

Объектно-ориентированное  
программирование  
с использованием языка

C++



# Урок 2

Инициализаторы.  
Статические переменные-  
члены и функции-члены.  
Указатель `this` и конст-  
руктор копирования

## Contents

1. Унифицированная инициализация объектов .....	3
2. Инициализация членов класса.....	6
3. Делегирование конструкторов .....	18
4. Статические члены класса: переменные и функции .....	25
5. Указатель <code>this</code> .....	35
6. Конструктор копирования .....	44
7. Домашнее задание .....	61

Материалы урока прикреплены к данному PDF-файлу. Для доступа к ним, урок необходимо открыть в программе Adobe Acrobat Reader.

# 1. Унифицированная инициализация объектов

Для предсказуемой и корректной работы любая переменная должна быть инициализирована. При этом не важно, какого типа (`int`, `float`, `char`, `bool`, указатель, структура, класс и т.п.) переменная — не оставляйте ее не инициализированной. Ведь такая переменная не в полной мере полноценна — у нее нет начального состояния, и соответственно, невозможно получить ее текущее значение. В большинстве случаев, попытка использования не инициализированной переменной приводит к ошибке во время компиляции, но так случается не всегда. Разные версии интегрированных сред разработки и/или компиляторов могут и не выдать ошибки на этапе компиляции. Не следует полагаться на случай в столь важном вопросе!

Рассмотрим небольшой пример.

## Пример 1

```
#include <iostream>

int main()
{
    int numbers[2];
    numbers[0] += 2;
    std::cout << numbers[0] << '\n';

    return 0;
}
```

## Вывод результата работы кода из примера 1:



Рисунок 1

В среде VisualStudio 2019 фрагмент успешно компилируется и запускается. Вот только что мы видим в результате? Чему равен `numbers[0]`? Ничему корректному и пригодному к использованию! Ведь мы прибавили 2 к текущему, не определенному, значению `numbers[0]` и результат присвоили ему же. Это лишь один из множества случаев, когда не в полной мере удастся выявить не инициализированную переменную во время компиляции и как следствие — возможна совершенно неверная работа программы. Следует однако заметить, что в большинстве подобных случаев, на этапе компиляции, возникает предупреждение, (но не ошибка!), которое легко проигнорировать.

В C++ существует несколько способов инициализировать переменную:

```
int number = 0; // копирующая инициализация
int value(42); // прямая инициализация
int size{ 33 }; // унифицированная инициализация
```

Наиболее предпочтительным и однозначным способом является последний — унифицированная инициализация. Данный способ, не в пример другим, единообразно подходит, как для инициализации простой переменной, так и для инициализации массива, структуры,

класса и т.п. И именно данный способ инициализации мы будем использовать в дальнейшем.

### *Пример 2*

```
#include <iostream>

struct Point
{
    int x;
    int y;
};

int main()
{
    int answer{ 42 }; //Простая переменная
    const float goodTemp{ 36.6 }; //константа
    int grades[4]{ 3, 5, 4, 4 }; // одномерный массив
    int matrix[2][2]{ {1,2}, {3,4} }; // двумерный массив
    int* dataPtr{ nullptr }; // указатель
    char* str{ new char[14]{"Hello, world!"} }; // c-style
    строка, указатель по-сути
    int& reference{ answer }; // ссылка
    Point point{ 10,-6 }; // экземпляр (объект) структуры
    return 0;
}
```

## 2. Инициализация членов класса

В современном C++, существует несколько способов инициализировать члены класса. Наиболее гибкий и предпочтительный из них это уже знакомый по прошлому уроку, конструктор. Точнее любой из конструкторов в случае, если для класса их определено несколько. Основная задача конструктора — обеспечить инициализацию экземпляра класса при его создании. Для ее выполнения конструктор должен инициализировать все поля класса. Ведь из начального состояния каждого компонента класса, составляется общее начальное состояние экземпляра класса в целом. Если же какое-то из полей класса останется не инициализированным, это может привести к негативным и непредсказуемым последствиям в процессе жизненного цикла такого, не в полной мере инициализированного, экземпляра.

- **Рекомендация:** *обеспечивайте инициализацию всех полей класса в каждом конструкторе!*

Каким же образом правильно произвести инициализацию полей класса в конструкторе? Может показаться, что достаточно будет способа, как в примере ниже.

*Пример 3*

```
#include <iostream>

class Point
```

```

{
    int x;
    int y;
public:
    Point() { x = 0; y = 0; } // конструктор по умолчанию
    Point(int pX, int pY) { x = pX; y = pY; } // конструктор
                                     // с параметрами
};

int main()
{
    Point p1; // используется конструктор по умолчанию
    Point p2{ 42,33 }; // используется конструктор с параметрами
    return 0;
}

```

Хотя в конечном результате мы и получим экземпляры класса с предсказуемым начальным состоянием, в теле конструктора происходит присваивание значений полям, а не их инициализация! В примере выше, при создании экземпляра класса, сперва происходит создание полей `x` и `y` без их истинной инициализации, а уже потом, в теле конструктора, происходит присваивание значений не инициализированным переменным. Вместо одного действия, вполне достаточного для решения задачи, выполняется еще одно дополнительное «лишнее» присваивание. Более того, таким способом мы не сможем инициализировать константы или ссылки. Ведь им нельзя присвоить значение, их можно только по-настоящему инициализировать. Так же не оптимально будет «инициализировать» таким образом вложенные классы — для них сперва будет вызван конструктор по умолчанию, а лишь

затем, в теле конструктора, произойдет присваивание. Пример, иллюстрирующий описанную проблему.

#### Пример 4

```
#include <iostream>
class Point
{
    // поля определены с помощью public специально!
public:
    int x;
    int y;
    // конструктор по умолчанию
    Point() { x = 0; y = 0; std::cout << "Point Default
                                                    constructor\n";}

    // конструктор с параметрами
    Point(int pX, int pY)
    {
        x = pX;
        y = pY;
        std::cout << "Point Parametrized constructor\n";
    }
};

class Rectangle
{
    Point leftUpperCorner;
    int width;
    int height;
public:
    // конструктор по умолчанию
    Rectangle()
    {
        leftUpperCorner.x = 10;
        leftUpperCorner.y = 10;
        width = 0;
        height = 0;
        std::cout << "Rectangle Default constructor\n";
    }
}
```



```

// конструктор с параметрами
Rectangle(int x, int y, int widthP, int heightP)
{
    leftUpperCorner.x = x;
    leftUpperCorner.y = y;
    width = widthP;
    height = heightP;
    std::cout << "Rectangle Parametrized constructor\n";
}
};

int main()
{
    // используется конструктор по – умолчанию
    Rectangle rect;
    // используется конструктор с параметрами
    Rectangle rect1{ 42, 33, /*вершина*/ 10,
                    /*ширина*/ 5 /*высота*/ };
    return 0;
}

```

Вывод результата работы кода из примера 4:



Рисунок 2

Проанализируем результат подробнее: первая и вторая строка в Выводе — результат создания экземпляра `rect`. В процессе его создания на первом шаге создается экземпляр `Point`, вызовом его конструктора по умолчанию. Затем, в конструкторе по умолчанию для `Rectangle` происходит присваивание начальных значений, как для

полей `leftUpperCorner` класса `Point`, так и для полей `width` и `height`. При этом мы специально сделали поля `Point` `public`, чтобы в данном примере не использовать сеттеры. Если оставить поля `private`, мы не смогли бы без сеттеров присвоить им значения из конструкторов класса `Rectangle`! Ну и наконец, последние две строки Вывода показывают процесс создания экземпляра `rect1` — сперва создается экземпляр `Point`, используя конструктор по умолчанию, затем в теле конструктора с параметрами `Rectangle` на основе переданных значений полям присваиваются начальные значения. То есть в обоих случаях сперва создается экземпляр `Point` с начальным состоянием по умолчанию, а лишь затем ему присваивается желаемое начальное состояние — две операции вместо одной корректной инициализации!

