

# Object-Oriented Programming Using

# C++



# Lesson 2

Initializers.  
Static Class Member  
Variables and Static  
Class Member Functions.  
"This" Pointer and Copy  
Constructor.

## Contents

1. Uniform Object Initialization.....	3
2. Class Members Initialization.....	6
3. Delegating Constructors.....	18
4. Static Class Member Variables and Static Class Member Functions.....	25
5. "This" Pointer .....	34
6. Copy Constructor.....	43
7. Homework.....	59

Lesson materials are attached to this PDF file. In order to get access to the materials, open the lesson in Adobe Acrobat Reader.

# 1. Uniform Object Initialization

By adhering to a predictable and working plan, the important thing is any variable must be initialized. It does not matter what type the variable is (`int`, `float`, `char`, `bool`, pointer, structure, class, etc.) — do not leave it uninitialized. After all, such a variable is not fully complete. It does not have an initial state, and accordingly, it is impossible to get its current value. In most cases, attempting to use uninitialized variable results in a compile-time error, but this happens not so often. Different versions of integrated development environments (IDEs) and/or compilers may not generate compile-time errors. You should not rely on chance in such an important matter!

Let's look at a small example:

## *Example 1*

```
#include <iostream>

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

    return 0;
}
```

## *Output 1*



Figure 1

In Visual Studio 2019, the snippet compiles and launches successfully. So, what can we see as a result? What are `numbers[0]` equal to? Nothing correct and usable! After all, we added 2 to `numbers[0]` — the current undefined value, and assigned the result. This is just one of many cases where it is impossible to detect an uninitialized variable during compilation fully. As a result, a completely incorrect program operation is possible. However, it should be noted that at the compilation stage, a warning (but it is not an error!) occurs, which is easy to ignore.

There are several methods to initialize a variable in C++:

```
int number = 0;    // copy initialization
int value(42);     // direct initialization
int size{ 33 };   // uniform initialization
```

The most preferred and unambiguous way is uniform initialization. This method, unlike others, is uniformly suitable both for initializing a simple variable and for initializing an array, structure, class, etc. And this particular method of initialization we are going to use further.

### *Example 2*

```
#include <iostream>

struct Point
{
    int x;
    int y;
};

int main()
{
```

```
int answer{ 42 }; // variable
const float goodTemp{ 36.6 }; // constant
int grades[4]{ 3, 5, 4, 4 }; // one-dimensional array
int matrix[2][2]{ {1,2}, {3,4} }; // two-dimensional
                                // array
int*dataPtr{nullptr }; // pointer
char*str{new char[14]{"Hello world!"} }; // c-style
                                           // string, a pointer
int& reference{ answer }; // link
Point point{ 10,-6 }; // instance (object)
                       // of the structure
return 0;
}
```

## 2. Class Members Initialization

There are several methods to initialize class members in C++. We are already familiar with the most flexible and preferred one from the previous lesson: a constructor. Any of the constructors, if only several of them are defined for the class. The primary constructor issue is to initialize the class object while creating. To execute this issue, the constructor must initialize all the class fields. After all, the general initial state of the class object is compiled from the initial state of each class component as a whole. If any of the class fields are uninitialized, this can lead to negative and unpredictable consequences during the life cycle of such an incompletely initialized instance.

- **Recommendation:** *initialize all class fields in every constructor!*

What is the right way to initialize the class fields in the constructor? It may seem that a method from the example below will suffice:

*Example 3*

```
#include <iostream>

class Point
{
    int x;
    int y;
public:
    Point() { x = 0; y = 0; } // default constructor
```

```

    Point(int pX, int pY) { x = pX; y = pY; }
                          // constructor with the parameters
};

int main()
{
    Point p1; // the constructor is used by default
    Point p2{ 42,33 }; // constructor with the parameters
    return 0;
}

```

Although, we will get. As a result, the class objects with a predictable initial state. And what happens in the constructor body is assignment values to the fields, not their initialization! In the example above, the `x` and `y` fields are created first without their initialization when creating a class object. Only then are values assigned to uninitialized variables in the constructor body. And instead of one action, which is sufficient to solve the problem, one more additional “extra” assignment is performed. Moreover, we will not be able to initialize constants or references. After all, they cannot be assigned a value, and they can only be initialized instead. It will also not be optimal to “initialize” nested classes in this way. A default constructor will be called for them at first, and only then will the assignment occur in a constructor body. The described issue is shown in the example below:

#### *Example 4*

```

#include <iostream>

class Point
{

```

```

    // fields are defined with public on purpose!
public:
    int x;
    int y;
    // default constructor
    Point() { x = 0; y = 0; std::cout
            << "Point Default constructor\n"; }
    // constructor with the parameters
    Point(int pX, int pY)
    {
        x = pX;
        y = pY;
        std::cout << "Point Parametrized constructor\n";
    }
};

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

public:
    // default constructor
    Rectangle()
    {
        leftUpperCorner.x = 10;
        leftUpperCorner.y = 10;
        width = 0;
        height = 0;
        std::cout << "Rectangle Default constructor\n";
    }

    // constructor with the parameters
    Rectangle(int x, int y, int widthP, int heightP)
    {
        leftUpperCorner.x = x;

```



```

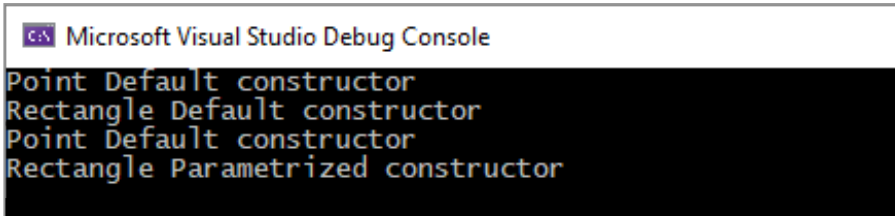
        leftUpperCorner.y = y;
        width = widthP;
        height = heightP;
        std::cout << "Rectangle Parametrized constructor\n";
    }
};

int main()
{
    // the constructor is used by default
    Rectangle rect;
    // parametrized constructor is used
    Rectangle rect1{ 42, 33, /*vertex*/ 10, /*width*/
                    5 /*height*/ };

    return 0;
}

