

Object-Oriented Programming Using

C++



Lesson 3

Const Methods.
Explicit constructor.
Operator Overloading

Contents

1. The Const Method	4
1.1. Declaration Syntax.....	5
1.2. Features of ‘this’ Pointer in a Constant Method	7
1.3. Constants as The Class Members	9
1.4. Examples of applying.....	9
2. Declaring a Constructor	
Using the ‘explicit’ Keyword.....	15
2.1. Examples, displaying an implicit object creation	15
2.2. The ‘Explicit’ Keyword and Its Usage	19
3. The Need to Use Operator Overloading.....	25
3.1. Code examples (implementation of classes through standard member methods like Sum, Mult, and others).....	25

3.2. The Logic of Using Standard Operators (+, — , > , < , etc.)	29
4. Operator Overloading.....	32
4.1. Fundamentals of operator overloading.....	32
4.2. Operator Overloading Examples	40
5. Summary	68
Section 1. The Const Method	68
Section 2. Declaring a constructor using the ‘explicit’ keyword	69
Section 3. The Need to Use Operator Overloading	70
6. Homework	78
Create the “Set of integers” class	78
7. Terminology	80

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

1. The Const Method

We know that C++ has constants, which (unlike many other programming languages) are often used, for example:

```
const int size = 3;  
const string s{"first day"};
```

Constant pointers and constant arrays are also used too:

```
const char* s{"Next day"};  
const int days{29,30, 31};
```

Class objects can also be const:

```
class Date  
{  
    int month;  
    int day;  
    int year;  
};  
  
const Date electionDay{ 11, 03, 2020 };
```

Const objects and references to the constant objects frequently appear in the text of programs as parameters to the class methods, e.g., copy constructor parameters:

```
Date(const Date& date)
```

When defining an object, the `const` keyword indicates that the object is immutable, and any attempt to modify

that object will be an error. A value cannot be assigned to a constant object; therefore, such an object can only receive a value by initialization. We cannot possibly change the value of a const object neither by direct assignment nor through the class functions.

```
electionDay.setYear(2024); // error!
```

Thus, all you can do with a const object is get the values of its elements (`get...`) or display them (`write...`). At first glance, strange things can occur with the constant objects.

```
int day = electionDay.getDay(); // error!
```

`getDay()` is a method of the `Date` class that returns the values of the date day.

```
int getDay()  
{  
    return day;  
}
```

This method does not change any data in the `election-Day...` object. But the compiler, when processing the `election-Day.getDay()` call knows nothing about it. And just in case, it does not allow calling this method on a constant object to prevent its possible change.

1.1. Declaration Syntax

However, there is a simple way out of this situation. Methods that do not change the value of the object (and therefore can be called for constants) should be marked as follows:

```
int getDay() const
{
    return day;
}
```

After the closing parenthesis in the function name, the `const` keyword means that the function does not change the object value. Moreover, the compiler checks whether this declaration is valid.

Such a function is called the `const` class member function or the `const` method. `Const` methods (and only they) can be called for the `const` objects. The method is specified as `constant` in the prototype and the method definition.

It should be noted that it is possible to overload a function in a class in such a way as to have a `constant` and a `non-const` version of this function at one stroke.:

```
double getValue();
double getValue() const;
```

The need for two such versions occurs when overloading operators (which we will discuss later). A `non-const` overload is needed simultaneously for reading/writing objects and the `const` overloading for the `const` objects.

Constructors and destructors cannot be declared as the `const`. Indeed, the purpose of these particular methods is an indispensable changing of the object (creation or destruction); therefore, they cannot be `constant`.

1.2. Features of 'this' Pointer in a Constant Method

Let's consider a small example (the program code is in the folder *Lesson03\les03_00_01*):

```
#include <iostream>
#include <stdio.h>
class Date
{
private:
    int day;
    int month;
    int year;
public:
    void setDay(int value)
    {
        day = value;
    }

    int getDay() const
    {
        return day;
    }
};

int main()
{
    Date aDate;
    aDate.setDay(10);
    std::cout << aDate.getDay() << std::endl;

    getchar();
    return 0;
}
```

When processing the `aDate.setDay(10);` operator, compiler perceives the `aDate` object as a hidden parameter, and transforms the `setDay(10)` call in a functions `setDay(&aDate, 10)`

call. Accordingly to that, the `setDate` method is converted to a function with two parameters via the compiler:

```
void setDate(Date* const this, int value)
{
    this->day = value;
}
```

When compiling an ordinary (non-static) class method, the compiler implicitly adds an implicit `*this` parameter to the parameter list. `*this` Pointer is a hidden pointer containing the object's address that calls the class method.