Разберем, как достичь *оптимальной инициализации* полей класса и какой наиболее предпочтительный для этого способ. Итак, предпочтительно инициализировать поля класса в конструкторе — используя «список инициализации членов класса» (иное название — «список инициализаторов членов класса»). Данный список следует непосредственно за сигнатурой (имя и список формальных параметров) конструктора и разделен с ней двоеточием.

```
Point(int pX, int pY) : x { pX }, y { pY }
```

После двоеточия следуют имена полей класса и их инициализаторы (значения, которыми производится соответствующая инициализация) заключенные в фигурные скобки `{}` — то есть по сути имеет место *унифицированная инициализация*! Поля класса с инициализаторами

в фигурных скобках разделены запятыми. Важно: в конце списка инициализации членов класса точки с запятой нет! Далее после списка инициализаторов *следует тело соответствующего конструктора*, заключенное в свои фигурные скобки. Важно: даже если тело конструктора *пустое*, после списка инициализаторов *блок фигурных скобок обязателен!*

```
Point() : x{ 0 }, y{ 0 } {}
```

Если список инициализаторов помещается на одной строке с сигнатурой — оптимально все разместить в строку. Слишком длинный список инициализаторов следует размещать на следующей за сигнатурой конструктора строке, обязательно с отступом перед двоеточием (для лучшей читаемости кода). Если же список инициализаторов крайне велик, то, опять же, для лучшей читаемости — рекомендуется каждый элемент помещать с новой строки с отступом.

Важно отметить, что поля класса инициализируются не в том порядке, в котором *они указаны* в списке инициализаторов, а в *порядке декларации* соответствующих полей в классе. Именно поэтому поля класса в списке инициализаторов следует располагать в том же, что и при декларации полей — порядке. Так не будет путаницы, кто за кем инициализируется. Так же следует обратить внимание, на то, чтобы одно поле класса *не инициализировалось* непосредственно или в результате вычислений — *значением другого* поля класса, которое, согласно вышеописанного свойства будет проинициализировано позднее.

## Пример 5

```
#include <iostream>

class BadOrder
{
    int fieldOne;
    int fieldTwo;
public:
    BadOrder(int param) : fieldTwo{ param },
                        fieldOne{ fieldTwo + 10 } {}
    void print()
    {
        std::cout << "fieldOne = " << fieldOne << '\n'
                    << "fieldTwo = " << fieldTwo << '\n';
    }
};

class GoodOrder
{
    int fieldOne;
    int fieldTwo;
public:
    GoodOrder(int param) : fieldOne{ param + 10 },
                        fieldTwo{ param } {}
    void print()
    {
        std::cout << "fieldOne = " << fieldOne << '\n'
                    << "fieldTwo = " << fieldTwo << '\n';
    }
};

int main()
{
    std::cout << "BadOrder\n";
    BadOrder t1{ 42 };
    t1.print();
}
```

```

std::cout << "GoodOrder\n";
GoodOrder t2{ 33 };
t2.print();

return 0;
}

```

Вывод результата работы кода из примера 5:



```

Microsoft Visual Studio Debug Console
BadOrder
fieldOne = -858993450
fieldTwo = 42
GoodOrder
fieldOne = 43
fieldTwo = 33

```

Рисунок 3

Как видно из Вывода 5, в случае с `BadOrder`, `fieldOne` инициализирован на основе не определенного на момент инициализации `fieldOne`, значения `fieldTwo`, что привело к неожиданному результату. Вариант `GoodOrder` лишен данного недостатка, здесь инициализация происходит на основе значения параметра конструктора и не зависит от порядка инициализации полей.

Вооружившись новыми знаниями, исправим недостатки и проблемы, увиденные в Примере 4.

*Пример 6*

```

#include <iostream>

class Point
{
    int x;
    int y;
}

```

```

public:
    // конструктор по умолчанию
    Point() : x{ 0 }, y{ 0 }
    { std::cout << "Point Default constructor\n"; }
    // конструктор с параметрами
    Point(int pX, int pY) : x{ pX }, y{ pY }
    { std::cout << "Point Parametrized constructor\n"; }
};

class Rectangle
{
    Point leftUpperCorner;
    int width;
    int height;

public:
    // конструктор по умолчанию
    Rectangle()
        : leftUpperCorner{ 10, 10 }, width{ 0 }, height{ 0 }
    { std::cout << "Rectangle Default constructor\n"; }

    // конструктор с параметрами
    Rectangle(int x, int y, int widthP, int heightP)
        : leftUpperCorner{ x, y }, width{ widthP },
          height{ heightP }
    { std::cout << "Rectangle Parametrized constructor\n"; }
};

int main()
{
    // используется конструктор по умолчанию
    Rectangle rect;
    // используется конструктор с параметрами
    Rectangle rect1{ 42, 33, /*вершина*/ 10 /*ширина*/,
                    5 /*высота*/ };

    return 0;
}

```

## Вывод результата работы кода из примера 6:



Рисунок 4

Как мы можем судить из Вывода, экземпляры `Point`, внутри `Rectangle` создаются сразу с «правильными» начальными значениями, используя конструктор с параметрами — за одну операцию, а не за две!

Иной способ инициализации полей класса — инициализировать их непосредственно при декларации соответствующего поля в классе.

*Пример 7*

```

#include <iostream>

class Point
{
    int x{ -100 };
    int y{ -100 };
public:
    int getX() { return x; }
    int getY() { return y; }
};

int main()
{
    Point point;
    std::cout << "point.x = " << point.getX() << '\n';
    std::cout << "point.y = " << point.getY() << '\n';
    return 0;
}

```

## Вывод результата работы кода из примера 7:



Рисунок 5

Отдельно следует рассмотреть случай инициализации как непосредственно при декларации поля, так и средствами конструктора — в таком случае «победу» одерживает конструктор.

*Пример 8*

```
#include <iostream>

class Point
{
    int x{ -100 };
    int y{ -100 };

public:
    // конструктор по умолчанию
    Point() : x{ 0 }, y{ 0 }
    { std::cout << "Point Default constructor\n"; }
    // конструктор с параметрами
    Point(int pX, int pY) : x{ pX }, y{ pY }
    {
        std::cout << "Point Parametrized constructor\n";
    }
    int getX() { return x; }
    int getY() { return y; }
};

int main()
{
    // поля x и y инициализируются конструктором
```



```

// по умолчанию в 0,0 соответственно
// инициализаторы -100,-100 игнорируются!
Point point;
std::cout << "point.x = " << point.getX() << '\n';
std::cout << "point.y = " << point.getY() << '\n';

// поля x и y инициализируются конструктором
// с параметрами в 42,33 соответственно
// инициализаторы -100,-100 игнорируются!
Point point1{ 42,33 };
std::cout << "point1.x = " << point1.getX() << '\n';
std::cout << "point1.y = " << point1.getY() << '\n';
return 0;
}

```

Вывод результата работы кода из примера 8:



```

Microsoft Visual Studio Debug Console
Point Default constructor
point.x = 0
point.y = 0
Point Parametrized constructor
point1.x = 42
point1.y = 33

```

Рисунок 6

Данный способ рекомендуется применять лишь в совершенно тривиальных классах, у которых возможно даже нет явно определенных конструкторов. Не рекомендуется так же злоупотреблять и смешивать инициализацию при декларации поля и инициализацию поля в конструкторе во избежание путаницы.

### 3. Делегирование конструкторов

Нередко класс содержит сразу несколько конструкторов, при этом часть кода в них дублируется:

*Пример 9*

```
class Person
{
    char* name;
    uint16_t age;
    /* uint16_t – unsigned integer 16 bit type
       рекомендуемая современным стандартом нотация
       целочисленных типов, занимающая предсказуемое
       количество байт на любой архитектуре.
       Аналог unsigned short */
    uint32_t socialId;
    // Аналогично предыдущему unsigned integer 32 bit type –
    // аналог unsigned int

public:
    Person() : name{ nullptr }, age{ 0 }, socialId{ 0 }
    {
        std::cout << "Person constructed\n";
    }

    Person(const char* nameP)
        : name{ new char[strlen(nameP) + 1] }, age{ 0 },
          socialId{ 0 }
    {
        strcpy_s(name, strlen(nameP) + 1, nameP);
        std::cout << "Person constructed\n";
    }
}
```

```

Person(const char* nameP, uint16_t ageP)
    : name{ new char[strlen(nameP) + 1] },
      age{ ageP }, socialId{ 0 }
{
    strcpy_s(name, strlen(nameP) + 1, nameP);
    std::cout << "Person constructed\n";
}

Person(const char* nameP, uint16_t ageP,
        uint32_t socialIdP)
    : name{ new char[strlen(nameP) + 1] },
      age{ ageP }, socialId{ socialIdP }
{
    strcpy_s(name, strlen(nameP) + 1, nameP);
    std::cout << "Person constructed\n";
}

~Person()
{
    delete[] name;
    std::cout << "Person destructed\n";
}
};

```

Как видно, у нас есть 4 конструктора для разного рода инициализации класса. Немалая часть кода в данных конструкторах дублируется. Как избежать повторов? Возможно для этого следует вызывать один конструктор из другого? В принципе это возможно, но такой вызов приведет к совершенно неожиданным последствиям — создастся временный объект класса на основе вызванного конструктора, при этом инициализации вновь создаваемого объекта не произойдет. Правильным решением будет использовать механизм делегирования

конструктора. Суть его заключается в использовании конструктора в уже знакомом нам списке инициализации членов класса.