```

### Output 4



```

C:\> Microsoft Visual Studio Debug Console
Point Default constructor
Rectangle Default constructor
Point Default constructor
Rectangle Parametrized constructor

```

Figure 2

Let's analyze the result in more detail. The first and the second lines in the [output](#) result from the [rect](#) object creation. While its creating, the first step is to create a [Point](#) object by calling its default constructor. Then, in the default constructor, initial values are assigned for [Rectangle](#), as for the [leftUpperCorner](#) fields of the [Point](#) class, just as for the width

and height fields. At the same time, we made the `Point` public fields not to use setters in this example. If you leave the private fields, we could not assign values to them from the `Rectangle` class constructors without setters! And, the last two lines in the `Output` show the `rect1` object creating process. A `Point` object is created using the default constructor, and then initial values are assigned to the fields in the constructor body with the `Rectangle` parameters according to the parameters. A `Point` object is created first with a default initial state in both cases. Only then is it assigned the desired initial state — two operations instead of one correct initialization!

Let's figure out how to achieve optimal initialization of the class fields and the most preferred method to implement it. So, it is preferable to initialize the class fields in the constructor using the “class member initialization list” (another name is “class member initializer list”. This list follows the constructor's signature (name and formal parameters list), and it is separated from it by a colon.

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

After the colon, follow the names of the class fields and their initializers (values that perform the corresponding initialization) enclosed in curly brackets `{}` — that is, there is a *unified initialization*! Commas separate class fields with initializers in curly braces.

- **Note:** *there is no semicolon at the end of the class member initialization list! Next after the list of initializers followed by the corresponding constructor body enclosed in curly brackets.*

- **Note:** even if the constructor body is empty, a curly brackets block is required after the initializer list!

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

If the initializer list fits in the exact string as the signature, it is best to place it in a string. The too-long list of initializers should be placed in the string following the constructor signature, always indented before the colon (for better code readability). If the list of initializers is huge, it is recommended to place each element in a new string with an indent, again for better code readability.

It is important to mention that class fields are not initialized in the same order as *indicated* in the initializer list but in a *declaration procedure* of the corresponding fields in the class. That is why the class fields in the initializer list should be placed in the same order when declaring the fields. That's why there will be no confusion about what and in what order is initialized. You should also pay attention to the fact that one field of the class is *not initialized* directly or due to calculations by *the meaning of another* class field, which will be initialized later according to the property described above.

### Example 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';
    }
};

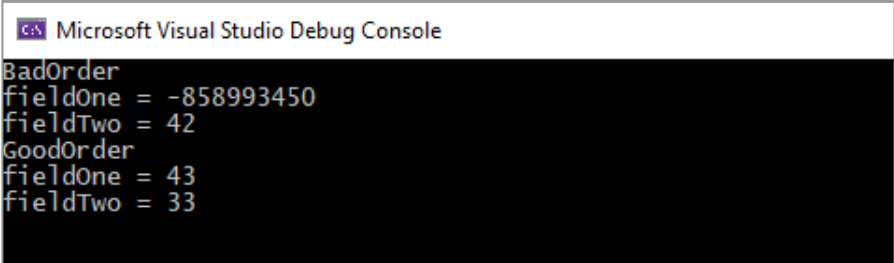
class GoodOrder
{
    int fieldOne;
    int fieldTwo;
public:
    GoodOrder(int param) : fieldOne{ param + 10 },
                        fieldTwo{ param } {}

    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;
}
```

*Output 5*


```

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

```

Figure 3

As you can see on Output 5, in the case of the `BadOrder`, the `fieldOne` is initialized based on the `fieldTwo` value undefined at the time of the `fieldOne` initialization, which led to an unexpected result. By comparison, the `GoodOrder` does not have such a drawback. Their initialization occurs based on the value of the constructor parameter, and it does not depend on the field's initialization order.

Having received knowledge, we will eliminate some bugs from Example 4:

*Example 6*

```

#include <iostream>

class Point
{
    int x;
    int y;

public:
    // default constructor
    Point() : x{ 0 }, y{ 0 }
    { std::cout << "Point D efault constructor\n"; }

```

```

    // constructor with the parameters
    Point(int pX, int pY) : x{ pX }, y{ pY }
    { std::cout << "Point Parametrized constructor\n"; }
};

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

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

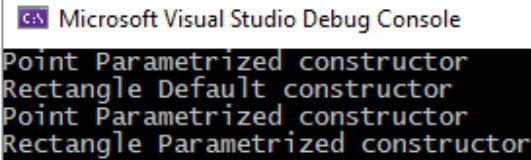
    // constructor with the parameters
    Rectangle(int x, int y, int widthP, int heightP)
        : leftUpperCorner{ x, y }, width{ widthP },
          height{ heightP }
    { std::cout << "Rectangle Parametrized constructor\n"; }
};

int main()
{
    // the constructor is used by default
    Rectangle rect;

    // parametrized constructor is used
    Rectangle rect1{ 42, 33, /*vertex*/ 10 /*width*/,
                    5 /*height*/ };

    return 0;
}

```

*Output 6*


```

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

```

Figure 4

As seen from the [Output](#), the [Point](#) objects inside the [Rectangle](#) are created with the “correct” initial values using a constructor with parameters in one operation, not two!

Another way to initialize class fields is to initialize them directly when the corresponding field is declared in the class;

*Example 7*

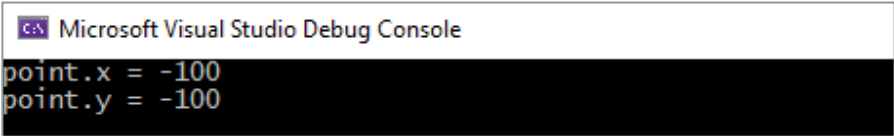
```

#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;
}

```

*Output 7*


```
Microsoft Visual Studio Debug Console
point.x = -100
point.y = -100
```

Figure 5

We should consider the cases initialization separately: during the direct declaration of the field and through the constructor. In this case, the constructor wins:

*Example 8*

```
#include <iostream>

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

public:
    // default constructor
    Point() : x{ 0 }, y{ 0 }
    { std::cout << "Point Default constructor\n"; }

    // constructor with the parameters
    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()
{
```



