**Методические материалы. Перегрузка операторов**

Во многих языках программирования используются операторы: как минимум, присваивания (=, := или похожие) и арифметические операторы (+, -, \* и /). В большинстве языков со статической типизацией эти операторы привязаны к типам. Например, в Java сложение с оператором + возможно лишь для целых чисел, чисел с плавающей запятой и строк. Если мы определим свои классы для математических объектов, например, для матриц, мы можем реализовать метод их сложения, но вызвать его можно лишь чем-то вроде этого: a = b.add(c). В C++ этого ограничения нет — мы можем перегрузить практически любой известный оператор. По факту, перегрузка операторов – это способ сделать код понятнее

Без перегрузки операторов:

Point a;

Point b;

a.input();

b.input();

Point c = sum(a, b);

c.print();

С перегрузкой операторов:

Point a;

Point b;

std::cin >> a >> b;

std::cout << (a + b);

Возможностей не счесть: можно выбрать любую комбинацию типов операндов, единственным ограничением является необходимость того, чтобы присутствовал как минимум один операнд пользовательского типа. То есть определить новый оператор над встроенными типами или переписать существующий **нельзя**.

**Какие операторы можно перегружать**

В С++ можно перегрузить практически любой оператор, учитывая следующие исключения и ограничения:

* Нельзя определить новый оператор, например, operator\*\*.
* Следующие операторы перегружать нельзя:
  1. ?: (тернарный оператор);
  2. :: (доступ к вложенным именам);
  3. . (доступ к полям);
  4. .\* (доступ к полям по указателю);
  5. sizeof, typeid и операторы каста.
* Следующие операторы можно перегрузить только в качестве методов:
  1. = (присваивание);
  2. -> (доступ к полям по указателю);
  3. () (вызов функции);
  4. [] (доступ по индексу);
  5. ->\* (доступ к указателю-на-поле по указателю);
  6. операторы конверсии и управления памятью.
* По крайней мере один из типов данных для перегружаемого оператора должен быть пользовательским. (Нельзя перегрузить оператор сложения int + double, но можно перегрузить оператор int + MyDouble)

**Перегрузка оператора через дружественную функцию**

Рассмотрим класс Point (точка на координатной оси). Определим для нее операцию сложения, и попробуем реализовать ее, используя перегрузку оператора сложения

class Point {

public:

Point(int x, int y) : m\_x(x), m\_y(y) {};

friend Point operator+(const Point &left, const Point &rigth);

private:

int m\_x;

int m\_y;

};

Point operator+(const Point &left, const Point &rigth) {

return Point(left.m\_x + rigth.m\_y, left.m\_y + rigth.m\_y);

}

В этом примере мы:

1. Объявили дружественную функцию operator+();
2. Задали в качестве параметров два операнда, с которыми хотим работать — два объекта класса Point (передаем их по константной ссылке - гарант того, что их значения не изменятся)
3. Указали соответствующий тип возврата — Point
4. Записали реализацию операции сложения.

**Перегрузка оператора через функцию**

Попробуем решить ту же задачу, используя обычные функции. Тогда для обращения к приватным полям m\_x и m\_y нам понадобится написать гетеры

class Point {

public:

Point(int x, int y) : m\_x(x), m\_y(y) {};

int get\_x() const { return m\_x; }

int get\_y() const { return m\_y; }

private:

int m\_x;

int m\_y;

};

Point operator+(const Point &left, const Point &rigth) {

return Point(left.get\_x() + rigth.get\_x(), left.get\_x() + rigth.get\_y());

}

Встает вопрос, какому способу стоит отдать предпочтение? Обычным функциям, или дружественным? Ответ такой: если для реализации оператора не придется реализовывать новые методы класса (например гетеры для доступа к приватным полям), то лучше использовать обычные функции. А если требуется доступ к приватным полям/методам класса, и нет публичного интерфейса взаимодействия, используйте friend-функции.

**Перегрузка оператора через метод класса**

Также, есть способ реализовать оператор, как метод класса. Давайте рассмотрим на примере оператора +=

class Point {

public:

Point(int x, int y) : m\_x(x), m\_y(y) {};

int get\_x() const { return m\_x; }

int get\_y() const { return m\_y; }

Point operator+=(const Point &other);

private:

int m\_x;

int m\_y;

};

Point Point::operator+=(const Point &other) {

this->m\_x += other.get\_x();

this->m\_y += other.get\_y();

return \*this;

}

Заметьте, что методы get\_x() и get\_y() помечены модификатором const. Это гарантия того, что при вызове этих методов не будет происходить изменение полей объекта.

