Перегрузка операторов

На самом деле: определение действий операторов для пользовательских типов путём определения функций-операторов

April 11, 2017

Стандартные обозначения

Предметная область определяет действия символов

Алгебра: действительные числа

$$a + b \cdot c$$

Умножение и сложение действительных чисел.

Аналитическая геометрия: вектора

$$\vec{z} = \vec{a} \cdot 0.6 + \vec{c}$$

Умножение вектора на скаляр и сложение векторов.

Программирование: строки

```
string s = "ABCDE", t = "ZZZ", x;
s += "atu_example_string_x42";   // конкатенация строк
x = t + s;    // аналогично
```

Реализация действий в C++
Стандартный поход: функции

```
class Vect {
  public:
    Vect( double ax, double ay )
        : x(ax), y(ay) {};

    void addfrom( const Vect &r );
    friend Vect add( const Vect &l, const Vect &r );
  private:
    double x, double y;
};
Vect a(1,2), b(1,1), c(0,0);
a.addfrom(b);
c = add( a, b );
```

Достоинство подхода: используются привычная для программиста форма записи.
Недостатки: непривычная форма записи для специалиста в предметной области, слишком длинная форма записи.

Решение проблемы в C++

- действия в любом случае выполняют функции;
- функциям даются специальные имена, состоящие из слова “**operator**” и собственно символа оператора (например “+”, “<<”, “+=”);
- компилятор, когда встречает в тексте программы оператор, **хотя бы один из операндов которого имеет пользовательский тип**, вызывает функцию с требуемым именем (если такова есть).

Пример функций-операторов

Предыдущий пример с использованием функций-операторов

```
class Vect {
  public:
    Vect( double ax, double ay )
        : x(ax), y(ay) {};

    Vect& operator+=( const Vect &r );
    friend Vect operator+( const Vect &l,
                           const Vect &r );
  private:
    double x, double y;
};

Vect a(1,2), b(1,1), c(0,0);
a += b;      // a.operator+=(b);
c = a + b;   // c = operator+( a, b );
//  точнее
c.operator=( operator+( a, b ) );
```

Ограничения

Какие функции-операторы определять нельзя

Пользовательские типы

Хотя бы один из операторов должен быть пользовательского типа. Нельзя определять действие оператора только для фундаментальных или производных типов.

Неизменность характеристик оператора

Нельзя изменить приоритет, арность и ассоциативность оператора. Нельзя определять новые операторы.

Запрещено перегружать операторы

```
::    .    .*    ?:    sizeof    typeid    throw
static_cast    dynamic_cast    reinterpret_cast    const_cast
```

Способ реализации функции-оператора

Функция член класса

В предыдущем примере оператор “**+=**” был определен как функция-член, а оператор “**+**” – как функция не-член (друг). Как следует делать выбор?
Если функция-оператор реализована как член класса, то ее вызов выглядит так:

```
a.operatorX(b) ; // binary
a.operatorX();   // unary
```

В этом случае на первый (левый) операнд, так как он передается с помощью указателя **this** накладывается существенное ограничение: это должен быть объект, *являющийся* экземпляром данного класса. Для него **не производятся** *неявные преобразования*. Если такое ограничение – свойство самого оператора, то следует использовать функцию-член класса. Например, операторы “**=**”, “**&**”, “**[]**”, “**->**” должны быть членами класса.

Способ реализации функции-оператора

Функция не-член класса

Если операнды у оператора равноправны, то следует реализовывать функцию-оператор как не-член класса (как друга или помощника). При этом разрешены неявные преобразования как для первого, так и для второго аргумента.

Бинарные операторы “**+**”, “**-**”, “*” и т.д. правильно определять как функции не-члены класса. Если уже определены операторы “**+=**”, “**-=**”, то можно использовать их функциональность и реализовать “**+**”, “**-**”, “*” как функции-помощники, не имеющие доступ к реализации.

Если первый аргумент – объект **не нашего типа**, то функция **не может** быть реализована как функция-член класса. Например, многие классы определяют оператор “**<<**” для вывода в ostream (например в cout).

Унарный оператор

Передача параметров унарному оператору (1)

Унарный оператор – как функция член класса

В этом случае у функции не должно быть явных аргументов – единственный операнд передается с помощью указателя **this**.

Пример:

```
class A {
  public:
  A& operator++();          // prefix:  ++a
  const A operator++(int);// postfix: a++
};
```

При этом следующие вызовы эквивалентны:

```
A a;
++a;   a.operator++();
```

Для определения постфиксных ++ и –– следует передать фиктивный параметр типа int.

Унарный оператор – как функция не-член класса

В этом случае у функции должен быть один явный аргумент – единственный операнд передается через него.

Пример:

```
class A {
  public:
  friend const A& operator+(const A& arg);
};
```

При этом следующие вызовы эквивалентны:

```
A a;
+a;   operator+(a);
```

Бинарный оператор
Передача параметров бинарному оператору (1)

Бинарный оператор – как функция член класса

В этом случае у функции должен быть 1 явный аргумент – **правый** операнд передается через него, а левый – с помощью указателя **this**.