```

// x and y fields are initialized by default
// constructor to 0,0 respectively
// initializers -100,-100 are ignored!
Point point;

std::cout << "point.x = " << point.getX() << '\n';
std::cout << "point.y = " << point.getY() << '\n';

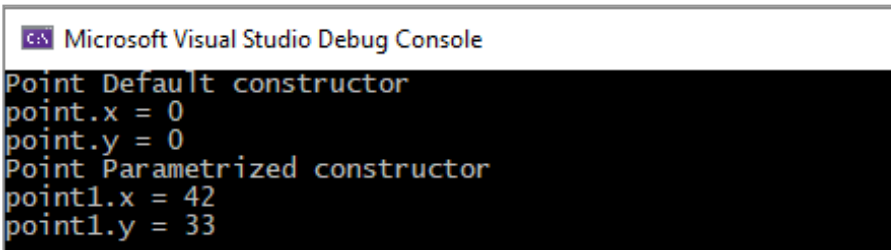
// x and y fields are initialized by the constructor
// with to 42,33 parameters respectively
// initializers -100,-100 are ignored!
Point point1{ 42,33 };

std::cout << "point1.x = " << point1.getX() << '\n';
std::cout << "point1.y = " << point1.getY() << '\n';

return 0;
}

```

### Output 8



```

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

```

Figure 6

This method is recommended for use only in completely trivial classes, which may not even have explicitly defined constructors. To avoid confusion, it is also not recommended to misuse and mix initialization while declaring a field and initialization a field in a constructor.

### 3. Delegating Constructors

Often, a class contains several constructors at once, while some of the code is duplicated in them:

#### *Example 9*

```
class Person
{
    char* name;

    uint16_t age;
    /* uint16_t – unsigned integer 16-bit type recommended
       by current standard integer type notation that
       occupies a predictable number of bytes in any
       architecture. Unsigned short analogue
    */

    uint32_t socialId;
    // Similar to the previous unsigned integer
    // 32-bit type – the unsigned int analogue

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";
}
};

```

We have 4 constructors for various types of class initialization, as you can see. A large part of the code in these constructors is duplicated. But how to avoid repeated actions? Perhaps, is it necessary to call a constructor from another one? This is generally possible, but such a call will lead to entirely unexpected consequences: a temporary class object will be created based on the called constructor, while the newly created object will not be initialized. The correct solution would be using a constructor delegation technique. Its core is to use the constructor in the list of class members initialization familiar to us:

*Example 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 } {}
    /* The default constructor delegates (redirects)
       its work to the constructor with specified
       parameters.
    */

    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();

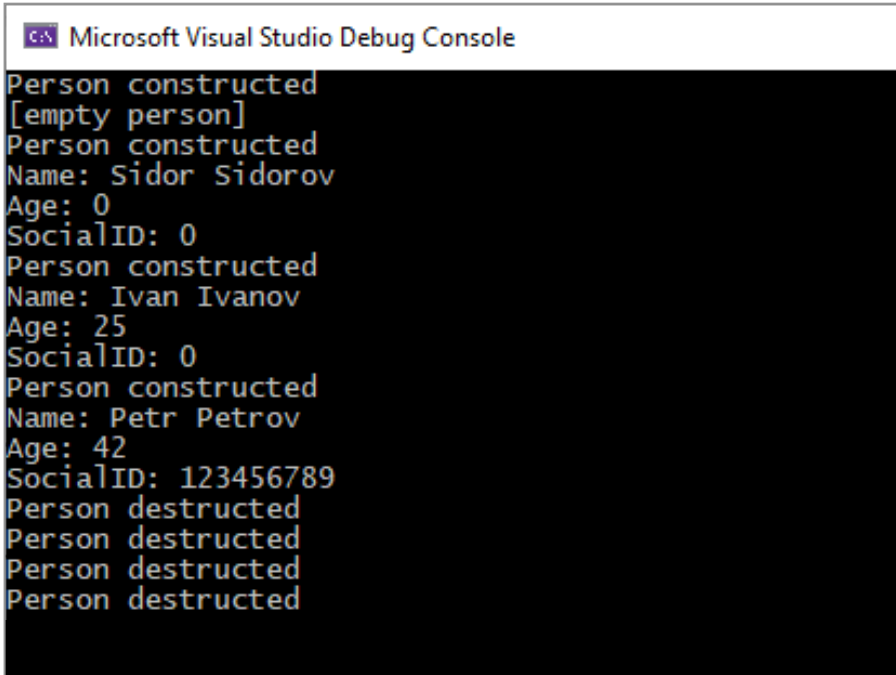
    Person person1{ "Sidor Sidorov" };
    person1.print();

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

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

    return 0;
}

```

*Output 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

```

Figure 7

Let's take a closer look at how it works. First, we declare the most specific and “detailed” constructor according to three parameters: name, age, and social insurance number.

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

Note how the name field is initialized. We use ternary operator and `nameP` depending on parameter value, allocate dynamic memory to store strings with the `name`, and save the pointer or write `nullptr` value to the `name` field.

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

Other fields of the `age` and `socialId` class are initialized with the values of the corresponding constructor parameters.

Then, if the field contains existed name, not the `nullptr` pointer, in the constructor body, we will copy a string parameter to the allocated memory block. We also display a message about a `Person` object creation. Next, for contrast, is the least specific and “detailed” default constructor. This is where the constructor delegation mechanism takes effect! We use the three-parameter constructor in the initializer list defined above, passing the most common values as parameters — `nullptr` for `name` and zeros for the remaining two fields. *Remember about* specifying empty curly brackets after the list of field initializers.

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

The default constructor is ready! Without any string with a duplicate code! Similarly, we will delegate work to the most specific constructor of two constructors remains, setting the appropriate parameters in each case. Without code repetition, indeed.

Add a print member function and destructor to the class to represent a result. And we create 4 copies of `Person` for testing, using all the constructors in turn. The program output indicates the correctness of the work done.

There are several limitations to be aware of while using the constructor delegation technique:

- it is not allowed to initialize class fields in the initializer list along with using a constructor delegation;
- you should avoid the cyclic calling of delegating constructors when `A` constructor delegates work to `B` con-

structor, which in turn delegates the initialization to **A** constructor (**A** and **B** conventional constructor names, actual constructors have the same name!)

The above example is not the only way to use constructor delegation. An arbitrary constructor can generally delegate work to any other, not just one “chosen” constructor. The main thing is to prevent the absence of cyclic calls!



## 4. Static Class Member Variables and Static Class Member Functions

---

When a class copy is created, memory is individually allocated for each of its fields. While creating subsequent copies *for each* of them, memory is also allocated separately *for each* field. Any copy can work separately with its fields without affecting (or even “unknowing” about their presence) other copies of the class.

### *Example 11*

