Object-Oriented Programming Using C++

STEP computer ACADEMY

Lesson 4

Move Constructor. Overloading Special Operators. Friend functions and Overloading

Contents

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

1. Move Constructor

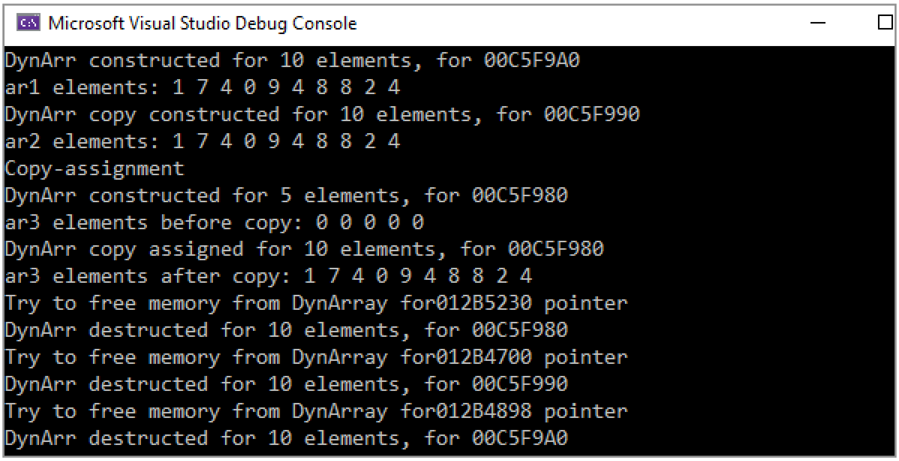
To study the new topic, we need to recall the previous one, — “Copy constructor”, and more specifically, — the DynArray example. Based on the material from the previous lesson about overloading various operators, including the assignment operator, we will add the overloading of the copy assignment operator to the class based on the existing DynArray.

Example 1.

```
#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 the class field initializers
           above, allocate a new block of the dynamic
           memory of the same size as in the copied
           instance of the DynArray class.
           The next loop copies the elements
           from the source 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 ';
    }

    DynArray& operator=(const DynArray& object)
    {
        // checking for self-assignment
        if (!(this == &object))
        {
            /* checking for the impossibility of "reusing
"
               the storage block allocated for
               the existing array */
            if (size != object.size)
            {
                /* in case of impossibility to "reuse "
                   you need to deallocate the memory
                   occupied with the elements
                   of the current dynamic array */
                delete[] arr;
                /* allocate a new storage block according
                   to the size of the copied array */
                arr = new int[object.size];
            }
            size = object.size;
            /* The next loop copies the elements
               from the source memory block to the newly
               allocated. */
            for (int i{ 0 }; i < size; ++i)
            {
                arr[i] = object.arr[i];
            };
        }
        std::cout << “DynArr copy assigned for “
                  << size <<  " elements, for  " << this
                  <<  '\n ';
        return *this;
    }

    int getElem(int idx)const { return arr[idx]; }
    void setElem(int idx, int val) { arr[idx] = val; }
    void print()const;
    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()const
{
    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;
    }
}
```

```cpp
int main()
{
    DynArray ar1{ 10 };
    ar1.randomize();
    std::cout <<  "ar1 elements:  ";
    ar1.print();
    DynArray ar2{ ar1 };
    std::cout <<  "ar2 elements:  ";
    ar2.print();
    std::cout <<  "Copy-assignment test\n ";
    DynArray ar3{ 5 };
    std::cout <<  "ar3 elements before copy:  ";
    ar3.print();
    ar3 = ar2;
    std::cout <<  "ar3 elements after copy:  ";
    ar3.print();

    return 0;
}
```

The output code result of Example 1:



```
DynArr constructed for 10 elements, for 00C5F9A0
ar1 elements: 1 7 4 0 9 4 8 8 2 4
DynArr copy constructed for 10 elements, for 00C5F990
ar2 elements: 1 7 4 0 9 4 8 8 2 4
Copy-assignment
DynArr constructed for 5 elements, for 00C5F980
ar3 elements before copy: 0 0 0 0 0
DynArr copy assigned for 10 elements, for 00C5F980
ar3 elements after copy: 1 7 4 0 9 4 8 8 2 4
Try to free memory from DynArray for012B5230 pointer
DynArr destructed for 10 elements, for 00C5F980
Try to free memory from DynArray for012B4700 pointer
DynArr destructed for 10 elements, for 00C5F990
Try to free memory from DynArray for012B4898 pointer
DynArr destructed for 10 elements, for 00C5F9A0
```

Figure 1

The comments in Example 1 and the outputted result indicate that we have a class that fully supports copy semantics, both during initialization (copy constructor) and assignment (copy assignment operator). Tests have confirmed that deep copying occurs when allocating a new memory block for copies.

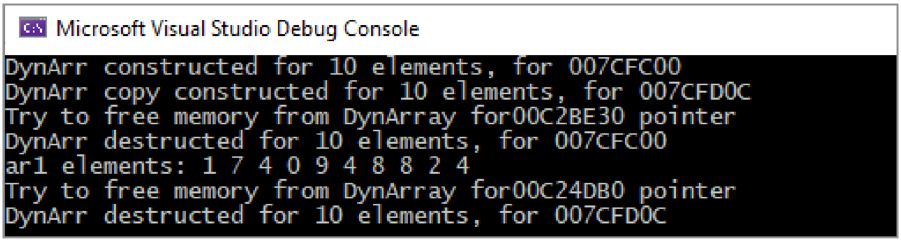
However, the two questions arise: is deep copying always necessary? And, how is it optimal? Such deep copying is not always needed. It is often possible to avoid deep copying and replace it with a more efficient option. What does "moving" mean, and when and in what cases is it necessary? — We will consider it further.

We often created an array in the dynamic array examples and filled it with random numbers. Well, let's get rid of code repetition and write a function for this! The function parameter will be the number of elements in the array, while the function will return a dynamic array of a given size, filled with the random numbers. So, add the above function to Example 1 and replace the 'main' function to test the work.

Example 2

```
DynArray arrayFactory(int arrSize)
{
    DynArray arr{ arrSize };
    arr.randomize();
    return arr;
}

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

    return 0;
}
```

The output code result of Example 2:



Microsoft Visual Studio Debug Console

```
DynArr constructed for 10 elements, for 007CFC00
DynArr copy constructed for 10 elements, for 007CFD0C
Try to free memory from DynArray for00C2BE30 pointer
DynArr destructed for 10 elements, for 007CFC00
ar1 elements: 1 7 4 0 9 4 8 8 2 4
Try to free memory from DynArray for00C24DB0 pointer
DynArr destructed for 10 elements, for 007CFD0C
```

Figure 2

Let’s take a closer look at the ‘array factory’ function. To perform the tasks assigned with its application, we should create a copy of the DynArray class of the size specified by the value of the arrSize parameter. A copy is created as a local variable by calling the DynArray constructor taking the number of array elements as a parameter. Then, the member randomize() function is called on the class copy, and the array is filled with the random numbers. It seems that the task is already solved, but this is where the most interesting part begins.

But, how can we return the function result? Can we return a reference to a class copy created inside a function? No, because it will terminate its processing with the function completion, and we will get a “dangling” reference to the already deleted DynArray copy. Perhaps, if we allocate a class copy in dynamic memory, can we return a pointer to a class copy then? It will work, but it will be highly inconvenient to apply such a function. And we will have to work not with a class copy but with a pointer to the copy, which is not so handy. However, a more critical problem is the need to free the memory manually allocated for the class copy; simply termination of the processing of the pointer is not enough to free the memory occupied by the class copy. So, what remains, and what should we do? And all that remains is to return a class copy by a value.

Return by a value generally occurs by copying the value returned by the function into a temporary object that holds the function's return value after it completes. Copying a value for class copy leads to running a copy constructor, which may not be the best solution, as shown below.

The temporary storage object of the return value is available at the function call site, and its lifetime is the expression in which the function was called. The value of this object is used as the right side of the assignment operator, if the function's return value is stored in some variable.

When copying the return value into this temporary object, the compiler can apply optimizations and get rid of additional copying under certain conditions. The description of this optimization process (RVO, Return Value Optimization) is beyond the scope of the topic we are currently studying; details and subtleties of implementation are not required to understand the topic itself. It is only necessary to know that such optimization can take place.

Let's come back to Example 2. We create a copy of the DynArray class in the 'main' function by initializing this copy with the result of the arrayFactory function. The first line in Output 2 is shown the created copy of the DynArray class inside the function. After the function completes, the result is obtained in a temporary object (in this case, the RVO optimization described above takes place), which as a copy of the DynArray class, shows as the initializer of the ar1 array we created. Here is a copy initialization that occurs in the second string of Output 2. In the fourth string of Output 2, the lifetime of the temporary object has ended, and its destructor is called. Then we display the array data created in such a way — see the fifth string of Output 2. The 'main' function and the test application stop the processing, the lifetime of ar1 ends, and the destructor is called for it, which we can see at the last string of Output 2.

Example 2 works without errors, but is the copy optimal occurred in the example? When a DynArray copy is created inside the function, a dynamic memory block is allocated where the elements of the array are stored. Further, while copying the return value to ar1, another dynamic memory block is created, where the values are copied from the memory block belonging to the temporary object. And all these "heavy" and resource-intensive operations of memory allocation and copying generate a copy of the memory block to free it. Significantly suboptimal position, but necessary for the consistent operation of the DynArray class.

It would be much more optimal to transfer the command of the dynamic memory block with the elements stored from the temporary object to the ar1 object for making not a deep but a shallow copy. But we already know that it is looking at advance free of the dynamic memory block by the destructor of a temporary object and the failure of the ar1 destructor when trying to free the already deallocated dynamic memory block. It is because the temporary object has a pointer to a dynamic memory block that does not require deletion, according to the optimization mentioned above.

It would be desirable to remove this pointer from the temporary object. Let's look at the required process illustrated below.

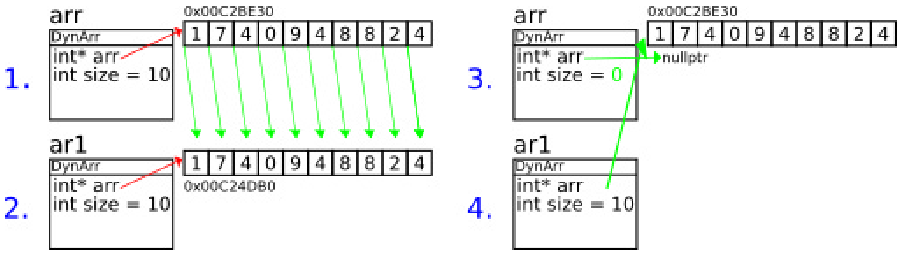


Figure 3

Fragments 1 and 2 reflect the standard copying process. Fragment 1 shows the state of the arr object inside the function, and Fragment 2 shows the state of the ar1 object after allocating a new memory block and copying (green arrows) all the elements into it from the memory block owned by arr. Fragments 3 and 4 reflect the state of objects arr and ar1 in the process of returning a value from the arrayFactory function. Here we notice that the ar1 object "took ownership" of a memory block from arr without allocating it and performing element copying.

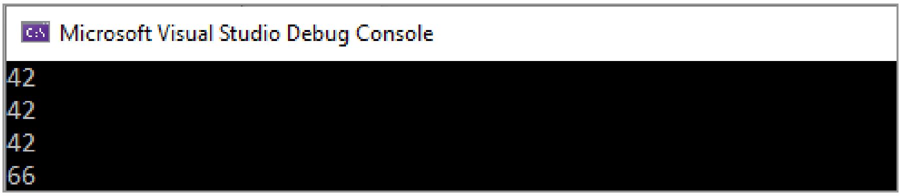
But how can we achieve this process? After all, all this should happen during the work of a modified copy constructor, and its only parameter is the const DynArray&, that is, a const reference. The need for the constness of this reference was described in the lesson related to the topic of the copy constructor. Does it appear that the desired optimization is impossible and only a “fantasy”? It is so based only on the tools and constructions of the programming language we have. But we are not going to stop and study new tools and programming language constructs. Then, we will return to solving our problem with copy optimization.

First, we should recall what a reference is. A reference is an aliased name for some named object in memory.

Example 3.

```
#include <iostream>

int main()
{
    int x{ 42 };
    int& refX{ x };
    const int& cRefX{ x };
    const int& cRefXX{ x + 24 };

    std::cout << x <<  '\n '
        << refX <<  '\n '
        << cRefX <<  '\n '
        << cRefXX <<  '\n ';

    return 0;
}
```

The output code result of Example 3:


Microsoft Visual Studio Debug Console

```
42
42
42
66
```

Figure 4

So, x is some variable that has its location in RAM associated with the actual name x. RefX is another name associated with the same memory area. We can do everything with x and with refX as well. cRefX is another name for the memory area that holds x, allowing only reading but not modifying this memory area. And the most interesting is cRefXX. It's a const reference, but to what? Is it a reference to an expression? To a place in memory storing the expression result, its value. It is impossible to create this reference without the const modifier; since then, it would be possible to change the expression's result, which seems very strange. At least for now. So, what is the feature of the "strange" cRefXX reference? Mostly, the lifetime of the temporary object that results from the expression is the expression itself. After the entire expression x+24 has been evaluated, the temporary object is deleted. The const cRefXX reference extends the lifetime of this temporary object to the lifetime of the reference itself. RefX and cRefX are lvalue references familiar to us from the course. The lvalue reference is a reference to an object in memory that has its name. The cRefXX reference does not correspond with such an argument since a temporary object doesn't have its name, which was an exception to the general rule in older C++ standards.

Starting with the C++11 standard, it became possible to create non-constant references to temporary, unnamed objects, that is, the references to values. Such references are called the rvalue references. Declaration of the rvalue references is made by adding two ampersands after the reference type:

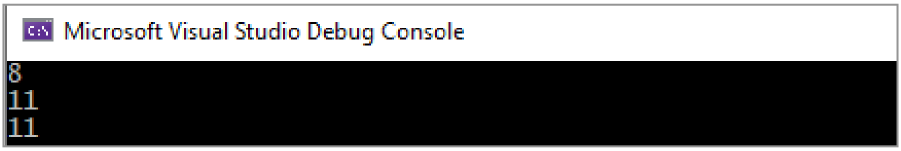
```
RefType&& refName{ rvalue object };
```

Let's analyze some typical examples:

Example 4.

```
#include <iostream>
int max(int a, int b)
{
    return a > b ? a : b;
}

int main()
{
    int&& rvalRef { 2 + 3};
    rvalRef += 3;
    std::cout << rvalRef <<  '\n ';
    int&& res{ max(3, 5) };
    res += max(6, 4);
    std::cout << res <<  '\n ';
    int x{ 42 };

    // The following strings are specially commented!
    // int&& rvalBad{ x }; /* It is impossible to
    // initialize the rvalue reference with the lvalue
    // object. */
    // int&& rvalBad1{ res }; /* rvalue reference is
    // the lvalue object */
    int& lvalRef{ res };
    std::cout << lvalRef <<  '\n ';

    return 0;
}
```

The output code result of Example 4:



```
Microsoft Visual Studio Debug Console
8
11
11
```

Figure 5

Let's discuss the processes in Example 4. So, the rvalue reference can only be initialized with the rvalue object. It is exactly what happens with rvalRef: it is initialized with the result of the expression 2+3, — extending the lifetime of the temporary object, storing the expression result to the lifetime of the reference, and making it possible to change this object.

Another possible way to initialize the rvalue reference is to initialize it with the result of a value-returning function — the res reference is in the example. The max(3,5) function result is a temporary rvalue object that sets great as an initializer. Similar to rvalRef, the lifetime is extended, and it is possible to change the result object of the function.

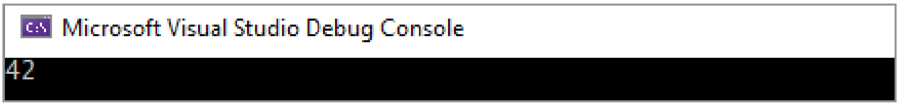
You should pay special attention to the fact that the rvalue reference cannot be initialized, i.e., "attached" to the lvalue object. The rvalBad reference from Example 4 cannot be attached to the x variable because x is a named object in memory, and therefore it is the lvalue. The rvalBad1 reference cannot be created because 'res' is the lvalue object! Why is it so? After all, the 'res' is the rvalue reference, isn't it? The thing is that res is associated with an anonymous, unnamed result object of the function. But the 'res' object is already named, and it just names deanonymizes the function result object! Therefore, the rvalue reference is the lvalue object, as demonstrated by the lvalRef reference in Example 4.

Let's come up with a conclusion. There are two types of references since the C++11 standard was performed. Lvalue references are references to the named objects, and rvalue references are references to the unnamed values and temporary objects. Both types of references can be const and non-const, and they are required to be attached to the corresponding object during initialization. They do not support reattaching to another object after initialization. It is not possible to create a non-const lvalue reference to the rvalue value and the rvalue reference directly to the lvalue value. However, it is possible to convert the lvalue value to the rvalue value by calling std::move. Using converted the lvalue to rvalue can have certain consequences, but we will discuss it later. Also, the practical use and application of such a transformation will be shown additionally. For now, it is only essential to know about such an option.

Example 5.

```
#include <iostream>

int main()
{
    int x{ 42 };
    int&& rvalRef{ std::move(x) };

    std::cout << rvalRef <<  '\n ';

    return 0;
}
```

The output code result of Example 5:



```
Microsoft Visual Studio Debug Console
42
```

Figure 6

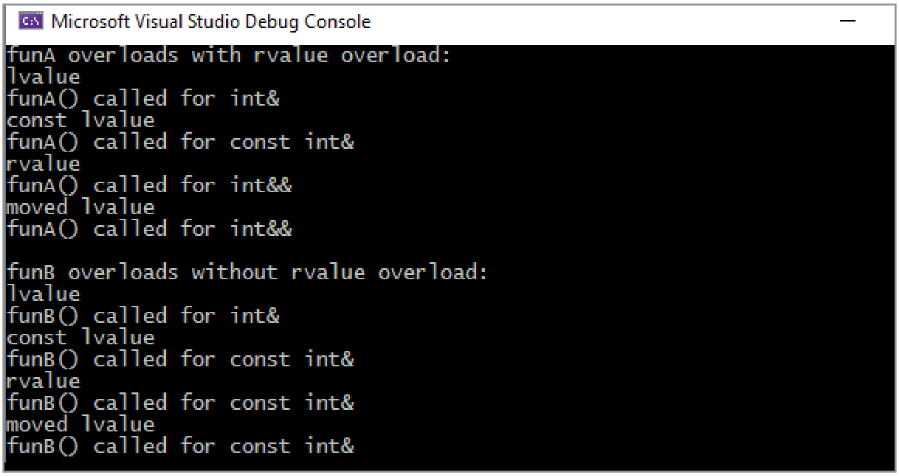
At the end of the description of rvalue references, it is necessary to note the possibility of creating different function overloads for different categories of lvalues and rvalues.

Example 6.

```
#include <iostream>
// funA overloads with rvalue overload
void funA(int& val)
{
    std::cout <<  "funA() called for int&\n ";
}

void funA(const int& val)
{
    std::cout <<  "funA() called for const int&\n ";
}

void funA(int&& val)
{
    std::cout <<  "funA() called for int&&\n ";
}

// funB overloads without rvalue overload
void funB(int& val)
{
    std::cout <<  "funB() called for int&\n ";
}

void funB(const int& val)
{
    std::cout <<  "funB() called for const int&\n ";
}

int main()
{
    int val{ 42 };
    const int cVal{ 26 };
    std::cout <<  "funA overloads with rvalue overload:\n ";
    std::cout <<  "lvalue\n ";
    funA(val); // lvalue -> int&
    std::cout <<  "const lvalue\n ";
```

```cpp
    funA(cVal); // const lvalue -> const int&
    std::cout <<  "rvalue\n ";
    funA(80 + 1); // rvalue -> int&&
    std::cout <<  "moved lvalue\n ";
    funA(std::move(val)); // moved lvalue -> int&&

    std::cout <<  '\n ';
    std::cout <<  "funB overloads without rvalue overload:\n ";
    std::cout <<  "lvalue\n ";
    funB(val); // lvalue -> int&
    std::cout <<  "const lvalue\n ";
    funB(cVal); // const lvalue -> const int&
    std::cout <<  "rvalue\n ";
    funB(80 + 1); // rvalue -> const int&
    std::cout <<  "moved lvalue\n ";
    funB(std::move(val)); // moved lvalue ->const int&

    return 0;
}
```

The output code result of Example 6:



```
funA overloads with rvalue overload:
lvalue
funA() called for int&
const lvalue
funA() called for const int&
rvalue
funA() called for int&&
moved lvalue
funA() called for int&&

funB overloads without rvalue overload:
lvalue
funB() called for int&
const lvalue
funB() called for const int&
rvalue
funB() called for const int&
moved lvalue
funB() called for const int&
```

Figure 7

If there is a separate function overload for the rvalue, calling the function with the rvalue arguments will call it, but if there is no such overloading, the const lvalue overloading will replace it. The ability to “roll back” to a const lvalue overloading in the absence of the rvalue overload will also prove to be very suitable and functional further.

