операциите върху примитивните типове
- Пример:
- bool operator==(Rational const& r1, Rational const& r2) {
  return r1.getNumerator() == r2.getNumerator() &&
  r1.getDenominator() == r2.getDenominator();
  }

Извикване на операторни функции

- Изразите с операции приложени върху класове автоматично се преобразуват до извиквания на операторни функции
- r1 * r2 ↔ r1.operator*(r2)
- -r1 ↔ r1.operator-()
- r1 == r2 ↔ operator==(r1, r2)

Извикване на операторни функции

- Кога се налага да пишем операторни функции, които не са член-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)
- Решение: Приятелските функции са функции, на които се позволява вътрешен достъп до компонентите на класа
- friend <тип> <име>(<параметри>);
  friend <тип> <име>(<параметри>) { <тяло> }
- Пример:

```
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 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
- Стремете се операциите, които предефинирате да се използват по същия начин както за примитивните типове

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

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

Предефиниране на някои операции

Операция []

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

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 използваме външна операторна функция
- Примери:

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

Операция =

- 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 такава, която присвоява сляпо всички полета от единия обект на другия

Операция =

- Пример:
- Student& operator=(Student const& s) {
  delete[] name;
  name = new char[strlen(s.name)+1];
  strcpy(name, s.name); fn = s.fn; grade = s.grade;
  return *this;
  }
- Защо връщаме Student&?
  - за да можем да използваме резултата като lvalue
  - (s1 = s2).setName(„Иван Иванов“);
- Какво се случва, ако напишем s = s?

Операция =

- При s = s се получава разрушаване на обекта!
- Решение: игнорираме самоприсвоявания
- Student& operator=(Student const& s) {
  if (this != &s) {
  delete[] name;
  name = new char[strlen(s)+1];
  strcpy(name, s.name); fn = s.fn; grade = s.grade;
  }
  return *this;
  }
- А защо не (*this != s)?

Операции @=

- Операциите от вида @= трябва да връщат lvalue, както =
- Можем да използваме операция @= за дефиниране на @
- Rational& Rational::operator*=(Rational const& r) {
  numer *= r.numer; denom *= r.denom;
  return *this;
  }
- Rational Rational::operator*(Rational const& r) {
  Rational temp = *this;
  return temp *= r;
  }

Операции ++ и --

- Операциите ++ и -- съществуват в два варианта
  - префиксна (++r) - връща новата стойност след промяната (lvalue)
  - постфиксна (r++) - връща старата стойност преди промяната (rvalue)
- Как се разбира коя от двете операции предефинираме?
  - фиктивен аргумент от тип int за постфиксния вариант
- Rational& Rational::operator++() { // ++r, префиксна
  numer += denom; return *this;
  }
- Rational Rational::operator++(int) { // r++, постфиксна
  Rational old = *this; numer += denom; return old;
  }

Операция ()

- Операцията () е операция извикване на 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

- operator<тип>() { <тяло> }
- Дефинират правило за преобразуване <клас> → <тип>
- Обекти от <клас> могат да се използват навсякъде, където се очаква <тип>
- Обратни по действие на конструкторите за преобразуване
  - те дефинират правило <тип> → <клас>
- Примери:
  Rational::operator int() { return numer / denom; }
  Rational::operator double() { return (double)numer / denom; }
  Student::operator char const*() { return name; }
  Student s; cout << s;