```
#include <iostream>

// simplified Point class with the public fields
class Point
{
public:
    int x;
    int y;
};

int main()
{
    // Memory is allocated and initialized for "personal"
    // x and y for the pointOne
    Point pointOne{ 1,1 };
    // Memory is allocated and initialized for "personal"
    // x and y for the pointTwo
    Point pointTwo{ 2,2 };

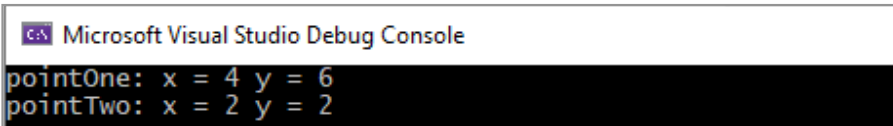
    // modify the x and y fields of the PointOne
    pointOne.x = 4;
    pointOne.y = 6;
}
```

```

// check the changed x and y values for pointOne
std::cout << "pointOne: x = " << pointOne.x
           << " y = " << pointOne.y << '\n';
// check the unchanged x and y values for pointTwo
std::cout << "pointTwo: x = " << pointTwo.x
           << " y = " << pointTwo.y << '\n';
return 0;
}

```

### Output 11



```

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

```

Figure 8

Thus, the state of one instance is entirely unrelated to the state of the others. However, it is necessary to have some shared state to solve individual tasks. In simpler terms, that means to have a class field whose value would be the same and common for all the copies. Identity is easy to achieve. It is enough to define, for example, the `const int maxX{1500};` in the `Point` class, and now the value of the `maxX` field is the same and unchanging, i.e., it is a constant of all the `Point` copies. However, the additional constant property is not always appropriate. Also, you should pay attention that despite the identity of the `maxX` field value, memory for it will be allocated for each copy of the `Point` class. It means that we have not reached the field generality. To solve such issues correctly, it is necessary to use the static-class fields. Such fields can be of any valid type, and they do not have to be constant.

One of the particular characteristics of the static fields is that they are not a part of a class instance; they are not inherent to each model individually. On the contrary, such fields are inherent to the class as a whole, and they are available from an arbitrary instance!

Static fields are created at the start of the program and exist until the end of the program. Working with the static fields has some features. Start with the fact that we cannot use constructors to initialize such fields. We can initialize constant integer fields or enum-fields while declaring them in the class. For initializing the static fields of an arbitrary type, which are not obligatory constant, the syntax specifics should be noted: initialization is performed in a global scope outside the class and outside any function, indicating the field's type, its full name, and initializer. It is no need to specify the 'static' keyword! The action of the `private` and `protected` access specifiers does not apply to the static class field initialization but applies to the subsequent access.

### *Example 12*

```
#include <iostream>

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

int Demo::common{ 0 };
```

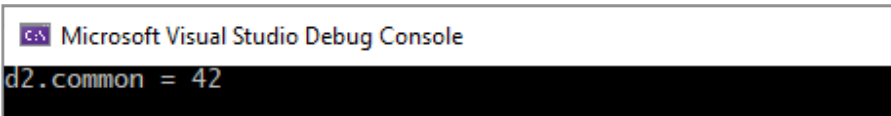
```

int main()
{
    // create an instance and initialize the personal
    // field for d1
    Demo d1{ 1 };
    // create an instantiate and initialize the personal
    // field for d2
    Demo d2{ 2 };

    // assign a value to the common field
    d1.common = 42;
    // tast the value of the common field in d2
    std::cout << "d2.common = " << d2.common << '\n';
    // 42 is on screen
    return 0;
}

```

### Output 12



The screenshot shows a window titled "Microsoft Visual Studio Debug Console". Inside the console, the text "d2.common = 42" is displayed on a single line.

Figure 9

As already mentioned, static fields are not a part of the class instance. So, why then do we access them as ordinary fields? In fact, this is not the only way to access the static fields, but it is needed for convenience and integrity. Another way to access is to specify the field name with a clarification via the `::` class name operator, — `Demo::common` in our example. The key point is that for the field `Demo::common`, the memory is allocated even if no instance of the `Demo` class has been created.

*Example 13*

```

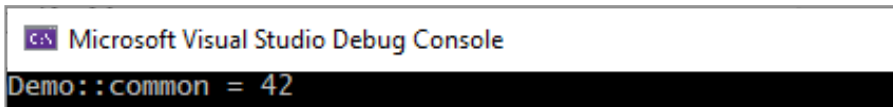
#include <iostream>

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

int Demo::common{ 42 };

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

```

*Output 13*


Microsoft Visual Studio Debug Console

Demo::common = 42

Figure 10

Note that we have not created a single instance of the `Demo` class, but at the same time, we got the value of the `common` field.

Let's consider using a static class field on the example of a `NumberStorage` demo class. Its task is to store in a dynamic memory an array of integers. The only defined constructor takes the number of required elements as a parameter, allocates a block for them in dynamic memory, and fills that block with random numbers. Also, the constructor displays a message

about allocating extra memory amount and the capacity of a dynamic memory used by all class instances. Additionally, there is a member function for displaying the entire array. In the class destructor, the memory is deallocated, and the diagnostics similar to the constructor are displayed. The only difference is that all objects' freed memory and the total amount of memory occupied are displayed. Let's consider the code example:

### *Example 14*

```
#include <iostream>
#include <ctime> // for the time function

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()
{
    // Set the sequence number of the random ones based
    // on the current time.
    srand(time(nullptr));