Having learned the new concept of the rvalue references, we can return to the initial problem with copy optimization. Remember that the optimization should eliminate unnecessary copying of a memory block from a temporary object. The solution is to create special overloading of the copy constructor, which will reference the rvalue parameter to a copy of the DynArray class. Such a constructor is called a move constructor.

A move constructor is a particular constructor that allows moving resources owned by an existing temporary class copy into a new copy of the same class omitting the deep copying. An example of that resource would be a block of dynamic memory. The move constructor can be defined explicitly by using a unique signature:

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

Where ClassName means the name of the class for which the constructor is being defined. A move constructor is a constructor that takes a copy of the same class by the rvalue reference. Signature example for the DynArray:

```
DynArray(const DynArray&& object);
```

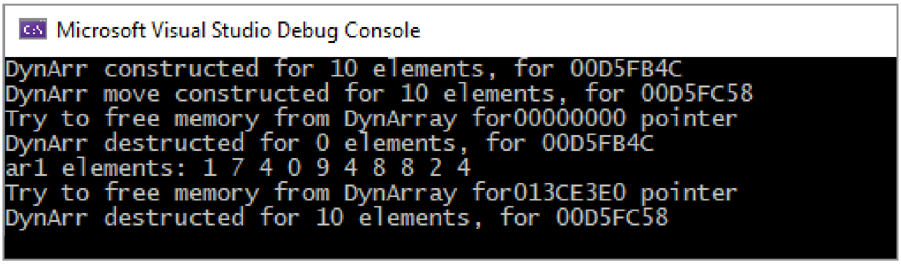
Let’s explicitly define a move constructor for the DynArray class and test it.

Example 7.

```
#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 of
           the same size as in the copied instance of
           the DynArray class. The next loop copies
           the elements from the source 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 ';
    }

    DynArray(DynArray&& object)
        : arr{ object.arr }, size{ object.size }
    /* copy the pointer to the block of dynamic memory
       allocated in the source object, and the size of
       this block to the object initialized by
       the constructor */
    {
        /*  "get the ownership " from the source object with
            the dynamic memory block and set the block
            size to 0 */
        object.arr = nullptr;
        object.size = 0;
        std::cout <<  "DynArr move constructed for  "
                  << size <<  " elements, for  " << this
                  <<  '\n ';
    }

    DynArray& operator=(const DynArray& object)
    {
        // checking for self-assignment
        if (!(this == &object))
        {
            /* checking for the impossibility of  "reusing"
               the storage block allocated for the existing
               array */
            if (size != object.size)
            {
             /* in case of impossibility to  "reuse "
                you need to deallocate the memory occupied
                with the elements of the current dynamic
                array */
                delete[] arr;
                /* allocate a new storage block according
                   to the size of the copied array */
                arr = new int[object.size];
            }
            size = object.size;

            /* an alternative way to copy the array,
               more efficient one in terms of execution
               time due to three additional pointers */
               // pointer to the start of the target copy
               // array
            int* dest{ arr };
            // pointer to the start of the source copy array
            int* src{ object.arr };
            /* a constant pointer to the next after
               the last element in a target array -
                "the end of the source array " */
            int* const end{ arr + size };
            // while the target pointer is not exceeding
               the  "end "...
            while (dest < end)
            {
                /* set the value at the dest pointer
                   address stored at src. Then increment
                   both pointers */
                *dest++ = *src++;
            }
            // array is copied to arr from obj.arr
        }
        std::cout <<  "DynArr copy assigned for  "
                  << size <<  " elements, for  " << this
                  <<  '\n ';
        return *this;
    }

    int getElem(int idx)const { return arr[idx]; }
    void setElem(int idx, int val) { arr[idx] = val; }
    void print()const;
    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 ';
    }
};
```

```cpp
void DynArray::print()const
{
    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;
    }
}

DynArray arrayFactory(int arrSize)
{
    DynArray arr{ arrSize };
    arr.randomize();
    return arr;
}

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

    return 0;

}
```

The output code result of Example 7:



```
DynArr constructed for 10 elements, for 00D5FB4C
DynArr move constructed for 10 elements, for 00D5FC58
Try to free memory from DynArray for00000000 pointer
DynArr destructed for 0 elements, for 00D5FB4C
ar1 elements: 1 7 4 0 9 4 8 8 2 4
Try to free memory from DynArray for013CE3E0 pointer
DynArr destructed for 10 elements, for 00D5FC58
```

Figure 8

Example 7 is identical to Example 2. The only difference is that Example 7 has an explicitly defined move constructor and demonstrates an alternative way of copying an array in the assignment operator to broaden the reader's mind. This copy version compiles to more efficient, faster machine code. Moreover, working with an array through the pointers anticipates our upcoming work with arrays and similar containers in the subsequent parts of our course (topic "Iterators").

Similar to Example 2, a copy of the DynArray class is created inside the function in Example 7 (the first Output line). The result is returned as a temporary object with a copy of DynArray. And since a temporary object is the rvalue, and the DynArray class is overloaded specifically with a constructor, then the corresponding overloading is launched for the rvalue, that is, the move constructor. We should consider the work of the move constructor in detail.

First, we copy a pointer to the dynamic memory block and the number of the elements in the source object into the newly created class copy via the list of initializers. The shallow copying is performed by analogy.

Then, in the constructor's body, we will assign the nullptr value into the pointer to the dynamic memory block of the source object, and the value of zero is assigned to the number of its elements. It became possible because the source object is passed to the move constructor via a non-const rvalue reference, unlike the copy constructor. Here is the previously promised example of modifying an object by applying the rvalue reference. The second Output line shows the work of the move constructor ends with the output of debugging information that the move constructor worked. The copy of the DynArray class, initialized by the move constructor after its completion, uses the dynamic memory block allocated when the class copy was created inside the arrayFactory function without allocating a new block and copying data from the source one into it. Optimization in use!

You may think that all these complexities are not worth the extra copy of 10 int values. Perhaps, it is so. But, in the general case, a dynamic array can store many more elements. Then the overhead of creating a copy will increase significantly, and optimization will no longer seem insignificant.

One crucial question remains: what will happen to the temporary object when the destructor is called? Will any surprises have occurred? No, because the temporary object has been modified with the move constructor. There will be a useless and harmless attempt in the destructor of the temporary object to free the memory at the nullptr pointer. It is even what we see in the third line of the Output. Thus, the temporary object will be released without negative consequences, without affecting the block of dynamic memory allocated and "owned" — the fourth line of the Output.

The next implementation process in our example is the elements output of the array obtained as a work result of the move constructor — the fifth line of the Output. The main function and the test application stop the processing, the lifetime of ar1 ends, and the destructor is called for it, shown in the Output's last line.

Thus, we managed to realize our unrealistic desire to avoid unnecessary actions, such as deep copying from an object that lifetime is terminating. It only clarifies how Example 2 worked without implemented constructor that takes the rvalue reference. And here we should remember the ability to "roll back" to a const lvalue& overloading, in case of absence the rvalue& overloading. And, as you know, the const lvalue& is nothing more than a copy constructor. So, it turns out that in the absence of an explicitly defined move constructor, a copy constructor can take over its role. Although, it is usually less efficient.

2. Applying Move Constructor While Assignment Operator Overloading

Similar to the “pairing” of the copy constructor and the copy assignment operator, there is a “pairing” of the move constructor and the move assignment operator. You will need such an assignment operator to assign the value of a temporary object of the same class to a class copy without deep copying. The signature of such an operator looks as follows:

```
ClassName& operator=(ClassName&& object);
```

Where ClassName is the class name for which the operator is being defined. Signature example for the DynArray:

```
DynArray& operator=(DynArray&& object);
```

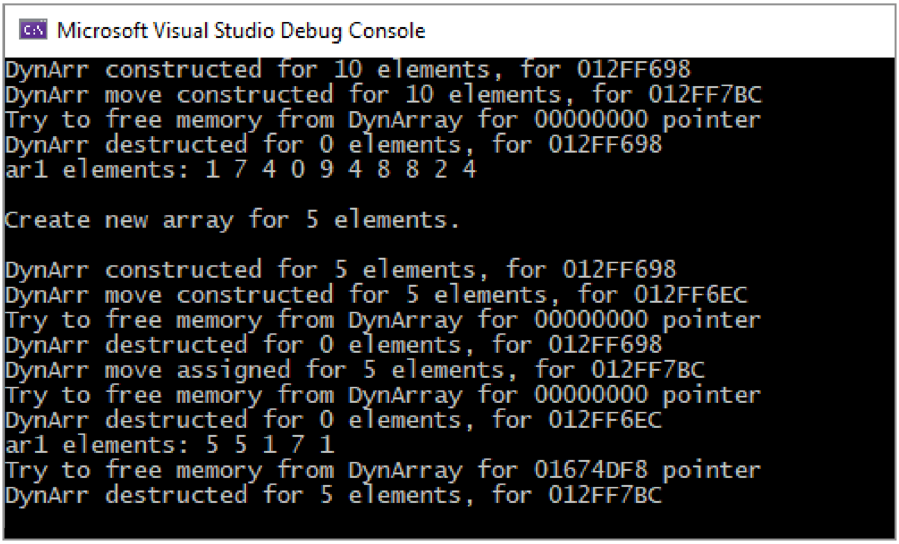
Let’s explicitly define a move assignment operator for the DynArray class and test it.

Example 8.

```
#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 }
    {
        for (int i{ 0 }; i < size; ++i)
        {
            arr[i] = object.arr[i];
        };
        std::cout <<  "DynArr copy constructed for  "
                  << size <<  " elements, for  " << this
                  <<  '\n ';
    }

    DynArray(DynArray&& object)
        : arr{ object.arr }, size{ object.size }
    {
        object.arr = nullptr;
        object.size = 0;
        std::cout <<  "DynArr move constructed for  "
            << size <<  " elements, for  " << this
            <<  '\n ';
    }

    DynArray& operator=(const DynArray& object)
    {
        if (!(this == &object))
        {
            if (size != object.size)
            {
                delete arr;
                arr = new int[object.size];
            }
            size = object.size;
            int* dest{ arr };
            int* src{ object.arr };
            int* const end{ arr + size };

            while (dest < end)
            {
                *dest++ = *src++;
            }
        }

        std::cout <<  "DynArr copy assigned for  "
                  << size <<  " elements, for  " << this
                  <<  '\n ';

        return *this;
    }

    DynArray& operator=(DynArray&& object)
    {
        if (!(this == &object))
        {
            delete arr;

            arr = object.arr;
            size = object.size;

            object.arr = nullptr;
            object.size = 0;

        }
        std::cout <<  "DynArr move assigned for  "
                  << size <<  " elements, for  " << this
                  <<  '\n ';

        return *this;
    }
    int getElem(int idx)const { return arr[idx]; }
    void setElem(int idx, int val) { arr[idx] = val; }
    void print()const;
    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()const
{
    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;
    }
}

DynArray arrayFactory(int arrSize)
{
    DynArray arr{ arrSize };
    arr.randomize();
    return arr;
}
```

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

    std::cout << "\nCreate new array for 5 elements.\n\n ";
    ar1 = arrayFactory(5);
    std::cout << "ar1 elements:  ";
    ar1.print();

    return 0;
}
```

The output code result of Example 8:



Microsoft Visual Studio Debug Console

```
DynArr constructed for 10 elements, for 012FF698
DynArr move constructed for 10 elements, for 012FF7BC
Try to free memory from DynArray for 00000000 pointer
DynArr destructed for 0 elements, for 012FF698
ar1 elements: 1 7 4 0 9 4 8 8 2 4

Create new array for 5 elements.

DynArr constructed for 5 elements, for 012FF698
DynArr move constructed for 5 elements, for 012FF6EC
Try to free memory from DynArray for 00000000 pointer
DynArr destructed for 0 elements, for 012FF698
DynArr move assigned for 5 elements, for 012FF7BC
Try to free memory from DynArray for 00000000 pointer
DynArr destructed for 0 elements, for 012FF6EC
ar1 elements: 5 5 1 7 1
Try to free memory from DynArray for 01674DF8 pointer
DynArr destructed for 5 elements, for 012FF7BC
```

Figure 9

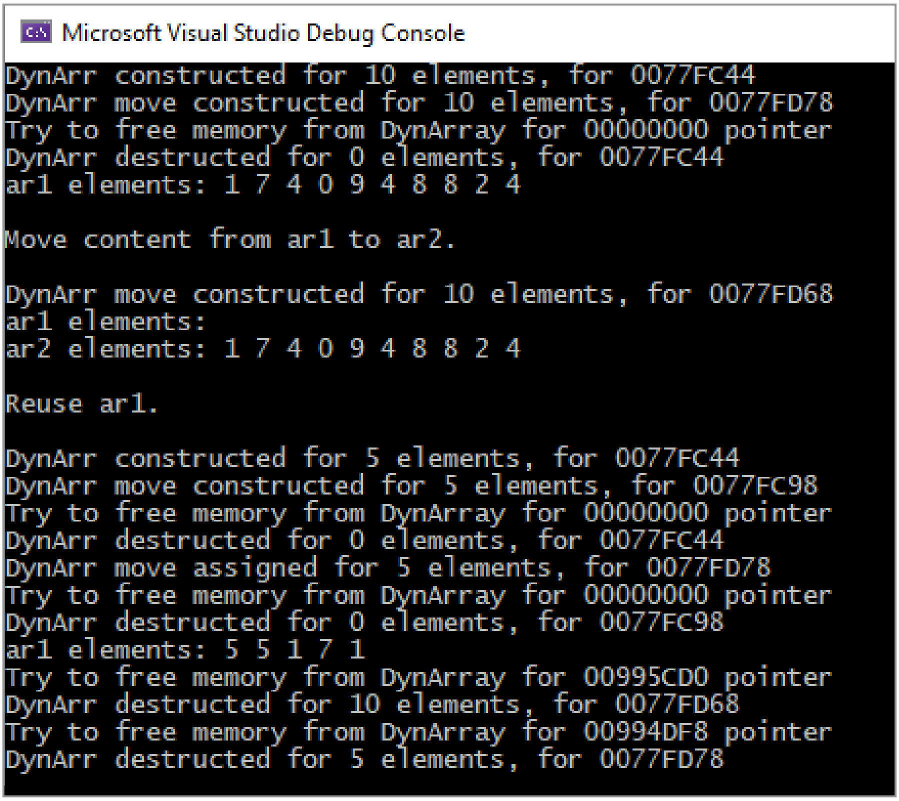
Let’s consider the execution in Example 8. As in Example 7, we create an instance of ar1 at first using the arrayFactory(10) function and output the resulting array's elements. Then, we assign the arrayFactory(5) function result to the existing ar1 copy. Let's take a closer look at how the assignment operator works. The arrayFactory(5) function creates a copy of the DynArray class (line 9 of the Output), fills it with random numbers, and returns it to a temporary object as the result of the function. The temporary object is created using a move in line 10 of the Output. The object's lifetime created with the function ends, and a destructor is executed for it, which does not affect the memory block owned by the object-result of the function (lines 11 and 12 of the Output). Then, the move assignment operator is called. Its implementation is similar to the implementation of the move constructor, but some important differences exist. Since this is not a constructor, creates a completely new object but an assignment operator, it is necessary to check and exclude self-assignment. Then it is important to free the dynamic memory block that the ar1 instance was using before the assignment operator was called. And finally, perform the actions, similar to the move constructor, by returning the ownership of the dynamic memory block from the temporary object (line 13 of the Output). Thus, the dynamic memory block, initially allocated inside the arrayFactory(5) function, having changed two "owners", appeared assigned to the ar1 instance. Next, the lifetime of the temporary function object-result will end, and it will be deleted without affecting the memory block that no longer belongs to it (lines 14 and 15 of the Output). And finally, the array is displayed on the screen.

A pair of a move constructor and a move assignment operator made it possible to avoid inappropriate deep copying twice by implementing move semantics for the DynArray class. Move semantics allows, along with copy semantics, not only to copy one class copy to another but also to move one class object to another, even if the move is not from a temporary object. How can you call the overloading that takes the rvalue reference on a non-temporary named lvalue object? Obviously, with the help of std::move conversions. Let's change the test code in the main function from Example 8.

Example 9.

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

    std::cout <<  "\nMove content from ar1 to ar2.\n\n ";
    DynArray ar2{ std::move(ar1) }; /* ar1 is empty now,
                                       but it is exist. */
    std::cout <<  "ar1 elements:  ";
    ar1.print();
    std::cout <<  "ar2 elements:  ";
    ar2.print();

    std::cout <<  "\nReuse ar1.\n\n ";
    ar1 = arrayFactory(5); /* ar1 is exist, and therefore
                              it can be reused */
    std::cout <<  "ar1 elements:  ";
    ar1.print();

    return 0;

}
```

The output code result of Example 9:



Microsoft Visual Studio Debug Console

```
DynArr constructed for 10 elements, for 0077FC44
DynArr move constructed for 10 elements, for 0077FD78
Try to free memory from DynArray for 00000000 pointer
DynArr destructed for 0 elements, for 0077FC44
ar1 elements: 1 7 4 0 9 4 8 8 2 4

Move content from ar1 to ar2.

DynArr move constructed for 10 elements, for 0077FD68
ar1 elements:
ar2 elements: 1 7 4 0 9 4 8 8 2 4

Reuse ar1.

DynArr constructed for 5 elements, for 0077FC44
DynArr move constructed for 5 elements, for 0077FC98
Try to free memory from DynArray for 00000000 pointer
DynArr destructed for 0 elements, for 0077FC44
DynArr move assigned for 5 elements, for 0077FD78
Try to free memory from DynArray for 00000000 pointer
DynArr destructed for 0 elements, for 0077FC98
ar1 elements: 5 5 1 7 1
Try to free memory from DynArray for 00995CD0 pointer
DynArr destructed for 10 elements, for 0077FD68
Try to free memory from DynArray for 00994DF8 pointer
DynArr destructed for 5 elements, for 0077FD78
```

Figure 10

Before we analyze the actions from Example 9, we need to get to know the features of the std::move function. The only thing that a call to the std::move function does is, given an object of the lvalue or rvalue, performs an unconditional conversion to the rvalue, and that is all. The further object transformations depend on the actions performed with it. The std::move function cannot modify the internal state of its argument in any way.

3. Default Methods and Delete Methods

Among the various member functions that a programmer can create in a class, many such functions have a special meaning. They are the so-called special member functions. If the programmer has not provided his own implementations for the above special member functions, the compiler will generate the default versions. Such member functions include:

- default constructor;
- copy constructor;
- copy assignment operator;
- move constructor;
- move assignment operator;
- destructor.

Generating special member functions allows user-defined classes to interact similarly with C++ base types and structures. It becomes possible to create, delete, copy, and move classes that have not implemented such member functions. It is convenient for simple classes that don't implement something that requires non-trivial versions of special member functions. However, the compiler does not automatically generate all or some of the special member functions. Here are a few rules that govern the automatic generation of special member functions:

- explicitly defining your version of an arbitrary constructor prevents the compiler from automatically generating a default constructor;

- explicitly defining a move constructor or move assignment operator prevents the compiler from automatically developing both the copy constructor and the copy assignment operator;
- explicitly defining a copy constructor, copy assignment operator, move constructor, move assignment operator, or destructor prevents the compiler from automatically generating the move constructor and the move assignment operator.

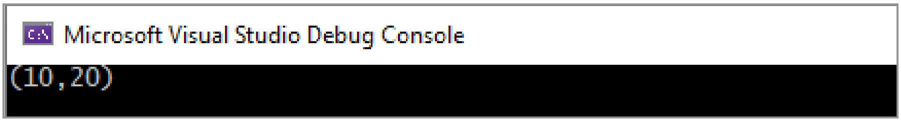
All of the above can also affect the inheritance mechanism. For example, suppose the parent class does not have a default constructor that explicitly defines or compiler-generated constructor with no parameters. In that case, it will be impossible for descendant classes to generate a default constructor automatically.

Let's look at an example illustrated preventing the automatic generation of a default constructor. But first, we should create a simple class that models a point without constructors initially, only with the corresponding setter functions.

Example 10.

```
#include <iostream>

class Point
{
    int x;
    int y;

public:
    Point& setX(int pX) { x = pX; return *this; }
    Point& setY(int pY) { y = pY; return *this; }
    void showPoint() const
    {
        std::cout << '( ' << x << ', ' << y << ') ';
    }
};

int main()
{
    Point p0;
    p0.setX(10).setY(20);
    p0.showPoint();
    std::cout << '\n ';

    return 0;
}
```

The output code result of Example 10:



```
Microsoft Visual Studio Debug Console
(10,20)
```

Figure 11

The example result is evident since there is no explicitly defined default constructor. The compiler generated its own version, which we use when creating the Point p0 class instance. But what happens if we add any constructor with parameters? A constructor like this:

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

The example above will stop compiling with an error.

```
E0291no default constructor exists for class  "Point "
```

That's right. According to the rules above, explicitly defining a constructor with parameters prevented the compiler from automatically generating its version of the default constructor. How can we solve this problem? The obvious solution would be to add an explicit definition of a default constructor with an empty body — a "dummy constructor".

```
Point() {}
```

However, it is more elegant and optimal to use the default keyword to inform the compiler to generate a default version.

```
Point() = default;
```

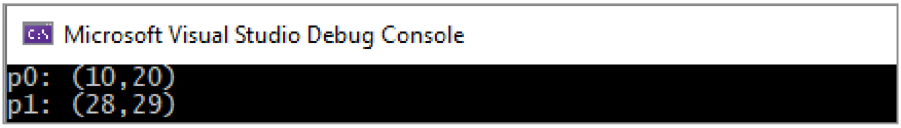
Consider the updated example.