### Пример 10

```
#include <iostream>

class Person
{
    char* name;
    uint16_t age;
    uint32_t socialId;
public:
    Person(const char* nameP, uint16_t ageP,
           uint32_t socialIdP)
        : name{ nameP ? new char[strlen(nameP) + 1]
          : nullptr },
          age{ ageP },
          socialId{ socialIdP }
    {
        if (name)
        {
            strcpy_s(name, strlen(nameP) + 1, nameP);
        }
        std::cout << "Person constructed\n";
    }

    Person() : Person{ nullptr, 0, 0 } {}
    /* Конструктор по умолчанию делегирует (перенаправляет)
       свою работу конструктору с параметрами, указывая
       желаемые параметры. */

    Person(const char* nameP) : Person{ nameP, 0, 0 } {}

    Person(const char* nameP, uint16_t ageP) :
        Person{ nameP, ageP, 0 } {}
}
```

```

~Person()
{
    delete[] name;
    std::cout << "Person destructed\n";
}

void print()
{
    if (name)
    {
        std::cout << "Name: " << name << '\n' <<
            "Age: " << age << '\n' <<
            "SocialID: " << socialId << '\n';
    }
    else
    {
        std::cout << "[empty person]" << '\n';
    }
}
};

int main()
{
    Person nobody;
    nobody.print();

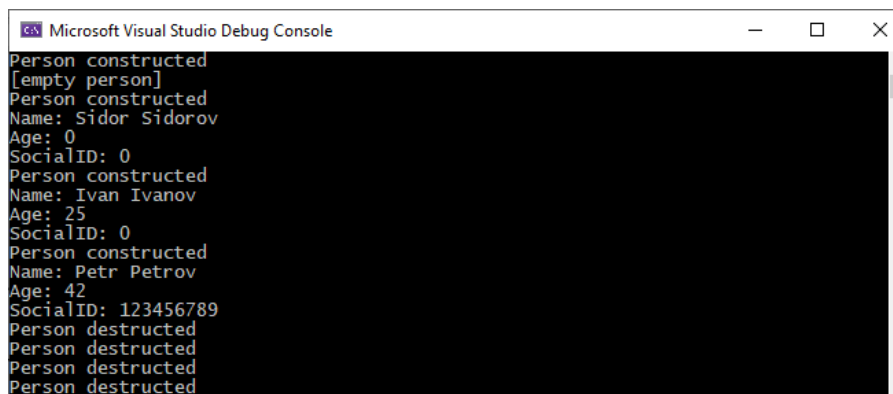
    Person person1{ "Sidor Sidorov" };
    person1.print();

    Person person2{ "Ivan Ivanov", 25 };
    person2.print();

    Person person3{ "Petr Petrov", 42, 123456789 };
    person3.print();
    return 0;
}

```

## Вывод результата работы кода из примера 10:



```

Microsoft Visual Studio Debug Console
Person constructed
[empty person]
Person constructed
Name: Sidor Sidorov
Age: 0
SocialID: 0
Person constructed
Name: Ivan Ivanov
Age: 25
SocialID: 0
Person constructed
Name: Petr Petrov
Age: 42
SocialID: 123456789
Person destructed
Person destructed
Person destructed
Person destructed

```

Рисунок 7

Рассмотрим внимательнее, как все устроено. Сперва объявляем наиболее специфичный, самый «подробный» конструктор с тремя параметрами, именем, возрастом и номером соц. страхования.

```
Person(const char* nameP, uint16_t ageP, uint32_t socialIdP)
```

Обратите внимание, как инициализируется поле `name` — мы используем тернарный оператор и в зависимости от значения параметра `nameP`, либо выделяем динамическую память для хранения строки с именем и сохраняем указатель, либо записываем в поле `name` значение `nullptr`.

```
name{ nameP ? new char[strlen(nameP) + 1] : nullptr }
```

Остальные поля класса `age` и `socialId` инициализируются значениями соответствующих параметров конструктора.

Далее, в теле конструктора, в случае наличия в поле `name` действительного, не `nullptr` указателя, копируем строку-параметр в выделенный под нее блок памяти. Так же выводим информационное сообщение о создании экземпляра `Person`. Дальше для контраста следует наименее специфичный и «подробный» конструктор по умолчанию. Вот тут-то и вступает в действие механизм делегирования конструктора! В списке инициализаторов мы используем выше определенный конструктор с тремя параметрами, передавая в качестве параметров самые общие значения, а именно `nullptr` для `name` и нули для оставшихся двух полей. *Не забываем* указать пустые фигурные скобки после списка инициализаторов полей.

```
Person() : Person{ nullptr, 0, 0 } {}
```

Конструктор по умолчанию готов! Ни единой строки дублирующего кода! Аналогичным образом делегируем работу самому специфичному конструктору из оставшихся двух конструкторов, в каждом из случаев передавая ему соответствующие параметры. Опять же без повторений кода.

Для иллюстрации работы добавляем в класс функцию-член `print` и деструктор. Для теста создаем 4 экземпляра `Person`, пользуясь поочередно всеми конструкторами. Вывод программы свидетельствует о правильности проделанной работы.

При использовании механизма делегирования конструктора следует учитывать ряд ограничений:

- не допустимо инициализировать поля класса в списке инициализаторов вместе с использованием делегирования конструктора;

- следует не допускать циклического вызова делегирующих конструкторов, когда конструктор **A** делегирует работу конструктору **B**, а тот в свою очередь делегирует инициализацию конструктору **A** (**A** и **B** условные имена конструкторов, реальные конструкторы имеют одинаковые имена!)

Представленный выше пример, не единственный способ использования делегирования конструкторов. В общем случае произвольный конструктор может делегировать работу любому другому, а не лишь одному «избранному» конструктору. Главное — следить за отсутствием циклических вызовов!



## 4. Статические члены класса: переменные и функции

При создании экземпляра класса для него индивидуально выделяется память под каждое его поле. При создании последующих экземпляров, *для каждого из них*, так же индивидуально выделяется память *под каждое поле*. Любой экземпляр может независимо работать со своими полями, не затрагивая (и даже не «зная» о их наличии!) остальные экземпляры класса.

*Пример 11*

```
#include <iostream>

// упрощенный класс Point с public полями
class Point
{
    public:
        int x;
        int y;
};

int main()
{
    // Выделяется и инициализируется память под
    // "персональный" x и y для pointOne
    Point pointOne{ 1,1 };

    // Выделяется инициализируется память под
    // "персональный" x и y для pointTwo
    Point pointTwo{ 2,2 };
```

```

// модифицируем PointOne поля x и y
pointOne.x = 4;
pointOne.y = 6;

// проверяем, что значения x и y для pointOne изменились
std::cout << "pointOne: x = " << pointOne.x <<
    " y = " << pointOne.y << '\n';
// проверяем, что значения x и y для pointTwo не затронуты
std::cout << "pointTwo: x = " << pointTwo.x <<
    " y = " << pointTwo.y << '\n';
return 0;
}

```

Вывод результата работы кода из примера 11:



```

Microsoft Visual Studio Debug Console
pointOne: x = 4 y = 6
pointTwo: x = 2 y = 2

```

Рисунок 8

Таким образом, состояние одного экземпляра совершенно не связано с состоянием других. Однако, для решения отдельных задач необходимо иметь некоторое разделяемое состояние. Говоря проще — иметь поле класса, значение которого было бы одинаковым и общим для всех экземпляров. Идентичности достичь легко — достаточно, определить, к примеру, `const int maxX{1500};` в классе `Point` и теперь у всех экземпляров `Point` значение поля `maxX` одинаковое и неизменное, то есть константное. Однако, дополнительное свойство неизменности не всегда уместно. Также, следует обратить внимание на то, что несмотря на идентичность значения поля `maxX`, память под него будет выделяться для каждого экземпляра

класса `Point` персонально, то есть общности поля мы так и не достигли. Для корректного решения такого рода задач, необходимо использовать `static`-поля класса. Такие поля могут быть произвольного допустимого типа и не обязательно должны быть константными. Специфика `static`-полей заключается в том, что они не являются частью экземпляра класса, они не присущи каждому экземпляру персонально. Напротив — такие поля присущи классу в целом и доступны из произвольного экземпляра!