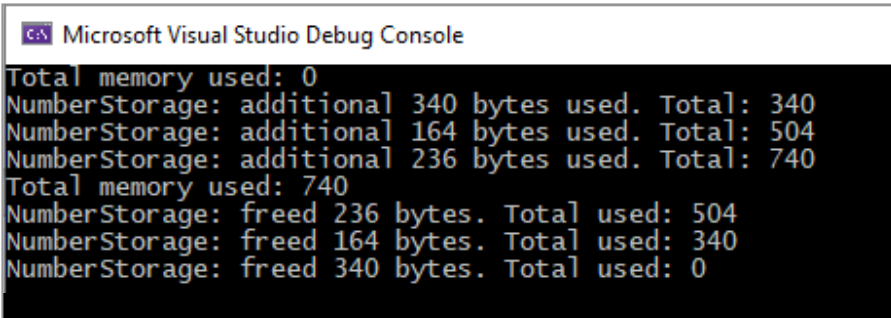
    std::cout << "Total memory used: "
                << NumberStorage::getUsedMemory() << '\n';

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

```

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

### Output 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
```

Figure 11

We do not display array elements on the screen so as not to “clog” the program output with insignificant information, in this case. The example shows another new concept called as static class member functions. If any member function in a class accesses only to the static members of that class, then it can also become fixed, and it can also be called in the context of the class, not an instance of the class. This is precisely the kind of member function in our example: the `getUsedMemory()`. Calling a static member function is similar to accessing a static field of a `class`: the class name, the name qualification operator `::` and the name of the static member function itself are specified with the actual parameters in the parentheses (if any). A static member function can also be called through an instance of the class if necessary.



To declare a class member function as a static one, you must specify the 'static' keyword before its declaration in the class. Not every member function can be made static. In addition to the static fields, a member function that requires the same common non-static fields for its work can't *be static*. This is quite logical since, without a class instance, we do not have any ordinary fields, any memory is allocated for them, and, accordingly, there is no access to them. At the same time, a non-static, common class member function can manipulate both non-static and static class fields without restrictions.

Static fields and class member functions are used to count created/active class instances or assign a unique ordinal number to each sample. In a general case, if some data is not tied to a specific object, but they are inherent to all the objects as a whole, or if it reflects the general characteristics/states of all the objects and not a specific one, then such fields should be made static. A good example would be constants used by the class as a whole or tables (arrays) with some set of constant values. The same applies to the member functions. If a particular member function does not work for a specific class instance, it should be made static for the class in general. While technically, nothing prevents you from creating a class containing only static fields and member functions, it is not recommended to do so! It is impossible to create instances of this class, and the practical value of its use does not have any sense.

## 5. “This” Pointer

In C++, a class is like a draft from which object instances are created. At the same time, it is necessary to understand how memory is allocated for fields and member functions for the specific class instance. So, let's analyze this simple example:

### *Example 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()
{
    /* Create and initialize an instance of the Date class
       Memory is allocated for the date1 instance and its
       private fields date1.day, date1.month, date1.year
       */
    Date date1{ 1,1,2020 };
```

```

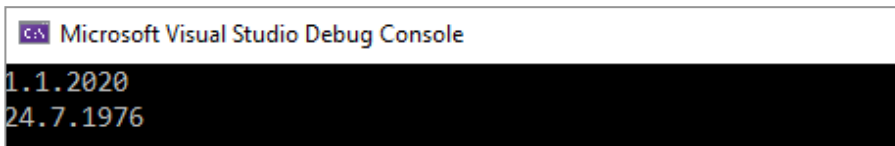
    /* Create and initialize an instance of the Date class
       Memory is allocated for the date2 instance and its
       private fields date2.day, date2.month, date2.year
    */
    Date date2{ 24,07,1976 };

    /*
       Call the print member function. The member function
       code is standard for all Date instances.
    */
    date1.print();
    date2.print();

    return 0;
}

```

### Output 15



```

Microsoft Visual Studio Debug Console
1.1.2020
24.7.1976

```

Figure 12

As we already know, creating a class instance leads to the memory allocation for all class fields, individually for each instance. But, what happens to member functions? We could assume that member functions are also unique for each instance. However, this is not true. According to their size, all member functions occupy memory within a class only once. When multiple instances are created, no extra memory is allocated for member functions. In other words, the fields' data are unique, and the code for processing member functions is

shared for class instances. How does the same method work with different class instances? Let's move back to the code from Example 15. The `print()` member function does not take any parameters, nor does it create any local variables inside. But, what are identifiers: `day`, `month`, `year` inside this method? — Class fields. It is true, but how does the one method “know” which class instance to work with? Which specific fields do this method deal with? And how to get access to the memory block allocated for a particular class instance?

There is nothing special here. The whole point is that each non-static member function in the class gets, as its first implicit parameter, a pointer to the class instance due to which the member function was called. This implicit first parameter is a constant pointer to an instance of the class, called “`this`”.

```
Date* const this /* this for Date member functions*/
```

Note that it is unnecessary to point out explicitly `this` as the first parameter. Moreover, it will lead to an error if you do so! Despite being implicit at the declaration stage, ‘`this`’ pointer can be used inside the non-static member functions explicitly or implicitly. The presence of such a pointer explains how the member function interacts with the data of a particular instance. Access to the fields of a specific instance can be done explicitly inside a member function via ‘`this`’ pointer and “`->`” operator:

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

Such a method of accessing fields is optional. But accessing with “`this`” pointer to a class instance is required to specify the field name inside a non-static class member function. The presence of an implicit `this` explains all the interaction “magic” of a general code with unique data for the class instances.

Should we always use `this` to access member fields and functions? Often we should not. However, it may be necessary to specify “`this`” explicitly when it resolves the ambiguity. For example, using the same member function parameter name as the class field or a local variable inside the member function.

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

In the shown example, “`this`” resolves the ambiguity associated with the parameter name and field name. At the same time, you should avoid such conflicts with naming and ambiguities, e.g., by adding `P` suffix or the prefix to `the` parameter names that could conflict with the names used for class components.

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

Another example of application is the member functions creation that supports chaining. Let’s compare two implementations of setters in the `Date` class.

### *Example 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;
}

