

**Об'єктно-орієнтоване  
програмування  
з використанням мови**

**C++**



# Урок №2

Ініціалізатори.  
Статичні змінні-члени  
та функції-члени.  
Показчик `this` та  
конструктор копіювання

## ЗМІСТ

1. Уніфікована ініціалізація об'єктів .....	3
2. Ініціалізація членів класу .....	6
3. Делегування конструкторів .....	18
4. Статичні змінні-члени і статичні функції-члени класу .....	25
5. Показчик <code>this</code> .....	34
6. Конструктор копіювання .....	43
7. Домашнє завдання .....	59

Додаткові матеріали уроку прикріплені до даного PDF-файлу.  
Для доступу до матеріалів, урок необхідно відкрити в програмі  
Adobe Acrobat Reader.

# 1. Уніфікована ініціалізація об'єктів

Для передбачуваної і коректної роботи будь-яка змінна повинна бути ініціалізованою. При цьому не важливо, якого типу (`int`, `float`, `char`, `bool`, покажчик, структура, клас тощо) змінна — не залишайте її НЕ ініціалізованою. Адже така змінна неповноцінна — у неї немає початкового стану і, відповідно, неможливо отримати її поточне значення.

У більшості випадків, спроба застосування неініціалізованої змінної призводить до помилки під час компіляції, але так трапляється не завжди. Різні версії інтегрованих середовищ розробки і / або компіляторів можуть і не видати помилки на етапі компіляції. Не слід покладатися на випадок в такому важливому питанні!

Розглянемо невеликий приклад:

*Приклад 1.*

```
#include <iostream>

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

    return 0;
}
```



Рисунок 1

У середовищі VisualStudio 2019 фрагмент успішно компілюється і запускається. Ось тільки що ми бачимо в результаті? Чому дорівнює `numbers[0]`? Нічому коректному і придатному до використання! Адже ми додали `2` до поточного, невизначеного, значення `numbers [0]` і результат присвоїли йому ж. Це лише один з безлічі випадків, коли не повністю вдається виявити неініціалізовану змінну під час компіляції і, як наслідок, можлива абсолютно неправильна робота програми. Однак слід зауважити, що в більшості подібних випадків на етапі компіляції виникає попередження (але не помилка!), яке легко проігнорувати.

У C++ існує кілька способів ініціалізувати змінну:

```
int number = 0; // ініціалізація з копіюванням
int value(42); // пряма ініціалізація
int size{ 33 }; // уніфікована ініціалізація
```

Найбільш доцільним і однозначним способом є останній — уніфікована ініціалізація. Цей спосіб, на відміну від інших, однаково підходить як для ініціалізації простої змінної, так і для ініціалізації масиву, структури, класу і т.д. І саме цей спосіб ініціалізації ми будемо використовувати надалі.

*Приклад 2.*

```
#include <iostream>
struct Point
{
```

```
int x;  
int y;  
};  
  
int main()  
{  
    int answer{ 42 }; //Проста змінна  
    const float goodTemp{ 36.6 }; //константа  
    int grades[4]{ 3, 5, 4, 4 }; // одновимірний масив  
    int matrix[2][2]{ {1,2}, {3,4} }; // двовимірний  
                                     // масив  
    int* dataPtr{ nullptr }; // покажчик  
    char* str{ new char[14]{ "Hello, world!" } };  
    // c-style рядок, фактично покажчик  
    int& reference{ answer }; // посилання  
    Point point{ 10,-6 }; // екземпляр (об'єкт) структури  
  
    return 0;  
}
```

## 2. Ініціалізація членів класу

У сучасній мові C++ існує декілька способів форматування членів класу. Найгнучкіший і найкращий з них — це вже знайомий за минулим уроком конструктор. Точніше, будь-який з конструкторів в разі, якщо для класу їх визначено декілька. Основне завдання конструктора — забезпечити ініціалізацію екземпляра класу під час його створення. Для її виконання конструктор повинен ініціалізувати всі поля класу. Адже з початкового стану кожного компонента класу складається загальний початковий стан екземпляра класу в цілому. Якщо ж якесь з полів класу залишиться неініціалізованим, це може призвести до негативних і непередбачуваних наслідків у процесі життєвого циклу такого, не повністю ініціалізованого, екземпляра.

**Рекомендація:** *забезпечуйте ініціалізацію всіх полів класу в кожному конструкторі!*

Яким же чином правильно виконати ініціалізацію полів класу в конструкторі? Може здатися, що досить буде способу, як у прикладі нижче:

*Приклад 3.*

```
#include <iostream>

class Point
{
    int x;
    int y;
```

```

public:
    Point() { x = 0; y = 0; } // конструктор
                               // за замовчуванням
    Point(int pX, int pY) { x = pX; y = pY; } //
                               // конструктор з параметрами
};

int main()
{
    Point p1; // використовується конструктор
              // за замовчуванням
    Point p2{ 42, 33 }; // використовується
                       // конструктор з параметрами

    return 0;
}

```

Хоч у кінцевому результаті ми й отримаємо екземпляри класу з передбачуваним початковим станом, в тілі конструктора відбувається присвоювання значень полів, а не їхня ініціалізація! В наведеному вище прикладі, під час створення екземпляра класу, спершу відбувається створення полів **x** і **y** без їхньої справжньої ініціалізації, а вже потім, в тілі конструктора, відбувається присвоювання значень неініціалізованим змінним. Замість однієї дії, цілком достатньої для вирішення завдання, виконується ще одне додаткове «зайве» присвоєння. Більше того, таким способом ми не зможемо ініціювати константи або посилання. Адже їм не можна присвоїти значення, їх можна тільки по-справжньому ініціалізувати. Так само не оптимально буде «форматувати» таким чином вкладені класи — для них спершу буде викликаний конструктор за замовчуванням, а лише потім, в тілі конструктора, від-

будеться присвоєння. Приклад, який ілюструє описану проблему:

#### Приклад 4.

```
#include <iostream>

class Point
{
    // поля описані за допомогою public спеціально!
public:
    int x;
    int y;
    // конструктор за замовчуванням
    Point() { x = 0; y = 0; std::cout <<
        "Point Default constructor\n"; }
    // конструктор з 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 за замовчуванням
    Rectangle()
    {
        leftUpperCorner.x = 10;
        leftUpperCorner.y = 10;
        width = 0;
        height = 0;
    }
};
```



```

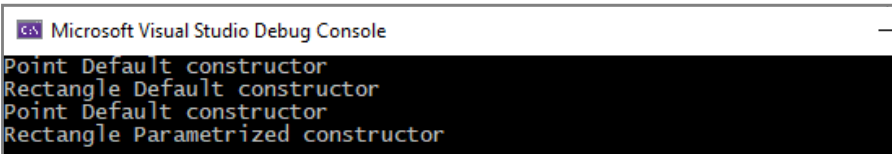
        std::cout << "Rectangle Default constructor\n";
    }

    // конструктор з 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()
{
    // використання конструктора за замовчуванням
    Rectangle rect;
    // використовується конструктор з параметрами
    Rectangle rect1{ 42, 33, /*вершина*/
                    10, /*ширина*/ 5 /*висота*/ };

    return 0;
}