Example 11.

```
#include <iostream>

class Point
{
    int x;
    int y;

public:
    Point() = default;
    Point(int pX, int pY) : x{ pX }, y{ pY } {}
    Point& setX(int pX) { x = pX; return *this; }
    Point& setY(int pY) { y = pY; return *this; }

    void showPoint() const
    {
        std::cout << '( ' << x << ', ' << y << ') ';
    }
};
```

```
int main()
{
    Point p0;
    Point p1{ 28,29 };
    p0.setX(10).setY(20);
    std::cout <<  "p0:  ";
    p0.showPoint();
    std::cout <<  '\n ';
    std::cout <<  "p1:  ";
    p1.showPoint();
    std::cout <<  '\n ';
    return 0;
}
```

The output code result of Example 11:



```
p0:  (10,20)
p1:  (28,29)
```

Figure 12

Now we can use an explicitly defined constructor with parameters to initialize an instance of p1 and a version of the default constructor generated by the compiler to initialize an instance of p0.

Another way to use the default keyword is to explicitly declare a special class member function that the compiler must generate with a non-public access modifier.

```
class Point
{
    int x;
    int y;
    Point() = default;

public:
    Point(int pX, int pY) : x{ pX }, y{ pY } {}
    Point& setX(int pX) { x = pX; return *this; }
    Point& setY(int pY) { y = pY; return *this; }

    void showPoint() const
    {
        std::cout << '( ' << x << ', ' << y << ') ';
    }
};
```

Now you can use the default constructor only inside the Point class, and attempts to use it from outside will lead to errors at the compilation stage.

The use of the 'default' keyword is only possible for special member functions that automatically generate a compiler. For other class member functions, the usage is not possible and is meaningless. The 'default' keyword may not seem very useful or necessary, but it is not so. With each new standard, the C++ language introduces more convenient means to increase the code semantics so that the program works and its text clearly expresses the author's thoughts. The 'default' keyword is one of such innovations that came with the C++11 standard. After all, what does a special member function mean with an empty body? It is a gap: did the author forget to write the body? It is a template: will the body of the corresponding function appear in the next version? And then use of the 'default' keyword represents that this special member function must be automatically generated with the compiler.

The next, the 'delete' keyword, increases the clarity and semantics of the code; it is applied to the arbitrary class member functions and standard functions, which are not members of any class. Its meaning is to prohibit using a function with a given signature explicitly.

In the previous lesson section, we talked about move semantics and its positive aspects in terms of memory efficiency and avoiding unnecessary copying. Having learned move semantics, we may desire to stop copying semantics for some classes. How can we express the prohibition against copying the class copies in a C++ program? The simplest way would be not to declare the copy constructor and the assignment operator by copying. Then, the compiler implicitly prohibits copying if there is a move constructor and/or a move assignment operator, attempts to copy class instances will lead to compilation errors. To represent it on practice, we will consider a familiar example of a dynamic array from which we are going to remove the copy constructor and copy assignment operator:

Example 12:

```
#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(DynArray&& object)
    : arr{ object.arr }, size{ object.size }
{
    object.arr = nullptr;
    object.size = 0;
    std::cout <<  "DynArr move constructed for  "
              << size <<  " elements, for  " << this
              <<  '\n ';
}

DynArray& operator=(DynArray&& object)
{
    if (!(this == &object))
    {
        delete arr;
        arr = object.arr;
        size = object.size;

        object.arr = nullptr;
        object.size = 0;
    }

    std::cout <<  "DynArr move assigned for  "
              << size <<  " elements, for  " << this
              <<  '\n ';
    return *this;
}

int getElem(int idx)const { return arr[idx]; }
void setElem(int idx, int val) { arr[idx] = val; }
void print()const;
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()const
{
    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;
    }
}

DynArray arrayFactory(int arrSize)
{
    DynArray arr{ arrSize };
    arr.randomize();
    return arr;
}

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

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

    return 0;
}
```

When compiling this example, we will get two compilation errors:

```
E1776function  "DynArray::DynArray(const DynArray &) "
     (declared implicitly) cannot be referenced --
     it is a deleted function

C2280 'DynArray::DynArray(const DynArray &) ': attempting
     to reference a deleted function
```

The first error says that the compiler automatically declared implicitly some "remote" version of the constructor that cannot be used. And the second error says that an attempt was made to use a "remote" function. Generally, everything is as it should be, but it is not clear whether the lack of member functions for copying is a mistake or it was intentional.

A quote "Explicit is better than implicit" from the so-called "Zen of Python", a collection of 19 "guiding principles" for writing computer programs that influence the design of the Python programming language, is the best fit in this case. Consider how we can explicitly express the prohibition on copying the class copies. Before the C++11 standard, the classic way to explicitly prohibit copying was to declare the copy constructor and assignment operator as private without defining those member functions.

```
private:
    DynArray(DynArray& object);
    DynArray& operator=(DynArray& object);
```

However, this method is also not without drawbacks. It is still possible to use “hidden” member functions from the class itself, which will lead to errors that are not obvious at first sight. And it is not clear still whether the lack of a definition is a mistake or it was done on purpose.

The clearest and most consistent way is to use the ‘delete’ to indicate explicitly that these member functions are “deleted” and prohibited.

```
public:
    DynArray(DynArray& object) = delete;
    DynArray& operator=(DynArray& object) = delete;
```

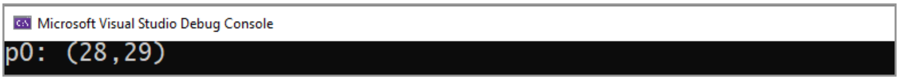
By adding such declarations to Example 12, we will get the same error messages as while we compiled Example 12. The only difference is that the member functions were removed by us this time, not implicitly by the compiler.

Let’s look at another problem where the ‘delete’ can help us apply to a class member function. The Point class will help us:

Example 13

```
#include <iostream>

class Point
{
    int x;
    int y;
public:
    Point() = default;
    Point(int pX, int pY) : x{ pX }, y{ pY } {}
    Point& setX(int pX) { x = pX; return *this; }
    Point& setY(int pY) { y = pY; return *this; }

    void showPoint() const
    {
        std::cout << '( ' << x << ', ' << y << ') ';
    }
};

int main()
{
    Point p0;

    p0.setX(28.02).setY(29.02);
    std::cout << "p0:  ";
    p0.showPoint();
    std::cout << '\n ';

    return 0;
}
```

The output code result of Example 13:



Microsoft Visual Studio Debug Console

```
p0:  (28,29)
```

Figure 13

We used the setX and setY member functions with double literals as the parameters in Example 13. It became possible to call setters declared with int parameters due to the implicit conversion from double to int. Often such an implicit conversion is a desirable one. But, what if the class author wants to explicitly disallow an ambiguous usage of some member functions due to unwanted implicit type conversion? To use 'delete' to explicitly disallow applying a member function with unwanted parameters, of course. Let's add a snippet to the definition of the Point class from Example 13:

```
public:
    Point& setX(double pX) = delete;
    Point& setY(double pY) = delete;
```

Thus, we explicitly prohibited using the double type values as the parameters and implicitly, — through the 'float' type conversion.

Similarly, it is possible to use 'delete' for accessible functions that are not members of any class:

Example 14

```
#include <iostream>

int max(int a, int b) { return a > b ? a: b; }
template <typename T1, typename T2> int max(T1 a, T2 b) =
         delete;

int main()
{
    std::cout << max(20, 30) <<  '\n ';
    std::cout << max(20.5, 30) <<  '\n ';
    std::cout << max( 'z ', false) <<  '\n ';
    std::cout << max( 'a ',  'b ') <<  '\n ';

    return 0;
}
```

Example 14 analysis. It seems a bit difficult to understand at first sight, but we add a few clarifications, and everything will become clear. So, we have defined a free function with two int parameters to find the maximum of two integers. As we remember, all basic C++ types are convertible to each other, so we can potentially call the max function with any combination of the two types of the actual parameters. It is common in some cases, but what if the function developer doesn't want it? How to explicitly prevent a function from being called with parameters other than the int? That is easy, and we should apply to define the function template and add the 'delete' to solve this problem.

Let's declare a 'max' function template with two template parameters of different types and mark it as a deleted with the keyword. When you try to call the max function with two int parameters, the compiler will call the existing max function. However, the compiler will be forced to look for the most appropriate overloading if the max function is called with any other types and combinations of actual parameter types. According to the rules of searching for function overloads, the template will be the most suitable that we have defined. In fact, it will prohibit the use of the function — explicitly and not ambiguously. The only question that could remain is why there are two type parameters in the template, but not the one? When the max function is called with two different actual parameters, such a template will not work according to the same rules for finding the most suitable overload because it is designed for two parameters of the same type. The template from the example will prohibit all, but the first max(20,30) function calls with any combination of actual parameter types.

4. Overloading Special Operators: Operator[]

Learning topic by topic, we mainly use the DynArray as an example. We will turn to it again to represent this part of our lesson. Example 12 contains the latest version of the Dyn-Array class, so let's start with it. At the moment, access to the array elements is implemented by a couple of methods:

```
int getElem(int idx)const { return arr[idx]; }
void setElem(int idx, int val) { arr[idx] = val; }
```

Is such an approach convenient? Definitely not. Splitting into a member function for reading the element and writing is not the most suitable solution. Moreover, you can meet the hesitating when using the setElem member function. For example, does setElem(1,2) mean to set the element at index 1 to 2, or to set the value of 1 to the element at index 2? Without seeing the function signature, it's not clear at all. To correct the situation, accessing elements via the square brackets will help us, which is familiar to us when working with the standard arrays. To do this, we need to overload the operator[] with a member function of the DynArray class. Overloading of this operator is allowed only as a class member function. The operator[] overloading can take only one parameter and return a value via the return statement.

Before moving on to implementing the operator[] overloading, there are two important questions to consider. The first question is, what exactly will the operator[]

overloading return? If the array element is by value, then it will not be possible to use this overloading to modify the array element; in fact, we need this overloading not only for convenient modification. If we return an array element by a non-constant reference, how should we ensure the constness semantics of the array? The solution to this issue will be to implement two overloads of the operator[]: one as a const member function that returns an array element by value, and the second as a regular member function that returns an array element by a non-constant reference.

The second question that we need to solve is whether to check the validity of the array index and, if so, what to do if the index goes out the allowable limits? Lack of index validation in basic C++ arrays is a tribute to performance at the expense of reliability and security. It is believed that the programmer can track the validity of the index. Our array is a training sample, so you should not entirely rely on the student programmers, but you should still check the validity of the index. But, another problem appeared with this solution: what if the index is invalid? What value should we return as an error? There is no appropriate answer at this stage of our training yet. We will get acquainted with the C++ exception mechanism later, which is suitable for solving this problem. At the moment, we will use the assert macros as a temporary solution. The essence of the asserted work is to calculate some logical statement, and if it is false, — display the false message, the string number in the program with the assert macro, and crash the program. To make the asserted output more informative, you can add an arbitrary string through && (and) after the statement is checked, which will also be reflected on the screen if the assert is executed.

The signature of the indexing operator, the "square-brackets-operator", is as follows:

```
ReturnType operator[](IndexType index)const // 1
ReturnType& operator[](IndexType index) // 2
```

where the ReturnType is a type of the return value, and the IndexType is a type of the formal index-parameter. Option 1 is a const member function overloading, and option 2 is non-const overloading.

Example for theDynArray:

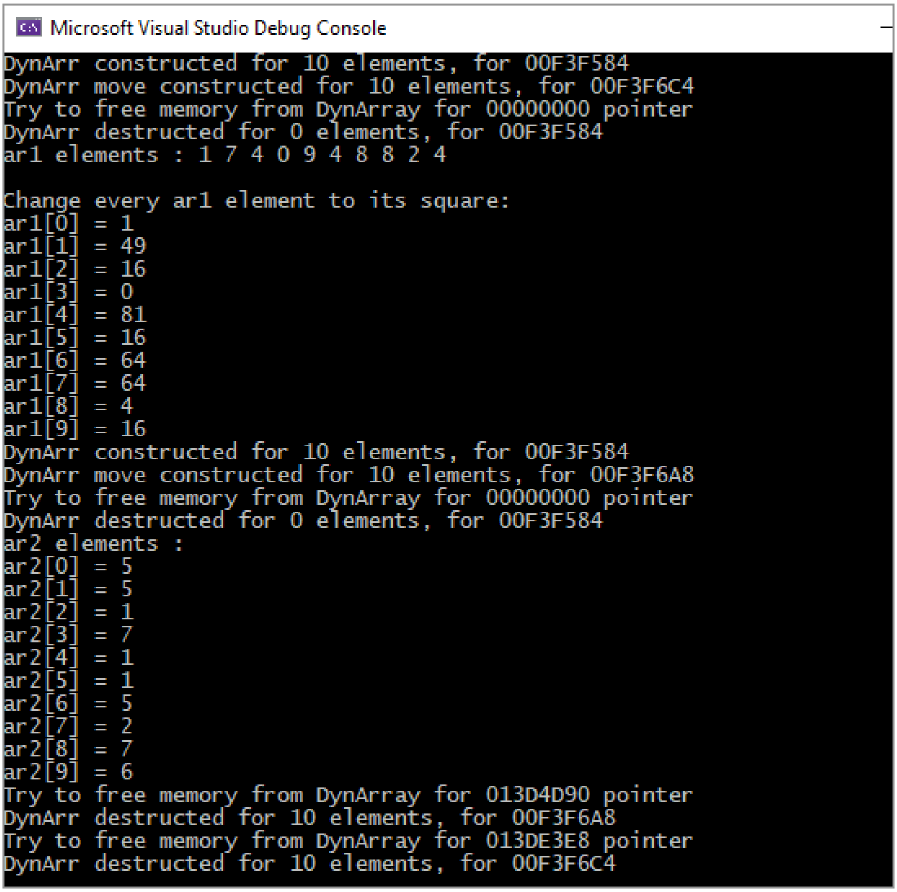
```
int operator[](int idx)const // 1
int& operator[](int idx) // 2
```

Now, having everything you need, you can start implementing the indexing operator for the DynArray class.

Example 15.

```
#include <iostream>
#include <cassert>

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(DynArray& object) = delete;
    DynArray& operator=(DynArray& object) = delete;
    DynArray(DynArray&& object)
        : arr{ object.arr }, size{ object.size }
    {
        object.arr = nullptr;
        object.size = 0;
        std::cout <<  "DynArr move constructed for  "
                  << size <<  " elements, for  " << this
                  <<  '\n ';
    }

    DynArray& operator=(DynArray&& object)
    {
        if (!(this == &object))
        {
            delete arr;
            arr = object.arr;
            size = object.size;
            object.arr = nullptr;
            object.size = 0;
        }

        std::cout <<  "DynArr move assigned for  "
                  << size <<  " elements, for  " << this
                  <<  '\n ';
        return *this;
    }

    // const overloading that returns an element by value
    int operator[](int idx)const
    {
        assert(idx >= 0 and idx < size and “Index is out
                                           of range! ");
        return arr[idx];
    }

    // non-const overloading that returns an element
    // by reference
    int& operator[](int idx)
    {
        assert(idx >= 0 and idx < size and  "Index is out
                                           of range! ");
        return arr[idx];
    }

    void print()const;
    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()const
{
    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;
    }
}

DynArray arrayFactory(int arrSize)
{
    DynArray arr{ arrSize };
    arr.randomize();
    return arr;
}

int main()
{
    const int arrSize{ 10 };
    DynArray ar1{ arrayFactory(arrSize) };
    std::cout << "ar1 elements :  ";
    ar1.print();
    std::cout << "\nChange every ar1 element to its
                  square:\n ";
    for (int i{ 0 }; i < arrSize; ++i)
    {
        ar1[i] *= ar1[i];
        std::cout << "ar1[ " << i <<  "] =  " << ar1[i]
                  << '\n ';
    }

    const DynArray ar2{ arrayFactory(arrSize) };
    std::cout << "ar2 elements :\n ";
    for (int i{ 0 }; i < arrSize; ++i)
    {
       std::cout << "ar2[ " << i <<  "] = " << ar2[i] <<  '\n';
    }

    return 0;
}
```

The output code result of Example 15.



Microsoft Visual Studio Debug Console

```
DynArr constructed for 10 elements, for 00F3F584
DynArr move constructed for 10 elements, for 00F3F6C4
Try to free memory from DynArray for 00000000 pointer
DynArr destructed for 0 elements, for 00F3F584
ar1 elements : 1 7 4 0 9 4 8 8 2 4

Change every ar1 element to its square:
ar1[0] = 1
ar1[1] = 49
ar1[2] = 16
ar1[3] = 0
ar1[4] = 81
ar1[5] = 16
ar1[6] = 64
ar1[7] = 64
ar1[8] = 4
ar1[9] = 16
DynArr constructed for 10 elements, for 00F3F584
DynArr move constructed for 10 elements, for 00F3F6A8
Try to free memory from DynArray for 00000000 pointer
DynArr destructed for 0 elements, for 00F3F584
ar2 elements :
ar2[0] = 5
ar2[1] = 5
ar2[2] = 1
ar2[3] = 7
ar2[4] = 1
ar2[5] = 1
ar2[6] = 5
ar2[7] = 2
ar2[8] = 7
ar2[9] = 6
Try to free memory from DynArray for 013D4D90 pointer
DynArr destructed for 10 elements, for 00F3F6A8
Try to free memory from DynArray for 013DE3E8 pointer
DynArr destructed for 10 elements, for 00F3F6C4
```

Figure 14

Example 15 defines two overloads of the operator[] member function: one overloading is for the non-const objects, and the second is for the const objects. Then, we check the performance of both options. We replace each element for ar1 with its square and, using the same overloading, we display the array on the screen in a “non-standard”, distinctive way, then provided by the print() member function. Similarly, we will check the performance of the constant overloading, displaying the array on the screen.

It is important to note that only the number of formal parameters is defined for overloading the operator[] member function — strictly one, but not its type! The index parameter type can be anything the class designer finds necessary and appropriate.

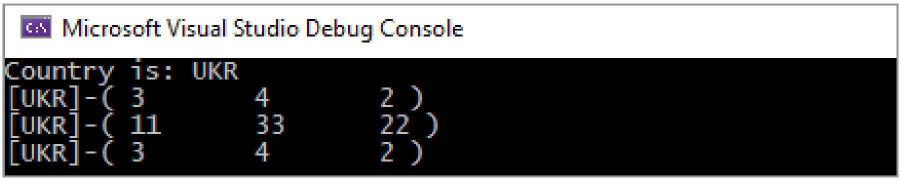
Consider an example with a more non-standard parameter for the operator[] as a string parameter. Our example will create the leaderboard that represents the medals statistic by country. Three-letter abbreviations will be used instead of the full country name. The following example will be a little sketchy, but it is sufficient to represent the possibility of using strings as indexes in a container. For simplifying the example, it will not use dynamic memory. As a result, there will be no need to implement the copy explicitly and move semantics. The compiler automatically generated special member functions will fully cope with the necessary tasks.

To begin with, we need a class that creates the row of our board with medals. This class will store the string abbreviation of the country and the total number for gold, silver, and bronze medals as an integer array.

Example 16.

```
#include <iostream>
#include <cassert>

class MedalRow
{
    char country[4];
    int medals[3];

public:
    /* define constants for convenient and unambiguous
       access to the array elements */
    static const int GOLD{ 0 };
    static const int SILVER{ 1 };
    static const int BRONSE{ 2 };

    MedalRow(const char* countryP, const int* medalsP)
    {
        strcpy_s(country, 4, countryP ? countryP: "NON ");

        for (int i{ 0 }; i < 3; ++i)
        {
            medals[i] = medalsP ? medalsP[i] : 0;
        }
    }

    MedalRow() : MedalRow(nullptr, nullptr) {}
    MedalRow& setCountry(const char* countryP)
    {
        if (countryP)
        {
            strcpy_s(country, 4, countryP);
        }
        return *this;
    }

    const char* getCountry()const { return country; }
    int& operator[](int idx)
    {
        assert((idx >= 0 and idx < 3) and  "Index out
                                           of range! ");
        return medals[idx];
    }
    int operator[](int idx)const
    {
        assert((idx >= 0 and idx < 3) and  "Index out
                                           of range! ");
        return medals[idx];
    }

    void print()const
    {
        std::cout <<  '[ ' << country <<  "]-(  ";

        for (int i{ 0 }; i < 3; ++i)
        {
            std::cout << medals[i];
            if (i < 2) { std::cout <<  '\t '; }
        }
        std::cout <<  " )\n ";
    }
};

int main()
{
    MedalRow mr;
    mr.setCountry( "UKR ");
    std::cout <<  "Country is:  " << mr.getCountry()
              <<  '\n ';

    mr[MedalRow::GOLD] = 3;
    mr[MedalRow::BRONSE] = 2;
    mr[MedalRow::SILVER] = 4;

    /* Create a copy string based on mr */
    MedalRow mr1{ mr };

    /* Certain that a copy is made */
    mr1.print();
    /* Modify the values of the copy string, and display
       it on the screen */
    mr1[MedalRow::GOLD] = 11;
    mr1[MedalRow::BRONSE] = 22;
    mr1[MedalRow::SILVER] = 33;
    mr1.print();

    /* Certain that the source remains unmodified */
    mr.print();

    return 0;
}
```

The output code result of Example 16.