```

### Example 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;
}

```

*Output 16, 17*

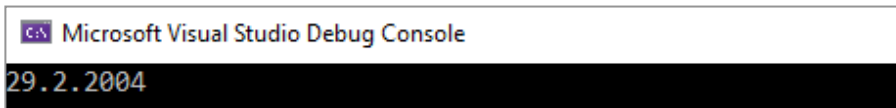


Figure 13

So, the getters in Example 16 take one corresponding parameter (day, month, year), assign the parameter value to a class instance field, and end without returning a value (the return type of the member function is `void`). To set all three values, you must sequentially call appropriate setters. It turns

out that this task can be solved more nicely than usual. In Example 17, the setters make the exact implementation. But at the end, they additionally return a dereferenced “`this`” pointer (return type of the member function — `Date&`). This addition allows you to call setters in a chain: calling the first setter (`setDay`) takes place on the `date1` instance. Then, the `setDay` returns a reference to the `date1` and, as a result, it is possible to call the `setMonth` setter using this link, and similarly then — to call the `setYear`. There is no difference due to setters’ operation, but the code with a call chain has become more beautiful and understandable. Looking ahead, the call output on the screen of `<<` operator is similarly implemented along the chain. In general, if a class member function returns `void`, it should be an appropriate `void` replacing to the `ClassName&`, making it possible to call following member functions in the chain.

Let’s consider another way to use `this`. We will add a visualization of the execution order of constructors and destructors for instances of the `Date` class.

### Example 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; }
};

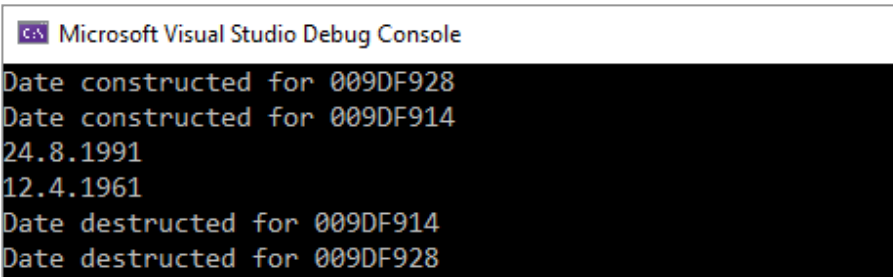
int main()
{
    Date date1{ 24,8,1991 };
    Date date2{ 12,4,1961 };

    date1.print();
    date2.print();

    return 0;
}

```

### Output 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

```

Figure 14

Example 18 shows this output value on screen from constructor and destructor. Now it is possible to trace the order of

the instance creation-destruction. First, the `date1{24,8,1991}` instance is created with the corresponding address: `009DF928`. Then the `date2{12,4,1961}` instance is created with the address: `009DF914`. According to the destructor's messages, the class instances destruction occurs in reverse order: for the `009DF914` first, and then for the `009DF928`. The specific value of the addresses will differ from the outputted.

Moreover, the values will most likely change from one program launch to another. It is not the absolute values of the addresses that are important, but their "pairing". "This" value output on a screen from various member functions can significantly simplify the program debugging.

## 6. Copy Constructor

Various ways of initializing objects were described in the previous sections of the lesson. So, let's consider one more method of initializing objects: copy initialization. For this, we need a small class — a simple fraction as a basic implementation. The example below will implement only the required minimum of functionality: a constructor with parameters, a default constructor, a destructor, and a member function to display a fraction on the screen. They show debugging information on the screen to represent the actions of constructors and destructors.

### *Example 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()
{
    /* Create and initialize the Fraction-a instance with
       the values of the numerator and denominator
    */
    Fraction a{ 2,3 };

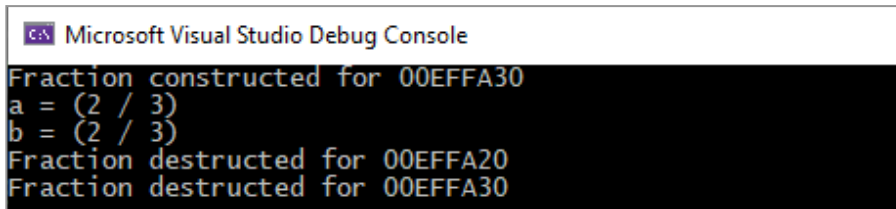
    /* Create and initialize the Fraction - b with
       the a instance using the current value
    */
    Fraction b{ a };

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

    return 0;
}

```

### Output 19



```

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

```

Figure 15

Output 19 shows that the constructor for the `a{2,3}` instance is called first with the corresponding `00EFFA30` address. Then, according to the code from Example 19, the `b{a}` instance is created without a message about the constructor processing.

Then, the screen displays `a`, `b`, and we see the destructor's working for the “not created” `b` instance (with the corresponding address: `00EFFA20`), and ending the process — the work of the destructor for the `a` instance with the address: `00EFFA30`. Shown may seem a little odd.

Let's consider the reasons for such an action. In the `Fraction` class, two constructors have been explicitly defined: one takes two integer parameters. The second is a default constructor without any parameters, which delegates initialization to the previous constructor with two parameters. We initialize the `a` instance with a constructor with two integer parameters, and what about the `b` instance?

Should we create it with the default constructor? Or should we set the values of the numerator and denominator explicitly? No, the deal is the `b` instance initialization takes place according to the current value of the `a` instance. And none of the constructors we explicitly defined are involved. That is why there is no debug output about instantiation, and it has nowhere to come from.

How will we initialize the `b` instance? After all, there is  $(2 / 3)$  according to Output 19 results. The initialization occurs with an automatically generated compiler, that is, with a *copy constructor*. Without knowing the inner point of the class, the compiler automatically creates a copy constructor that performs a shallow or a bitwise copy of one initializer object into another one just created that requires initialization.

A copy constructor is used to copy an existing class instance into a new instance of the same class. If a copy constructor is not explicitly defined in the class, then, similarly to the default constructor, the compiler will automatically generate a public constructor that will perform a member-wise initialization of the new instance fields with the values of the corresponding fields of the existing class instance. According to the example above, the `b.numerator` will be initialized with the `a.numerator` value, and the `b.denominator`, — with the `a.denominator`, correspondingly.