```



```

Microsoft Visual Studio Debug Console
Point Default constructor
Rectangle Default constructor
Point Default constructor
Rectangle Parametrized constructor

```

Рисунок 2

Проаналізуємо результат детальніше: перший і другий рядок на рисунку 2 — результат створення екземпляра `rect`. У процесі його створення на першому етапі створюється екземпляр `Point` викликом його конструктора за замов-

чуванням. Потім в конструкторі за замовчуванням для `Rectangle` відбувається присвоєння початкових значень як для полів `leftUpperCorner` класу `Point`, так і для полів `width` і `height`. При цьому ми спеціально зробили поля `Point` `public`, щоб в цьому прикладі не використовувати сетери. Якщо залишити поля `private`, ми не змогли б без сетерів присвоїти їм значення з конструкторів класу `Rectangle`! Ну і нарешті, останні два рядки виведення в консоль показують процес створення екземпляра `rect1` — спершу створюється екземпляр `Point`, використовуючи конструктор за замовчуванням, потім в тілі конструктора з параметрами `Rectangle` на основі переданих значень полям присвоюються початкові значення. Тобто, в обох випадках спершу створюється екземпляр `Point` з початковим станом за замовчуванням, а лише потім йому присвоюється бажаний початковий стан — дві операції замість однієї коректної ініціалізації!

Розберемо, як досягти оптимальної ініціалізації полів класу і який найкращий для цього спосіб. Отже, бажано формувати поля класу в конструкторі — використовуючи «список ініціалізації членів класу» (інша назва — «список ініціалізаторів членів класу»). Цей список розміщують безпосередньо за сигнатурою (ім'я та список формальних параметрів) конструктора і ставлять між ними двокрапку.

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

Після двокрапки йдуть імена полів класу і їхні ініціалізатори (значення, якими виконується відповідна ініціалі-

зація), вкладені у фігурні дужки `{}` — тобто по суті маємо уніфіковану ініціалізацію! Поля класу з ініціалізаторами у фігурних дужках розділені комами. Важливо: в кінці списку ініціалізації членів класу крапки з комою немає! Далі, після списку ініціалізаторів, розташоване тіло відповідного конструктора, вкладене в свої фігурні дужки. Важливо: навіть якщо тіло конструктора пусте, після списку ініціалізаторів блок фігурних дужок обов'язковий!

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

Якщо список ініціалізаторів вміщується в одному рядку з сигнатурою — оптимально все розмістити в рядок. Занадто довгий список ініціалізаторів слід розмістити на наступному за сигнатурою конструктора рядку, обов'язково з відступом перед двокрапкою (для кращого читання коду). Якщо ж список ініціалізаторів вкрай великий, то, знову ж таки, для кращого читання рекомендується кожен елемент розміщувати з нового рядка з відступом.

Важливо зазначити, що поля класу ініціалізуються не в тому порядку, в якому вони розміщені в списку ініціалізаторів, а в порядку декларування відповідних полів у класі. Саме тому поля класу в списку ініціалізаторів слід розташовувати в тому ж порядку, що і при декларації полів. Тоді не буде плутанини, хто за ким ініціалізується. Так само слід звернути увагу на те, щоб одне поле класу не ініціалізувалося, безпосередньо або в результаті обчислень, значенням іншого поля класу, яке, згідно з вищеописаним порядком, буде ініціалізуватися пізніше.

## Приклад 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;
}

```



Рисунок 3

Як видно з рисунку 3, у випадку з `BadOrder`, `fieldOne` ініціалізований на основі невизначеного на момент ініціалізації `fieldOne`, значенням `fieldTwo`, що призвело до несподіваного результату. Варіант `GoodOrder` позбавлений такого недоліку, тут ініціалізація відбувається на основі значення параметра конструктора і не залежить від порядку ініціалізації полів.

Озброївшись новими знаннями, виправимо недоліки і проблеми, побачені в Прикладі 4:

*Приклад 6.*

```

#include <iostream>

class Point
{
    int x;
    int y;
public:
    // конструктор за замовчуванням

```

```

    Point() : x{ 0 }, y{ 0 }
    {
        std::cout << "Point Default constructor\n";
    }
    // конструктор з 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 за замовчуванням
    Rectangle()
        : leftUpperCorner{ 10, 10 }, width{ 0 },
          height{ 0 }
    { std::cout << "Rectangle Default constructor\n";
    }

    // конструктор з 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()
{
    // використовується конструктор за замовчуванням
    Rectangle rect;
    // використовується конструктор з параметрами

```

```

    Rectangle rect1{ 42, 33, /*вершина*/ 10 /*ширина*/,
                    5 /*висота*/ };

    return 0;
}

```

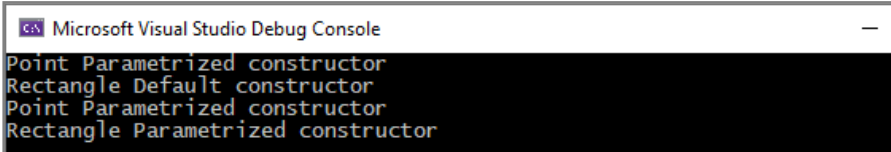


Рисунок 4

Як можемо побачити на рисунку 4, екземпляри **Point**, всередині **Rectangle** створюються одразу з «правильними» початковими значеннями, використовуючи конструктор з параметрами — за одну операцію, а не за дві!

Інший спосіб ініціалізації полів класу — формувати їх безпосередньо під час декларування відповідного поля в класі;

*Приклад 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;
}

```

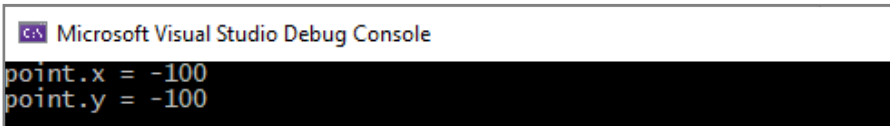


Рисунок 5

Окремо слід розглянути випадок ініціалізації як безпосередньо під час декларації поля, так і засобами конструктора — в такому випадку «перемогу» здобуває конструктор:

### Приклад 8.

```

#include <iostream>

class Point
{
    int x{ -100 };
    int y{ -100 };
public:
    // конструктор за замовчуванням
    Point() : x{ 0 }, y{ 0 }
    { std::cout << "Point Default constructor\n"; }
    // конструктор з параметрами
    Point(int pX, int pY) : x{ pX }, y{ pY }
    {
        std::cout << "Point Parametrized constructor\n";
    }
    int getX() { return x; }
}