Статические поля создаются в момент старта программы и существуют до ее завершения. Работа со `static`-полями имеет некоторые особенности. Начнем с того, что для инициализации таких полей мы не можем использовать конструкторы. Мы можем инициализировать константные целочисленные поля или `enum`-поля при их декларации в классе. Для инициализации `static`-полей произвольного типа, не обязательно являющегося константой, следует отметить специфику синтаксиса — инициализация производится в глобальной области видимости, вне класса, вне какой-либо функции, с указанием типа поля, его полного имени и инициализатора. Ключевое слово `static` указывать не нужно! Действие модификаторов доступа `private` и `protected` не распространяются на инициализацию статического поля класса, но на последующий доступ — да!

### Пример 12

```
#include <iostream>

class Demo
{
```

```
public:
    int personal;
    static int common;
};

int Demo::common{ 0 };

int main()
{
    // создаем экземпляр и инициализируем персональное
    // поле personal для d1
    Demo d1{ 1 };

    // создаем экземпляр и инициализируем персональное
    // поле personal для d2
    Demo d2{ 2 };

    // присваиваем значение общему полю common
    d1.common = 42;

    // проверяем значение общего поля в d2
    std::cout << "d2.common = " << d2.common << '\n'; //
                                                         // 42 на экране

    return 0;
}
```

Вывод результата работы кода из примера 12:



Рисунок 9

Как уже было сказано, статические поля не являются частью экземпляра класса, так почему же тогда мы обращаемся к ним как к обычным полям? На самом деле, это не единственный способ доступа к static-полям

и присутствует он для удобства и единообразия. Иным способом получения доступа, служит указание имени поля с уточнением, через оператор `::` имени класса — `Demo::common` в нашем примере. Ключевым моментом, важным для понимания, есть то, что память под поле `Demo::common` выделяется даже если не создано ни единого экземпляра класса `Demo`!

### Пример 13

```
#include <iostream>

class Demo
{
public:
    int personal;
    static int common;
};

int Demo::common{ 42 };
int main()
{
    std::cout << "Demo::common = " << Demo::common << '\n';
    return 0;
}
```

Вывод результата работы кода из примера 13:



Рисунок 10

Обратите внимание, мы не создали ни единого экземпляра класса `Demo`, но при этом смогли получить значение поля `common`.

Рассмотрим использования статического поля класса на примере демонстрационного класса `NumberStorage`. Его задача — хранение в динамической памяти некоторого массива из целых чисел. Единственный, явно определенный конструктор, принимает параметром количество требуемых элементов, выделяет под них блок в динамической памяти и заполняет этот блок случайными числами. Также, конструктор выводит на экран диагностическое сообщение о том, сколько дополнительной памяти выделено и сколько всего занято динамической памяти всеми экземплярами класса. Дополнительно, есть функция-член для вывода на экран всего массива. В деструкторе класса освобождается память и выводится, аналогичная конструктору, диагностика с той лишь разницей, что отображается освобожденная память и общее количество, занятой всеми объектами, памяти. Рассмотрим код примера.

### *Пример 14*

```
#include <iostream>
#include <ctime> // для функции time

class NumberStorage
{
    int* storage;
    uint32_t elementsCount;
    static uint32_t usedMemory;

public:
    NumberStorage(uint32_t elementsCountP)
        :storage{ new int[elementsCountP] },
        elementsCount{ elementsCountP }
```

```

{
    uint32_t used{ elementsCount * sizeof(int) };
    usedMemory += used;
    std::cout << "NumberStorage: additional " << used
              << " bytes used. Total: "
              << usedMemory << '\n';
    for (uint32_t i{ 0 }; i < elementsCount; ++i)
    {
        storage[i] = rand() % 10;
    }
}
~NumberStorage()
{
    uint32_t freed{ elementsCount * sizeof(int) };
    delete[] storage;
    usedMemory -= freed;
    std::cout << "NumberStorage: freed " << freed
              << " bytes. Total used: "
              << usedMemory << '\n';
}

void print()
{
    for (uint32_t i{ 0 }; i < elementsCount; ++i)
    {
        std::cout << storage[i] << ' ';
    }
    std::cout << '\n';
}

static uint32_t getUsedMemory()
{
    return usedMemory;
}
};

uint32_t NumberStorage::usedMemory{ 0 };

```

```

int main()
{
    // Задаем номер последовательности случайных чисел
    // исходя из текущего времени.
    srand(time(nullptr));

    std::cout << "Total memory used: "
              << NumberStorage::getUsedMemory() << '\n';

    const int poolSize{ 3 };
    NumberStorage pool[poolSize]{ rand() % 101,
                                   rand() % 101, rand() % 101 };

    std::cout << "Total memory used: "
              << NumberStorage::getUsedMemory() << '\n';
    return 0;
}

```

Вывод результата работы кода из примера 14:

```

Microsoft Visual Studio Debug Console
Total memory used: 0
NumberStorage: additional 340 bytes used. Total: 340
NumberStorage: additional 164 bytes used. Total: 504
NumberStorage: additional 236 bytes used. Total: 740
Total memory used: 740
NumberStorage: freed 236 bytes. Total used: 504
NumberStorage: freed 164 bytes. Total used: 340
NumberStorage: freed 340 bytes. Total used: 0

```

Рисунок 11

Мы намеренно не выводим элементы массивов на экран, чтобы не «засорять» вывод программы не существенной, в данном случае, информацией. Пример демонстрирует еще одно новое понятие — статические функции-члены класса. Если некоторая функция-член в классе обращается лишь к статическим членам данного класса, то она так же может стать статической и ее так



же можно будет вызывать в контексте класса, а не экземпляра класса! Именно такой функцией-членом и является `getUsedMemory()` в нашем примере. Вызов статической функции-члена, аналогичен доступу к статическому полю класса — указывается имя класса, оператор уточнения имени `::` и собственно имя статической функции-члена вместе с фактическими параметрами в круглых скобках (если они есть). При желании или необходимости статическую функцию-член можно, так же, вызвать и через экземпляр класса.

Для того, чтобы объявить функцию-член класса статической, необходимо указать ключевое слово `static` перед ее декларацией в классе. Не всякую функцию-член можно сделать статической. Функция-член, которая помимо статических полей *требует для своей работы так же обычные*, не статические поля, *не может быть статической!* Это вполне логично, так как без экземпляра класса у нас нет обычных полей, под них не выделена память и, соответственно, к ним нет доступа. При этом не статичная, обычная, функция-член класса может без ограничений манипулировать как не статическими, так и статическими полями класса.

Статические поля и функции-члены класса можно так же использовать для подсчета созданных/активных экземпляров класса, либо же для присваивания каждому экземпляру уникального порядкового номера. В общем случае, если некие данные не привязаны к конкретному объекту, а присущи всем объектам в целом, либо же отражают общие характеристики/состояния всех объектов, а не какого-то конкретно — то такие поля следует

делать статическими. Хорошим примером могут быть константы, используемые классом в целом или таблицы (массивы) с некоторым набором константных значений используемые в классе. То же самое относится и к функциям-членам — если некая функция-член выполняет работу не для конкретного экземпляра класса, а для класса вообще, то ее следует делать статической. Хотя технически ничто не мешает создавать класс, содержащий только статические поля и функции-члены, но делать этого не следует! Невозможно создавать экземпляры данного класса и практический смысл его использования весьма сомнителен.

## 5. Указатель this

Класс в C++ — это чертеж, на основе которого создаются экземпляры-объекты. При этом необходимо понимать, как происходит выделение памяти под поля и под функции-члены для каждого конкретного экземпляра класса. Итак, рассмотрим простой пример.