Microsoft Visual Studio Debug Console

```
Country is: UKR
[UKR]-( 3       4       2 )
[UKR]-( 11      33      22 )
[UKR]-( 3       4       2 )
```

Figure 15

Example 16 is also used for two-member function overloading: a const and a non-const for operator[], and the 'assert'. Pay attention to three static constants in the MedalRow class for convenience and better code readability;

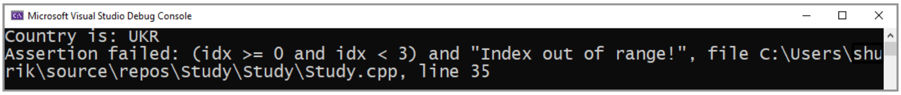
```
static const int GOLD{ 0 };
static const int SILVER{ 1 };
static const int BRONSE{ 2 };
```

Let's test the 'assert' in the operator[] member function overloading by specifying an invalid index for a string with a medal.

Example 17:

```
#include <iostream>
#include <cassert>

class MedalRow
{
    char country[4];
    int medals[3];
public:
    /* define constants for convenient and unambiguous
       access to the array elements */
    static const int GOLD{ 0 };
    static const int SILVER{ 1 };
    static const int BRONSE{ 2 };

    MedalRow(const char* countryP, const int* medalsP)
    {
        strcpy_s(country, 4, countryP ? countryP : "NON  ");
        for (int i{ 0 }; i < 3; ++i)
        {
            medals[i] = medalsP ? medalsP[i] : 0;
        }
    }

    MedalRow() : MedalRow(nullptr, nullptr) {}
    MedalRow& setCountry(const char* countryP)
    {
        if (countryP)
        {
            strcpy_s(country, 4, countryP);
        }
        return *this;
    }

    const char* getCountry()const { return country; }
    int& operator[](int idx)
    {
        assert((idx >= 0 and idx < 3) and "Index out
                                            of range! ");
        return medals[idx];
    }

    int operator[](int idx)const
    {
        assert((idx >= 0 and idx < 3) and "Index out
                                            of range! ");
        return medals[idx];
    }

    void print()const
    {
        std::cout << '[ ' << country << "]-( ";
        for (int i{ 0 }; i < 3; ++i)
        {
            std::cout << medals[i];
            if (i < 2) { std::cout << '\t '; }
        }
        std::cout << " )\n ";
    }
};

int main()
{
    MedalRow mr;
    mr.setCountry( "UKR ");
    std::cout << "Country is: " << mr.getCountry()
              << '\n ';
    mr[MedalRow::GOLD] = 3;
    mr[MedalRow::BRONSE] = 2;
    /* intentional error! There is no element in the array
       with such index!
     * An error will occur while the program is running!
     * It is appropriate for this example.
     */
    mr[8] = 4;

    return 0;
}
```

The output code result of Example 17


Microsoft Visual Studio Debug Console

```
Country is: UKR
Assertion failed: (idx >= 0 and idx < 3) and "Index out of range!", file C:\Users\shu
rik\source\repos\Study\Study\Study.cpp, line 35
```

Figure 16

As you can see from output 17, the macro 'assert' is executed when trying to access an invalid index, causing the program to crash and display the filename and the string with the code where the error occurred. Remember that the macro 'assert' is only a temporary solution until C++ exceptions are explored.

And we continue to create a leaderboard with medals. Let's implement the MedalsTable class. This class will store a static array of 10 elements of the MedalRow type and a size member variable containing the number of the "filled" rows in the table. Initially, the default constructor of MedalRow creates an object where the number of medals of all kinds is 0, and the abbreviation for the country is "NON". Such a row in the medals table will be considered a "dummy string". We create an array of such "dummy strings" in the MedalsTable class by default. And the default constructor sets the size member variable to 0; that is, the table is considered empty.

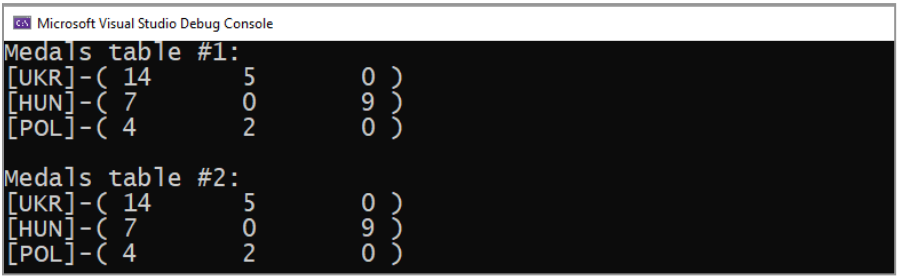
Next, we define two member function overloads for the operator[] to access our table. This is where the most remarkable part begins, for which this example was created. The parameter for the operator[] will not be an integer index but a string! By specifying the country abbreviation string as the actual parameter, the corresponding row of the medal table will be accessed. If a row with such an abbreviation is not in the table and the table size is not exceeded, a new row corresponding to the three-letter abbreviation of the country will be added to it. The non-const overloading returns a reference to the found or added row of the MedalRow type. The const version of the operator[] member function does not have a technique for automatically adding strings to preserve the object's constness semantics. If no string corresponds to the abbreviation, the macro assert is acted, and the process crashes. Also, a const overload returns the MedalRow object by the const reference.

The 'print' member function is provided to display the completed part of the medal table. A private findCountry member function returns the index of the corresponding MedalRow row into the MedalsTable array, or -1 if it is absent, to search for a row by a three-letter abbreviation.

Example 18:

```
#include <iostream>
#include <cassert>

class MedalRow
{
    char country[4];
    int medals[3];
public:
    static const int GOLD{ 0 };
    static const int SILVER{ 1 };
    static const int BRONSE{ 2 };

    MedalRow(const char* countryP, const int* medalsP)
    {
        strcpy_s(country, 4, countryP ? countryP : "NON");
        for (int i{ 0 }; i < 3; ++i)
        {
            medals[i] = medalsP ? medalsP[i] : 0;
        }
    }

    MedalRow() : MedalRow(nullptr, nullptr) {}
    MedalRow& setCountry(const char* countryP)
    {
        if (countryP)
        {
            strcpy_s(country, 4, countryP);
        }
        return *this;
    }

    const char* getCountry()const { return country; }
    int& operator[](int idx)
    {
        assert((idx >= 0 and idx < 3) and  "Index out
                                            of range! ");
        return medals[idx];
    }

    int operator[](int idx)const
    {
        assert((idx >= 0 and idx < 3) and  "Index out
                                            of range! ");
        return medals[idx];
    }

    void print()const
    {
        std::cout <<  '[ ' << country <<  "]-(  ";
        for (int i{ 0 }; i < 3; ++i)
        {
            std::cout << medals[i];
            if (i < 2) { std::cout <<  '\t '; }
        }
        std::cout <<  " )\n ";
    }
};

class MedalsTable
{
public:
    static const int maxSize{ 10 };
private:
    MedalRow medalRows[MedalsTable::maxSize];
    int size;
    int findCountry(const char* country)const
    {
        for (int i{ 0 }; i < size; ++i)
        {
            if (strcmp(medalRows[i].getCountry(),
                country) == 0)
            {
                return i;
            }
        }
        return -1;
    }
public:
    MedalsTable() : size{ 0 } {};
    MedalRow& operator[](const char* country)
    {
        int idx{ findCountry(country) };

        if (idx == -1)
        {
            assert(size < MedalsTable::maxSize and
                   "Table is FULL! ");
            idx = size++;
            medalRows[idx].setCountry(country);
        }

        return medalRows[idx];
    }
    const MedalRow& operator[](const char* country)const
    {
        int idx{ findCountry(country) };
        assert(idx != -1 and  "Country not found
                               on const table ");
        return medalRows[idx];
    }

    void print()const
    {
        for (int i{ 0 }; i < size; ++i)
        {
            medalRows[i].print();
        }
    }
};

int main()
{
    MedalsTable mt1;

    std::cout <<  "Medals table #1:\n ";
    mt1[ "UKR "][MedalRow::GOLD] = 14;
    mt1[ "UKR "][MedalRow::SILVER] = 5;
    mt1[ "HUN "][MedalRow::BRONSE] = 9;
    mt1[ "HUN "][MedalRow::GOLD] = 7;
    mt1[ "POL "][MedalRow::GOLD] = 4;
    mt1[ "POL "][MedalRow::SILVER] = 2;
    mt1.print();
    // create a const copy of the table #1
    std::cout <<  "\nMedals table #2:\n ";
    const MedalsTable mt2{ mt1 };
    mt2.print();

    // by uncommenting the following string you can test
    // a country for its absence in the const table of medals
    // the program will crash, that it is appropriate!
    // mt2[ "SLO "].print();

    return 0;
}
```

The output code result of Example 18



Microsoft Visual Studio Debug Console

```
Medals table #1:
[UKR]-( 14      5       0 )
[HUN]-( 7       0       9 )
[POL]-( 4       2       0 )

Medals table #2:
[UKR]-( 14      5       0 )
[HUN]-( 7       0       9 )
[POL]-( 4       2       0 )
```

Figure 17

So, at the beginning of Example 18, a table of sports medals mt1 is created. Then, using the operator[] overloading with the “UKR” parameter, a new MedalRow is created and returned with a non-const reference. A result is a MedalRow object that also uses the operator[] overloading to specify the number of gold medals. Although the next row looks similar to the previous one but works a little differently. The table already has a MedalRow corresponding to “UKR”, and it is this row that is returned by a non-const reference as the result of mt1[“UKR”]. Then, similar to the previous row, the number of silver medals is set using the operator[] overloading for MedalRow. After then, everything is implemented similarly for “HUN” and “POL”. Calling the mt1.print() member function, the completed part of the sports board is displayed on the screen. Based on mt1, we create a copy mt2 table, and its content is displayed on the screen to check the correct operation of a const copy of the MedalsTable class. By uncommenting the line mt2[“SLO”].print(); you can test the access to a non-existent element of a const object: the macro assert will work, and the program will crash. Thus, we observed the possibility of using various types of actual parameters to implement the operator[] overloading shown in Example 18.

Summarizing all of the above, following the next. For implementing access to the elements of a certain container, it is common to use the square brackets as the indexing operator as a rule. Overloading the indexing operator is implemented by defining a member function of the operator[] class. This operator can only be overloaded by a class member function, as a requirement of C++ standard. It is possible to create two versions of the operator[] overload to provide const semantics: a const and a non-const. This overloading has only one formal parameter of arbitrary type: there is no restriction on the integer parameter, although it is the integer parameter that is most often used.

5. Overloading Special Operators: Operator()

It’s time to learn one more special operator in C++ for overloading, known as a function call operator. Otherwise, it is a parentheses operator or the operator(). A class member function can only overload such an operator. Also, operator() has many features that open up interesting possibilities for its application.

So, let’s begin with the features of the function call operator. Almost all overloaded operators have a strictly defined number of arguments, in some cases even regulating the type of arguments, leaving only the return value type at the discretion of the class developer. However, the function call operator is not like that. How many arguments an arbitrary function has in general? Arbitrary quantity. So in the case of overloading the function call operator, the number of arguments and their type and order are not regulated. This feature allows you to implement operators with an atypical, particular purpose.

What is the purpose of overloading the function call operator? It is a challenging question. On the one hand, great flexibility gives excellent opportunities, but on the other hand, it requires a responsible approach while using. The semantics of standard operators is formalized a bit. For example, the addition operator carries out addition, merge, union, concatenation. The equivalence test operator (operator==) checks the equivalence of two class copies according to specific criteria. And what is the semantics of the function call operator? It is public (open), and it does not correlate with an arbitrary class.

On the one hand, the public (open) class allows arbitrary implementation. Still, on the other hand, the essence and the meaning of the actions implemented by overloading the function call operator are not clear. Even worse, you can mix up calling an overloaded function call operator for a class copy with calling a function. Therefore, we recommend not to use the overloadings of this operator excessively to avoid ambiguity in the code.

It could complete the description without even starting it. Still, we will consider a practical example of a semantically overloaded function call operator, based on a two-dimensional dynamic array class. And we will finish the section with the creation of functors as a completely new applying the function call operator.

Let's start with the signature and general form of represented overloading:

```
ReturnType operator()() {} // 1
ReturnType operator()(ParamTypeA paramA) {} // 2
ReturnType operator()(ParamTypeA paramA,
                      ParamTypeB paramB) {} // 3
```

where the ReturnType is a type of the return value, and the ParamTypeA and ParamTypeB are formal parameters. Option 1 is overloading a member function without parameters. Option 2 — overloading with one proper parameter. Option 3 is overloading with two parameters. Again, let us note that there can be an arbitrary number of formal parameters when overloading the function call operator. And parentheses appear twice when defining overloading, first as a part of the operator name and then — as parentheses for formal parameters.

The following example for overloading returns an integer and takes two formal integer parameters.

```
int operator()(int y, int x){}
```

Let's move on to an application example that illustrates the semantically reasonable use of overloading the function call operator. The example will not be entirely standard for the two-dimensional dynamic array class. Mainly, a two-dimensional dynamic array is created in two steps: first, memory is dynamically allocated for the row container of our array, and then memory is allocated one by one for individual rows of a two-dimensional dynamic array. The result is a construction that is no different from a two-dimensional static array at the stage of use. The same access to elements with the square bracket operator, which must be used twice: to access an array of elements from the container, and as a second time, — access directly to the row array element.

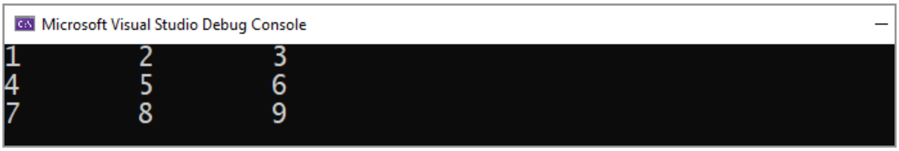
Let's give a simplified example of a class that implements a technique for creating a two-dimensional dynamic array described above.

Example 19.

```
#include <iostream>

class Dyn2DArr
{
    int sizeY;
    int sizeX;
```

```
public:
    int** data;
    Dyn2DArr(int sizeYP, int sizeXP)
        : sizeY{ sizeYP }, sizeX{ sizeXP },
          data{ new int*[sizeYP] }
    {
        for (int y{ 0 }; y < sizeY; ++y)
        {
            data[y] = new int[sizeX];
        }
    }

    void print()const
    {
        for (int y{ 0 }; y < sizeY; ++y)
        {
            for (int x{ 0 }; x < sizeX; ++x)
            {
                std::cout << data[y][x] <<  '\t ';
            }
            std::cout <<  '\n ';
        }
        std::cout <<  '\n ';
    }

    ~Dyn2DArr()
    {
        for (int y{ 0 }; y < sizeY; ++y)
        {
            delete[] data[y];
        }
        delete[] data;
    }
};

int main()
{
    int rows{ 3 };
    int columns{ 3 };
    int counter{ 1 };

    Dyn2DArr arr2d{ rows, columns };

    for (int y{ 0 }; y < rows; ++y)
    {
        for (int x{ 0 }; x < columns; ++x)
        {
            arr2d.data[y][x] = counter++;
        }
    }

    arr2d.print();

    return 0;
}
```

The output code result of Example 19.



```
Microsoft Visual Studio Debug Console
1       2       3
4       5       6
7       8       9
```

Figure 18

In example 19, we made the 'data' member variable public, which is not suitable for the state class encapsulating. But it was done to preserve the methodology for accessing array elements through double square brackets. Standard access to elements with square brackets is used in the 'print' member function and the 'main' function while filling the array with test values. The bigger problem than the 'data' public member variable represents the methodology for creating a two-dimensional dynamic array through multiple allocation/deallocation of a dynamic memory using 'new'/'delete'. As you know, calling the 'new'/'delete' leads to loss that should be preferably minimized.

To eliminate multiple allocation/deallocation of memory, let's radically change the data model for a two-dimensional dynamic array. Instead of allocating memory for the actual array elements in parts, we will allocate memory in one large block so that the size of the block can accommodate all the elements of a two-dimensional array at once. However, sticking to this approach, we will have lost the usual way of accessing the elements with square brackets because there is only one memory block. This issue could be solved by creating a container, as in Example 19, and filling it with the addresses of the corresponding elements row by row from a memory block.

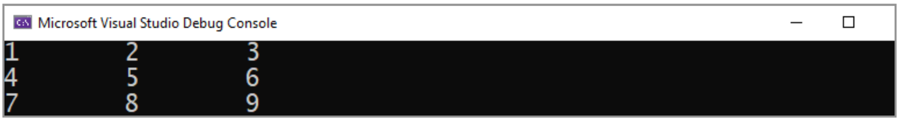
Here is the modified version of the class:

Example 20.

```
#include <iostream>

class Dyn2DArrLinear
{
    int sizeY;
    int sizeX;

public:
    int** data;
    Dyn2DArrLinear(int sizeYP, int sizeXP)
        : sizeY{ sizeYP }, sizeX{ sizeXP },
          data{ new int*[sizeYP] }
    {
        /* allocate a memory block to store all
         * the elements two-dimensional dynamic array.
         */
        int* dataElements{ new int[sizeY * sizeX] };
        for (int y{ 0 }; y < sizeY; ++y)
        {
            // set the block row by row
            data[y] = dataElements + y * sizeX;
        }
    }

    void print()const
    {
        for (int y{ 0 }; y < sizeY; ++y)
        {
            for (int x{ 0 }; x < sizeX; ++x)
            {
                std::cout << data[y][x] <<  '\t ';
            }
            std::cout <<  '\n ';
        }
        std::cout <<  '\n ';
    }

    ~Dyn2DArrLinear()
    {
        /* address of starting a large dataElements block
         * in the constructor matches the address of
         * the first row of the two-dimensional dynamic
         * array.
         * We free the memory from the array elements
         * first, and we free row container memory after then.
         */
        delete[] data[0];
        delete[] data;
    }
};
```

```
int main()
{
    int rows{ 3 };
    int columns{ 3 };
    int counter{ 1 };

    Dyn2DArrLinear arr2d{ rows, columns };

    for (int y{ 0 }; y < rows; ++y)
    {
        for (int x{ 0 }; x < columns; ++x)
        {
            arr2d.data[y][x] = counter++;
        }
    }
    arr2d.print();

    return 0;
}
```

The output code result of Example 20.



Microsoft Visual Studio Debug Console

```
1        2        3
4        5        6
7        8        9
```

Figure 19

As you can see, Example 20 has no functional difference from Example 19. We just got rid of multiple 'new'/'delete' due to accessing the elements with the square brackets. However, this usual method leads to allocating an additional memory block for the container of rows addresses and to double dereferencing: the row address at first and the direct element after then.

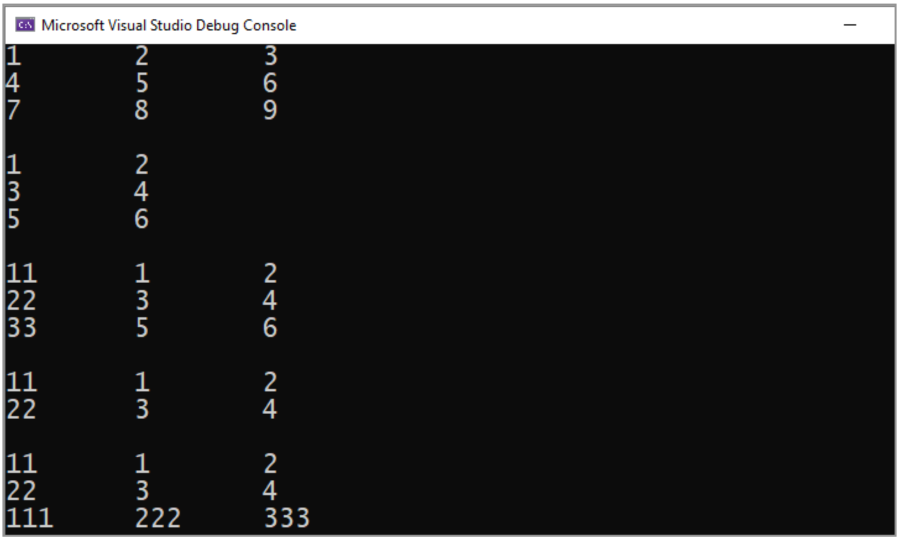
Let's see how we can eliminate the drawbacks in Examples 19 and 20. We will abandon using the row container to do this, leaving the allocated memory block for the elements as a single unbroken fragment. As a result, we will lose access to elements using double square brackets. The two-dimensional array model requires "two-dimensional" access to elements by row index and column index. As we know from the previous section, overloading the square bracket operator to specify two indices will fail because operator[] gets only one formal index parameter. And this is where the function call operator feature comes to the rescue: the possibility of using an arbitrary number of standard parameters.