```



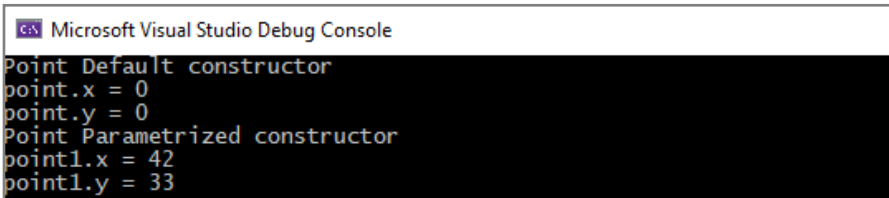
```

    int getY() { return y; }
};

int main()
{
    // поля x та y ініціалізуються конструктором
    // за замовчуванням в 0,0 відповідно
    // ініціалізатори -100,-100 ігноруються!
    Point point;
    std::cout << "point.x = " << point.getX() << '\n';
    std::cout << "point.y = " << point.getY() << '\n';

    // поля x та y ініціалізуються конструктором
    // з параметрами в 42,33 відповідно
    // ініціалізатори -100,-100 ігноруються!
    Point point1{ 42,33 };
    std::cout << "point1.x = " << point1.getX() << '\n';
    std::cout << "point1.y = " << point1.getY() << '\n';
    return 0;
}

```



```

CA 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

Цей спосіб рекомендується застосовувати лише в абсолютно очевидних класах, в яких, можливо, навіть немає явно визначених конструкторів. Не рекомендується так само зловживати і змішувати ініціалізацію під час декларації поля та ініціалізацію поля в конструкторі, щоб уникнути плутанини.

## 3. Делегування конструкторів

Нерідко клас містить відразу декілька конструкторів, при цьому частина коду в них дублюється:

*Приклад 9.*

```
class Person
{
    char* name;
    uint16_t age;
    /*
        uint16_t – unsigned integer 16 bit type
        рекомендована сучасним стандартом нотація
        цілочисельних типів, яка займає передбачувану
        кількість байт на будь-якій архітектурі.
        Аналог unsigned short
    */
    uint32_t socialId;
    // Аналогічно попередньому 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";
}
};

```

Як бачимо, у нас є 4 конструктори для різноманітних ініціалізацій класу. Чимала частина коду в цих конструкторах дублюється. Як уникнути повторень? Можливо, для цього слід викликати один конструктор з іншого? У принципі це можливо, але такий виклик призведе до абсолютно несподіваних наслідків — створиться тимчасовий об'єкт класу на основі викликаного конструктора, при цьому ініціалізації новостворюваного об'єкта не відбудеться. Правильним рішенням буде використовувати механізм делегування конструктора. Суть його полягає

у використанні конструктора у вже знайомому нам списку ініціалізації членів класу:

### Приклад 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 за замовчуванням делегує
        (перенаправляє) свою роботу конструктору
        з параметрами, вказуючи бажані параметри.
    */

    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';
    }
}

};

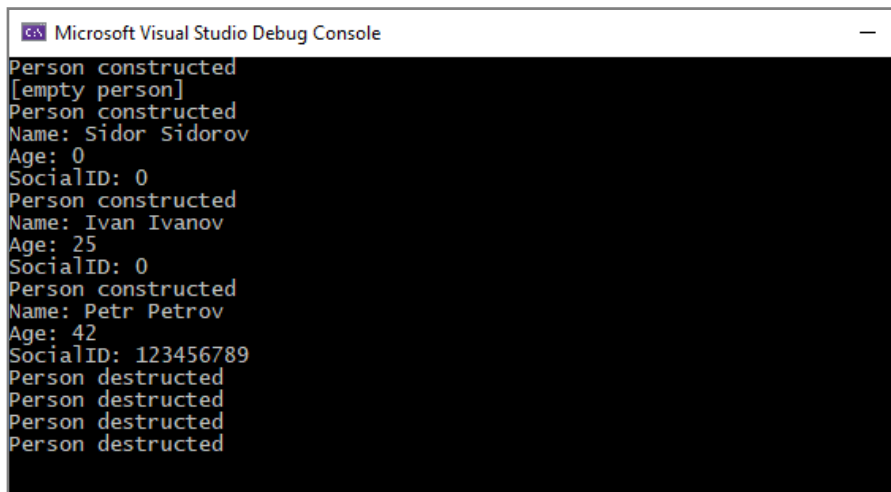
int main()
{
    Person nobody;
    nobody.print();

    Person person1{ "Petro Petrenko" };
    person1.print();

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

    Person person3{ "Pavlo Pavlenko", 42, 123456789 };
    person3.print();
    return 0;
}

```



```

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

Розглянемо уважніше, як все влаштовано. Спершу оголошуємо найспецифічніший, «найдокладніший» конструктор з трьома параметрами — ім'ям, віком і номером соц. страхування.

```

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

```

Зверніть увагу, як ініціалізується поле `name` — ми використовуємо тернарний оператор `i`, залежно від значення параметра `nameP`, або виділяємо динамічну пам'ять для зберігання рядка з ім'ям і зберігаємо покажчик, або записуємо в поле `name` значення `nullptr`.

```

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

```

Решта полів класу `age` і `socialId` ініціалізуються значеннями відповідних параметрів конструктора.

Далі, у тілі конструктора, у разі наявності в полі `name` дійсного, не `nullptr` покажчика, копіюємо рядок-параметр у виділений під нього блок пам'яті. Так само виводимо інформаційне повідомлення про створення екземпляра `Person`. Далі для контрасту описано найменш специфічний і «докладний» конструктор за замовчуванням. Ось тут-то і починає діяти механізм делегування конструктора! У списку ініціалізаторів ми використовуємо конструктор з трьома параметрами, який було визначено вище, передаючи як параметри найзагальніші значення, а саме `nullptr` для `name` і нулі для решти двох полів. Не забуваємо вказати порожні фігурні дужки після списку ініціалізаторів полів.

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

Конструктор за замовчуванням готовий! Жодного рядка дублюючого коду! Аналогічним чином делегуємо роботу найспецифічнішому конструктору з решти двох конструкторів, у кожному з випадків передаючи йому відповідні параметри. Знову ж, без повторень коду.

Для ілюстрації роботи додаємо в клас функцію-член `print` і деструктор. Для тесту створюємо 4 екземпляри `Person`, користуючись по черзі всіма конструкторами. Вивід програми свідчить про правильність зробленого.

Під час використання механізму делегування конструктора слід враховувати наступні обмеження:

- не припустимо форматувати поля класу в списку ініціалізаторів разом з використанням делегування конструктора;
- слід не допускати циклічного виклику делегуючих конструкторів, коли конструктор `A` делегує роботу кон-

структуру **B**, а той в свою чергу делегує ініціалізацію конструктору **A** (**A** і **B** — умовні імена конструкторів, реальні конструктори мають однакові імена!)

Поданий вище приклад — не єдиний спосіб використання делегування конструкторів. У загальному випадку довільний конструктор може делегувати роботу будь-якому іншому, а не лише одному «обраному» конструктору. Головне — стежити за відсутністю циклічних викликів!



## 4. Статичні змінні-члени і статичні функції-члени класу

Під час створення екземпляра класу для нього індивідуально виділяється пам'ять під кожне його поле. Під час створення наступних екземплярів, для кожного з них так само індивідуально виділяється пам'ять під кожне поле. Будь-який екземпляр може незалежно працювати зі своїми полями, не зачіпаючи (і навіть не «знаючи» про їхню наявність!) інші екземпляри класу.

*Приклад 11.*