The copy constructor can be defined explicitly. For this, you need to use a unique signature:

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

The `ClassName` means the name of the class for which the constructor is being defined. A copy constructor is a constructor that takes an instance of the same class by the constant reference. The reason for this will be discussed later. Signature example for the `Fraction`:

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

Let's explicitly define a copy constructor for the `Fraction` class:

### *Example 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()
{
    /* Create and initialize the Fraction – a instance
       with the numerator and denominator values
    */
    Fraction a{ 2,3 };

    /* Create and initialize the Fraction – b with the a
       instance using the current value*/
    Fraction b{ a };
}

```

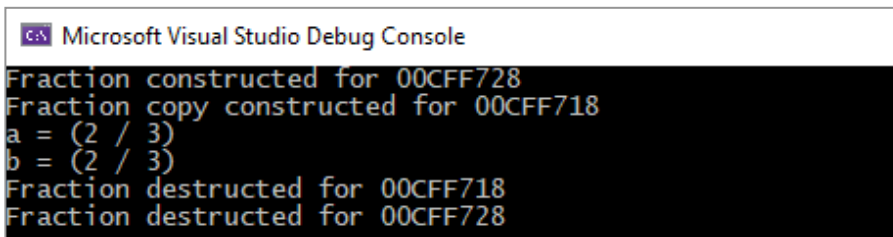
```

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

return 0;
}

```

### Output 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

```

Figure 16

In Example 20, the explicitly defined copy constructor initializes the fields member by member and outputs debugging information about creating a new instance by copy. According to Output 20, there are no more “uncreated” instances of the `Fraction` class. The `a` instance is initialized with the constructor with two integer parameters, and the `b` instance — with a copy constructor that takes the `Fraction` instance via the constant reference. An explicitly defined constructor for the `Fraction` performs the same actions as a compiler-generated copy constructor. We just added debug output to inform us that the copy constructor was executed. If we do not need to display debugging information, so then for the `Fraction` class, there is no need for an explicitly defined copy constructor. But



it does not occur with all classes. Later, we will analyze an example of avoiding an explicit definition of a copy constructor.

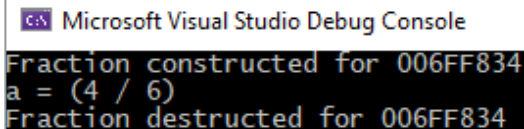
Suppose we initialize a class instance with a temporary anonymous instance. In that case, the compiler may not call the copy constructor but perform optimization and initialize the class instance with temporary object initializers without creating a temporary object. So, let's change the `main` function in Example 20:

### Example 21

```
int main()
{
    /* Create and initialize with a temporary, anonymous
       Fraction – a instance. The copy constructor will
       not be called; the temporary object will not be
       created! The compiler will initialize the a
       instance using the values 4 and 6.
    */
    Fraction a{ Fraction{4,6} };

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

### Output 21



```
Microsoft Visual Studio Debug Console
Fraction constructed for 006FF834
a = (4 / 6)
Fraction destructed for 006FF834
```

Figure 17

This optimization of unnecessary object creation with its subsequent copying is called elision. In this case, even if in the copy constructor some additional work was performed in addition to the actual copying — e.g., displaying on the screen, the body of the copy constructor will not be executed.

Let's analyze the copy constructor signature. Why is it essential that the only parameter is a constant reference to a class instance? And what happens if an instance is passed by value? Like this:

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

If such a code could be compiled and run, then while trying to use copy initialization, it would be necessary to create a **copy** of an initializer object! It would require calling the copy constructor! There would be an infinite recursion! Fortunately, modern compilers do not compile such a code, and errors are generated at the compilation stage. Why is the `const` modifier needed? By passing a parameter by reference, it is possible to modify it inside the copy constructor. That is technically possible, but it is completely unacceptable and inappropriate to modify the original object when copying. To prevent even the potential possibility of modification, even by mistake, it is the constant reference that is used.

Is an automatically generated copy constructor always appropriate? Or is an explicit copy constructor definition appropriate that implements only shallow copying? Consider an example of a simplified dynamic array class to answer this question. We will create a copy constructor by executing a surface copy in this example. The compiler automatically generates such a constructor when the programmer does

not provide own version of the copy constructor suitable for the class.

### *Example 22*

```
#include <iostream>
/* Attention! The program crashes! It is appropriate
   for this example.
*/

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';
}

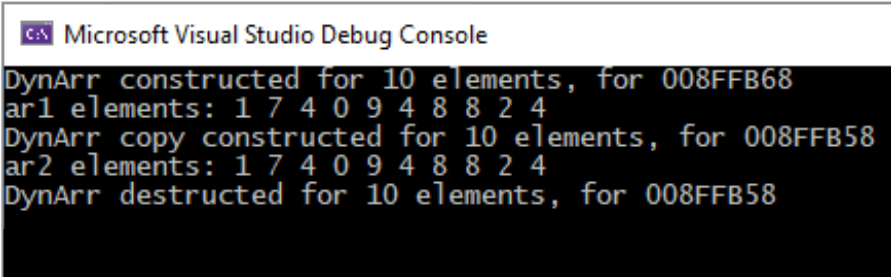
void DynArray::randomize()
{
    for (int i{ 0 }; i < size; ++i)
    {
        arr[i] = rand() % 10;
    }
}

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

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

    return 0;
}

```

*Output 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

```

Figure 18

Let's analyze the processes in Example 22. First, we create an `ar1` instance of the `DynArray` class of 10 integers and fill it with random numbers. Output the `ar1` data to the screen. Then, we create an `ar2` instance as an `ar1` copy and display it on the screen. For copying, we use an explicitly defined copy constructor, which performs memberwise initialization of the new instance fields based on the values of the existing instance of the `DynArray` class; in other words, we perform a shallow copy. The program ends by calling the destructors in the reverse order of class instance creation. First, the destructor is called `ar2`. After then, the program crashes during the operation of the destructor for `ar1`! Output 22 shows what has been displayed on the screen. Why does it occur so? Let's change the destructor a bit to figure it out.

