# Лекция 7. Классы.

# Перегрузка конструкторов и аргументы по умолчанию

## Пример класса

//-----------------класс DBDate----------------------------

class DBDate

{

friend string DateToStr(DBDate& date);

friend ostream& operator<<(ostream& out,DBDate& date);

int day, month, year;

public:

DBDate(string date);//формат строки: dd.mm.yyyy

DBDate(int d,int m,int y);

DBDate():day(0),month(0),year(0){};//конструктор по умолчанию

//конструктор копирования

DBDate(DBDate& dat):day(dat.day),month(dat.month),year(dat.year){}

int GetDay();

int GetMonth();

int GetYear();

private: //вспомогательные функции (для внутреннего использования)

bool IsLeapYear (int year); //год высокосный?

int GetDaysInMonth(int month,int year);//Количество дней в месяце

int DaysInCurYear();//Количество дней от начала года до текущей даты

public:

bool operator==(DBDate& date);

bool operator<(DBDate& date);

bool operator>(DBDate& date);

bool operator<= (DBDate& date); //дополнительная операция

bool operator>= (DBDate& date); //дополнительная операция

bool operator!= (DBDate& date); //дополнительная операция

DBDate& operator+=(int days); //Прибавляет к текущей дате days дней

DBDate& operator-=(int days); //Вычитает из текущей даты days дней

int operator-(DBDate& date);//Количество дней между текущей датой и date

//Если текущая дата > date, результат < 0.

};

Создавая объект (переменную), вы связываете *имя* с некоторой областью памяти, а функция является *именем* операции, выполняемой с этой областью памяти. Если система описывается с помощью разумно выбранных и понятных *имен*, программа будет легко читаться и редактироваться.   
В естественных языках присутствует некоторая *избыточность*, так что даже если пропустить пару слов, смысл все равно останется понятным. Человеку не нужна абсолютная точность в определениях — смысл часто удается определить по контексту.

Во многих языках программирования каждая функция должна снабжаться уникальным идентификатором. Если программа работает с тремя разными типами данных int, char и float, то, например, для их вывода *программисту* приходится создавать три разные функции: print\_jnt, print\_char и print\_float. Лишние функции усложняют жизнь как программисту, пишущему программу, так и читателям, которые пытаются в ней разобраться.  
Принцип перегрузки функций заключается в использовании в программе *одинаковых имен* функций с разными списками аргументов, а различающиеся внутренние имена генерирует компилятор.

В C++ также существует дополнительный фактор, требующий перегрузки имен функций. Речь идет о конструкторах. Поскольку имя конструктора определяется именем класса, возникает впечатление, что конструктор может быть только один. А что, если объекты могут создаваться несколькими способами? Предположим, вы построили класс, объекты которого могут инициализироваться либо стандартным способом, либо на основании данных, загруженных из файла. В этом случае необходимы два конструктора: первый будет вызываться без аргументов (конструктор по умолчанию), а второй получит аргумент типа string — имя файла с данными инициализации. *Обе функции* являются конструкторами, поэтому они должны иметь одинаковые имена, совпадающие с именем класса. Следовательно, перегрузка функций абсолютно необходима для того, чтобы одно *имя* функции (в данном случае конструктор) могло использоваться с разными типами аргументов (в объектном коде это будут две разные функции).   
Чтобы механизм перегрузки работал, *компилятор* должен включить в имя функции информацию о типах аргументов. Для функций из предыдущего примера будут сгенерированы внутренние имена вида \_print\_jnt, \_print\_char и \_print\_float. Правила «украшения» имен компилятором не оговорены в стандарте, поэтому результаты различаются в зависимости от компилятора.

Перегрузка позволяет задействовать одно имя для нескольких функций, но существует другая возможность сделать вызов функции более удобным. Если функция вызывается с большим количеством аргументов, многие из которых имеют одни и те же значения, повторять их неудобно и утомительно, не говоря уже о неудобстве чтения. В таких случаях в C++ применяются аргументы по умолчанию, то есть значения, которые автоматически подставляются компилятором, если аргумент не был указан при вызове.