```
#include <iostream>

// спрощений клас Point з public-полями
class Point
{
public:
    int x;
    int y;
};

int main()
{
    // Виділяється та ініціалізується пам'ять під
    // "персональний" x та y для pointOne
    Point pointOne{ 1,1 };
    // Виділяється та ініціалізується пам'ять під
    // "персональний" x та y для pointTwo
    Point pointTwo{ 2,2 };
    // модифікуємо PointOne поля x та y
    pointOne.x = 4;
```

```

pointOne.y = 6;

// перевіряємо, що значення x та y для pointOne
// змінилися
std::cout << "pointOne: x = " << pointOne.x
           << " y = " << pointOne.y << '\n';
// перевіряємо, що значення x та y для pointTwo
// не змінені
std::cout << "pointTwo: x = " << pointTwo.x
           << " y = " << pointTwo.y << '\n';
return 0;
}

```

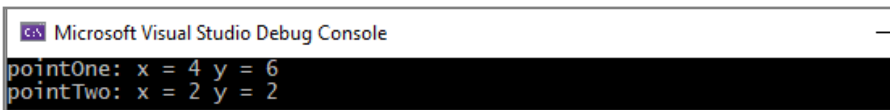


Рисунок 8

Таким чином, стан одного екземпляра абсолютно не пов'язаний зі станом інших. Однак, для вирішення окремих завдань необхідно мати деякий роздільний стан. Кажучи простіше — мати поле класу, значення якого було б однаковим і загальним для всіх екземплярів. Ідентичності досягти легко — достатньо визначити, наприклад, `const int maxX {1500};` в класі `Point` і тепер у всіх екземплярів `Point` значення поля `maxX` однакове і незмінне, тобто константне. Однак додаткова властивість незмінності не завжди доречна. Також слід звернути увагу на те, що незважаючи на ідентичність значення поля `maxX`, пам'ять під нього буде виділятися для кожного екземпляра класу `Point` персонально, тобто спільності поля ми так і не досягли. Для коректного вирішення такого роду завдань,

необхідно використовувати static-поля класу. Такі поля можуть бути довільного допустимого типу і не обов'язково повинні бути константними. Специфіка static-полів полягає в тому, що вони не є частиною екземпляра класу, вони не належать кожному екземпляру персонально. Навпаки — такі поля належать класу в цілому і доступні з довільного екземпляра!

Статичні поля створюються в момент старту програми й існують до її завершення. Робота зі static-полями має деякі особливості. Почнемо з того, що для ініціалізації таких полів ми не можемо використовувати конструктори. Ми можемо ініціалізувати константні цілочисельні поля або enum-поля під час їхньої декларації в класі. Для ініціалізації static-полів довільного типу, які не обов'язково є константними, слід зазначити специфіку синтаксису — ініціалізація виконується в глобальному контексті, поза класом, поза будь-якою функцією, із зазначенням типу поля, його повного імені та ініціалізатора. Ключове слово **static** вказувати не потрібно! Дія модифікаторів доступу **private** і **protected** не поширюється на ініціалізацію статичного поля класу, але на наступний доступ — так!

*Приклад 12.*

```
#include <iostream>

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

```

int Demo::common{ 0 };

int main()
{
    // створюємо екземпляр та ініціалізуємо
    // персональне поле personal для d1
    Demo d1{ 1 };
    // створюємо екземпляр та ініціалізуємо
    // персональне поле personal для d2
    Demo d2{ 2 };

    // присвоюємо значення спільному полю common
    d1.common = 42;
    // перевіряємо значення загального поля в d2
    std::cout << "d2.common = " << d2.common << '\n';
    // 42 на екрані
    return 0;
}

```

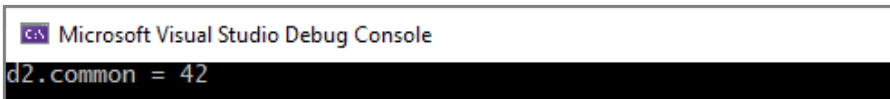


Рисунок 9

Як уже було сказано, статичні поля не є частиною екземпляра класу, чому ж тоді ми звертаємося до них як до звичайних полів? Насправді, це не єдиний спосіб доступу до static-полів і присутній він для зручності і одноманітності. Іншим способом отримання доступу слугує вказання імені поля з уточненням, через оператор `::` імені класу — `Demo :: common` в нашому прикладі. Ключовим моментом, важливим для розуміння, є те, що пам'ять під поле `Demo :: common` виділяється навіть якщо не створено жодного екземпляра класу `Demo`!

## Приклад 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;
}

```

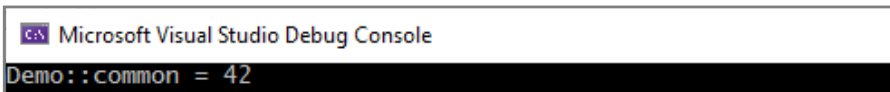


Рисунок 10

Зверніть увагу, ми не створили жодного екземпляра класу **Demo**, але при цьому змогли отримати значення поля **common**.

Розглянемо використання статичного поля класу на прикладі демонстраційного класу **NumberStorage**. Його завдання — зберігання в динамічній пам'яті деякого масиву з цілих чисел. Єдиний, явно визначений конструктор, приймає як параметр кількість необхідних елементів, виділяє під них блок у динамічній пам'яті і заповнює цей блок випадковими числами. Також конструктор виводить на екран

діагностичне повідомлення про те, скільки додаткової пам'яті виділено і скільки всього зайнято динамічної пам'яті всіма екземплярами класу. Додатково, є функція-член для виведення на екран всього масиву. У деструкторі класу звільняється пам'ять і виводиться діагностика, аналогічна конструктору, з тією лише різницею, що відображається звільнена пам'ять і загальна кількість зайнятої всіма об'єктами пам'яті. Розглянемо код прикладу:

### Приклад 14.

```
#include <iostream>
#include <ctime> // для функції 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()
{
    // Задаємо номер послідовності випадкових чисел
    // на базі поточного часу.
    srand(time(nullptr));

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

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

```

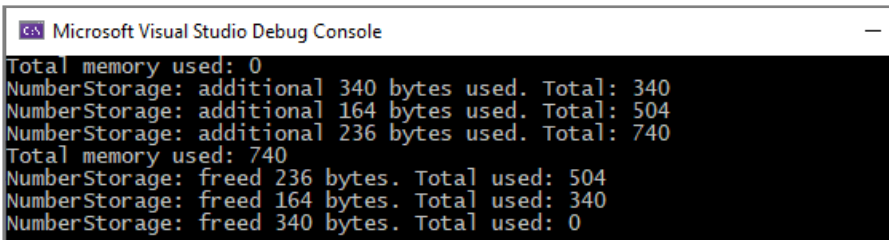
```

rand() % 101, rand() % 101 };

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

    return 0;
}

```



The screenshot shows the Microsoft Visual Studio Debug Console with the following output:

```

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

Ми навмисне не виводимо елементи масивів на екран, щоб не «засмічувати» вивід програми несуттєвою в цьому випадку інформацією. Приклад показує ще одне нове поняття — статичні функції-члени класу. Якщо деяка функція-член в класі звертається лише до статичних членів цього класу, то вона так само може стати статичною і її так само можна буде викликати в контексті класу, а не екземпляра класу! Саме такою функцією-членом і є `getUsedMemory()` в нашому прикладі. Виклик статичної функції-члена, аналогічний доступу до статичного поля класу — вказується ім'я класу, оператор уточнення імені `::` і власне ім'я статичної функції-члена разом з фактичними параметрами в круглих дужках (якщо вони є). За бажання або необхідності статичну функцію-член можна так само викликати і через екземпляр класу.