### *Пример 15*

```
#include <iostream>

class Date
{
    int day;
    int month;
    int year;

public:
    Date(int dayP, int monthP, int yearP)
        : day{ dayP }, month{ monthP }, year{ yearP }
    {}
    Date() : Date(1, 1, 1970) {}
    void print() { std::cout << day << '.' << month
        << '.' << year << '\n'; }
};

int main()
{
    /* Создаем и инициализируем экземпляр класса Date
       Выделяется память для экземпляра date1 и его
       персональных полей date1.day, date1.month,
       date1.year */
    Date date1{ 1,1,2020 };
```

```
/* Создаем и инициализируем экземпляр класса Date
   Выделяется память для экземпляра date2 и его
   персональных полей date2.day, date2.month,
   date2.year */
Date date2{ 24,07,1976 };

/*
   Вызываем функцию-член print.
   Код функции-члена общий для всех экземпляров Date.
*/
date1.print();
date2.print();

return 0;
}
```

Вывод результата работы кода из примера 15:

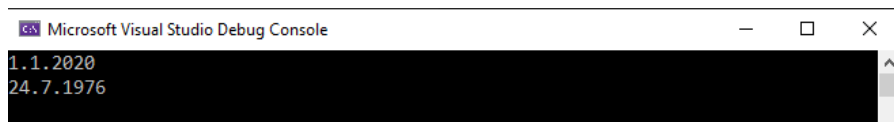


Рисунок 12

Как мы уже знаем, создание экземпляра класса приводит к выделению памяти под все поля класса индивидуально для каждого экземпляра. Что же происходит с функциями членами? Можно было бы предположить, что и функции-члены так же индивидуальны для каждого экземпляра. Однако это не так. На самом деле, все функции-члены в рамках класса занимают память, соответственно их размеру, один раз. При создании нескольких экземпляров не происходит дополнительного выделения памяти для функций-членов. Иными словами данные

(поля) индивидуальны, а код по их обработке (функции-члены) — разделяемые для экземпляров класса. Каким же образом один и тот же метод производит работу с разными экземплярами класса? Вернемся к коду из Примера 15. Функция-член `print()` не принимает ни одного параметра и не создает внутри себя ни одной локальной переменной. Чем же тогда являются идентификаторы `day`, `month`, `year` внутри этого метода? Полями класса! Да, но как один единственный метод «узнает» с каким именно экземпляром класса ему работать, с какими конкретно полями он имеет дело и как получить доступ к блоку памяти, выделенному под определенный экземпляр класса?

На самом деле, здесь нет ничего сверхъестественного, все дело в том, что каждая не статическая функция-член в классе своим первым, неявным параметром получает указатель на экземпляр класса, через который была вызвана функция-член. Этот неявный первый параметр представляет из себя константный указатель на экземпляр класса и называется `this`.

```
Date* const this /* this для функций-членов Date*/
```

Обратите внимание, явно указывать `this` в качестве первого параметра не надо, более того если сделать так — это приведет к ошибке! Несмотря на свою неявность на этапе объявления, указатель `this` можно использовать внутри не статических функций-членов как явно, так и не явно. Именно наличие такого указателя и объясняет то, как функция-член связывается с данными конкретного экземпляра. Внутри функции-члена обращение к полям

конкретного экземпляра можно производить явно, через указатель `this` и оператор «->»:

```
void print() {
    std::cout << this->day << ' ' << this->month << ' '
    << this->year << '\n';
}
```

Данный способ доступа к полям не является обязательным, но именно доступ через указатель `this` на экземпляр класса, подразумевается, когда мы просто указываем имя поля внутри не статической функции-члена класса. Наличие неявного `this` и объясняет всю «магию» связи общего кода с индивидуальными для экземпляров класса данными.

Следует ли всегда использовать `this` для доступа к полям и функциям-членам? Чаще всего нет. Тем не менее, бывает необходимо явно указывать `this` в случае, когда это устраняет неоднозначность, например в случае использования идентичного с полем класса имени параметра функции-члена или локальной переменной внутри функции-члена.

```
void setDay(int day) {
    this->day = day;
}
```

Здесь использование `this` устраняет неоднозначность, связанную с именем параметра и именем поля. В то же время следует избегать появления таких конфликтов имен и неоднозначности, например добавляя суффикс `P` или префикс `the` к именам параметров, которые могли

бы потенциально конфликтовать с именами, используемыми для компонентов класса.

```
void setDay(int dayP) { day = dayP; }
```

Еще одним примером применения `this` может послужить создание функций-членов, которые поддерживают вызов по цепочке. Сравним две реализации сеттеров в классе `Date`.

### Пример 16

```
#include <iostream>

class Date
{
    int day;
    int month;
    int year;
public:
    Date(int dayP, int monthP, int yearP)
        : day{ dayP }, month{ monthP }, year{ yearP }
    {}
    Date() : Date(1, 1, 1970) {}
    void print() { std::cout << day << '.' << month << '.'
        << year << '\n'; }
    void setDay(int dayP) { day = dayP; }
    void setMonth(int monthP) { month = monthP; }
    void setYear(int yearP) { year = yearP; }
};

int main()
{
    Date date1;
    date1.setDay(29);
    date1.setMonth(2);
    date1.setYear(2004);
```

```

    date1.print();

    return 0;
}

```

### Пример 17

```

#include <iostream>

class Date
{
    int day;
    int month;
    int year;
public:
    Date(int dayP, int monthP, int yearP)
        : day{ dayP }, month{ monthP }, year{ yearP }
    {}
    Date() : Date(1, 1, 1970) {}
    void print() { std::cout << day << '.' << month << '.'
        << year << '\n'; }
    Date& setDay(int dayP) { day = dayP; return *this; }
    Date& setMonth(int monthP) { month = monthP;
        return *this; }
    Date& setYear(int yearP) { year = yearP;
        return *this; }
};

int main()
{
    Date date1;
    date1.setDay(29).setMonth(2).setYear(2004);
    date1.print();

    return 0;
}

```



## Вывод результата работы кода из примера 16, 17:



Рисунок 13

Итак, в Примере 16 геттеры принимают один соответствующий параметр (день, месяц, год) присваивают значение параметра полю экземпляра класса и завершают свою работу, не возвращая значения (тип возвращаемого значения функции-члена — `void`). Для задания всех трех значений, необходимо последовательно вызвать соответствующие сеттеры. Оказывается, данную задачу можно решить немногим более элегантно. В Примере 17 сеттеры выполняют ту же работу, но в завершении, дополнительно возвращают разыменованный указатель `this` (тип возвращаемого значения функции-члена — `Date&`). Данное дополнение позволяет вызывать сеттеры по цепочке: вызов первого сеттера, `setDay`, происходит на экземпляре `date1` затем `setDay` возвращает ссылку на экземпляр `date1` и, как следствие — возможно используя эту ссылку вызвать сеттер `setMonth` и затем — аналогично вызвать `setYear`. Принципиальной разницы в результате работы сеттеров нет, но сама цепочка вызовов стала нагляднее, а код — элегантнее. Забегая вперед, подобным способом реализован вывод на экран по цепочке вызовов оператора `<<`. В общем случае, если функция-член класса возвращает `void` следует рассмотреть целесообразность замены `void` на `ClassName&` где `ClassName` — имя класса, сделав возможным вызов последующих функций-членов по цепочке.

И в завершении, рассмотрим еще один способ применения `this`. Мы добавим визуализацию порядка запуска конструкторов и деструкторов для экземпляров класса `Date`.