Используя аргументы по умолчанию, необходимо помнить о двух правилах.  
Во-первых, значения по умолчанию могут назначаться только для аргументов, завершающих список. Нельзя объявить аргумент по умолчанию, за которым будет следовать аргумент с явно задаваемым значением.   
Во-вторых, после первого аргумента по умолчанию все последующие аргументы в списке также должны быть аргументами по умолчанию (следствие из первого правила).

### Пример перегрузки

DBDate(string date);//формат строки: dd.mm.yyyy

DBDate(int d,int m,int y);

DBDate():day(1),month(1),year(1){};//конструктор по умолчанию

Если использовать аргументы по умолчанию, то два последних конструктора можно  
объединить в один:  
 DBDate(int d=1,int m=1,int y=1);

Аргументы по умолчанию задаются только в объявлении функции (которое обычно находится в заголовочном файле). Компилятор должен знать о существовании значения по умолчанию, прежде чем он сможет его использовать.

DBDate::DBDate(int d,int m,int y)  
//в определении функции аргументы по умолчанию не используются

{

bool isNulDate=0;

if((y>9999 || y<1)||(m>12 || m<1))

isNulDate=1;

else{

year=y;

month=m;

}

if(d>GetDaysInMonth(m,y) || d<1)

isNulDate=1;

else

day=d;

if(isNulDate){

day=1; month=1; year=1;

cout<<"Недопустимая дата. Установлено 1.1.1\n";

}

}

DBDate является хорошим примером типа, для которого при реализации конструкторов и перегрузке операций обязательно нужно выполнять проверку на допустимость присваиваемых значений.   
Например, не может быть даты 29.02.1900, так как 1900 год не високосный.  
Так как проверку на допустимость присваиваемых значений придется выполнять многократно, то для её реализации имеет смысл ввести в класс вспомогательные функции:

bool IsLeapYear (int year); //год высокосный?

int GetDaysInMonth(int month,int year);//Количество дней в месяце

Для перегрузки операций с датами потребуется функция

int DaysInCurYear();//Количество дней от начала года до текущей даты

# 2. Копирующий конструктор

Это особая разновидность конструкторов, создающих новые объекты на базе существующих объектов того же типа. Работа копирующего конструктора неразрывно связана со ссылками. В частности, копирующий конструктор используется компилятором при вызове функций для передачи и возврата объектов по значению.  
Копирующие конструкторы настолько важны, что компилятор автоматически генерирует копирующий конструктор, если этого не делает программист. Конструктор копирования, создаваемый по умолчанию, выполняет поразрядное копирование объектов. Для объектов простых типов это является приемлемым решением. Однако, если конструктор кроме инициализации делает что то ещё (как в приведенном ниже примере HowMany), или если в классе имеются указатели (что делать: копировать указатели или выделять для значений, на которые они ссылаются, новый блок памяти?), то простое копирование объектов скорее всего приведёт к ошибкам, обнаружить которые будет не просто. Поиск ошибок усложняется неявным вызовом конструкторов и деструкторов, а также созданием временных объектов.

К счастью, вы можете вмешаться в этот процесс и предотвратить поразрядное копирование. Для этого необходимо определить собственный конструктор копирования, который будет вызываться каждый раз, когда компилятору понадобится создать новый объект на базе существующего объекта.   
Это происходит в трёх случаях:  
- в *операторе объявления* переменной при инициализации её значением существующего объекта (не путать с оператором присваивания);  
при передаче в функцию параметров «по значению»;  
при возврате из функции параметра «по значению».  
Единственным аргументом этого конструктора должна быть *ссылка* на объект, на базе которого конструируется новый объект. Объект нельзя передавать конструктору по значению, потому что вы определяете функцию, которая должна обеспечивать передачу по значению.

Функции хранятся в ОП в единственном экземпляре, а вызываются многократно с различными значениями параметров. Как это реализуется в При вызове функции в стеке выделяется память (кадр функции) под *аргументы функции, адрес возврата и локальные переменные*. Адрес этой памяти передается в функцию в скрытом параметре.