Для того, щоб оголосити функцію-член класу статичною, необхідно вказати ключове слово **static** перед її декларацією в класі. Не всяку функцію-член можна зробити статичною. *Функція-член, яка крім статичних полів вимагає для своєї роботи також звичайних, не статичних полів, не може бути статичною!* Це цілком логічно, оскільки без екземпляра класу в нас немає звичайних полів, під них не виділена пам'ять і, відповідно, до них немає доступу. При цьому нестатична, звичайна функція-член класу може без обмежень маніпулювати як нестатичними, так і статичними полями класу.

Статичні поля і функції-члени класу можна так само використовувати для підрахунку створених/активних екземплярів класу, або ж для присвоєння кожному екземпляру унікального порядкового номера. У загальному випадку, якщо якісь дані не прив'язані до конкретного об'єкта, а властиві всім об'єктам в цілому, або ж відображають загальні характеристики/стани всіх об'єктів, а не якогось конкретно — то такі поля слід робити статичними. Хорошим прикладом можуть бути константи, що використовуються класом в цілому або таблиці (масиви) з деяким набором константних значень, які використовуються в класі. Те ж саме стосується і функцій-членів — якщо якась функція-член виконує роботу не для конкретного екземпляра класу, а для класу в цілому, то її слід робити статичною. Хоча технічно ніщо не заважає створювати клас, який містить лише статичні поля і функції-члени, але робити цього не слід! Неможливо створити екземпляри такого класу і практичний сенс його використання досить сумнівний.

## 5. Показчик this

Клас у C++ — це креслення, на основі якого створюються екземпляри-об'єкти. При цьому необхідно розуміти, як відбувається виділення пам'яті під поля і під функції-члени для кожного конкретного екземпляра класу. Отже, розглянемо простий приклад:

*Приклад 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()
{
    /*
        Створюємо й ініціалізуємо екземпляр класу
        Date. Виділяється пам'ять для екземпляра date1
        і його персональних полів date1.day,
        date1.month, date1.year
    */
    Date date1{ 1,1,2020 };
```

```

/*
    Створюємо й ініціалізуємо екземпляр класу Date.
    Виділяється пам'ять для екземпляра date2
    і його персональних полів date2.day,
    date2.month, date2.year
*/
Date date2{ 24,07,1976 };

/*
    Викликаємо функцію-член print.
    Код функції-члена загальний для всіх
    екземплярів Date.
*/
date1.print();
date2.print();

return 0;
}

```

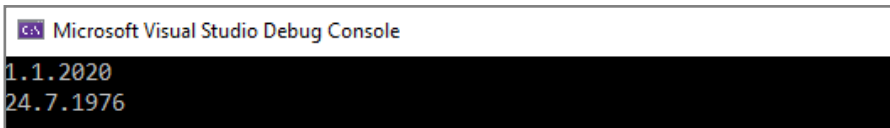


Рисунок 12

Як ми вже знаємо, створення екземпляра класу призводить до виділення пам'яті під всі поля класу індивідуально для кожного екземпляра. Що ж відбувається з функціями членами? Можна було б припустити, що і функції-члени так само індивідуальні для кожного екземпляра. Однак це не так. Насправді, всі функції-члени в межах класу займають пам'ять відповідно до їхнього розміру один раз. При створенні декількох екземплярів не відбувається додаткового виділення пам'яті для

функцій-членів. Іншими словами, дані (поля) індивідуальні, а код для їхнього опрацювання (функції-члени) — спільні для екземплярів класу. Яким же чином один і той же метод виконує роботу з різними екземплярами класу? Повернемося до коду з Прикладу 15. Функція-член `print()` не приймає жодного параметра і не створює всередині жодної локальної змінної. Чим же тоді є ідентифікатори `day`, `month`, `year` всередині цього методу? Полями класу! Так, але як один-єдиний метод «дізнається» з яким саме екземпляром класу йому працювати, з якими конкретно полями він має справу і як отримати доступ до блоку пам'яті, виділеного під певний екземпляр класу?

Насправді, тут немає нічого надприродного, вся справа в тому, що кожній нестатичній функції-члену в класі першим, неявним параметром передається показчик на екземпляр класу, через який була викликана функція-член. Цей неявний перший параметр є константним показчиком на екземпляр класу і називається `this`.

```
Date* const this /* this для функцій-членів Date*/
```

Зверніть увагу, явно вказувати `this` як перший параметр непотрібно; до того ж, якщо зробити так, це призведе до помилки! Незважаючи на свою неявність на етапі оголошення, показчик `this` можна використовувати всередині нестатичних функцій-членів як явно, так і неявно. Саме наявність такого показчика і пояснює те, як функція-член зв'язується з даними конкретного екземпляра. Всередині функції-члена звернення до полів конкретного екземпляра можна виконувати явно, через показчик `this` і оператор «->»:

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

Цей спосіб доступу до полів не є обов'язковим, але саме доступ через показчик `this` на екземпляр класу розуміється, коли ми просто вказуємо ім'я поля всередині нестатичної функції-члена класу. Наявність неявного `this` і пояснює всю «магію» зв'язку загального коду з індивідуальними для екземплярів класу даними.

Чи слід завжди використовувати `this` для доступу до полів і функцій-членів? Найчастіше ні. Проте, буває необхідно явно вказувати `this` в разі, коли це усуває неоднозначність, наприклад в разі використання ідентичного з полем класу імені параметра функції-члена або локальної змінної всередині функції-члена.

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

Тут використання `this` усуває неоднозначність, пов'язану з ім'ям параметра й ім'ям поля. У той же час слід уникати появи таких конфліктів імен та неоднозначності, наприклад додаючи суфікс `P` або префікс `the` до імен параметрів, які могли б потенційно конфліктувати з іменами, які використовуються для компонентів класу.

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

Ще одним прикладом застосування **this** може слугувати створення функцій-членів, які підтримують виклик за ланцюжком. Порівняємо дві реалізації сетерів у класі **Date**

### Приклад 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;
}
```