We will create a Matrix class to access the elements of a linearly allocated memory block as a two-dimensional array. Functionally, it repeats the previous classes because its data member variable has no need to make it public, which fixes the encapsulation problem. Likewise, there are methods of adding/deleting rows/columns in the Matrix class example to demonstrate the simplicity and efficiency of these utility subtasks, taking into account the linearity of the data block. The class implements two overloads of the function call operator: a constant and a non-constant one, to access the essential elements of our two-dimensional array with the most convenient semantics.

Example 21.

```
#include <iostream>

class Matrix
{
    int sizeY;
    int sizeX;
    int* data;
    int index2D(int y, int x)const { return y * sizeX + x;
}
    int index2D(int y, int x, int sizeXP)
        const { return y * sizeXP + x; }
public:
    Matrix(int sizeYP, int sizeXP)
        : sizeY{ sizeYP }, sizeX{ sizeXP },
          data{ new int[sizeYP * sizeXP] }{}

    int operator()(int y, int x)
        const { return *(data + index2D(y, x)); }
    int& operator()(int y, int x)
        { return *(data + index2D(y, x)); }

    void deleteColumn(int columnPos)
    {
        --sizeX;
        int* newData{ new int[sizeY * sizeX] };
        for (int y{ 0 }; y < sizeY; ++y)
        {
            for (int x{ 0 }; x < sizeX; ++x)
            {
                *(newData + index2D(y, x)) = *(data +
                  index2D(y, x + (x >= columnPos)));
            }
        }
        delete[] data;
        data = newData;
    }

    void addColumn(int columnPos, int* newCol = nullptr)
    {
        int* newData{ new int[sizeY * (sizeX + 1)] };
        for (int y{ 0 }; y < sizeY; ++y)
        {
            for (int x{ 0 }; x < sizeX; ++x)
            {
                *(newData + index2D(y, x + (x >=
                  columnPos), sizeX + 1)) = *(data +
                  index2D(y, x));
            }
            *(newData + index2D(y, columnPos, sizeX + 1))
              = newCol ? *(newCol + y) : 0;
        }
        delete[] data;
        data = newData;
        ++sizeX;
    }

    void deleteRow(int rowPos)
    {
        --sizeY;
        int* newData{ new int[sizeY * sizeX] };
        for (int y{ 0 }; y < sizeY; ++y)
        {
            for (int x{ 0 }; x < sizeX; ++x)
            {
                *(newData + index2D(y, x)) = *(data +
                  index2D(y + (y >= rowPos), x));
            }
        }
        delete[] data;
        data = newData;
    }

    void addRow(int rowPos, int* newRow = nullptr)
    {
        int* newData{ new int[(sizeY + 1) * sizeX] };
        for (int y{ 0 }; y < sizeY; ++y)
        {
            for (int x{ 0 }; x < sizeX; ++x)
            {
                *(newData + index2D(y + (y >= rowPos), x))
                  = *(data + index2D(y, x));
            }
        }

        for (int x{ 0 }; x < sizeX; ++x)
        {
            *(newData + index2D(rowPos, x)) =
              newRow ? *(newRow + x) : 0;
        }
        delete[] data;
        data = newData;
        ++sizeY;
    }

    void print()const
    {
        for (int y{ 0 }; y < sizeY; ++y)
        {
            for (int x{ 0 }; x < sizeX; ++x)
            {
                std::cout << (*this)(y, x) <<  '\t ';
            }
            std::cout <<  '\n ';
        }
        std::cout <<  '\n ';
    }

    ~Matrix() { delete[] data; }
};

int main()
{
    /* Set USER_INPUT into 1 to allow
       user input for matrix size */
#define USER_INPUT 0;
    int rows{ 3 };
    int columns{ 3 };
    int counter{ 1 };

#if USER_INPUT == 1
    std::cout <<  "Enter matrix rows count\n ";
    std::cin >> rows;
    std::cout <<  "Enter matrix columns count\n ";
    std::cin >> rows;
#endif

    Matrix matrix{ rows, columns };
    for (int y{ 0 }; y < rows; ++y)
    {
        for (int x{ 0 }; x < columns; ++x)
        {
            matrix(y, x) = counter++;
        }
    }
    matrix.print();

    matrix.deleteColumn(2);
    matrix.print();

    int* newColumn{ new int[columns] {11,22,33} };
    matrix.addColumn(0, newColumn);
    matrix.print();

    matrix.deleteRow(2);
    matrix.print();

    int* newRow{ new int[rows] {111,222,333} };
    matrix.addRow(2, newRow);
    matrix.print();

    delete[] newRow;
    delete[] newColumn;
    return 0;
}
```

The output code result of Example 21.



Microsoft Visual Studio Debug Console

```
1          2          3
4          5          6
7          8          9

1          2
3          4
5          6

11         1          2
22         3          4
33         5          6

11         1          2
22         3          4

11         1          2
22         3          4
111        222        333
```

Figure 20

The operations of Example 21 require some explanations. The critical data model in a class is an ordinary linearly allocated memory block as a kind of one-dimensional array. There are two-member function overloadings of index2D in the class Matrix for getting a linear index by a pair of indices, as in a two-dimensional array. The first overloading uses the row size from the sizeX member variable to calculate the index. And the second overloading allows us to specify coefficients of the row size corrections. Such overloading is used in member functions of adding/removing columns.

The member functions described above use overloadings of the function call operator to access elements stored in a linearly allocated memory block with two indices: row and column. The need of two overloadings is due to the need to implement constancy semantics, as in the example from the previous section on overloading the element access operator by index with operator[]. Overloading special operator() allows you to access an element of a matrix, a two-dimensional array, by specifying the row and column index in parentheses after the class instance name.

```
Matrix matrix{ 3,3 };
matrix(1, 2) = 42;
```

Function call operator overloading for a Matrix class is semantically justified but requires clarification before using it. This method of accessing elements is used inside the Matrix class in a 'print' member function and the 'main' function, filling the array with test values.

To test the functionality of a Matrix class in example 21, we created a class copy and fill it with the test values. Next, we removed/added a column, the same for a row after then. Output 21 is shown the correct functioning of the features implemented in the class.

Also extremely interesting, although it goes beyond the lesson's topic, is the question of comparing methods for creating a two-dimensional dynamic array in terms of efficiency depending on various options for the subsequent use of a two-dimensional dynamic array. We recommend considering it separately.

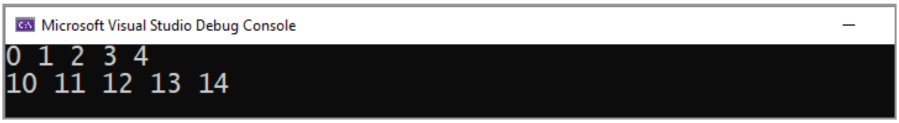
Using function call operator overload to access elements in container classes is an application example of using the operator(). However, the functor classes are increasing even more interest and semantic meaning.

The functor class implements the operator() overloading. Copies of this class are used as ordinary functions but with several unique features and capabilities. Let's start with a simple example of a functor class that implements a counter. Each function call returns the next integer in sequence. The class constructor can get the starting value of the counter as a formal parameter, and the default constructor creates a counter starting from zero. The resetTo member function will allow us to change the countdown position to the value specified by the formal parameter.

Example 22.

```
#include <iostream>

class Counter
{
    int cnt;
public:
    Counter(int start) : cnt{ start } {};
    Counter() : Counter(0) {};

    int operator()() { return cnt++; }
    void resetTo(int start) { cnt = start; }
};

int main()
{
    const int maxCnt{ 5 };
    Counter cnt1{};

    for (int i{ 0 }; i < maxCnt; ++i)
    {
        std::cout << cnt1() << ' ';
    }
    std::cout << '\n';
    cnt1.resetTo(10);

    for (int i{ 0 }; i < maxCnt; ++i)
    {
        std::cout << cnt1() << ' ';
    }

    std::cout << '\n';
    return 0;
}
```

The output code result of Example 22.



Microsoft Visual Studio Debug Console

```
0 1 2 3 4
10 11 12 13 14
```

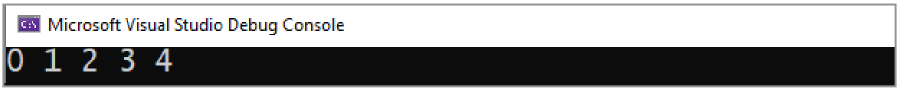
Figure 21

The operations of the shown example are not difficult to understand. The cnt member variable is in the Counter class. Its value is returned, incrementing any time, overloading the function call operator. Having created a class copy, we call the cnt1() function to get the next counter value. Does it look complicated, or is it easy to realize? After all, we can create a function with a static variable with the same effect. Or not the same?

Example 23.

```
#include <iostream>

int cnt()
{
    static int counter{ 0 };
    return counter++;
}

int main()
{
    const int maxCnt{ 5 };

    for (int i{ 0 }; i < maxCnt; ++i)
    {
        std::cout << cnt() << ' ';
    }
    std::cout << '\n ';

    return 0;
}
```

The output code result of Example 23.



Microsoft Visual Studio Debug Console

```
0 1 2 3 4
```

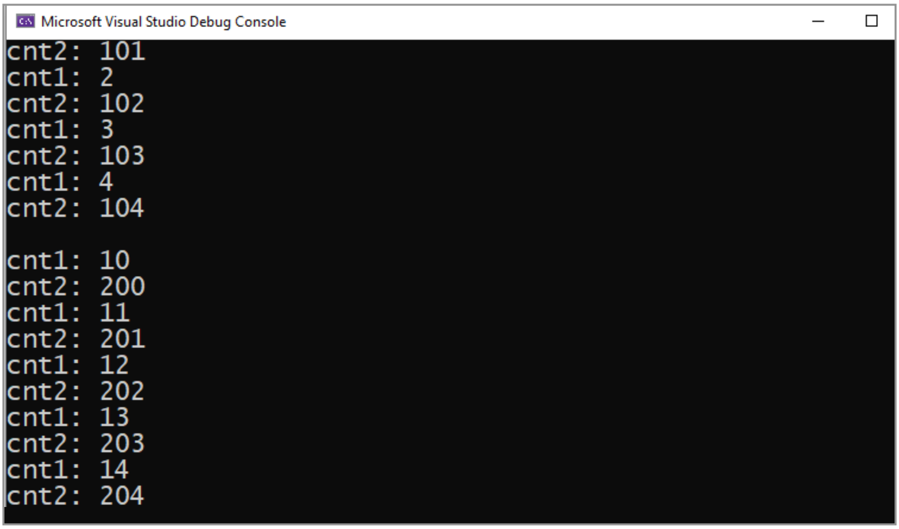
Figure 22

A function with a static variable only partially implements the functor class's capabilities. After all, there is only one function, and a static variable is also in it. After all, there is no convenient way to set the value of a static variable from outside the function. It is impossible to have several different counters using a function with an inert variable cause of only one static variable! The functor does not contain all these drawbacks. You should consider a functor as a function that can have a state. Suppose function has a state condition with the same function call and actual (or without any) parameters. In that case, it will get a different result, based on the condition and not only on the actual parameters. Functors also have a convenient and flexible way to get/modify the stored state if necessary.

Example 24.

```
#include <iostream>

class Counter
{
    int cnt;
public:
    Counter(int start) : cnt{ start } {};
    Counter() : Counter(0) {};

    int operator()() { return cnt++; }
    void resetTo(int start) { cnt = start; }
};

int main()
{
    const int maxCnt{ 5 };
    Counter cnt1{};
    Counter cnt2{100};

    for (int i{ 0 }; i < maxCnt; ++i)
    {
        std::cout <<  "cnt1:  " << cnt1() <<  '\n ';
        std::cout <<  "cnt2:  " << cnt2() <<  '\n ';
    }
    std::cout <<  '\n ';
    cnt1.resetTo(10);
    cnt2.resetTo(200);

    for (int i{ 0 }; i < maxCnt; ++i)
    {
        std::cout <<  "cnt1:  " << cnt1() <<  '\n ';
        std::cout <<  "cnt2:  " << cnt2() <<  '\n ';
    }
    std::cout <<  '\n ';

    return 0;
}
```

The output code result of Example 24.


Microsoft Visual Studio Debug Console

```
cnt2:  101
cnt1:  2
cnt2:  102
cnt1:  3
cnt2:  103
cnt1:  4
cnt2:  104

cnt1:  10
cnt2:  200
cnt1:  11
cnt2:  201
cnt1:  12
cnt2:  202
cnt1:  13
cnt2:  203
cnt1:  14
cnt2:  204
```

Figure 23

Thus, we created two independent counters in Example 24 with its initial count in each of them. The counters were also restarted to new reference points. At the same time, getting the next counter value is just a "function call". Nothing like this can be achieved using the function from Example 23!

Let's consider one more example. We will implement several function templates using the gained knowledge about function templates, pointers, and pointers to functions. The first will be the function template output for a one-dimensional array. The print template takes a pointer to the beginning of the array and a pointer to the next after the last element of the array. A separator will be displayed between the elements of the array — optionally. The copy_if template is a little more complicated. It gets some pointers at the beginning/end (the end is also in the form of the next after the last element of the array) of the source array, and a predicate function that will be called for each element of the source array; and only in case of returning the 'true' from this function, the element will be copied to the destination array. The copy_if template is the same counts the number of copied elements filtered by the predicate function.

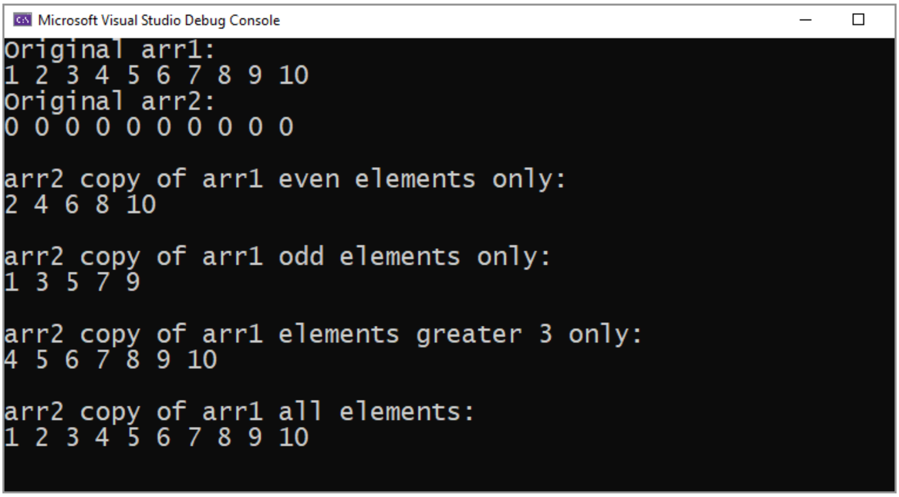
The practical meaning and advantage of using the copy_if template consist in a generalized solution of the problem by copying the elements of the source array to the destination array according to some conditions. Instead of implementing its copying variant for each condition, the condition itself becomes a generalized copying algorithm parameter, a conditionally independent one. The copy conditions will be the standard functions that take an integer as a formal parameter and return true/false. The 'even' function is presented for copying the even values only, 'odd' — for odd values, 'greater3' — for values greater than 3, and 'all' — for copying all values.

Example 25.

```
#include <iostream>

template <typename T>
void print(T* begin, T* end, char delimiter = '  ')
{
    while (begin != end)
    {
        std::cout << *begin++ << delimiter;
    }
    std::cout <<  '\n ';
}

template <typename T, typename Predicate>
int copy_if(T* srcB, T* srcE, T* destB, T* destE,
            Predicate pred)
{
    int copyCount{ 0 };
    while (destB != destE and srcB != srcE)
    {
        if (pred(*srcB))
        {
            *destB++ = *srcB;
            ++copyCount;
        }
        ++srcB;
    }
    return copyCount;
}

bool odd(const int el)
{
    return el % 2 != 0;
}

bool even(const int el)
{
    return el % 2 == 0;
}

bool greater3(const int el)
{
    return el > 3;
}
```

```
bool all(const int el)
{
    return true;
}

int main()
{
    const int size{ 10 };
    int arr1[size]{ 1,2,3,4,5,6,7,8,9,10 };
    int arr2[size]{};

    /*
     * pointer to the first element, the arr1 -
     * arr1Begin is the beginning of the array
     * pointer to the past-the-end element, arr1 -
     * arr1End is the end of the array
     */
    int* const arr1Begin{ arr1 };
    int* const arr1End{ arr1 + size };

    /*
     * pointer to the first element, arr2 - arr2Begin is
     * the beginning of the array pointer to the past-
     * the-end element, arr2 - arr2End is the end
     * of the array
     */
    int* const arr2Begin{ arr2 };
    int* const arr2End{ arr2 + size };

    /*
     * pointer to the past-the-end element copied
     * the arr2NewEnd into the arr2
     */
    int* arr2NewEnd{};

    std::cout <<  "Original arr1:\n ";
    print(arr1, arr1 + size);
    std::cout <<  "Original arr2:\n ";
    print(arr2, arr2 + size);
    std::cout <<  '\n ';

    std::cout <<  "arr2 copy of arr1 even elements
                   only:\n ";
    arr2NewEnd = arr2Begin + copy_if(arr1Begin, arr1End,
                 arr2Begin, arr2End, even);
    print(arr2, arr2NewEnd);
    std::cout <<  '\n ';

    std::cout <<  "arr2 copy of arr1 odd elements
                   only:\n ";
    arr2NewEnd = arr2Begin + copy_if(arr1Begin, arr1End,
                 arr2Begin, arr2End, odd);
    print(arr2, arr2NewEnd);
    std::cout <<  '\n ';

    std::cout <<  "arr2 copy of arr1 elements greater
                   3 only:\n ";
    arr2NewEnd = arr2Begin + copy_if(arr1Begin, arr1End,
                 arr2Begin, arr2End, greater3);
    print(arr2, arr2NewEnd);
    std::cout <<  '\n ';

    std::cout <<  "arr2 copy of arr1 all elements:\n ";
    arr2NewEnd = arr2Begin + copy_if(arr1Begin, arr1End,
                 arr2Begin, arr2End, all);
    print(arr2, arr2NewEnd);
    std::cout <<  '\n ';

    return 0;


}
```

The output code result of Example 25.



```
Original arr1:
1 2 3 4 5 6 7 8 9 10
Original arr2:
0 0 0 0 0 0 0 0 0 0

arr2 copy of arr1 even elements only:
2 4 6 8 10

arr2 copy of arr1 odd elements only:
1 3 5 7 9

arr2 copy of arr1 elements greater 3 only:
4 5 6 7 8 9 10

arr2 copy of arr1 all elements:
1 2 3 4 5 6 7 8 9 10
```

Figure 24

Let's analyze Example 25. A couple of test arrays is created. The arr1 filled in with values from 1 to 10. The arr2 was created for copying elements according to some conditions. After then, the conditional copying functionality is tested in the form of a predicate function using the described function templates and functions. Output 25 indicates the correctness of the program.

Note that two template type parameters are included in the copy_if template. The first is a type parameter to generalize the type of the array elements, and the second type is a parameter to generalize the type of the predicate function. We have not specified the type of the predicate function as a pointer to a function as a parameter that returns bool and accepts a T type argument, the first type parameter in the template. You will get to know the reason for this decision further.

Using predicate functions and a generic copy algorithm is an elegant solution, but is generic copy capable of more complex tasks? For example, to copy the source array elements into the destination array without identical elements following one-by-one. Only the first element should be copied, but not exact copies of the elements following one-by-one. Or another task is to copy the elements to the destination array so that their sum does not exceed a particular value. To write the appropriate function predicates would seem like the problem is solved! However, it is not easy to write a common function to solve this task. It is because now the predicate function cannot determine the need for copying only by the value of the next element; this is not enough. You will need a retrospective view of information about the previous elements and their conditions. Of course, it is possible to write a function expending some effort that uses static variables for solving this task. But this is entirely massive, inflexible, and it's just not an elegant solution. And since there is an opinion that what looks beautiful and works even better, we will use just such an attractive solution.

And it is possible to resolve the above task elegantly due to a functor, a functional object with a specific condition. In this case, the presence of the condition couldn't be more appropriate for us.

