# Лекция 8. Классы. Перегрузка операторов

# Перегрузка конструктора копирования и оператора присваивания

На предыдущей лекции мы убедились в том, компилятор вставляет в программу вызов конструктора копирования каждый раз, когда понадобится создать новый объект на базе существующего объекта. Конструктор копирования, создаваемый компилятором по умолчанию, выполняет поразрядное копирование объектов.   
Однако, если в классе имеются указатели, то возникает вопрос, что делать: копировать указатели или выделять для значений, на которые они ссылаются, новый блок памяти?   
Поразрядное копирование объектов скорее всего приведёт к ошибкам. В этом случае оба объекта - оригинал и его копия, будут ссылаться на одну и ту же область памяти.   
Чтобы исключить поразрядное копирование, необходимо определить собственный конструктор копирования (перегрузить конструктор копирования).  
Аналогичная ситуация возникает при использовании *оператора присваивания*, созданного по умолчанию.

Эта проблема решается путем дублирования данных, на которые ссылаются указатели.

Простое присваивание часто называют "поверхностным" копированием, а дублирование

данных - "глубоким" копированием.

Решение проблемы с раздвоением указателей посредством дублирования данных рассмотрим на примере, приведенном в программе Лекция8::copyPtr.срр.

В этом примере используются два класса Dog и DogHouse.  
 Простой класс Dog содержит только объект string, в котором хранится имя объекта

(собаки). Для получения информации об основных событиях, происходящих с

экземплярами Dog, вставим в конструкторы и деструкторы вывод сообщений о своем

вызове. Обратите внимание: второй конструктор немного похож на копирующий

конструктор, только он получает указатель на Dog вместо ссылки, и еще у него есть

второй аргумент — сообщение, присоединяемое к идентификатору аргумента Dog.

Второй аргумент изменяет идентификатор объекта, помогая тем самым наблюдать за

поведением программы.

Класс DogHouse содержит указатель Dog\*. В нем представлены четыре функции,

которые всегда необходимо определять, если класс содержит указатели:

все необходимые «обычные» конструкторы,

копирующий конструктор,

функция operator=,

деструктор.

//L8: copyPtr.срр

// Решение проблемы с раздвоением указателей посредством дублирования данных,

//на которые они ссылаются, при присваивании или конструировании копий.

#include <string>

#include <iostream>

using namespace std;

//======================class Dog=============================

class Dog {

string nm; //имя собаки (значение)

public:

// Перегружать копирующий конструктор и оператор = не нужно.

//Перегрузка "обычных" конструкторов:

//Создание Dog по значению имени

Dog(const string& name) : nm(name) {

cout << "Creating Dog: " << \*this << endl;

}

// Создание Dog по указателю на Dog:

Dog(const Dog\* dp, const string &msg) : nm(dp->nm + msg) {

cout << "Copied dog " << \*this << " from " << \*dp << endl;

}

//Перегрузка деструктора

~Dog() {

cout << "Deleting Dog: " << \*this << endl;

}

void rename(const string& newName) {

nm = newName;

cout << "Dog renamed to: " << \*this << endl;

}

//Перегрузка оператора включения в поток

friend ostream& operator<<(ostream& os, const Dog& d) {

return os <<"[" << d.nm << "]";

}

};

//===========================class DogHouse============================

class DogHouse {

Dog\* p;

string houseName;

public:

DogHouse(Dog\* dog, const string& house) : p(dog), houseName(house) {}

// Перегрузка копирующего конструктора

DogHouse(const DogHouse& dh) : p(new Dog(dh.p, " copy-constructed")),

houseName(dh.houseName + " copy-constructed") {}

// Перегрузка оператора присваивания

DogHouse& operator=(const DogHouse& dh) {

if(&dh != this) {// проверка на самоприсваивание

p = new Dog(dh.p, " assigned");

houseName = dh.houseName + " assigned";

}

return \*this;

}

void renameHouse(const string& newName) {

houseName = newName;

}

Dog\* getDog() const { return p; }

//Перегрузка деструктора

~DogHouse() { delete p; }

//Перегрузка оператора включения в поток

friend ostream& operator<<(ostream& os, const DogHouse& dh) {

return os << "[" << dh.houseName << "] contains " << \*dh.p;

}

};

//======================функция main()===============================

int main() {

DogHouse fidos(new Dog("Fido"), "FidoHouse");

cout << fidos << endl;

DogHouse fidos2 = fidos; // Конструирование копии

cout << fidos2 << endl;

fidos2.getDog()->rename("Spot");

fidos2.renameHouse("SpotHouse");

cout << fidos2 << endl;

fidos = fidos2; // Присваивание

cout << fidos << endl;

fidos.getDog()->rename("Max");

fidos2.renameHouse("MaxHouse");

system("pause");

}

Из листинга видно, что при выводе информации функции класса не обращаются

к этой информации напрямую, а посылают \*this в cout. В свою очередь, это

приводит к вызову функции operator<< для ostream. Такая организация весьма

полезна, потому что если нам потребуется изменить формат вывода данных

Dog (добавить вывод символов "[" и "]" в приведенном примере), изменения

достаточно внести только в одном месте.

Функция operator= выполняет проверку на самоприсваивание, чтобы объект

не удалил сам себя.