### Пример 18

```
#include <iostream>

class Date
{
    int day;
    int month;
    int year;

public:
    Date(int dayP, int monthP, int yearP)
        : day{ dayP }, month{ monthP }, year{ yearP }
    { std::cout << "Date constructed for " << this << '\n'; }
    Date() : Date(1, 1, 1970) {}
    ~Date() { std::cout << "Date destructed for "
        << this << '\n'; }
    void print() { std::cout << day << '.' << month
        << '.' << year << '\n'; }
    Date& setDay(int dayP) { day = dayP; return *this; }
    Date& setMonth(int monthP) { month = monthP; return *this; }
    Date& setYear(int yearP) { year = yearP; return *this; }
};

int main()
{
    Date date1{ 24,8,1991 };
    Date date2{ 12,4,1961 };
    date1.print();
    date2.print();
    return 0;
}
```

## Вывод результата работы кода из примера 18:



```
Microsoft Visual Studio Debug Console
Date constructed for 009DF928
Date constructed for 009DF914
24.8.1991
12.4.1961
Date destructed for 009DF914
Date destructed for 009DF928
```

Рисунок 14

В Примере 18 добавлен вывод на экран значения `this` из конструктора и деструктора. Теперь есть возможность проследить порядок создания-уничтожения экземпляров. Сперва создается экземпляр `date1{24,8,1991}` ему соответствует адрес `009DF928`, затем создается экземпляр `date2{12,4,1961}` с адресом `009DF914` соответственно. По сообщениям из деструктора видно, что уничтожение экземпляров класса происходит строго в обратном порядке, сперва для `009DF914` и затем — для `009DF928`. Конкретное значение адресов будет отличаться от приведенных в выводе, более того значения вероятнее всего будут изменяться от одного запуска программы к другому, на самом деле важны не абсолютные значения адресов, а их «парность». Вывод значения `this` на экран из различных функций-членов может существенно упростить отладку программы.

## 6. Конструктор копирования

В предыдущих разделах урока описывались различные способы инициализации объектов, рассмотрим еще один — копирующую инициализацию. Для этого нам понадобится небольшой класс — базовая реализация сущности простая дробь. В примере ниже будет реализован лишь самый необходимый минимум функциональности, конструктор с параметрами, конструктор по умолчанию, деструктор и функция-член вывода дроби на экран. Для визуализации работы конструкторов и деструктора они выводят отладочную информацию на экран.

### *Пример 19*

```
#include <iostream>

class Fraction
{
    int numerator;
    int denominator;
public:
    Fraction(int num, int denom)
        : numerator{ num }, denominator{ denom }
    {
        std::cout << "Fraction constructed for "
                    << this << '\n';
    }
    Fraction() : Fraction(1, 1) {}
    ~Fraction() { std::cout << "Fraction destructed for "
                        << this << '\n'; }
    void print()
    {
```

```

        std::cout << '(' << numerator << " / "
                << denominator << ")\n";
    }
};

int main()
{
    /* Создаем и инициализируем значениями числителя
       и знаменателя экземпляра Fraction – a*/
    Fraction a{ 2,3 };
    /* Создаем и инициализируем текущим значением
       экземпляра a, экземпляра Fraction – b*/
    Fraction b{ a };

    std::cout << "a = ";
    a.print();
    std::cout << "\nb = ";
    b.print();
    std::cout << '\n';

    return 0;
}

```

Вывод результата работы кода из примера 19:



```

Microsoft Visual Studio Debug Console
Fraction constructed for 00EFA30
a = (2 / 3)
b = (2 / 3)
Fraction destructed for 00EFA20
Fraction destructed for 00EFA30

```

Рисунок 15

Вывод 19 демонстрирует нам, что сперва вызывает-ся конструктор для экземпляра `a{2,3}` ему соответствует адрес `00EFA30`, затем, согласно коду в Примере 19 создается экземпляр `b{a}` но мы не видим соответствующего сообщения о работе конструктора!

Затем на экран выводятся **a**, **b** и мы видим работу деструктора для «не созданного» экземпляра **b** (ему соответствует адрес **00EFFA20**) и в завершении — работу деструктора для экземпляра **a** с адресом **00EFFA30**. Увиденное может показаться немного странным.

Рассмотрим причины такого поведения. В классе **Fraction** явно определены два конструктора, один — принимающий два целочисленных параметра и один, конструктор по умолчанию, вовсе без параметров, который делегирует инициализацию предыдущему конструктору с двумя параметрами. Экземпляр **a** мы инициализируем при помощи конструктора с двумя целочисленными параметрами, а как же с экземпляром **b**?

Разве мы создаем его, используя конструктор по умолчанию? Или явно задавая значения числителя и знаменателя? Нет, все дело в том, что инициализация экземпляра **b** происходит согласно текущему значению экземпляра **a**. И в этом случае не задействован ни один из явно определенных нами конструкторов! Именно поэтому нет вывода отладки о создании экземпляра, ему неоткуда взяться.

Как же тогда инициализируется экземпляр **b**, ведь в конечном результате, согласно Вывода 19 там (2 / 3)? Все дело в том, что инициализация происходит автоматически генерируемым компилятором *конструктором копирования*. Без знания внутренней сути класса компилятор автоматически создает конструктор копирования, проводящий поверхностное или побитовое копирование одного объекта, инициализатора, в другой, только что созданный, объект, требующий инициализации.

Конструктор копирования — это специальный конструктор, служащий для копирования существующего экземпляра класса в новый экземпляр того же класса. Если в классе явно не определен конструктор копирования, то аналогично конструктору по умолчанию, компилятор автоматически сгенерирует `public`-конструктор, который выполнит почленную инициализацию полей нового экземпляра значениями соответствующих полей имеющегося экземпляра класса. То есть в примере выше, `b.numerator` проинициализируется значением `a.numerator` и соответственно `b.denominator` — `a.denominator`.

Конструктор копирования можно определить явно, для этого необходимо использовать специальную сигнатуру:

```
ClassName(const ClassName& object);
```

Где `ClassName` — имя класса, для которого определяется конструктор. По сути, конструктор копирования — это конструктор, принимающий экземпляр того же класса по константной ссылке. Почему именно так — будет сказано немного позднее. Пример сигнатуры для `Fraction`:

```
Fraction(const Fraction& fract);
```

Давайте определим явно конструктор копирования для класса `Fraction`:

*Пример 20*

```
#include <iostream>

class Fraction
{
```

```

    int numerator;
    int denominator;

public:
    Fraction(int num, int denom)
        : numerator{ num }, denominator{ denom }
    {
        std::cout << "Fraction constructed for "
                    << this << '\n';
    }
    Fraction() : Fraction(1, 1) {}
    Fraction(const Fraction& fract)
        : numerator{ fract.numerator },
          denominator{ fract.denominator }
    {
        std::cout << "Fraction copy constructed for "
                    << this << '\n';
    }
    ~Fraction() { std::cout << "Fraction destructed for "
                            << this << '\n'; }
    void print()
    {
        std::cout << '(' << numerator << " / "
                    << denominator << ")";
    }
};

int main()
{
    /* Создаем и инициализируем значениями числителя
       и знаменателя экземпляра Fraction – a*/
    Fraction a{ 2,3 };
    /* Создаем и инициализируем текущим значением
       экземпляра a, экземпляр Fraction – b*/
    Fraction b{ a };

    std::cout << "a = ";

```



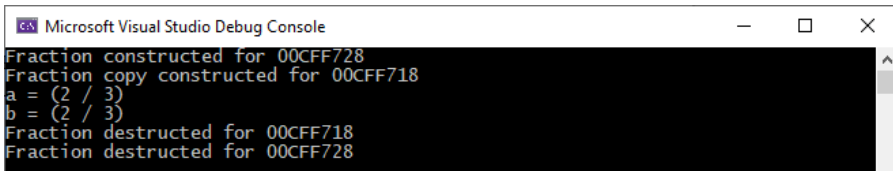
```

a.print();
std::cout << "\nb = ";
b.print();
std::cout << '\n';

return 0;
}

```

Вывод результата работы кода из примера 20:



```

Microsoft Visual Studio Debug Console
Fraction constructed for 00CFF728
Fraction copy constructed for 00CFF718
a = (2 / 3)
b = (2 / 3)
Fraction destructed for 00CFF718
Fraction destructed for 00CFF728

```

Рисунок 16

В Примере 20, в явно определенном конструкторе копирования производится почленная инициализация полей и вывод отладочной информации о создании нового экземпляра копированием. Теперь, согласно Вывода 20, более нет «не созданных» экземпляров класса `Fraction`! Просто экземпляр `a` инициализируется конструктором с двумя целочисленными параметрами, а экземпляр `b` — конструктором копирования, принимающим по константной ссылке экземпляр `Fraction`. Фактически, явно определенный конструктор для `Fraction` выполняет те же действия, что и генерируемый компилятором конструктор копирования, мы только добавили вывод отладочной информации, информирующий нас о том, что выполнен именно конструктор копирования. Если нас не интересует выводить отладочную информацию,

то для класса `Fraction` нет необходимости в явно определенном конструкторе копирования. Но так бывает далеко не со всеми классами! Немногим позднее мы рассмотрим пример, где без явного определения конструктора копирования не обойтись.