Let's implement the NoSequence functor to solve the first task:

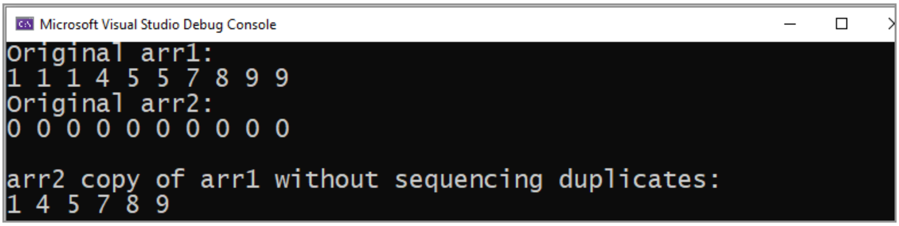
```
class NoSequence
{
    bool init;
    int prevEl;
public:
    NoSequence() : init{ false }, prevEl{ 0 } {}

    bool operator()(int el)
    {
        if (init)
        {
            bool result{ prevEl != el };
            if (result)
            {
                prevEl = el;
            }
            return result;
        }
        init = true;
        prevEl = el;
        return true;
    }
};
```

When calling the special overloading operator() with the next element of the array as the actual parameter, the following happens: first, you need to find out if this functor is called for the first time. If so, then whatever the element of the array is, it must be unconditionally copied to the destination array. Since even if the entire array is the source consists of exactly the same sequential elements, then the first one is still unique. The init member variable is assigned to the ‘ true’ to distinguish repeated runs of the functor from the first one. It is also necessary to store the value of the copied element in a prevEl member variable, the previous element, for later analysis. Suppose the functor is re-run, init == true. In that case, it is necessary to compare the next array element to copy the actual parameter with which the functor was launched with the previously copied pervEl. And only if they are not equal will you copy the next element to the destination array, updating the value of the prevEl member variable. If the next element of the source array is the same as the one previously stored in the prevEl member variable, copying will not take place. It is obvious that the functor represents its copying (or not) in the form of the operator(), 'true' (copying), and otherwise 'false'.

Example 26.

```
#include <iostream>

template <typename T>
void print(T* begin, T* end, char delimiter =  '   ')
{
    while (begin != end)
    {
        std::cout << *begin++ << delimiter;
    }
    std::cout <<  '\n ';
}

template <typename T, typename Predicate>
int copy_if(T* srcB, T* srcE, T* destB, T* destE,
            Predicate pred)
{
    int copyCount{ 0 };
    while (destB != destE and srcB != srcE)
    {
        if (pred(*srcB))
        {
            *destB++ = *srcB;
            ++copyCount;
        }
        ++srcB;
    }
    return copyCount;
}

class NoSequence
{
    bool init;
    int prevEl;
public:
    NoSequence() : init{ false }, prevEl{ 0 } {}

    bool operator()(int el)
    {
        if (init)
        {
            bool result{ prevEl != el };
            if (result)
            {
                prevEl = el;
            }
            return result;
        }
        init = true;
        prevEl = el;
        return true;
    }
};

int main()
{
    const int size{ 10 };
    int arr1[size]{ 1,1,1,4,5,5,7,8,9,9 };
    int arr2[size]{};

    int* const arr1Begin{ arr1 };
    int* const arr1End{ arr1 + size };
    int* const arr2Begin{ arr2 };
    int* const arr2End{ arr2 + size };

    int* arr2NewEnd{};

    std::cout <<  "Original arr1:\n ";
    print(arr1, arr1 + size);

    std::cout <<  "Original arr2:\n ";
    print(arr2, arr2 + size);

    std::cout <<  '\n ';
    std::cout <<  "arr2 copy of arr1 without sequencing
                   duplicates:\n ";
    arr2NewEnd = arr2Begin + copy_if(arr1Begin, arr1End,
                      arr2Begin, arr2End, NoSequence{});
    print(arr2, arr2NewEnd);
    std::cout <<  '\n ';

    return 0;

}
```

The output code result of Example 26.



Microsoft Visual Studio Debug Console

```
Original arr1:
1 1 1 4 5 5 7 8 9 9
Original arr2:
0 0 0 0 0 0 0 0 0 0

arr2 copy of arr1 without sequencing duplicates:
1 4 5 7 8 9
```

Figure 25

The output result from Example 26 confirms the correctness of the previously implemented functor. Only non-sequential elements were copied to the destination array. Of particular interest is the way the functor was passed to the copy_if template copy

```
copy_if(arr1Begin, arr1End, arr2Begin, arr2End, NoSequence{})
```

The functor copy was created anonymously and initialized directly when the template copy was called a NoSequence{} construct.

The functor SumLimit will help in solving the second task:

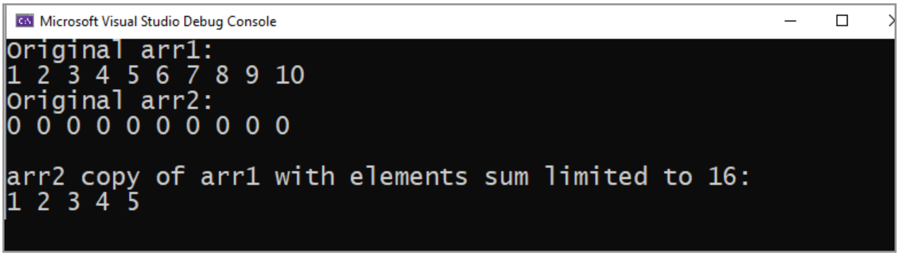
```
class SumLimit
{
    int sumLimit;
    int sum;

public:
    SumLimit(int sumLimitP, int startSumP) :
             sumLimit{ sumLimitP }, sum{ startSumP } {}
    SumLimit(int sumLimitP) : SumLimit{ sumLimitP, 0 } {}

    bool operator()(int el)
    {
        if (sum + el < sumLimit)
        {
            sum += el;
            return true;
        }
        return false;
    }
};
```

The functor has two constructors: one that sets a limit on the sum of the elements of the source array, after which copying stops, and the second one additionally allows to set the initial sum of the elements, in case you need to start not from zero value. The overloaded function call operator is quite simple. It only accumulates the sum of the array elements that it receives as an actual parameter and checks the limit's sum. If the limit is exceeded, the functor starts to return false, which is equivalent to a ban on copying.

Example 27.

```
#include <iostream>

template <typename T>
void print(T* begin, T* end, char delimiter = '  ')
{
    while (begin != end)
    {
        std::cout << *begin++ << delimiter;
    }
    std::cout << '\n ';
}

template <typename T, typename Predicate>
int copy_if(T* srcB, T* srcE, T* destB, T* destE,
            Predicate pred)
{
    int copyCount{ 0 };
    while (destB != destE and srcB != srcE)
    {
        if (pred(*srcB))
        {
            *destB++ = *srcB;
            ++copyCount;
        }
        ++srcB;
    }
    return copyCount;
}

class SumLimit
{
    int sumLimit;
    int sum;

public:
    SumLimit(int sumLimitP, int startSumP) :
             sumLimit{ sumLimitP }, sum{ startSumP } {}
    SumLimit(int sumLimitP) : SumLimit{ sumLimitP, 0 } {}

    bool operator()(int el)
    {
        if (sum + el < sumLimit)
        {
            sum += el;
            return true;
        }
        return false;
    }
};

int main()
{
    const int maxSum{ 16 };
    const int size{ 10 };

    int arr1[size]{ 1,2,3,4,5,6,7,8,9,10 };
    int arr2[size]{};

    int* const arr1Begin{ arr1 };
    int* const arr1End{ arr1 + size };
    int* const arr2Begin{ arr2 };
    int* const arr2End{ arr2 + size };
    int* arr2NewEnd{};
    std::cout <<  "Original arr1:\n ";
    print(arr1, arr1 + size);
    std::cout <<  "Original arr2:\n ";
    print(arr2, arr2 + size);
    std::cout <<  '\n ';
    std::cout << "arr2 copy of arr1 with elements sum
                  limited to  "
              << maxSum <<  ":\n ";
    arr2NewEnd = arr2Begin + copy_if(arr1Begin, arr1End,
                 arr2Begin, arr2End, SumLimit{ maxSum });
    print(arr2, arr2NewEnd);
    std::cout << '\n ';

    return 0;
}
```

The output code result of Example 27.



```
Original arr1:
1 2 3 4 5 6 7 8 9 10
Original arr2:
0 0 0 0 0 0 0 0 0 0

arr2 copy of arr1 with elements sum limited to 16:
1 2 3 4 5
```

Figure 26

Examples 26 and 27 show the practical use of non-trivial functors for elegant problem-solving. The demonstrated approach is also interesting. Instead of a monolithic solution of various copying tasks in the form of a task-specific function, the solution is divided into a number of simpler subtasks that can be solved, tested, and debugged the others independently, which is undoubtedly convenient and efficient. Also, this example represents the possibility of re-using previously written code to solve different tasks, which seems from first sight.

We will face methods and ways demonstrated in this section when we learn the C++ STL (Standard Template Library). STL is the most powerful tool of modern C++ to write clean and efficient code. But everything has its time. While we only slightly lifted the veil on the STL power.

6. Overloading Special Operators: Type Conversion Operators Overloading

One of the tasks that C++ classes solve with operator overloading is the ability for programmers to create new complex data types with a semantic condition, similar to the base types of the programming language. As you know, all base types have the possibility of mutual conversion. However, the issue of converting classes and types created by programmers has not been considered in detail yet. Well, it's time for us to do it.

It is possible to define a conversion operator for a class to an arbitrary base type or an arbitrary other class. Given the internal structure and purpose of the class, not every transformation makes sense and even is possible at all. The class developer determines possibilities and adopts a rational approach. Overloading a type conversion operator is only allowed with a class member function.

Consider the general form and the signature of the type conversion operator:

```
operator typename() {} // 1
operator typename() const {} // 2
explicit operator typename() const {} // 3
```

Option 1 — a non-constant member function implementing a class-to-type conversion type name. Since it is unlikely, and in most cases, it is pointless to modify the class state during type conversion, Option 1 is of practical interest. Option 2 — a constant member function that implements a class-to-type conversion typename. Option 3 — a const version of a member function that explicitly converts a class to typename type and forbids this operator's use in implicit conversions.

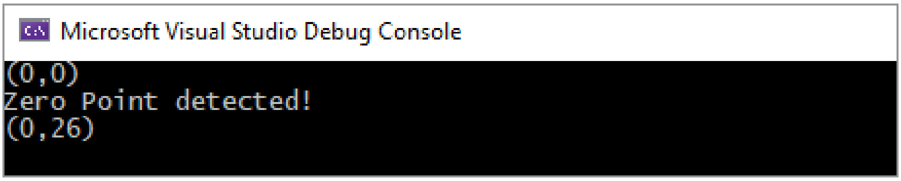
Pay attention that we do not use a return type for a type conversion operator. It is evident why it is necessary to return the type result where the conversion occurs. Also, it should be noted that it is forbidden to use the formal parameters in a type conversion operator.

Let's represent the overloading of the type conversion operator on the example of the Point class.

Example 28.

```
#include <iostream>

class Point
{
    int x;
    int y;

public:
    Point() = default;
    Point(int pX, int pY) : x{ pX }, y{ pY } {}

    Point& setX(int pX) { x = pX; return *this; }
    Point& setY(int pY) { y = pY; return *this; }

    void showPoint() const
    {
        std::cout << '( ' << x << ', ' << y << ') ';
    }
    operator bool() const { return x and y; }
};

int main()
{
    const int pointsCount{ 3 };
    Point points[pointsCount]{ {0,0}, {28,29}, {0,26} };
    bool isZero{false};

    for (auto point{points}, pointsEnd{points + pointsCount};
         point != pointsEnd; ++point)
    {
        isZero = *point;

        if (isZero)
        {
            std::cout << "Zero Point detected!\n ";
        }
        else
        {
            point->showPoint();
            std::cout << '\n ';
        }
    }

    return 0;
}
```

The output code result of Example 28.



Microsoft Visual Studio Debug Console

```
(0,0)
Zero Point detected!
(0,26)
```

Figure 27

Some explanations of processes from Example 28. For all C++ base, types exist a 'bool' type conversion as the "null value" principle corresponds to 'false', and all other values of the original type are 'true'. We will implement a Point type conversion logic similar to the 'bool' type. If both coordinates are zero, the result of the transformation will be 'false', in other cases — 'true'. The member function implemented conversion logic

```
operator bool() const { return x and y; }
```

by simply using the boolean 'and' for point coordinates.

In Example 28, we created an array of 3 points with both zero coordinates. Further in the 'for' loop iterates over the elements of the array, and using 'bool' with isZero variable and overloading the conversion operator for the 'bool' type, the checking if the point is "zero" is taking place. It is important to note that the isZero variable assignment implements with implicit type conversion from Point to 'bool'.

In some cases, it is desirable to prohibit type conversion operator overloading in implicit conversions. To do this, we will use 'explicit', similarly to the constructors. In the case of adding 'explicit' to the type conversion operator in Example 28

```
explicit operator bool() const { return x and y; }
```

the example will stop compiling with errors.

```
E0413no suitable conversion function from  "Point "
     to  "bool " exists

C2440 '= ': cannot convert from  'Point ' to  'bool '
```

To fix these errors, you must perform an explicit type conversion like this:

```
isZero = (bool)*point;
```

Besides, remember about performing a contextual type conversion in ‘bool’ in some cases like:

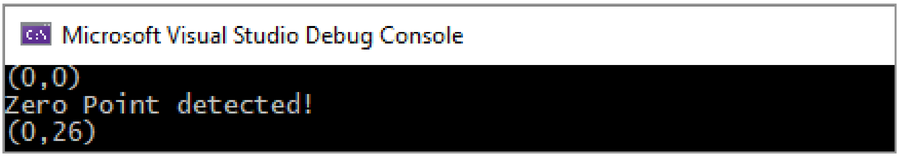
- in the control (conditional) expression of the conditional operator if, as well as loops: ‘while’, ‘do-while’, ‘for’;
- in arguments of built-in operators || boolean OR, &&, boolean AND, !, boolean negation.
- in the first expression of the ternary operator ?.

In the context of the cases above, there is an explicit type conversion to ‘bool’ as the context requires the ‘bool’ type of the corresponding statements or language constructs. In other words, even if overloading of the type conversion operator is declared to ‘bool’ as ‘explicit’, this operator will be performed the context type conversion ‘bool’.

Example 29.

```
#include <iostream>

class Point
{
    int x;
    int y;

public:
    Point() = default;
    Point(int pX, int pY) : x{ pX }, y{ pY } {}
    Point& setX(int pX) { x = pX; return *this; }
    Point& setY(int pY) { y = pY; return *this; }
    void showPoint() const
    {
        std::cout << '( ' << x << ', ' << y << ') ';
    }

    explicit operator bool() const { return x and y; }
};

int main()
{
    const int pointsCount{ 3 };
    Point points[pointsCount]{ {0,0}, {28,29}, {0,26} };

    bool isZero{ false };

    for (auto point{points}, pointsEnd{points+pointsCount};
         point != pointsEnd; ++point)
    {
        isZero = (bool)*point; /* used to show
                                  the necessity of
                                  explicit type conversion
                               */

        if (*point)
        {
            std::cout << "Zero Point detected!\n ";
        }
        else
        {
            point->showPoint();
            std::cout << '\n ';
        }
    }

    return 0;

}
```

The output code result of Example 29.



```
(0,0)
Zero Point detected!
(0,26)
```

Figure 28

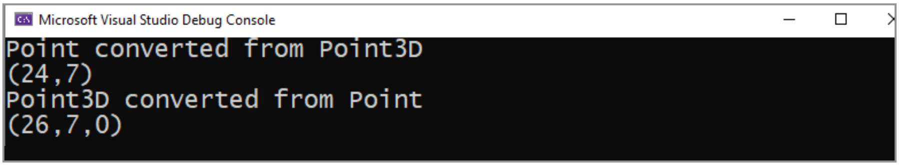
In Example 29, there is a contextual type conversion to 'bool' without explicit type conversion for an expression in a conditional if operator.

The type conversion operator overloading can also be used with base types. Let's consider this using the example of mutual type conversion Point to Point3D (the class is entirely similar to ordinary points, only for three dimensions instead of two).

Example 30.

```
#include <iostream>

class Point3D;
class Point
{
    int x;
    int y;
public:
    Point() = default;
    Point(int pX, int pY) : x{ pX }, y{ pY } {}
    Point& setX(int pX) { x = pX; return *this; }
    Point& setY(int pY) { y = pY; return *this; }

    void showPoint() const
    {
        std::cout << '( ' << x << ', ' << y << ') ';
    }

    explicit operator bool() const { return x and y; }
    explicit operator Point3D() const;
};

class Point3D
{
    int x;
    int y;
    int z;
public:
    Point3D() = default;
    Point3D(int pX, int pY, int pZ) : x{ pX }, y{ pY },
            z{ pZ } {}
    Point3D& setX(int pX) { x = pX; return *this; }
    Point3D& setY(int pY) { y = pY; return *this; }
    Point3D& setZ(int pZ) { z = pZ; return *this; }

    void showPoint() const
    {
        std::cout << '( ' << x << ', ' << y << ', '
                  << z << ') ';
    }

    explicit operator bool() const { return x and y and z; }
    explicit operator Point() const;
};

Point::operator Point3D() const { return { x,y,0 }; }
Point3D::operator Point() const { return { x, y }; }

int main()
{
    Point point{ 26,7 };
    Point pointConv{};
    Point3D point3D{ 24, 7, 76 };
    Point3D point3DConv{};

    pointConv = (Point)point3D;
    point3DConv = (Point3D)point;

    std::cout <<  "Point converted from Point3D\n ";
    pointConv.showPoint();

    std::cout <<  "\nPoint3D converted from Point\n ";
    point3DConv.showPoint();
    std::cout <<  '\n ';

    return 0;
}
```

The output code result of Example 30.


Microsoft Visual Studio Debug Console

```
Point converted from Point3D
(24,7)
Point3D converted from Point
(26,7,0)
```

Figure 29

Example 30 uses two classes: Point and its 3D version, obtained from Point with adding Z-coordinate. Inheritance is not explicitly used here so as not to complicate the example. Both classes provide the possibility of mutual conversion. Since the Point3D class has not been declared at the time of the Point class declaration yet, then to describe the type conversion operator to the Point3D inside the Point class, the Point3D class forward declaration is used. For the correct definition of type conversion operators in the Point and

Point3D, a complete description of these classes should be contained in the compiler regarding the memory occupied with classes. Therefore, only declarations of type conversion operators are placed inside the classes. And their actual definition is placed out of the corresponding classes. The output result from Example 26 confirms the correctness of the code.

Overloading type conversion operators allow you to implement semantically complete and easy-to-use classes.

7. List of Operators That Cannot Be Overloaded

C++ contains several operators that cannot be overloaded in any way. The list of such operators is as follows:

- :: scope resolution operator;
- .* operator pointer to member of the structure/class;
- . struct/class member access operator;
- ?: ternary operator.

The impossibility of overloading these operators is because they perform specific and independent operand types. The same ternary operator evaluates one or the other expression based on the 'bool' expression type before the question mark. As such, the operand types of a ternary operator do not play an active role in its operation. The same is true for the rest of the operators. Even if we assume the possibility of their overloading, it is difficult to imagine a practical and semantically coherent meaning that could be assigned to their overloadings. After all, one of the main operator overloading tasks in C++ is to endow new data types with the semantically consistent and predictable condition, with a processing method similar to base types in C++. And overloading the "non-overloadable" operators would most likely break this semantic and introduce an unexpected condition, which is unacceptable and unavailable.

8. Static Polymorphism and Operator Overloading as a Special Case

Operator overloading in C++ implements the possibility of semantically similar or identical processing of different data types. At the same time, there are not any restrictions and differences for what data types overloading is implemented: for basic C++ types or developed classes. The provision of a single interface to entities of different types or the use of a single symbol to represent multiple different types is what polymorphism means. And the ad-hock polymorphism is one of the major classes of polymorphism (refers to polymorphic functions). This kind of polymorphism refers to static polymorphism since choosing a particular code implementation for the corresponding types occurs at the stage of compilation, not at running the program. For better performance and analysis by a programmer or automatic software tools, the static polymorphism is preferable to the dynamic, which you will be acquainted with later in the course.

9. Overloading with Global Functions

The most optimal solution for overloading operators with unmodified operands is the global or friend functions. Let's begin our discussion with global functions overloading. And we will leave our consideration of the friend functions for the next section of this lesson.

So, look at the overloading of arithmetic operators + – * / from the example of the Fraction class. These operators are binary operators that do not modify their arguments and return the result as a new temporary object. The function for this kind of operator looks as follows:

```
ClassName operator+(const ClassName& left,
                    const ClassName& right); //1
ClassName operator+(const ClassName& left, int right); //2
ClassName operator+(int left, const ClassName& right); //3
```

Option 1 — the operator overloading for both arguments are the ClassName classes.

Option 2 and 3 — the operator overloading for the right (2) or left (3) operands of some base type, i.e., int. You should note that, despite its symmetry, it is not enough to implement only one of options 2 or 3. For semantically complete conditions, you need to implement both options.

Example 31.