При передаче объекта по значению вы создаете новый объект (*переданный объект* в *кадре* функции) на базе существующего объекта (*исходный объект*, находящийся вне *кадра* функции).

То же самое часто происходит и при возвращении объекта из функции.

2.1 Определение мест вызова конструкторов и деструкторов объектов в программе.

Знание механизма вызова конструкторов и деструкторов в программе поможет избежать недоразумений, связанных с инициализацией и уничтожением объектов.   
Для демонстрации работы этого механизма рассмотрим несложный пример программы с классом, который выводит на экран информацию о количестве и именах объектов «своего» типа, существующих в произвольный момент времени в ОП.   
Эта задача решается с помощью включения в класс статической переменной для подсчёта объектов и добавлением в конструкторы и деструктор операторов вывода сообщений, содержащих имя объекта и число объектов этого класса, которые будут находиться в ОП после их вызова.   
При правильном завершении программы счётчик объектов должен быть равен 0.  
//HowMany.cpp   
// Копирующий конструктор

#include <fstream>

#include <string>

using namespace std;

ofstream fout("HowMany.txt");

class HowMany {

string name; // Идентификатор объекта

static int objectCount;

public: HowMany(const string& id = "") : name(id) {

++objectCount;

print("HowMany()");

}

~HowMany() {

--objectCount;

print("~HowMany()");

}

// Копирующий конструктор:

HowMany(const HowMany& h) : name(h.name) {

name += " copy";

++objectCount;

print("HowMany(const HowMany&)");

}

void print(const string &msg = "") const {

if(msg.size() != 0)

fout << msg << endl;

fout << '\t' << name << ": "

<< "objectCount = " << objectCount << endl;

}

};

int HowMany::objectCount = 0;//инициализация статической переменной

//---------------------------------------------------------

// Передача и возврат ПО ЗНАЧЕНИЮ:

HowMany f(HowMany x) {

x.print("x argument inside f()");

fout << "Returning from f()" << endl;

return x;

}

int main() {

HowMany h("h");

fout << "Entering f()" << endl;

HowMany h2 = f(h);

h2.print("h2 after call to f()");

fout << "Call f(), no return value" << endl;

f(h);

fout << "After call to f()" << endl;

system("pause");

}

Класс HowMany содержит строку с именем объекта, счетчик static int objectCount, конструктор с необязательным строковым аргументом, конструктор копирования, деструктор и функцию print()для вывода значения objectCount. Конструкторы увеличивают счетчик при каждом создании объекта, а деструктор уменьшает его.

Функция print() выводит сообщение, идентификатор объекта и счетчик экземпляров в поток fout связанный с файлом "HowMany.txt".

Внутри функции main() имеется два вызова f(). В первом случае она инициализирует объект h2, во втором вызове возвращаемое значение игнорируется.   
Теперь, когда вы знаете, как организуется возврат значения (код внутри функции заносит результат в область памяти, адрес которой передается в скрытом аргументе), может возникнуть вопрос: что происходит, когда возвращаемое значение игнорируется? Выходные данные программы помогут найти ответ.  
 Но прежде чем обсуждать результаты, создадим небольшую программу, которая нумерует строки произвольного файла с использованием потоков ввода-вывода и выводит их на экран. Она нужна для того, чтобы можно было ссылаться на номера строк при объяснении результатов выполнения программы (вспомним, что в функции main() в файле HowMany.cpp мы специально выводили результаты не на экран, а в файл HowMany.txt).

#include <fstream>

#include <iostream>

#include <iomanip>

using namespace std;

int main(int argc, char\* argv[]) {

system("chcp 1251>nul");

if(argc<2){

cout<<"Добавьте в argv[] файл для нумерации строк"<<endl;//HowMany.txt

system("pause"); return 0;}

ifstream fin(argv[1]);

if(!fin.is\_open()){

cout<<"Добавьте в папку файл для нумерации строк"<<endl;

system("pause");

return 0;

}

string line;

vector<string> lines;

while(getline(fin, line))// Чтение всего файла argv[1]

lines.push\_back(line);

if(lines.size() == 0) return 0;

int num = 0;