# 2. Автоматическое приведение типа

Когда компилятор встречает выражение или вызов функции, в котором используется не совсем тот встроенный тип, который ему нужен, нередко он может привести его к нужному типу.   
В C++ можно добиться аналогичного эффекта для пользовательских типов, определяя для них функции автоматического приведения типа. Эти функции существуют в двух разновидностях: особый тип конструктора и перегруженный оператор.

Скрытые операции

*Предупреждение*. С автоматическим приведением типа связано больше скрытых действий, чем можно было бы предполагать. В большинстве случаев это связано с созданием временных объектов при передаче параметров и возврате результата из функции, а также при вычислении выражений. Поэтому к автоматическому приведению типа следует относиться с осторожностью. При наличии нескольких возможных путей преобразования компилятор может запутаться.

Общее правило для операций автоматического приведения типа состоит в том, чтобы для перехода от одного типа к другому определять только одно автоматическое приведение типа. При этом вы можете определять преобразования для других типов; просто они не должны быть всегда автоматическими. Для запрета автоматического приведения типа используется ключевое слово explicit.

### 2.1. Приведение типа с использованием конструктора

Если определить конструктор, единственный аргумент которого является объектом (или ссылкой) на другой тип, это позволит компилятору выполнять автоматическое приведение типа.   
2.2. Оператор приведения типа

Второй способ автоматического приведения типа основан на перегрузке операторов. Вы создаете функцию класса, которая получает текущий тип и приводит его к нужному типу; имя функции состоит из ключевого слова operator и типа, к которому осуществляется приведение. Эта форма перегрузки оператора уникальна тем, что для нее не указывается тип возвращаемого значения, — он определяется именем перегружаемого оператора.

При использовании конструктора *преобразование выполняется классом, к которому приводится тип,* а при использовании операторов *преобразование выполняется исходным классом*.   
Однако создание конструктора с одним аргументом (и даже конструктора с несколькими аргументами, если для остальных аргументов заданы значения по умолчанию) всегда определяет автоматическое приведение типа. Иногда это оказывается нежелательным, и тогда автоматическое приведение можно запретить при помощи ключевого слова explicit. Кроме того, приведение типа с конструктором не годится для перехода от пользовательского типа к встроенному типу; это возможно только путем перегрузки операторов.

### 2.3. Запрет на автоматическое приведение типа

Иногда автоматическое приведение типа с использованием конструктора или операторов приведения типа порождает лишние проблемы. Ключевым словом explicit мы запрещаем

компилятору выполнять какие-либо операции по автоматическому приведению типа с использованием этого конкретного конструктора или оператора преобразования типа (другие конструкторы и операторы преобразования, у которых слово explicit отсутствует, по-прежнему могут задействоваться для автоматических преобразований).

В случае использования *explicit* перед операторами приведения типа, если пользователь хочет выполнить приведение типа, он должен явно указать на это обстоятельство в программе.   
Пример преобразования типов приведен в программе   
Лекция8:: OverloadingConversion.срр

# 3. Перегрузка операторов

Перегружать операторы можно двумя способами:  
- как методы класса,  
- как глобальные функции.  
Во втором случае функции перегрузки объявляются как друзья класса.

Исключением является функция перегрузки оператора присваивания. Она может быть только методом класса.

Использование для перегрузки операторов методов класса является более эффективным.  
Одним из важнейших доводов в пользу использования глобальных перегруженных операторов вместо операторных функций классов является то, что в глобальных версиях автоматическое приведение типа может применяться к обоим операндам, тогда как в функциях классов левосторонний операнд должен изначально относиться к нужному типу.

### 3.1. Рефлексивность

Рефлексивность предполагает, что преобразование типов может применяться к обоим операндам. Если преобразуются оба операнда, глобальная версия сэкономит немало усилий. Рассмотрим небольшой пример:

Класс Number содержит функцию класса operator+ и объявление friend operator-. Поскольку у класса имеется конструктор с единственным аргументом типа int, тип int может автоматически приводиться к типу Number, но только при соблюдении необходимых условий. Из функции main() видно, что прибавление Number к другому объекту Number работает нормально, так как операция точно соответствует сигнатуре перегруженного оператора. Когда компилятор встречает идентификатор Number, за которым следуют + и int, он находит соответствие с функцией Number::operator+ и преобразует аргумент int в Number при помощи конструктора. Но, встречая int, + и Number, компилятор не знает, что делать, потому что у него есть лишь функция Number::operator+, которая требует, чтобы левосторонний операнд уже был объектом Number. Компилятор выдает сообщение об ошибке. С функцией friend operator- дело обстоит иначе. Компилятор должен заполнить оба аргумента; левосторонний аргумент не ограничен типом Number. Следовательно, когда компилятор встречает следующее выражение, он может привести первый аргумент к типу Number при помощи конструктора:   
 1 – а  
 Иногда объявление операторных функций в форме функций класса позволяет ограничивать выполняемые операции. Например, при умножении матрицы на вектор справа должен находиться вектор. Но если вы хотите, чтобы оператор мог преобразовать любой из аргументов, оформите его операторную функцию в виде дружественной функции. К счастью, для выражения 1-1 компилятор не пытается привести оба аргумента к типу Number и вызвать для них функцию operator-. Компилятор всегда начинает поиск с «простейшей» возможности, а для выражения 1-1 «простейшим» окажется встроенный оператор.