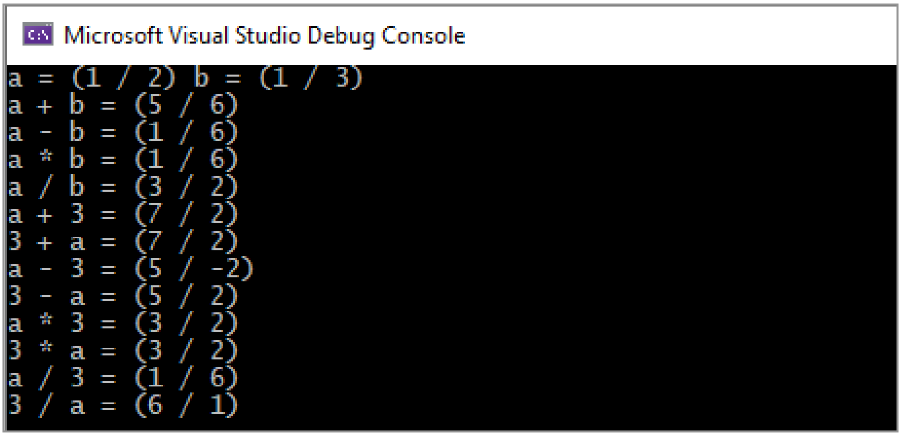
```cpp
#include <iostream>

class Fraction
{
    int numerator;
    int denominator;
    int gcd(int a, int b);
    void reduce();
public:
    Fraction(int num, int denom)
        : numerator{ num }, denominator{ denom ? denom : 1 }
    {
        reduce();
    };

    Fraction() : Fraction(1, 1) {};
    void setNumerator(int num) { numerator = num; reduce();};
    int getNumerator() const { return numerator; };
    void setDenominator(int denom) { denominator = denom ?
                                     denom : 1; reduce();};
    int getDenominator() const { return denominator; }
    void print() const;
};

int Fraction::gcd(int a, int b)
{
    int copy;
    while (b)
    {
        a %= b;
        copy = a;
        a = b;
        b = copy;
    }
    return a;
}
```

```
void Fraction::reduce()
{
    int gcdVal{ gcd(numerator, denominator) };
    numerator /= gcdVal;
    denominator /= gcdVal;

    if (denominator < 0 and numerator < 0)
    {
        denominator *= -1;
        numerator *= -1;
    }
}


void Fraction::print() const
{
    std::cout << '( ' << numerator << " / "
              << denominator << ") ";
}

Fraction operator+(const Fraction& left, const Fraction& right)
{
    return Fraction{ left.getNumerator() *
           right.getDenominator() + right.getNumerator() *
           left.getDenominator(), left.getDenominator() *
           right.getDenominator() };
}

Fraction operator+(const Fraction& left, int right)
{
    return Fraction{ left.getNumerator() + right *
           left.getDenominator(), left.getDenominator() };
}

Fraction operator+(int left, const Fraction& right)
{
    return right + left;
}
```

```
Fraction operator-(const Fraction& left,
                   const Fraction& right)
{
    return Fraction{ left.getNumerator() *
           right.getDenominator() - right.getNumerator() *
           left.getDenominator(), left.getDenominator() *
           right.getDenominator() };
}

Fraction operator-(const Fraction& left, int right)
{
    return Fraction{ left.getNumerator() - right *
           left.getDenominator(), left.getDenominator() };
}

Fraction operator-(int left, const Fraction& right)
{
    return Fraction{ left * right.getDenominator() -
           right.getNumerator(), right.getDenominator() };
}

Fraction operator*(const Fraction& left,
                   const Fraction& right)
{
    return Fraction{ left.getNumerator() *
           right.getNumerator(), left.getDenominator() *
           right.getDenominator() };
}

Fraction operator*(const Fraction& left, int right)
{
    return Fraction{ left.getNumerator() * right,
                     left.getDenominator()};
}

Fraction operator*(int left, const Fraction& right)
{
    return right * left;
}
```

```
Fraction operator/(const Fraction& left,
                   const Fraction& right)
{
    return Fraction{ left.getNumerator() *
           right.getDenominator(), left.getDenominator() *
           right.getNumerator() };
}

Fraction operator/(const Fraction& left, int right)
{
    return Fraction{ left.getNumerator() ,
                     left.getDenominator() * right };
}

Fraction operator/(int left, const Fraction& right)
{
    return Fraction{ left* right.getDenominator(),
                     right.getNumerator() };
}

int main()
{
    Fraction a{1,2};
    Fraction b{1,3};

    std::cout <<  "a =  "; a.print(); std::cout
              <<  " b =  "; b.print(); std::cout
              <<  '\n ';
    std::cout <<  "a + b =  "; (a + b).print(); std::cout
              <<  '\n ';
    std::cout <<  "a - b =  "; (a - b).print(); std::cout
              <<  '\n ';
    std::cout <<  "a * b =  "; (a * b).print(); std::cout
              <<  '\n ';
    std::cout <<  "a / b =  "; (a / b).print(); std::cout
              <<  '\n ';
    std::cout <<  "a + 3 =  "; (a + 3).print(); std::cout
              <<  '\n ';
    std::cout << "3 + a =  "; (3 + a).print(); std::cout
              << '\n ';
    std::cout << "a - 3 =  "; (a - 3).print(); std::cout
              << '\n ';
    std::cout << "3 - a =  "; (3 - a).print(); std::cout
              << '\n ';
    std::cout << "a * 3 =  "; (a * 3).print(); std::cout
              << '\n ';
    std::cout << "3 * a =  "; (3 * a).print(); std::cout
              << '\n ';
    std::cout << "a / 3 =  "; (a / 3).print(); std::cout
              << '\n ';
    std::cout << "3 / a =  "; (3 / a).print(); std::cout
              << '\n ';
    return 0;
}
```

The output code result of Example 31.


Microsoft Visual Studio Debug Console

```
a = (1 / 2) b = (1 / 3)
a + b = (5 / 6)
a - b = (1 / 6)
a * b = (1 / 6)
a / b = (3 / 2)
a + 3 = (7 / 2)
3 + a = (7 / 2)
a - 3 = (5 / -2)
3 - a = (5 / 2)
a * 3 = (3 / 2)
3 * a = (3 / 2)
a / 3 = (1 / 6)
3 / a = (6 / 1)
```

Figure 30

We added two private member functions for implementing arithmetic operations into the modified Fraction class, where ‘gcd’ searches for the greatest common divisor and the ‘reduce’ needs for reducing fractions. You should use the ‘reduce’ function call in the constructor and the corresponding so that the fraction stored in the class is always abbreviated. The arithmetic operations are implemented via overloading the corresponding operators with global functions. These functions return a new Fraction class copy as a result. It should also be noted that you are required to implement only one Fraction/int overloading due to the commutativity of addition and multiplication. And the next overloading you will implement through the previous one, changing the order of the arguments. The program output indicates the correctness of the work done.

Is everything correct in Example 31, after all? Formally, yes, because the problem has been solved, and the fraction/fraction and fraction/integer arithmetic operations have been implemented and tested. But, the operational efficiency is not the most optimal because of the permanent calling of the corresponding getters. Calling getters is necessary due to the Fraction class member variables; there is no other way to get to them from the global functions. And, what should we do in this case? Should we make Fraction class member variables public, eliminating the need to use getters in global functions? No way! By doing so, we violate the encapsulation principle. But there is a solution. We will make our global functions as class-friendly to Fraction.

10. Overloading with Friend Functions

Friend functions of the ClassName are not referred to such class and have access to 'private' and 'protected' the ClassName members. Friend functions can be global functions and class member functions as well. Friend functions are not the class member functions, though they are referred to it.

For a function to become class-friendly, it is necessary to indicate the function by setting the 'friend' as a keyword before entering the class definition.

```
class ClassB
{
    void bFunction() {/* function body */};
};

class ClassName
{
    // definig the class
    friend void nonMemberFunction(); // 1
    friend void ClassB::bFunction(); // 2
};

void nonMemberFunction() {/* function body */ }
```

Option 1 — specifying a global function as a class friend of the ClassName. Option 2 — setting a member function of the ClassB as a class friend of the ClassName.

It may seem that friend functions resist the encapsulation principle. However, it is not. Because only those functions that are declared as a friend of the class get access to 'private' and 'protected' class members. The "desire" (purpose) written inside of the function is not enough; the determining factor is the "allowing" and "accepting" (getting) the friendship of that class. The class developer has complete control over which functions are provided extended access and which — are not. The concept of the friend functions gains the problems with a misstatement of class member variables to a well-defined set of the friend functions. Accordingly, when debugging the class, it is necessary to pay attention to the class member functions and the friend functions of the class. The encapsulation principle has practically stayed unmodified.

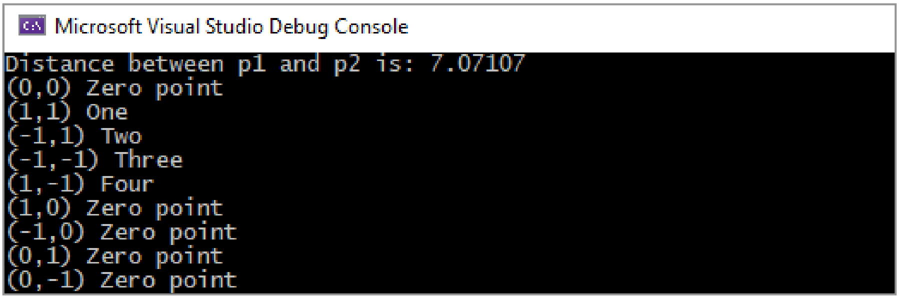
Let's consider a simple example of using friend functions based on the Point class from the previous sections of the lesson. We will implement several functions for working with points, such as calculating the distance between two points and determining which quadrant or coordinate axis a given point belongs to. We will also implement a service function that displays a string corresponding to a quadrant or coordinate axis constant.

Example 32.

```
#include <iostream>

enum Quadrants { ZeroPoint, First, Second, Third, Fourth,
                 XPositive, XNegative, YPositive, YNegative};

class Point
{
    int x;
    int y;
public:
    Point() = default;
    Point(int pX, int pY) : x{pX}, y{pY} {}
    Point& setX(int pX) {x = pX; return *this;}
    Point& setY(int pY) {y = pY; return *this;}
    int getX()const { return x;}
    int getY()const { return y;}

    void show() const
    {
        std::cout <<  '( ' << x <<  ', ' << y <<  ') ';
    }

    friend double distance(const Point& p1, const Point& p2);
    friend int quadrant(const Point& p)
    {
        if ((!p.y and !p.x)) { return
                                   Quadrants::ZeroPoint; }
        if (!p.y) { return p.x > 0 ? Quadrants::XPositive :
                                   Quadrants::XNegative; }
        if (!p.x) { return p.y > 0 ? Quadrants::YPositive :
                                   Quadrants::YNegative; }
        if (p.y > 0)
        {
            return p.x > 0 ? Quadrants::First :
                             Quadrants::Second;
        }
        return p.x > 0 ? Quadrants::Fourth :
                         Quadrants::Third;
    }
};

double distance(const Point& p1, const Point& p2)
{
    auto xLength{ p2.x - p1.x };
    auto yLength{ p2.y - p1.y };
    return sqrt(xLength * xLength + yLength * yLength);
}
```

```
void quadrantDecode(int quadrant)
{
    if (quadrant == Quadrants::ZeroPoint) { std::cout
                        <<  "Zero point "; }
    else if (quadrant == Quadrants::First) { std::cout
                        <<  "First quadrant "; }
    else if (quadrant == Quadrants::Second) { std::cout
                        <<  "Second quadrant "; }
    else if (quadrant == Quadrants::Third) { std::cout
                        <<  "Third quadrant "; }
    else if (quadrant == Quadrants::Fourth) { std::cout
                        <<  "Fourth quadrant "; }
    else if (quadrant == Quadrants::XPositive) { std::cout
                        <<  "X axis positive "; }
    else if (quadrant == Quadrants::XNegative) { std::cout
                        <<  "X axis negative "; }
    else if (quadrant == Quadrants::YPositive) { std::cout
                        <<  "Y axis positive "; }
    else if (quadrant == Quadrants::YNegative) { std::cout
                        <<  "Y axis negative "; }
}

int main()
{
    Point p1{ 5,5 };
    Point p2{ 10,10 };

    std::cout <<  "Distance between p1 and p2 is:  "
              << distance(p1, p2) <<  '\n ';
    const int testCases{ 9 };
    Point points[testCases]{
        {0,0},
        {1,1},
        {-1,1},
        {-1,-1},
        {1,-1},
        {1,0},
        {-1,0},
        {0,1},
        {0,-1}
    };

    for (int i{ 0 }; i < testCases; ++i)
    {
        points[i].show();
        std::cout <<  '  ';
        quadrantDecode(quadrant(points[i]));
        std::cout <<  '\n ';
    }

    return 0;
}
```

The output code result of Example 32.



Microsoft Visual Studio Debug Console

```
Distance between p1 and p2 is: 7.07107
(0,0) Zero point
(1,1) One
(-1,1) Two
(-1,-1) Three
(1,-1) Four
(1,0) Zero point
(-1,0) Zero point
(0,1) Zero point
(0,-1) Zero point
```

Figure 31

In Example 32, we defined two friend functions: distance and quadrant. Both functions got access to the Point class member variables without applying getters. We have fully defined the quadrant function inside the Point class. Such definition of a friend function is consistent with the standard, and it is convenient and appropriate in some cases.

It should be noted that no matter in what way you define friend functions: defining of friendliness inside the class, defining of the function outside the class, or defining the friendliness of the function and the function inside the class, — they will not become class member functions. Friend functions don't have an implicit pointer 'this', and they cannot be called in as s class copy.

It's time to modify the example from the previous lesson section to use the global functions, friendly to the class, as overloading of the corresponding operators.

Example 33.

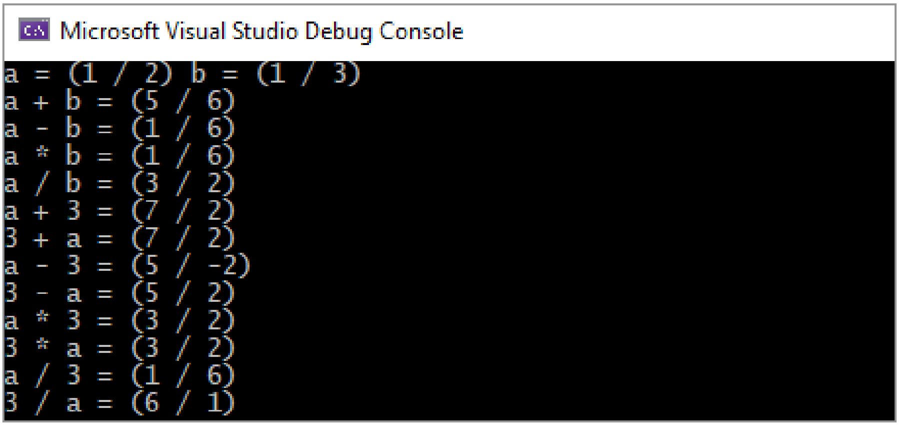
```
#include <iostream>

class Fraction
{
    int numerator;
    int denominator;
    int gcd(int a, int b);
    void reduce();

public:
    Fraction(int num, int denom)
        : numerator{num}, denominator{denom ? denom : 1}
    {
        reduce();
    };
    Fraction() : Fraction(1, 1) {};
    void setNumerator(int num) {numerator = num; reduce();};
    int getNumerator() const { return numerator; };
    void setDenominator(int denom) { denominator =
         denom ? denom : 1; reduce(); };
    int getDenominator() const { return denominator; }
    void print() const;
    friend Fraction operator+(const Fraction& left,
                              const Fraction& right);
    friend Fraction operator+(const Fraction& left,
                              int right);
    friend Fraction operator+(int left,
                              const Fraction& right);
    friend Fraction operator-(const Fraction& left,
                              const Fraction& right);
    friend Fraction operator-(const Fraction& left,
                              int right);
    friend Fraction operator-(int left,
                              const Fraction& right);
    friend Fraction operator*(const Fraction& left,
                              const Fraction& right);
    friend Fraction operator*(const Fraction& left,
                              int right);
    friend Fraction operator*(int left,
                              const Fraction& right);
    friend Fraction operator/(const Fraction& left,
                              const Fraction& right);
    friend Fraction operator/(const Fraction& left,
                              int right);
    friend Fraction operator/(int left,
                              const Fraction& right);
};

int Fraction::gcd(int a, int b)
{
    int copy;
    while (b)
    {
        a %= b;
        copy = a;
        a = b;
        b = copy;
    }
    return a;
}
```

```cpp
void Fraction::reduce()
{
    int gcdVal{ gcd(numerator, denominator) };
    numerator /= gcdVal;
    denominator /= gcdVal;

    if (denominator < 0 and numerator < 0)
    {
        denominator *= -1;
        numerator *= -1;
    }
}

void Fraction::print() const
{
    std::cout << '( ' << numerator << " / " 
              << denominator << ") ";
}

Fraction operator+(const Fraction& left,
                   const Fraction& right)
{
    return Fraction{ left.numerator * right.denominator +
                     right.numerator * left.denominator,
                     left.denominator * right.denominator };
}

Fraction operator+(const Fraction& left, int right)
{
    return Fraction{ left.numerator + right *
                     left.denominator, left.denominator };
}

Fraction operator+(int left, const Fraction& right)
{
    return right + left;
}
```

```
Fraction operator-(const Fraction& left,
                   const Fraction& right)
{
    return Fraction{ left.numerator * right.denominator -
                     right.numerator * left.denominator,
                     left.denominator * right.denominator };
}

Fraction operator-(const Fraction& left, int right)
{
    return Fraction{ left.numerator - right *
                     left.denominator, left.denominator };
}

Fraction operator-(int left, const Fraction& right)
{
    return Fraction{ left * right.denominator -
                     right.numerator, right.denominator };
}

Fraction operator*(const Fraction& left,
                   const Fraction& right)
{
    return Fraction{ left.numerator * right.numerator,
                     left.denominator * right.denominator };
}

Fraction operator*(const Fraction& left, int right)
{
    return Fraction{ left.numerator * right,
                     left.denominator };
}

Fraction operator*(int left, const Fraction& right)
{
    return right * left;
}
```

```
Fraction operator/(const Fraction& left,
                   const Fraction& right)
{
    return Fraction{ left.numerator * right.denominator,
                     left.denominator * right.numerator };
}

Fraction operator/(const Fraction& left, int right)
{
    return Fraction{ left.numerator ,
                     left.denominator * right };
}

Fraction operator/(int left, const Fraction& right)
{
    return Fraction{ left * right.denominator,
                     right.numerator };
}

int main()
{
    Fraction a{ 1,2 };
    Fraction b{ 1,3 };
    std::cout <<  "a =  "; a.print(); std::cout
              <<  " b =  "; b.print(); std::cout <<  '\n';
    std::cout <<  "a + b =  "; (a + b).print(); std::cout
              <<  '\n ';
    std::cout <<  "a - b =  "; (a - b).print(); std::cout
              <<  '\n ';
    std::cout <<  "a * b =  "; (a * b).print(); std::cout
              <<  '\n ';
    std::cout <<  "a / b =  "; (a / b).print(); std::cout
              <<  '\n ';
    std::cout <<  "a + 3 =  "; (a + 3).print(); std::cout
              <<  '\n ';
    std::cout <<  "3 + a =  "; (3 + a).print(); std::cout
              <<  '\n ';
    std::cout << "a - 3 =  "; (a - 3).print(); std::cout
              << '\n ';
    std::cout << "3 - a =  "; (3 - a).print(); std::cout
              << '\n ';
    std::cout << "a * 3 =  "; (a * 3).print(); std::cout
              << '\n ';
    std::cout << "3 * a =  "; (3 * a).print(); std::cout
              << '\n ';
    std::cout << "a / 3 =  "; (a / 3).print(); std::cout
              << '\n ';
    std::cout << "3 / a =  "; (3 / a).print(); std::cout
              << '\n ';

    return 0;
}
```

The output code result of Example 33.



```
a = (1 / 2) b = (1 / 3)
a + b = (5 / 6)
a - b = (1 / 6)
a * b = (1 / 6)
a / b = (3 / 2)
a + 3 = (7 / 2)
3 + a = (7 / 2)
a - 3 = (5 / -2)
3 - a = (5 / 2)
a * 3 = (3 / 2)
3 * a = (3 / 2)
a / 3 = (1 / 6)
3 / a = (6 / 1)
```

Figure 32

It is evident that the Output result of Example 33 corresponds to Example 31. The difference is that now the global function overloading operators are friends of the Fraction class (with its "allowing" and "accepting"). As a result, they can access the private member variables without using getters. The work of arithmetic operators in this form will become more efficient without additional actions associated with calling getters.

Overloading operators with friendly functions is a very effective tool. So, why do we commonly overload operators without applying the friend functions? The thing is that the class for which it is necessary to create overloading this or that operator may be outside your administrative control, i.e., this class may be developed by another programmer. The class author may not agree to include the "third-party" functions in the list of class-friendly functions, and, as we remember, the final decision "to assign friendship" or not is under the control of the class author. Another situation where such "friendship" is impossible occurs when the source code is missing in a specific class. As we can use a ready-made class, the implementation is not available to us. In this case, the solution is to use operator overloading with global functions, which, in turn, will access the class member variables with the corresponding getters/setters.

Some of the special operators in C++, that the class member functions can only overload:

- = — it is used to assign the values;
- () — it is used for function calls;
- [] — it is a subscript operator (access by the index);
- -> — it is a member access operator.

The operators listed above cannot be implemented with the friend or global functions.

11. Input/Output Overloading

Operator overloading in C++ allows us to implement conditions and semantics for new data types of C++ classes similar to those of the basic types, like int, double, bool, etc. All the basic types allow the output of their values and input the values using the keyboard. In general, input/output can be generalized to output to a file or another output device and input from a file or some input device. For the basic types, this condition is provided by overloads of the bitwise left shift operator, which referred to the "put in the stream", and the bitwise right shift operator referred to as "out of the stream". It's time to analyze specifics of input/output overloading for the classes we're implementing.