Также модификатором const помечен аргумент other в нашем операторе. Используя константную ссылку, мы опять же гарантируем, что значение other не изменится после вызова метода. По константной ссылке нельзя вызвать метод, который не помечен модификатором const

**Рекомендации по перегрузке операторов**

Как мы уже выяснили, существует три способа операторов — в виде функции класса, в виде функции, и в виде дружественной глобальной функции. Роб Мюррей, в своей книге [C++ Strategies and Tactics](http://www.amazon.com/Strategies-Tactics-Robert-B-Murray/dp/0201563827) определил следующие рекомендации по выбору формы оператора:

| **Оператор** | **Рекомендуемая форма** |
| --- | --- |
| Все унарные операторы | Член класса |
| = () [] -> ->\* | Обязательно член класса |
| += -= /= \*= ^= &= |= %= >>= <<= | Член класса |
| Остальные бинарные операторы | Не член класса |

**Примеры. Перегрузка оператора ввода и вывода**

Перегрузка оператора вывода << аналогична перегрузке оператора + (оба являются бинарными операторами), за исключением того, что их типы различны.

Рассмотрим выражение std::cout << point. Если оператором является <<, то чем тогда являются операнды? Левым операндом является объект std::cout, а правым — объект нашего класса Point. std::cout фактически является объектом типа std::ostream, поэтому перегрузка оператора << выглядит следующим образом:



friend std::ostream& operator<< (std::ostream &out, const Point &point);

Реализация данного оператора очень проста:

std::ostream& operator<< (std::ostream &out, const Point &point) {

out << "Point(" << point.m\_x << ", " << point.m\_y << ")";

return out;

}

Дополним наш класс еще и оператором ввода. Он реализуется точно также, как и оператор вывода

class Point {

public:

friend std::ostream& operator<< (std::ostream &out, const Point &point);

friend std::istream& operator>> (std::istream &in, Point &point);

private:

int m\_x;

int m\_y;

};

std::ostream& operator<< (std::ostream &out, const Point &point) {

out << "Point(" << point.m\_x << ", " << point.m\_y << ")";

return out;

}

std::istream& operator>> (std::istream &in, Point &point) {

in >> point.m\_x;

in >> point.m\_y;

return in;

}

**Примеры. Перегрузка оператора инкремента**

Перегрузка оператора инкремента (++) или декремента (--) очень проста. За исключением одной детали - есть постфиксная (a++) и префикасная (--a) запись. Давайте рассмотрим код:

int a = 10;

int b = 10;

std::cout << a++ << '\n';

std::cout << ++b << '\n';

Результат его работы:

10

11

Вспомним, в чем тут соль

Постфиксная запись:

1. Сохраняет старое значение в отдельную переменную
2. Инкрементирует значение
3. Возвращает старое значение в поток

Префиксная запись:

1. Инкрементирует значение
2. Возвращает значение в поток

Давайте определим класс DayTime, который отвечает за смену состояний Утро, День, Вечер, Ночь. Операция инкремента будет осуществлять изменение значения на следующее по порядку. Поскольку операторы ++ и -- унарные, и изменяют поля класса, реализуем их как методы класса

class DayTime {

public:

enum DayTimeNames {

MORNING,

AFTERNOON,

EVENING,

NIGTH

};

int current = MORNING;

DayTime& operator++(); // версия префикс

DayTime operator++(int); // версия постфикс

};

Здесь возникает вопрос, как же различить при перегрузке постфиксную и префиксную запись? Обратите внимание на (int). Этот инт не с проста

Дело в том, что язык C++ использует фиктивную переменную (или «фиктивный параметр») для операторов версии постфикс. Этот фиктивный целочисленный параметр используется только с одной целью: отличить версию постфикс операторов инкремента/декремента от версии префикс. Выполним перегрузку операторов инкремента версии префикс и постфикс в одном классе:

DayTime& DayTime::operator++() {

if (current == NIGTH) {

current = MORNING;

} else {

++current;

}

return \*this;

}

DayTime DayTime::operator++(int) {

// Создаем временный объект - копию текущего

DayTime tmp;

tmp.current = this->current;

// Инкрементируем текущий

++(\*this);

// Возвращаем временный

return tmp;

}

Давайте обратим внимание на следующее:

Во-первых, мы отделили версию постфикс от версии префикс использованием целочисленного фиктивного параметра в версии постфикс.

Во-вторых, поскольку фиктивный параметр не используется в реализации самой перегрузки, то мы даже не предоставляем ему имя. Таким образом, компилятор будет рассматривать эту переменную, как простую заглушку (заполнитель места), и даже не будет предупреждать нас о том, что мы объявили переменную, но никогда её не использовали.

В-третьих, операторы версий префикс и постфикс выполняют одно и то же задание: оба увеличивают/уменьшают значение переменной объекта. Разница между ними только в значении, которое они возвращают