## Приклад 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;
}

```

 Microsoft Visual Studio Debug Console

29.2.2004

Рисунок 13

Отже, в Прикладі 16 гетери приймають один відповідний параметр (день, місяць, рік) присвоюють зна-

чення параметра полю екземпляра класу і завершують свою роботу, не повертаючи значення (тип повернення значення функції-члена — `void`). Для задання всіх трьох значень необхідно послідовно викликати відповідні сетери. Виявляється, це завдання можна вирішити більш елегантно. У Прикладі 17 сетери виконують ту ж роботу, але на завершення, додатково повертають розіменований показчик `this` (тип значення функції-члена — `Date&`). Таке доповнення дозволяє викликати сетери за ланцюжком: виклик першого сетера, `setDay`, відбувається на екземплярі `date1` потім `setDay` повертає посилання на екземпляр `date1` і, як наслідок, можна використати це посилання, аби викликати сетер `setMonth`, і потім аналогічно викликати `setYear`. Принципової різниці в результаті роботи сетерів немає, але сам ланцюжок викликів став наочнішим, а код — елегантнішим. Забігаючи вперед, подібним чином реалізовано вивід на екран за ланцюжком викликів оператора `<<`. У загальному випадку, якщо функція-член класу повертає `void`, слід розглянути доцільність заміни `void` на `ClassName&`, де `ClassName` — ім'я класу, зробивши можливим виклик наступних функцій-членів за ланцюжком.

І на завершення розглянемо ще один спосіб застосування `this`. Ми додамо візуалізацію порядку запуску конструкторів і деструкторів для екземплярів класу `Date`.

### Приклад 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'; }

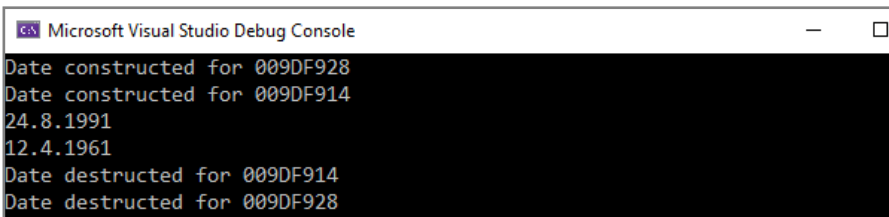
    void print() { std::cout << day << '.'
      << month << '.'
      << year << '\n'; }

    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;
}

```



```

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

У Прикладі 18 додано вивід на екран значення `this` з конструктора і деструктора. Тепер є можливість простежити порядок створення-знищення екземплярів. Спершу створюється екземпляр `date1 {24,8,1991}`, йому відповідає адреса `009DF928`, потім створюється екземпляр `date2 {12,4,1961}` з адресою `009DF914` відповідно. За повідомленнями з деструктора видно, що знищення екземплярів класу відбувається строго в зворотному порядку, спершу для `009DF914` і потім — для `009DF928`. Конкретне значення адрес буде відрізнятися від наведених у висновку, більше того, значення найімовірніше будуть змінюватися від одного запуску програми до іншого, насправді важливі не абсолютні значення адрес, а їхня «парність». Вивід значення `this` на екран з різних функцій-членів може істотно спростити відлагодження програми.

## 6. Конструктор копіювання

У попередніх розділах уроку описувалися різні способи ініціалізації об'єктів, розглянемо ще один — копіюючу ініціалізацію. Для цього нам знадобиться невеликий клас — базова реалізація сутності простий дріб. У прикладі нижче буде реалізований лише найнеобхідніший мінімум функціональності, конструктор з параметрами, конструктор за замовчуванням, деструктор і функція-член виведення дробу на екран. Для візуалізації роботи конструкторів і деструкторів вони виводять налагоджувальну інформацію на екран.

*Приклад 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 << ")";
    }
};

int main()
{
    /*
        Створюємо й ініціалізуємо значеннями
        чисельника і знаменника екземпляр Fraction - a
    */
    Fraction a{ 2,3 };
    /*
        Створюємо й ініціалізуємо значеннями
        чисельника і знаменника екземпляр Fraction - b
    */
    Fraction b{ a };

    std::cout << "a = ";
    a.print();
    std::cout << "\nb = ";
    b.print();
    std::cout << '\n';
    return 0;
}

```

```

Microsoft Visual Studio Debug Console
Fraction constructed for 00EFFA30
a = (2 / 3)
b = (2 / 3)
Fraction destructed for 00EFFA20
Fraction destructed for 00EFFA30

```

Рисунок 14

Рисунок 14 демонструє, що спершу викликається конструктор для екземпляра `a{2, 3}`, йому відповідає адреса `00EFFA30`, потім, згідно з кодом у Прикладі 19, створюється

екземпляр `b{a}`, але ми не бачимо відповідного повідомлення про роботу конструктора!

Потім на екран виводяться `a`, `b` і ми бачимо роботу деструктора для «нествореного» екземпляра `b` (йому відповідає адреса `00EFA20`) і на завершення — роботу деструктора для екземпляра `a` з адресою `00EFA30`. Побачене може здатися трохи дивним.

Розглянемо причини такої поведінки. У класі `Fraction` явно визначені два конструктори, один — приймає два цілочисельні параметри, і один, конструктор за замовчуванням, зовсім без параметрів, який делегує ініціалізацію попередньому конструктору з двома параметрами. Екземпляр `a` ми ініціалізуємо за допомогою конструктора з двома цілочисельними параметрами, а що ж з екземпляром `b`?

Хіба ми створюємо його, використовуючи конструктор за замовчуванням? Або явно задаючи значення чисельника і знаменника? Ні, вся справа в тому, що ініціалізація екземпляра `b` відбувається згідно з поточним значенням екземпляра `a`. І в цьому випадку не задіяний жоден з явно визначених нами конструкторів! Саме тому немає виводу налагодження про створення екземпляра, йому немає звідки взятися.

Як же тоді ініціалізується екземпляр `b`, адже в кінцевому результаті, згідно з Рисунком 14, там (2/3)? Вся справа в тому, що ініціалізація відбувається автоматично згенерованим компілятором конструктором копіювання. Без знання внутрішньої суті класу компілятор автоматично створює конструктор копіювання, виконує поверхнєве або побітове копіювання одного об'єкта, ініціалізатора, в інший, щойно створений, об'єкт, що вимагає ініціалізації.

**Конструктор копіювання** — це спеціальний конструктор, який використовується для копіювання існуючого екземпляра класу в новий екземпляр того ж класу. Якщо в класі явно не визначено конструктор копіювання, то аналогічно конструктору за замовчуванням, компілятор автоматично згенерує public-конструктор, який виконає почленну ініціалізацію полів нового екземпляра значеннями відповідних полів наявного екземпляра класу. Тобто, в прикладі вище, `b.numerator` ініціалізується значенням `a.numerator` і відповідно `b.denominator` — `a.denominator`.