First, let's look at the example of how the output of the basic int type acts:

```
std::cout << 42;
```

Here "<<" is a binary operator, 42 and std::cout its operands. The std::cout global object is a class copy, indirectly inherited from the std::ostream class, which implements a basic streaming output interface. For example, the ostream class is also inherited from the std::ofstream. And, as a consequence, if you implement an overload of the << operator for the std::ostream class, then both output to the screen and a file will be possible. Looking ahead a little, we note that the situation is precisely the same with input streams. It means that by implementing the >> overload for the std::istream class, it will be possible to implement the input from the keyboard and a file.

Let's move back to the put-in-stream operator. How should we implement its overloading? Will be a class member function relevant for us? No, because the left operand is the std::ostream class. Member functions are suitable for overloading operators with the left argument of the source function class. Accordingly, only the option with a global or a friend function remains.

Another feature of this operator is its ability "to work in chain order":

```
std::cout <<  "Answer is  " << 42;
```

For such implementation, it is necessary to return the object from the operator overloading, suitable for re-calling the stream insertion operator. For example, the expression above can be written like this:

```
((std::cout <<  "Answer is  ")/*1*/ << 42)/*2*/;
```

That is, expression 1 should return std::cout as a result so that expression 2 works correctly.

The reasoning above leads us to the following overloading signature of the put-in-stream operator:

```
std::ostream& operator<<(std::ostream& out,
                         const ClassName& object);
```

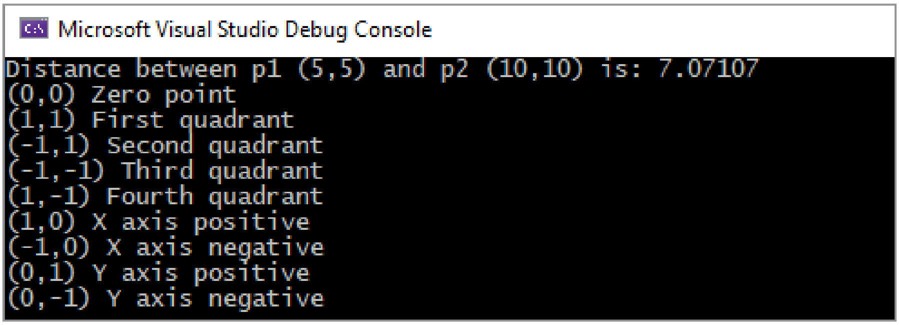
Note that both putting to a function and returning a copy of the std::ostream must be produced with a non-constant reference. Transfer/return is executed by reference to avoid copying which prohibits the classstd::ostream (materials from the previous sections of the lesson help you figure out this is an implementation). Also, note that the link to std::ostream is not const because the data placed in it will change the object. The actual object placed in the stream is passed to the function by a constant reference for saving on copying and prohibition of modification inside the source operator of the ClassName copy placed in the stream.

Let's modify Example 32 and implement for the Point class overloading for the stream insertion operator using a friend function, thus replacing the previously provided functionality with the 'show' member function:

Example 34.

```
#include <iostream>

enum Quadrants {ZeroPoint, First, Second, Third, Fourth,
                XPositive, XNegative, YPositive, YNegative};

class Point
{
    int x;
    int y;

public:
    Point() = default;
    Point(int pX, int pY) : x{ pX }, y{ pY } {}
    Point& setX(int pX) { x = pX; return *this; }
    Point& setY(int pY) { y = pY; return *this; }
    int getX()const { return x; }
    int getY()const { return y; }

    void show() const
    {
        std::cout << '( ' << x << ', ' << y << ') ';
    }
    friend double distance(const Point& p1, const Point& p2);
    friend int quadrant(const Point& p)
    {
        if ((!p.y and !p.x)) { return
                                  Quadrants::ZeroPoint; }
        if (!p.y) {return p.x > 0 ? Quadrants::XPositive :
                   Quadrants::XNegative;}
        if (!p.x) {return p.y > 0 ? Quadrants::YPositive :
                   Quadrants::YNegative; }
        if (p.y > 0)
        {
            return p.x > 0 ? Quadrants::First :
                             Quadrants::Second;
        }
        return p.x > 0 ? Quadrants::Fourth :
                         Quadrants::Third;
    }

    friend std::ostream& operator<<(std::ostream& out,
                                    const Point& point);
};

double distance(const Point& p1, const Point& p2)
{
    auto xLength{ p2.x - p1.x };
    auto yLength{ p2.y - p1.y };
    return sqrt(xLength * xLength + yLength * yLength);
}

void quadrantDecode(int quadrant)
{
    if (quadrant == Quadrants::ZeroPoint) { std::cout
                    <<  "Zero point "; }
    else if (quadrant == Quadrants::First) { std::cout
                    <<  "First quadrant "; }
    else if (quadrant == Quadrants::Second) { std::cout
                    <<  "Second quadrant "; }
    else if (quadrant == Quadrants::Third) { std::cout
                                << "Third quadrant "; }
    else if (quadrant == Quadrants::Fourth) { std::cout
                                << "Fourth quadrant "; }
    else if (quadrant == Quadrants::XPositive) { std::cout
                                << "X axis positive "; }
    else if (quadrant == Quadrants::XNegative) { std::cout
                                << "X axis negative "; }
    else if (quadrant == Quadrants::YPositive) { std::cout
                                << "Y axis positive "; }
    else if (quadrant == Quadrants::YNegative) { std::cout
                                << "Y axis negative "; }
}

std::ostream& operator<<(std::ostream& out, const Point& point)
{
    out << '( ' << point.x << ', ' << point.y << ') ';
    return out;
}

int main()
{
    Point p1{ 5,5 };
    Point p2{ 10,10 };

    std::cout << "Distance between p1 " << p1
              << " and p2 " << p2 << " is: "
              << distance(p1, p2) << '\n ';
    const int testCases{ 9 };
    Point points[testCases]{
        {0,0},
        {1,1},
        {-1,1},
        {-1,-1},
        {1,-1},
        {1,0},
        {-1,0},
        {0,1},
        {0,-1}
    };

    for (int i{ 0 }; i < testCases; ++i)
    {
        std::cout << points[i] << ' ';
        quadrantDecode(quadrant(points[i]));
        std::cout << '\n';
    }
    return 0;
}
```

The output code result of Example 34



Microsoft Visual Studio Debug Console

```
Distance between p1 (5,5) and p2 (10,10) is: 7.07107
(0,0) Zero point
(1,1) First quadrant
(-1,1) Second quadrant
(-1,-1) Third quadrant
(1,-1) Fourth quadrant
(1,0) X axis positive
(-1,0) X axis negative
(0,1) Y axis positive
(0,-1) Y axis negative
```

Figure 33

The output result of Example 34 is very similar to Output 32. However, the code has become more semantic, and it has become possible to place the Point class objects into a stream, displaying them on the screen, as is executed for all basic types inC++. When implementing stream insertion operator overloading, it is essential to remember returning a reference to the stream from the operator overloading. Such a return is correct because the output stream was before the operator was called, and it will be after its stop. Consequently, we will not get the so-called "dangling link"; on the contrary, we will get the possibility of massive, as a single-part, setting in the stream.

The overloading of the stream extraction operator is completely similar to the stream insertion operator, and only the std::istream class is used instead of the std::ostream class. All considerations regarding the overloading techniques, passing parameters, returning a value, compatibility with file input — all this is also true for overloading the stream extraction that implements input for class copies appropriate for C++. And yet, there is one crucial difference is the class instance for which the overloading is implemented and passed through a non-constant reference because the object will be modified while extracting out of the stream.

The stream extraction function looks as below:

```
std::istream& operator>>(std::istream& in, ClassName& object);
```

Similar to stream insertion, to support concatenated value input, you must remember to return a copy of the std::istream from the stream extraction operator overloading.

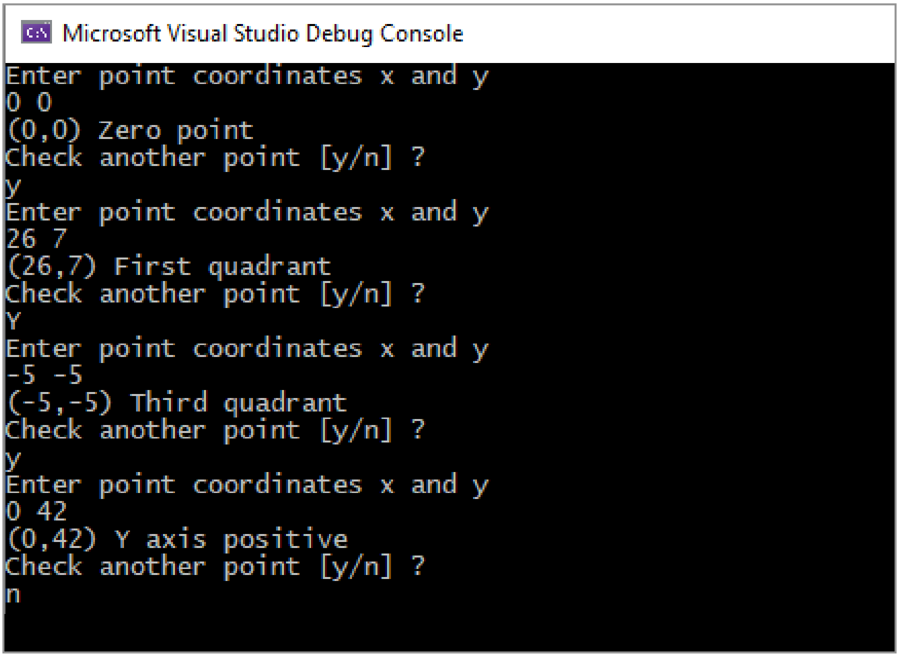
Implementing stream extraction could be a desire to display an input prompt directly in the function, explaining to the user what exactly to input and in what order. Such a desire is quite obvious. However, we should not forget that input can be made interactively — from the keyboard and automatically — from a file. In such a case displaying messages on the screen will be undesirable. It is recommended to avoid embedding prompt messages directly within the stream extraction operator overloading. Alternatively, consider a helper member function that displays an input prompt, in which case such a separate function can be used with interactive stream extraction.

To demonstrate it, let's add a stream extraction operator overloading to the Point class, as well as a helper function that displays the input prompt. And we will slightly modify the previous Example 34 to allow the user to enter points to determine if they belong to the corresponding quadrant or coordinate axis.

Example 35.

```
#include <iostream>

enum Quadrants {ZeroPoint, First, Second, Third, Fourth,
                XPositive, XNegative, YPositive, YNegative};

class Point
{
    int x;
    int y;

public:
    Point() = default;
    Point(int pX, int pY) : x{ pX }, y{ pY } {}
    Point& setX(int pX) { x = pX; return *this; }
    Point& setY(int pY) { y = pY; return *this; }
    int getX()const { return x; }
    int getY()const { return y; }
    void show() const
    {
        std::cout <<  '( ' << x <<  ', ' << y <<  ') ';
    }
    friend double distance(const Point& p1,
                           const Point& p2);
    friend int quadrant(const Point& p)
    {
        if ((!p.y and !p.x)) { return
                                   Quadrants::ZeroPoint; }
        if (!p.y) { return p.x > 0 ? Quadrants::XPositive :
                    Quadrants::XNegative; }
        if (!p.x) { return p.y > 0 ? Quadrants::YPositive :
                    Quadrants::YNegative; }
        if (p.y > 0)
        {
            return p.x > 0 ? Quadrants::First :
                             Quadrants::Second;
        }
        return p.x > 0 ? Quadrants::Fourth :
                         Quadrants::Third;
    }
    Point& inputPrompt() { std::cout
           <<  "Enter point coordinates x and y\n ";
           return *this; }
    friend std::ostream& operator<<(std::ostream& out,
                                    const Point& point);
    friend std::istream& operator>>(std::istream& in,
                                    Point& point);
};

double distance(const Point& p1, const Point& p2)
{
    auto xLength{ p2.x - p1.x };
    auto yLength{ p2.y - p1.y };
    return sqrt(xLength * xLength + yLength * yLength);
}

void quadrantDecode(int quadrant)
{
    if (quadrant == Quadrants::ZeroPoint) { std::cout
                                  <<  "Zero point "; }
    else if (quadrant == Quadrants::First) { std::cout
                                  <<  "First quadrant "; }
    else if (quadrant == Quadrants::Second) { std::cout
                                  <<  "Second quadrant "; }
    else if (quadrant == Quadrants::Third) { std::cout
                                << "Third quadrant "; }
    else if (quadrant == Quadrants::Fourth) { std::cout
                                << "Fourth quadrant "; }
    else if (quadrant == Quadrants::XPositive) { std::cout
                                << "X axis positive "; }
    else if (quadrant == Quadrants::XNegative) { std::cout
                                << "X axis negative "; }
    else if (quadrant == Quadrants::YPositive) { std::cout
                                << "Y axis positive "; }
    else if (quadrant == Quadrants::YNegative) { std::cout
                                << "Y axis negative "; }
}

std::ostream& operator<<(std::ostream& out, const Point& point)
{
    out << '( ' << point.x << ', ' << point.y << ') ';
    return out;
}

std::istream& operator>>(std::istream& in, Point& point)
{
    in >> point.x >> point.y;
    return in;
}

int main()
{
    Point p1{};
    char menu{ 'n ' };
    bool correct{ false };
    do
    {
        std::cin >> p1.inputPrompt();
        std::cout << p1 << '  ';
        quadrantDecode(quadrant(p1));
        std::cout << '\n ';
        do
        {
            std::cout << "Check another point [y/n] ?\n ";
            std::cin >> menu;
            correct = menu ==  'y ' or menu ==  'Y ' or
                      menu ==  'n ' or menu ==  'N ';
            if (!correct) { std::cout
                     << "Incorrect choice! Try again!\n";}
        } while (!correct);
    } while (menu == 'y ' or menu == 'Y ');

    return 0;
}
```

The output code result of Example 35.



```
Enter point coordinates x and y
0 0
(0,0) Zero point
Check another point [y/n] ?
y
Enter point coordinates x and y
26 7
(26,7) First quadrant
Check another point [y/n] ?
Y
Enter point coordinates x and y
-5 -5
(-5,-5) Third quadrant
Check another point [y/n] ?
y
Enter point coordinates x and y
0 42
(0,42) Y axis positive
Check another point [y/n] ?
n
```

Figure 34

Output 35 shows an interactive session where the user sequentially enters the coordinates of points 0,0; 26.7; -5, -5; 0.42. And the program determines if the entered points are to the corresponding quadrants or coordinates. In this example, a little “trick” is implemented. The inputPrompt() member function returns a reference to an instance of the Point class. So, it is possible to use std::cin >> p1.inputPrompt(); to interactively enter a point from a keyboard with simultaneously displaying an input prompt for the user. To evaluate the above expression, you must call the inputPrompt() member function first, which will output a message and return a reference to the copy of p1, which in turn will be used as the right argument in the stream extraction operator overloading.

In sum, we have learned how to overload stream insertion and stream extraction operators. And now we can implement the input and output of the classes as easily as the basic C++ types.

12. Friend Classes

In addition to friend functions, a class can also have friend classes. All member functions of the friend class will have access to private and protected members of the friend class. The syntax for defining, if the classes are friend classes, is similar to defining the friend functions:

```
class FriendClass
{
    // defining the FriendClass
};

class SomeClass
{
    // defining the SomeClass
    friend FriendClass;
};
```

In this example, the FriendClass becomes a friend class to the SomeClass. And accordingly, to that, all member functions of the FriendClass get access to all members of the SomeClass, as well as private and protected. However, these classes are not friend classes completely. The member functions of the Some-Class don't get access to private and protected members of the FriendClass.

When should we apply the friend classes? When classes closely interact, and their interaction is critical to implementation, it is quite appropriate to use friend classes. However, you should accurately choose the classes for assigning a "friendship" to them and apply class friendships if it is truly warranted. You should always consider friend member functions of a class as a safer alternative.

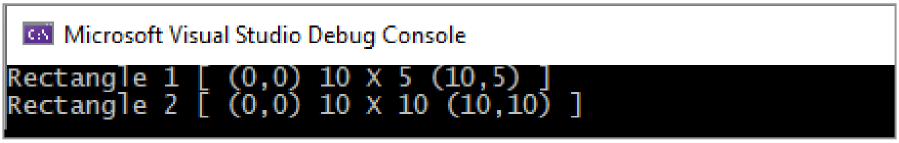
Show the friend classes by the example of the basic implementation of a class due to which a rectangle is created in two-dimensional space. Such an implementation does not apply for completeness and perfect correctness. It does not perform several checks and modifications to simplify the example and focus on class friendliness.

Example 36.

```
#include <iostream>

class Rectangle;

class Point
{
    int x;
    int y;
public:
    Point() = default;
    Point(int pX, int pY) : x{ pX }, y{ pY } {}
    Point& setX(int pX) { x = pX; return *this; }
    Point& setY(int pY) { y = pY; return *this; }
    int getX()const { return x; }
    int getY()const { return y; }
    void show() const
    {
        std::cout <<  '( ' << x <<  ', ' << y <<  ') ';
    }

    Point& inputPrompt() { std::cout <<  "Enter point
                           coordinates x and y\n ";
                           return *this; }
    friend std::ostream& operator<<(std::ostream& out,
                                    const Point& point);
    friend std::istream& operator>>(std::istream& in,
                                    Point& point);
    friend Rectangle;
};

class Rectangle
{
    Point leftUpCorner;
    Point rightDownCorner;
public:
    Rectangle() = default;
    Rectangle(const Point& leftUpCornerP, int sideAP,
              int sideBP)
        : leftUpCorner{ leftUpCornerP },
        rightDownCorner{leftUpCornerP.x + sideAP,
                        leftUpCorner.y + sideBP}
    {}
    Rectangle(const Point& leftUpCornerP,
              const Point& rightDownCornerP)
        : leftUpCorner{ leftUpCornerP },
        rightDownCorner{rightDownCornerP}
    {}

    int getSideA() const { return rightDownCorner.x -
                           leftUpCorner.x; }
    int getSideB() const { return rightDownCorner.y -
                           leftUpCorner.y; }
    friend std::ostream& operator<<(std::ostream& out,
                           const Rectangle& rectangle);
};

std::ostream& operator<<(std::ostream& out,
                         const Point& point)
{
    out <<  '( ' << point.x <<  ', ' << point.y <<  ') ';
    return out;
}
```

```
std::istream& operator>>(std::istream& in, Point& point)
{
    in >> point.x >> point.y;
    return in;
}

std::ostream& operator<<(std::ostream& out,
                         const Rectangle& rectangle)
{
    out << "[ " << rectangle.leftUpCorner << ' '
        << rectangle.getSideA()
        << " X "
        << rectangle.getSideB()
        << ' ' << rectangle.rightDownCorner
        << " ] ";
    return out;
}

int main()
{
    Rectangle rect1{ {0,0}, 10 ,5 };
    Rectangle rect2{ {0,0}, {10,10} };

    std::cout << "Rectangle 1 " << rect1 << '\n '
              << "Rectangle 2 " << rect2 << '\n ';

    return 0;
}
```

The output code result of Example 36.


Microsoft Visual Studio Debug Console

```
Rectangle 1 [ (0,0) 10 X 5 (10,5) ]
Rectangle 2 [ (0,0) 10 X 10 (10,10) ]
```

Figure 35

We implemented the Rectangle class with two member variables of the Point type: the leftUpCorner is the upper left corner coordinate, and the rightDownCorner is the lower right corner coordinate. The Rectangle class is listed in the Point class as a friend class. This is what allows us to implement a number of the Rectangle class member functions effectively — a constructor that takes the upper left corner and two sides of the rectangle, as well as member functions that return the lengths of the two sides of the rectangle: getSideA() and getSideB(). The stream insertion operator overloading is implemented for the Rectangle class, necessary for convenient output result of the rectangle. The output result of Example 36 indicates the correctness of the program.

13. Homework

Task 1

Using Example 18, implement in the MedalsTable class the ability to dynamically set the medal table's size. The current implementation uses a 10-element static array for simplicity. Change it to a dynamically allocated array. Implement copy and move semantics for the MedalsTable class (two pairs: constructor/assignment operator).

Task 2

Complete the solution from Task 1 by overloading for stream insertion operator (operator <<) for the MedalRow and MedalsTable classes replacing the corresponding print() member functions in them.

Task 3

Complete Task 2 by implementing the function call operator for the MedalsTable class. The overloading must get a country identifier as an argument and return one of the constants: MedalRow::GOLD, MedalRow::SILVER, MedalRow::BRONSE a constant corresponding to the maximum number of medals for a given country. It means if Poland has 2 golden, 4 silver and 1 bronze medals, then the overloading of the function call operator with a POL parameter will return MedalRow::SILVER.

Task 4

Modify the NoSequence functor from Task 26 to ignore at least N values in sequence, where N is a constructor parameter of this functor.

Lesson 4.
Move Constructor. Overloading Special Operators.
Friend functions and Overloading