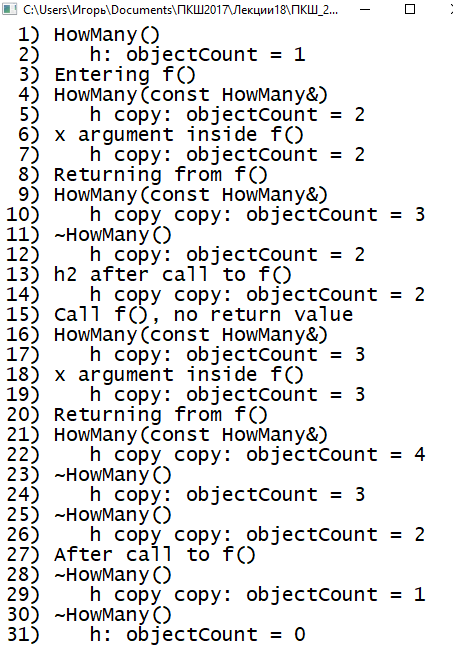
// число строк в файле argv[1] не больше 1000

const int width = 3;

for(int i = 0; i < lines.size(); i++) {// Печать всего файла argv[1]

cout <<setw(width)<< ++num << ") " << lines[i] << endl;

}  
 system("pause");  
}  
Если запустить пример Linenum.cpp для файла HowMany.txt, будет получен следующий результат:

\*

Как и следовало ожидать, работа программы начинается с вызова обычного конструктора для h, что приводит к увеличению счетчика ссылок на 1. Но затем при входе в f() компилятор автоматически вызывает копирующий конструктор для передачи объекта по значению. В кадре функции f() создается новый объект, который является копией h (отсюда и имя «h сору»). В результате выполнения копирующего конструктора счетчик объектов увеличивается до 2.  
 Строка 8 обозначает начало возврата из f(). Но перед уничтожением локальной переменной «h сору», выходящей из области видимости в конце функции, она должна быть скопирована в переменную возвращаемого значения h2.   
Объект h2 создается на базе существующего объекта (локальной переменной внутри f()); конечно, это приводит к очередному вызову копирующего конструктора в строке 9. Теперь идентификатор h2 принимает вид «h сору сору», потому что данные копируются из копии (локального объекта внутри f()). После возвращения объекта, но перед выходом из функции счетчик временно увеличивается до 3, после чего локальный объект «h сору» уничтожается. После завершения вызова f() в строке 13 в программе остаются всего два объекта h и h2, и мы видим, что h2 действительно был присвоен объект с идентификатором «h сору сору».

### 2.1.1. Временные объекты

Строка 15 начинается с вызова f(h). На этот раз возвращаемое значение функции игнорируется. Из строки 16 видно, что передача аргумента, как и прежде, сопровождается вызовом копирующего конструктора. Кроме того, как и прежде, строка 21 показывает, что для возвращаемого значения тоже вызывается копирующий конструктор. Но для работы копирующего конструктора нужен адрес, который будет использоваться в качестве приемника (указатель this). Откуда он берется? Оказывается, компилятор создает временные объекты всюду, где они нужны для правильного вычисления выражения. В данном случае объект создается для приема игнорируемого возвращаемого значения f(). Срок жизни этого временного объекта сокращается до абсолютного минимума, чтобы программа не загромождалась временными объектами, которые ожидают уничтожения и лишь зря расходуют ценные ресурсы. Иногда временный объект немедленно передается другой функции, но в нашем случае он не используется после вызова функции. Поэтому сразу же после вызова деструктора локального объекта (строки 23 и 24), завершающего вызов функции, временный объект уничтожается (строки 25 и 26). Наконец, в строках 28-31 объекты h2 и h уничтожаются, и счетчик экземпляров, как и положено, становится равным нулю.

### 2.2 Задание.

Используя эту методику убедитесь, что сгенерированный компилятором по умолчанию копирующий конструктор выполняет поразрядное копирование для примитивных типов и вызывает копирующий конструктор для пользовательских типов.