В случае, если мы инициализируем экземпляр класса временным, анонимным экземпляром, компилятор может не вызвать конструктор копирования, а провести оптимизацию и, не создавая временный объект, непосредственно инициализировать экземпляр класса инициализаторами временного объекта. В Примере 20 изменим функцию `main` :

### *Пример 21*

```
int main()
{
    /*
        Создаем и инициализируем временным, анонимным
        экземпляром Fraction – a.
        Конструктор копирования не вызовется, временный
        объект не создастся!
        Компилятор непосредственно проинициализирует
        экземпляр a значениями 4 и 6.
    */

    Fraction a{ Fraction{4,6} };

    std::cout << "a = ";
    a.print();
    std::cout << '\n';

    return 0;
}
```

## Вывод результата работы кода из примера 21:



Рисунок 17

Такая оптимизация лишнего создания объекта с его последующим копированием называется элизией. При этом даже если в конструкторе копирования, помимо собственно копирования выполнялась какая-то дополнительная работа, например вывод на экран, тело конструктора копирования не выполнится!

Рассмотрим сигнатуру конструктора копирования внимательнее. Почему так важно, чтоб единственным параметром была константная ссылка на экземпляр класса? Что будет, в случае передачи экземпляра по значению? Например так:

```
ClassName(ClassName object);
```

Если бы такой код можно было бы скомпилировать и запустить, то при попытке воспользоваться копирующей инициализацией, необходимо было бы создать копию объекта-инициализатора! Для этого потребовалось бы вызвать конструктор копирования! Случилась бы бесконечная рекурсия! К счастью, в современных компиляторах такой код даже не компилируется, будут выданы соответствующие ошибки на этапе компиляции. А почему необходим модификатор `const`? Передавая параметр по ссылке, возможно модифицировать его внутри конструктора копирования, что технически возможно,

но логически совершенно не приемлемо — при копировании совершенно нецелесообразно, как бы то ни было, модифицировать оригинал, исходный объект. Для того, чтобы предотвратить даже потенциальную возможность модификации, даже по ошибке — используется именно константная ссылка.

Всегда ли подходит автоматически генерируемый конструктор копирования или явное определение конструктора копирования, в котором проводится лишь поверхностное копирование? Для того, чтобы ответить на этот вопрос, рассмотрим пример упрощенного класса динамического массива. В данном примере мы создадим конструктор копирования, выполняющий поверхностное копирование. Именно такой конструктор автоматически генерируется компилятором, когда программист не предоставляет свою, подходящую классу, версию конструктора копирования.

### *Пример 22*

```
#include <iostream>
/* Внимание!Программа завершается аварийно! Для этого
  примера это нормально.*/
class DynArray
{
    int* arr;
    int size;
public:
    DynArray(int sizeP)
        : arr{ new int[sizeP] {} }, size{ sizeP }
    {
        std::cout << "DynArr constructed for " << size
                    << " elements, for " << this << '\n';
    }
};
```

```

}
DynArray() : DynArray(5) {}
DynArray(const DynArray& object)
    : arr{ object.arr}, size{ object.size }
{
    std::cout << "DynArr copy constructed for "
                << size << " elements, for " << this
                << '\n';
}
int getElem(int idx) { return arr[idx]; }
void setElem(int idx, int val) { arr[idx] = val; }
void print();
void randomize();
~DynArray()
{
    delete[] arr; std::cout << "DynArr destructed for"
                                << size << " elements, for"
                                << this << '\n';
}
};

void DynArray::print()
{
    for (int i{ 0 }; i < size; ++i)
    {
        std::cout << arr[i] << ' ';
    }
    std::cout << '\n';
}

void DynArray::randomize()
{
    for (int i{ 0 }; i < size; ++i)
    {
        arr[i] = rand() % 10;
    }
}

```

```

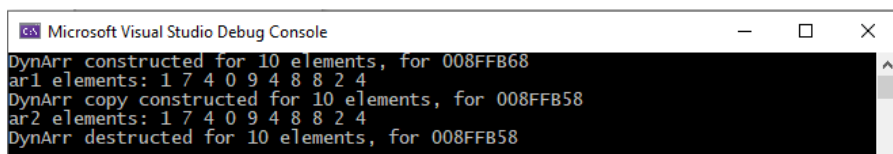
int main()
{
    DynArray ar1{ 10 };
    ar1.randomize();
    std::cout << "ar1 elements: ";
    ar1.print();

    DynArray ar2{ ar1 };
    std::cout << "ar2 elements: ";
    ar2.print();

    return 0;
}

```

Вывод результата работы кода из примера 22:



```

Microsoft Visual Studio Debug Console
DynArr constructed for 10 elements, for 008FFB68
ar1 elements: 1 7 4 0 9 4 8 8 2 4
DynArr copy constructed for 10 elements, for 008FFB58
ar2 elements: 1 7 4 0 9 4 8 8 2 4
DynArr destructed for 10 elements, for 008FFB58

```

Рисунок 18

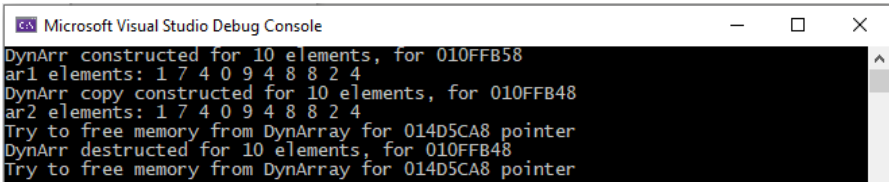
Рассмотрим, что происходит в Примере 22. Сперва создаем экземпляр `ar1` класса `DynArray` из 10 целых чисел и заполняем его случайными числами. Выводим содержимое `ar1` на экран. Затем создаем экземпляр `ar2` как копию `ar1` и так же выводим его на экран. Для копирования используем явно определенный конструктор копирования, который производит почленную инициализацию полей нового экземпляра на основе значений имеющегося экземпляра класса `DynArray`, то есть выполняем поверхностное копирование. Программа завершается вызовом деструкторов в обратном порядке создания экземпляров

класса, сперва вызывается деструктор для `ar2`, а затем — программа аварийно завершается в процессе работы деструктора для `ar1`! Вывод 22 показывает то, что успело отобразиться на экране. От чего так происходит? Немного изменим деструктор, чтоб попытаться в этом разобраться.

### Пример 23

```
~DynArray()
{
    std::cout << "Try to free memory from DynArray for "
               << arr << " pointer\n";
    delete[] arr;
    std::cout << "DynArr destructed for "
               << size << " elements, for "
               << this << '\n';
}
```

Вывод результата работы кода из примера 23:



```
Microsoft Visual Studio Debug Console
DynArr constructed for 10 elements, for 010FFB58
ar1 elements: 1 7 4 0 9 4 8 8 2 4
DynArr copy constructed for 10 elements, for 010FFB48
ar2 elements: 1 7 4 0 9 4 8 8 2 4
Try to free memory from DynArray for 014D5CA8 pointer
DynArr destructed for 10 elements, for 010FFB48
Try to free memory from DynArray for 014D5CA8 pointer
```

Рисунок 19

Итак, деструктор для `ar2` освобождает память занимаемую динамически выделенным массивом по указателю `014D5CA8`, позднее деструктор для `ar1` освобождает память по тому же указателю! Все верно, ведь в нашем конструкторе копирования мы всего лишь скопировали указатель! Мы не создали новую копию массива, не скопировали туда значения из имеющегося массива! Как

и было сказано, была создана поверхностная копия, совершенно не подходящая для данного класса!

Именно для подобного рода классов, где производится выделение динамической памяти, необходима верная, не автоматически созданная компилятором, реализация конструктора копирования. Такая реализация должна на основе имеющегося объекта класса создать его глубокую копию, с выделением новых блоков динамической памяти и копированием туда значений из исходных блоков динамической памяти. Рассмотрим пример такого конструктора.