Пример:

```
class A {
  public:
  A& operator+=( const A& rhs );
};
```

При этом следующие вызовы эквивалентны:

```
A a, b;
a+=b;   a.operator+=(b);
```

Бинарный оператор – как функция не-член класса

В этом случае у функции должно быть 2 явных аргумента – оба операнда передается явно.

Пример:

```
class A {
  public:
  friend const A operator+(const A& lhs,
                           const A& rhs);
};
```

При этом следующие вызовы эквивалентны:

```
A a, b, c;
c = a+b;   c = operator+(a,b);
```

Автоматически создаваемые операторы

Компилятор создаст сам, если не задал пользователь

Автоматически создаются функции для следующих операторов

```
=
```

Реализация: вызываются операторы “=” для всех вложенных и базовых объектов.

```
& //  унарный ; (+const)
```

Реализация: возвращает адрес объекта.

Правило

Если объект захватывает ресурсы (память, файлы), то у него должен быть определен оператор присваивания (также как конструктор копирования и деструктор). Можно запретить присваивание, описав оператор в “private” и не дав его определения (C++11: “= delete”).

Передача аргументов
По ссылке или по значению

Передача по ссылке (const T &arg)

Передача больших объектов по ссылке дешевле, чем по значению. Если оператор не должен изменять аргумент, то следует использовать ссылку на константный объект.

Передача по значению (T arg)

Если содержимое объекта мало (например один int или double), то имеет смысл передавать по значению.

Возврат значения

По ссылке или по значению

Возврат по ссылке

Возврат больших объектов по ссылке дешевле, чем по значению. Если результат содержится в одном из операндов, то можно вернуть ссылку на этот операнд.

Возврат по значению

Если результат не содержится ни в одном из операндов, то приходится возвращать объект по значению (может быть по константному значению).

Правило

Если сомневаешься, как получать аргументы или возвращать значение – делай так, как это сделано для фундаментальных типов.

Операторы new, new[]

Операторы new и new[] выполняют 2 действия:

- Выделяют память для объекта (объектов) с помощью **функции** operator new;
- Для создания объектов вызывают конструкторы.

Если надо переопределить распределение памяти для заданного типа, можно определить функцию operator new. Это статический (неявно) член класса, и возвращает **void***. Обычный new получает 1 аргумент типа *size_t*.
Если определена функция *operator new*, то почти всегда следует определить функцию *operator delete*, которая и освобождает память.

Операторы new, delete

Пример описания функций *operator new operator delete*

```
class SmallObj {
  public:
    SmallObj();
    static void* operator new( size_t size );
    //  – это базовая форма operator new,
    //  параметр size важен - эта функция наследуется.
    static void operator delete( void *ptr, size_t size);
};
void* SmallObj::operator new( size_t size )
{
  if( size == sizeof(SmallObj) )
    return xxxxxxx; //  свои действия
  return ::operator new(size);
}
```

Операторы преобразования

Задают пользовательские преобразования – из заданного типа в произвольный. Beware!

```
class FixedPoint {
  public:
    FixedPoint( double v )
      : ip(int(v), fp(int(INT_MAX*(v-ip))) {};

    operator double() const
      { return ip + (double)(fp)/INT_MAX; }

  private:
    int ip, fp;
};

FixedPoint a( 0.5 );
double v = sin( a );//неявно вызывается operator double()
```

В C++11 можно запретить неявные преобразования ключевым словом *explicit*.

Оператор []

Наличие в классе функции-оператора **[]** позволяет использовать объект этого класса как массив:

```
class Vect {
 public:
  explicit Vect( int sz );
  Vect( const Vect &rhs );
  virtual ~Vect();

  double& operator[]( int n );
  double& operator[]( const char *str );
  // ....

 protected:
  int size;
  double* arr;
};

Vect a(100); a[50] = 10;
```

Оператор ()

Наличие в классе функции-оператора **()** позволяет использовать объект этого класса (функтор) как функцию:

```
class Maxer {
  public:
   explicit Maxer( double inv = 0 ) : mv(inv) {} ;

   double operator()( double v );
  private:
   double mv;
};

  Maxer m, nm;
  cout << "m(5)=" << m(5) << "␣nm(7)" << nm(7) << endl;
```

Рекомендации по определению функций-операторов

- Сначала определяйте операторы с присваиваниями (+=), а потом, с использованием уже созданного – соответствующие бинарные операторы (+).
- Оператор “=” (а также +=. –=, . . .) должен возвращать ***this**.
- Оператор “=” должен или проверять на равенство левого и правого оператора, или использовать swap-технологию.
- В операторе “=” помните про все члены данных.
- Не перегружайте **&&**, **||**, “**,**” без особой необходимости.
- Осторожно используйте операторы преобразования.