Конструктор копіювання можна визначити явно, для цього необхідно використовувати спеціальну сигнатуру:

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

Де `ClassName` — ім'я класу, для якого визначається конструктор. По суті, конструктор копіювання — це конструктор, який приймає екземпляр того ж класу за константним посиланням. Чому саме так — ми розповімо пізніше. Приклад сигнатури для `Fraction`:

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

Давайте явно визначимо конструктор копіювання для класу `Fraction`:

*Приклад 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 << ")";
    }
};

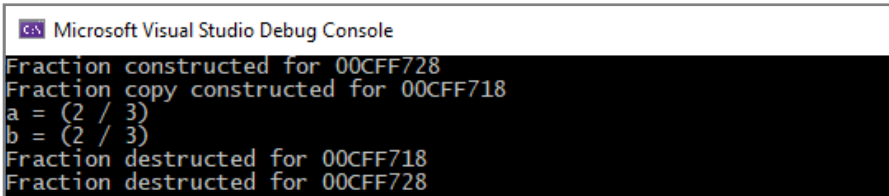
int main()
{
    /*
        Створюємо й ініціалізуємо значеннями чисельника
        і знаменника екземпляр Fraction – a
    */
    Fraction a{ 2, 3 };
    /*
        Створюємо й ініціалізуємо значеннями чисельника
        і знаменника екземпляр Fraction – b
    */
    Fraction b{ a };

```

```

std::cout << "a = ";
a.print();
std::cout << "\nb = ";
b.print();
std::cout << '\n';
return 0;
}

```



```

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 15

У Прикладі 20, у явно визначеному конструкторі копіювання виконується почленна ініціалізація полів і виведення налагоджувальної інформації про створення нового екземпляра копіюванням. Тепер, згідно з рисунком 15, більше немає «нестворених» екземплярів класу [Fraction](#)! Просто екземпляр `a` ініціалізується конструктором з двома цілочисельними параметрами, а екземпляр `b` — конструктором копіювання, які приймають за константним посиленням екземпляр [Fraction](#). Фактично, явно визначений конструктор для [Fraction](#) виконує ті ж дії, що і згенерований компілятором конструктор копіювання, ми тільки додали виведення налагоджувальної інформації, яка інформує нас про те, що виконався саме конструктор копіювання. Якщо нас не цікавить виведення налагоджувальної інформації, то для класу [Fraction](#) немає необхідності в явно визначеному конструкторі копіювання. Але так буває далеко не з усіма



класами! Трохи пізніше ми розглянемо приклад, де без явного визначення конструктора копіювання не обійтися.

У разі, якщо ми ініціалізуємо екземпляр класу тимчасовим, анонімним екземпляром, компілятор може не викликати конструктор копіювання, а виконати оптимізацію і, не створюючи тимчасовий об'єкт, безпосередньо ініціалізувати екземпляр класу ініціалізаторами тимчасового об'єкта. У Прикладі 20 змінимо функцію `main`:

*Приклад 21.*

```
int main()
{
    /*
        Створюємо й ініціалізуємо тимчасовим, анонімним
        екземпляром Fraction – a.
        Конструктор копіювання не викликається,
        тимчасовий об'єкт не створюється!
        Компілятор безпосередньо проініціалізує
        екземпляр a значеннями 4 і 6.
    */
    Fraction a{ Fraction{4,6} };

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

    return 0;
}
```

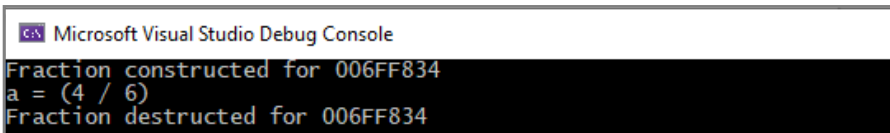


Рисунок 16

Така оптимізація зайвого створення об'єкта з його подальшим копіюванням називається елізією. При цьому навіть якщо в конструкторі копіювання, крім власне копіювання, виконувалася якась додаткова робота, наприклад вивід на екран, тіло конструктора копіювання не виконається!

Розглянемо сигнатуру конструктора копіювання уважніше. Чому так важливо, щоб єдиним параметром було константне посилання на екземпляр класу? Що буде в разі передачі екземпляра за значенням? Наприклад так:

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

Якби такий код можна було б скомпілювати і запустити, то при спробі скористатися копіюючою ініціалізацією, необхідно було б створити копію об'єкта-ініціалізатора! Для цього треба було б викликати конструктор копіювання! Сталася б нескінченна рекурсія! На щастя, в сучасних компіляторах такий код навіть не компілюється, будуть видані відповідні помилки на етапі компіляції. А чому потрібен модифікатор `const`? Передаючи параметр за посиланням, можливо модифікувати його всередині конструктора копіювання, що технічно можливо, але за логікою абсолютно неприйнятно — під час копіювання зовсім недоцільно модифікувати оригінал. Для того, щоб запобігти навіть потенційній можливості модифікації, навіть помилково, використовується саме константне посилання.

Чи завжди підходить автоматично згенерований конструктор копіювання або явне визначення конструктора копіювання, у якому виконується лише поверхнєве копі-

ювання? Для того, щоб відповісти на це запитання, розглянемо приклад спрощеного класу динамічного масиву. В цьому прикладі ми створимо конструктор копіювання, який виконує поверхнєве копіювання. Саме такий конструктор автоматично генерується компілятором, коли програміст не надає свою, відповідну класу, версію конструктора копіювання.

### Приклад 22.

```
#include <iostream>
/*
    Увага! Програма завершиться аварійно!
    Для цього прикладу це 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;
}

```

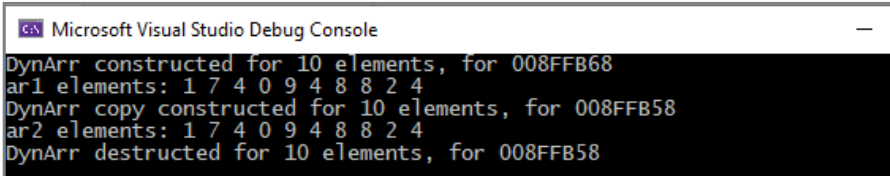


Рисунок 17

Розглянемо, що відбувається в Прикладі 22. Спершу створюємо екземпляр `ar1` класу `DynArray` з 10 цілих чисел і заповнюємо його випадковими числами. Виводимо вміст `ar1` на екран. Потім створюємо екземпляр `ar2` як копію `ar1` і так само виводимо його на екран. Для копіювання використовуємо явно визначений конструктор копіювання, який виконує почленну ініціалізацію полів нового екземпляра на основі значень наявного екземпляра класу `DynArray`, тобто виконуємо поверхневе копіювання.

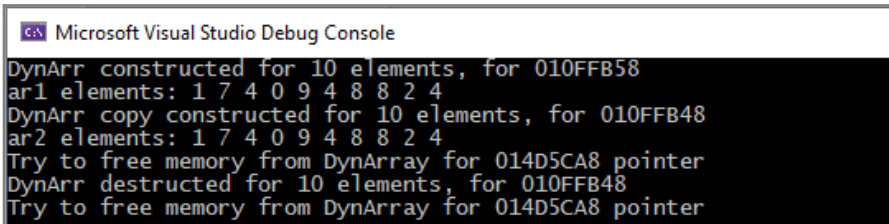
Програма завершується викликом деструкторів у зворотному порядку створення екземплярів класу, спершу викликається деструктор для `ar2`, а потім — програма аварійно завершується в процесі роботи деструктора для `ar1`!

Рисунок 17 показує те, що встигло відобразитися на екрані. Чому так відбувається? Трохи змінимо деструктор, щоб спробувати в цьому розібратися.

```

~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';
}

```



```

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 18

Отже, деструктор для **ar2** звільняє пам'ять, яку займає динамічно виділений масив за покажчиком **014D5CA8**, пізніше деструктор для **ar1** звільняє пам'ять за тим же покажчиком! Все вірно, адже в нашому конструкторі копіювання ми всього лише скопіювали покажчик! Ми не створили нову копію масиву, не скопіювали туди значення з наявного масиву! Як і було сказано, була створена поверхнева копія, яка абсолютно не підходить для цього класу!

Саме для подібного роду класів, де виконується виділення динамічної пам'яті, необхідна правильна, не автоматично створена компілятором, реалізація конструктора копіювання. Така реалізація повинна на основі наявного об'єкта класу створити його глибоку копію, з виділенням

нових блоків динамічної пам'яті і копіюванням туди значень з вихідних блоків динамічної пам'яті. Розглянемо приклад такого конструктора:

### Приклад 23.