*Example 23*

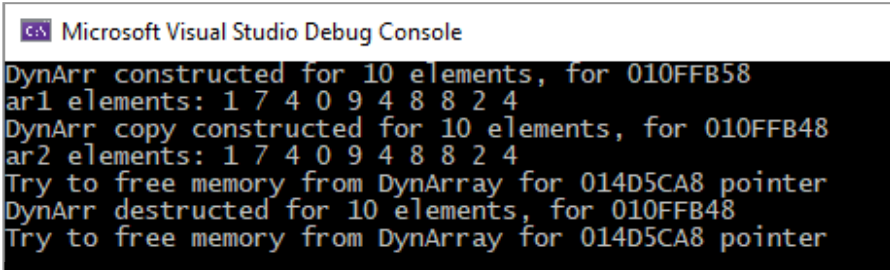
```

~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';
}
```

### Output 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
```

Figure 19

So, the destructor for `ar2` deallocates memory occupied by a dynamically allocated array by the `014D5CA8` pointer. After, the destructor for `ar1` deallocates memory at the same pointer. It is so because we just copied a pointer in our copy constructor. We have not created a new copy of the array, and we have not copied the values from the existing array. As mentioned, a shallow copy was created, entirely unsuitable for this class.

For this kind of class, where dynamic memory is allocated, a correct implementation of the copy constructor is necessary but not automatically created by the compiler. And based on the existing class object, such an implementation should create its deep copy, allocating new dynamic memory blocks and copying values from the original dynamic memory blocks there. Let's look at an example of such a constructor:

*Example 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 }
    {
        /*
         In the list of class field initializers above,
         allocate a new block of the dynamic memory
         with the same size as the copied instance of
         the DynArray class.
         The following loop copies the elements from
         the original memory block to the newly
         allocated one.
        */
        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();
}

```



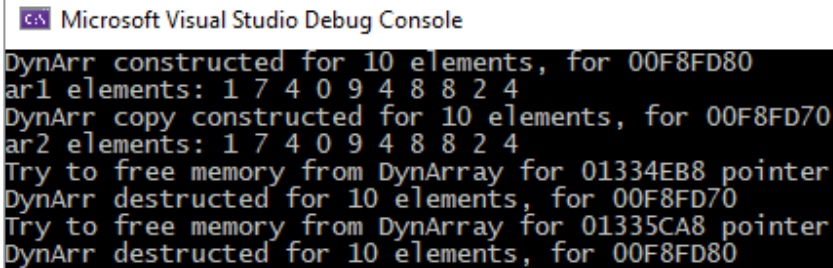
```

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

return 0;
}

```

### Output 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

```

Figure 20

Now the program works correctly, without failures. The copy constructor creates a deep copy of the `ar1` class instance. A memory block is allocated at its pointer when launching destructors, like in Output 23.

The implementation example of the copy constructor represents that for such classes, in general, the copying is an extremely “cost” action. It is necessary to allocate a new block or blocks of dynamic memory and copy the information from the original block or blocks. In the above example, we talked about a block of 10 integers. The copying was almost instantly, but memory blocks can be much larger in the general case. It follows from the above that you should avoid object copying where possible and acceptable alternatives are.

- **Important comment!** *When passing a class instance as a parameter to a function by value, a new class instance is created using a copy constructor.*

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

*Similarly, if a class instance is returned from a function by value, a copy of the class instance is also created.*

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

Further, as a part of our course, we will consider techniques to avoid unnecessary copying where possible, according to the requirements of actual cases. But for now, you should remember the “cost” of the copy operation. You should avoid copying by passing class instances by references where possible and `const` where it is appropriate. Also, return class instances from functions by references, where possible.

## 7. Homework

### 1. Create a “Fraction” class to represent a simple fraction.

*Fields:*

- numerator
- denominator

*Member functions:*

- constructor accepting numerator and denominator.

Use the list of class field initializers in the constructor.

- ▷ default constructor, implement via delegation to the constructor a numerator and a denominator with parameters;
- ▷ displaying a fraction;
- ▷ addition/subtraction/multiplication using simple fractions;
- ▷ addition/subtraction/multiplication of a simple fraction with an integer.

Provide for the possibility of calling operations in a chain using “[this](#)” pointer in arithmetic operations.

Consider fraction reduction. The reduction is recommended to be done in the constructor.

### 2. Create a Person class.

*Fields:*

- ID number;
- surname;
- name;

- second name (*For indicating a full name to allocate memory dynamically!*)
- date of birth (it is recommended to create an additional Date class (day, month, year).

*Member functions:*

- constructor with parameters: ID number, surname, name, second name, date of birth.  
Use the list of class field initializers in the constructor.
- default constructor. In the constructor, use constructor delegation.
- copy constructor;
- destructor;
- a member function for counting created instances of the “Person” class;
- setters/getters for the corresponding class fields;
- displaying the personal information.

### 3. Create a “String” class.

*Fields:*

- string length excluding the null terminator;
- pointer to the memory block where the string is stored.  
Allocate memory dynamically!

*Member functions:*

- constructor with a string parameter (`const char*`);
- constructor with a string length parameter;
- copy constructor;
- destructor;
- output string to screen;

- a setter that takes a string (`const char*`) as a parameter. If there is not enough of an already allocated dynamic memory block, perform a correct memory reallocation to copy the string parameter into it.
  - **Note:** *The list of member fields and functions in Tasks 1-3 is recommended, not a definitive. Optionally, you can add required fields and member functions.*
4. **Create a program that simulates an apartment building.** It is necessary to have such classes: Person, Apartment, House. The “Apartment” class contains a dynamic array of the Person class objects. The “House” class includes an array of the Apartment class objects. Each class has member variables and member functions, required for the class’s domain. Note: memory for string values is allocated dynamically. Remember to use different constructors (copy constructor is obligatory) and destructors in classes. The “Human” class should be used from Task 2.



## Lesson 2.

### Initializers. Static Class Member Variables and Static Class Member Functions. "This" Pointer and Copy Constructor.

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

© Aleksandr Rybchanskiy

All rights to protected pictures, audio, and video belong to their authors or legal owners.

Fragments of works are used exclusively in illustration purposes to the extent justified by the purpose as part of an educational process and for educational purposes in accordance with Article 1273 Sec. 4 of the Civil Code of the Russian Federation and Articles 21 and 23 of the Law of Ukraine "On Copyright and Related Rights". The extent and method of cited works are in conformity with the standards, do not conflict with a normal exploitation of the work, and do not prejudice the legitimate interests of the authors and rightholders. Cited fragments of works can be replaced with alternative, non-protected analogs, and as such correspond the criteria of fair use.

All rights reserved. Any reproduction, in whole or in part, is prohibited. Agreement of the use of works and their fragments is carried out with the authors and other right owners. Materials from this document can be used only with resource link.

Liability for unauthorized copying and commercial use of materials is defined according to the current legislation of Ukraine.