### *Пример 24*

```
#include <iostream>

class DynArray
{
    int* arr;
    int size;

public:
    DynArray(int sizeP)
        : arr{ new int[sizeP] {} }, size{ sizeP }
    {
        std::cout << "DynArr constructed for " << size
                    << " elements, for " << this << '\n';
    }
    DynArray() : DynArray(5) {}
    DynArray(const DynArray& object)
        : arr{ new int[object.size] }, size{ object.size }
    {
        /* В списке инициализаторов полей класса выше,
           выделяем новый блок динамической памяти того же
           размера, что и в копируемом экземпляре класса
        */
    }
};
```



```

        DynArray. Следующим циклом копируем элементы
        из оригинального блока памяти во вновь
        выделенный. */
    for (int i{ 0 }; i < size; ++i)
    {
        arr[i] = object.arr[i];
    };

    std::cout << "DynArr copy constructed for "
                << size << " elements, for " << this
                << '\n';
}
int getElem(int idx) { return arr[idx]; }
void setElem(int idx, int val) { arr[idx] = val; }
void print();
void randomize();
~DynArray()
{
    std::cout << "Try to free memory from DynArray for"
                << arr << " pointer\n";
    delete[] arr;
    std::cout << "DynArr destructed for " << size
                << " elements, for " << this << '\n';
}
};

void DynArray::print()
{
    for (int i{ 0 }; i < size; ++i)
    {
        std::cout << arr[i] << ' ';
    }
    std::cout << '\n';
}

void DynArray::randomize()
{

```

```

        for (int i{ 0 }; i < size; ++i)
        {
            arr[i] = rand() % 10;
        }
    }

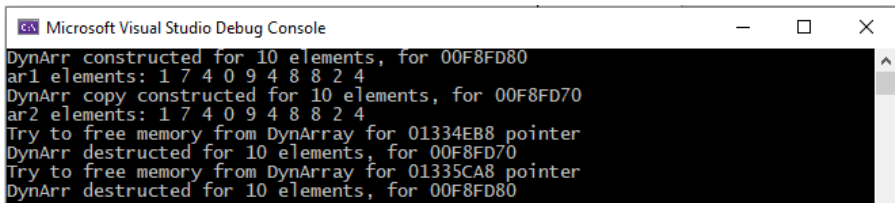
int main()
{
    DynArray ar1{ 10 };
    ar1.randomize();
    std::cout << "ar1 elements: ";
    ar1.print();

    DynArray ar2{ ar1 };
    std::cout << "ar2 elements: ";
    ar2.print();

    return 0;
}

```

Вывод результата работы кода из примера 24:



```

Microsoft Visual Studio Debug Console
DynArr constructed for 10 elements, for 00F8FD80
ar1 elements: 1 7 4 0 9 4 8 8 2 4
DynArr copy constructed for 10 elements, for 00F8FD70
ar2 elements: 1 7 4 0 9 4 8 8 2 4
Try to free memory from DynArray for 01334EB8 pointer
DynArr destructed for 10 elements, for 00F8FD70
Try to free memory from DynArray for 01335CA8 pointer
DynArr destructed for 10 elements, for 00F8FD80

```

Рисунок 20

Теперь программа работает как следует, без сбоев! Конструктор копирования создает глубокую копию экземпляра класса `ar1`. При запуске деструкторов каждый освобождает блок памяти по своему указателю, о чем свидетельствует Вывод 23.

Пример реализации конструктора копирования позволяет увидеть, что в общем случае для подобного рода классов операция копирования является крайне затратной! Необходимо выделить новый блок или блоки динамической памяти и произвести копирование информации из исходного блока или блоков. В приведенном примере речь шла о блоке на 10 целых чисел, весь процесс копирования происходил практически мгновенно, но в общем случае блоки памяти могут быть существенно большего размера! Из выше сказанного следует, что следует избегать копирования объектов, где это возможно и где для этого есть приемлемые альтернативы.

- **Важное замечание!** При передаче экземпляра класса в качестве параметра функции по значению, происходит создание нового экземпляра класса, используя конструктор копирования.

```
void printArray(DynArray array);
```

Аналогично, в случае возврата экземпляра класса из функции по значению, так же создается копия экземпляра класса.

```
DynArray createArray(int size)
{
    DynArray arr{ size };
    arr.randomize();
    return arr;
}
```

Далее, в рамках нашего курса будут рассмотрены способы, как избежать излишнего копирования, где это

возможно, согласно требованиям реальных задач, но пока следует запомнить о «затратности» операции копирования. Где возможно, избегать копирования, передавая экземпляры класса по ссылкам, где уместно — константным. Так же, по ссылкам, где это в принципе возможно, возвращать экземпляры класса из функций.

## 7. Домашнее задание

### 1. Создать класс «Дробь» для представления простой дроби.

*Поля:*

- числитель,
- знаменатель.

*Функции-члены:*

- конструктор принимающий числитель и знаменатель. В конструкторе использовать список инициализаторов полей класса.
  - ▷ конструктор по умолчанию, реализовать через делегирование конструктору с параметрами числитель и знаменатель;
  - ▷ вывод на экран дроби;
  - ▷ сложение/вычитание/умножение простой дроби с простой дробью;
  - ▷ сложение/вычитание/умножение простой дроби с целым числом.

В арифметических операциях предусмотреть возможность вызова операций по цепочке используя указатель `this`.

Предусмотреть сокращение дроби. Сокращение рекомендуется производить в конструкторе.

### 2. Создать класс Человек.

*Поля:*

- идентификационный номер;
- фамилия;

- имя;
- отчество (*для фамилии, имени и отчества память выделять динамически!*);
- дата рождения (рекомендуется создать дополнительный класс Дата (день, месяц, год);

*Функции-члены:*

- конструктор с параметрами идентификационный номер, фамилия, имя, отчество, дата рождения. В конструкторе использовать список инициализаторов полей класса;
- конструктор по умолчанию. В конструкторе использовать делегирование конструктора;
- конструктор копирования;
- деструктор;
- функцию-член для подсчета созданных экземпляров класса «Человек»;
- сеттеры/геттеры для соответствующих полей класса;
- вывод на экран информации о человеке.

### 3. Создать класс Строка

*Поля:*

- длина строки без учета нуль-терминатора;
- указатель на блок памяти, где хранится строка; память выделять динамически!

*Функции-члены:*

- конструктор с параметром-строкой ( `const char*` );
- конструктор с параметром длина строки;
- конструктор копирования;

- деструктор;
- вывод строки на экран;
- сеттер, принимающий в качестве параметра — строку ( `const char*` ). При нехватке уже выделенного блока динамической памяти для копирования в него строки-параметра — произвести корректное перераспределение памяти.

► **Примечание:** *перечень полей и функций-членов в заданиях 1-3 является рекомендованным, а не окончательным. При необходимости возможно добавление требуемых или желаемых полей и функций-членов.*

#### 4. **Создайте программу, имитирующую многоквартирный дом.**

Необходимо иметь классы Человек, Квартира, Дом. Класс Квартира содержит динамический массив объектов класса Человек. Класс Дом содержит массив объектов класса Квартира. Каждый из классов содержит переменные-члены и функции-члены, которые необходимы для предметной области класса. Обращаем ваше внимание, что память под строковые значения выделяется динамически. Не забывайте обеспечить классы различными конструкторами (конструктор копирования обязателен), деструкторами. Класс Человек следует использовать из Задания 2.



## Урок 2. Инициализаторы. Статические переменные-члены и функции-члены. Указатель this и конструктор копирования

© Компьютерная Академия «Шаг», [www.itstep.org](http://www.itstep.org)

© Александр Рыбчанский

Все права на охраняемые авторским правом фото-, аудио- и видеопроизведения, фрагменты которых использованы в материале, принадлежат их законным владельцам. Фрагменты произведений используются в иллюстративных целях в объеме, оправданном поставленной задачей, в рамках учебного процесса и в учебных целях, в соответствии со ст. 1274 ч. 4 ГК РФ и ст. 21 и 23 Закона Украины «Про авторське право і суміжні права». Объем и способ цитируемых произведений соответствует принятым нормам, не наносит ущерба нормальному использованию объектов авторского права и не ущемляет законные интересы автора и правообладателей. Цитируемые фрагменты произведений на момент использования не могут быть заменены альтернативными, не охраняемыми авторским правом аналогами, и как таковые соответствуют критериям добросовестного использования и честного использования.

Все права защищены. Полное или частичное копирование материалов запрещено. Согласование использования произведений или их фрагментов производится с авторами и правообладателями. Согласованное использование материалов возможно только при указании источника.

Ответственность за несанкционированное копирование и коммерческое использование материалов определяется действующим законодательством Украины.