```
#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 списку ініціалізаторів полів класу вище
         виділяємо новий блок динамічної пам'яті
         того ж розміру, що і в скопійованому
         екземплярі класу DynArray. Наступним
         циклом копіюємо елементи із оригінального
         блоку пам'яті в знову виділений.
        */
        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;
    }
}

```



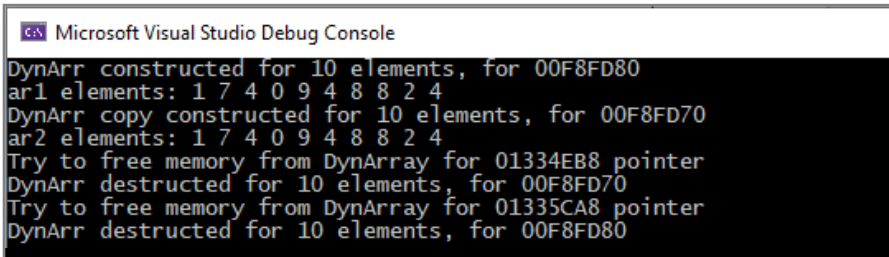
```

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

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

    return 0;
}

```



```

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 19

Тепер програма працює як слід, без збоїв! Конструктор копіювання створює глибоку копію екземпляра класу `ar1`. Під час запуску деструкторів кожен звільняє блок пам'яті за своїм покажчиком, про що свідчить рисунок 19.

Приклад реалізації конструктора копіювання дозволяє побачити, що в загальному випадку для подібного роду класів операція копіювання є вкрай витратною! Необхідно виділити новий блок або блоки динамічної пам'яті і зробити копіювання інформації з вихідного блоку або блоків. У наведеному прикладі мова йшла про блок на 10

цілих чисел, весь процес копіювання відбувався практично миттєво, але в загальному випадку блоки пам'яті можуть бути істотно більшого розміру! З вищесказаного випливає, що слід уникати копіювання об'єктів, де це можливо і де для цього є прийнятні альтернативи.

**Важливе зауваження!** Під час передачі екземпляра класу як параметра функції за значенням, відбувається створення нового екземпляра класу, використовуючи конструктор копіювання.

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

Аналогічно, у разі повернення екземпляра класу з функції за значенням, так само створюється копія екземпляра класу.

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

Далі, в рамках нашого курсу будуть розглянуті способи, як уникнути зайвого копіювання де це можливо, згідно з вимогами реальних завдань, але поки слід запам'ятати про «затратність» операції копіювання. Де можливо уникати копіювання, передаючи екземпляри класу за посиланнями, де доречно — константою. Так само, за посиланнями, де це в принципі можливо, повертати екземпляри класу з функцій.

## 7. Домашнє завдання

### 1. Створити клас «Дріб» для подання простого дробу.

*Поля:*

- чисельник,
- знаменник.

*Функції-члени:*

- конструктор приймає чисельник і знаменник.

У конструкторі використовувати список ініціалізаторів полів класу.

- ▷ конструктор за замовчуванням, реалізувати через делегування конструктору з параметрами чисельник і знаменник;
- ▷ виведення на екран дробу;
- ▷ додавання / віднімання / множення простого дробу з простим дробом;
- ▷ додавання / віднімання / множення простого дробу з цілим числом.

В арифметичних операціях передбачити можливість виклику операцій за ланцюжком використовуючи покажчик [this](#).

Передбачити скорочення дробу. Скорочення рекомендується виконувати в конструкторі.

### 2. Створити клас Людина.

*Поля:*

- ідентифікаційний номер;
- прізвище;

- ім'я;
- по батькові (для прізвища, імені та по батькові пам'ять виділяти динамічно!);
- дата народження (рекомендується створити додатковий клас Дата (день, місяць, рік).

Функції-члени:

- конструктор з параметрами ідентифікаційний номер, прізвище, ім'я, по батькові, дата народження. У конструкторі використовувати список ініціалізаторів полів класу;
- конструктор за замовчуванням. У конструкторі використовувати делегування конструктора;
- конструктор копіювання;
- деструктор;
- функцію-член для підрахунку створених екземплярів класу «Людина»;
- сетери / гетери для відповідних полів класу;
- виведення на екран інформації про людину.

### 3. Створити клас Рядок

Поля:

- довжина рядка без урахування нуль-термінатора;
- покажчик на блок пам'яті, де зберігається рядок;
- пам'ять виділяти динамічно!

Функції-члени:

- конструктор з параметром-рядком (`const char*`);
- конструктор з параметром довжина рядка;
- конструктор копіювання;
- деструктор;
- виведення рядка на екран;

- сетер, що приймає як параметр рядок (`const char*`). За нестачі вже виділеного блоку динамічної пам'яті для копіювання в нього рядків-параметрів — зробити коректне перевиділення пам'яті.

**Примітка:** *перелік полів і функцій-членів у завданнях 1-3 є рекомендованим, а не остаточним. За необхідності можливе додавання необхідних або бажаних полів і функцій-членів.*

#### 4. Створіть програму, що імітує багатоквартирний будинок.

Необхідні класи Людина, Квартира, Будинок. Клас Квартира містить динамічний масив об'єктів класу Людина. Клас Будинок містить масив об'єктів класу Квартира. Кожен з класів містить змінні-члени і функції-члени, які необхідні для предметної області класу. Звертаємо вашу увагу, що пам'ять під рядкові значення виділяється динамічно. Не забувайте забезпечити класи різними конструкторами (конструктор копіювання обов'язковий), деструкторами. Клас Людина слід використовувати із Завдання 2.

**STEP IT Academy, [www.itstep.org](http://www.itstep.org)**

Усі права на захищені авторським правом фото, аудіо та відеотвори, фрагменти яких використані в матеріалі, належать їхнім законним власникам. Фрагменти творів використовуються з ілюстративною метою в обсязі, виправданому поставленим завданням, в межах навчального процесу і в навчальних цілях. Відповідно до ст. 21 і 23 Закону України «Про авторське право й суміжні права». Обсяг і спосіб цитованих творів відповідає прийнятим нормам, не завдає шкоди нормальному використанню об'єктів авторського права і не обмежує законні інтереси автора та правовласників. Цитовані фрагменти творів на момент використання не можуть бути замінені альтернативними, що незахищені авторським правом аналогами, і як такі відповідають критеріям сумлінного і чесного використання.

Усі права захищені. Повне або часткове використання матеріалів заборонено. Узгодження використання творів або їхніх фрагментів проводиться з авторами і правовласниками. Узгоджене використання матеріалів можливе лише за умов згадування джерела.

Відповідальність за несанкціоноване копіювання і комерційне використання матеріалів визначається чинним законодавством України.