The `const` keyword specified after the closing parenthesis in the function name has nothing to do with the value returned by the function, e.g., the `int getDate() const`. It refers only to the hidden '`this`' pointer used in the function and means what is pointed to '`this`' (i.e., object data) — cannot be changed.

If a function is declared with a constant '`this`' pointer, it will be forbidden to change the data of class objects in the function body. Upon attempt to change an object in the code, a compilation error will occur (see Figure 1).

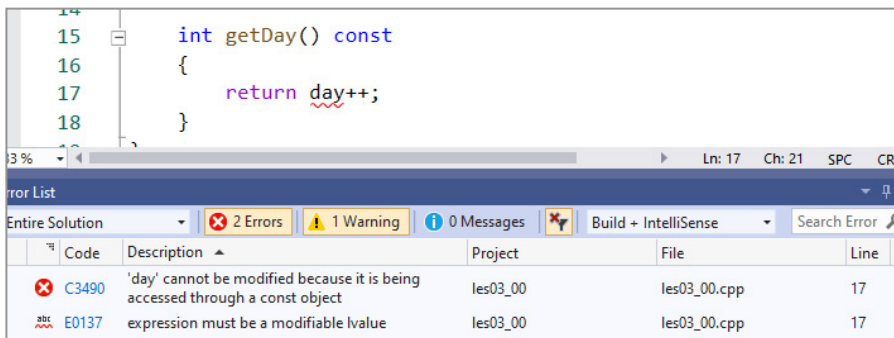


Figure 1. Constant method compilation error

When trying to change a class member variable ‘day’ in a constant function, the compiler outputs an error message: “‘day’ cannot be modified because it is being accessed through a const object”.

1.3. Constants as The Class Members

Class fields that are constants and fields that are references cannot receive values anywhere except at the moment of initialization in the class code. And upon that, they must be initialized only with the initializers (the program code is in the folder *Lesson03\les03_00_02*).

```
class Date
{
private:
    const int baseYear;
    int& currentYear;
    int day;
    int month;
    int year;
public:
    Date(int currYear) : baseYear{ 2000 },
                        currentYear(currYear)
    }
}
```

The `baseYear` constant and the `currentYear` reference get values in the class constructor initialization list.

1.4. Examples of applying

Let’s analyze the next program, which has the `Account` class and two objects of this class: a non-const `account 1` and a constant `account2` (the program code is in the folder *Lesson03\les03_01*).

Errors have been deliberately introduced into this code to demonstrate possible problems that arise when using (or not using) the constant methods.

```
#include <conio.h>
using namespace std;

class Account
{
private:
    double sum;
    const double rate;

public:
    Account(double Rate, double Sum)
    {
        // Error C2789
        rate = Rate; // "Account::rate": an object
                     // of const-qualified type must be
                     // initialized
                     // as a const class
                     // Error C2166
                     // l-value specifies
                     // the const object

        sum = Sum;
    }

    double getRate() const
    {
        return rate;
    }

    double getIncome()
    {
        return sum * rate / 100;
    }
}
```

```

double getIncome() const
{
    return sum * rate / 100;
}

double getSum()
{
    return sum;
}

double setSum() const
{
    sum += getIncome(); // Error    C3490
                        // "sum" cannot be modified
                        // because it is being accessed
                        // through a const object
    return sum;
}
};

int main()
{
    Account account1(5, 2000);
    const Account account2(8, 5000);

    account1.getRate();
    account2.getRate();    // correct

    account1.getSum();
    account2.getSum();    // Error    C2662
                        // double Account::getSum(void) :
                        // unable to convert "this" pointer
                        // from the "const Account"
                        // to the "Account &"

    account1.getIncome();
    account2.getIncome();
}

```

```

    account1.setSum();
    account2.setSum();

    _getch();
    return 0;
}

```

While starting the program, we see the compilation errors:

- `rate = Rate;` — when trying to set a value to a constant class field using an assignment operator;
- `sum += get Income();` — when trying to modify the value of a class field in a constant method;
- `account2.getSum();` — when calling a non-const method on a const object.

And, fix the errors (the program code is in the folder *Lesson03\les03_03*):

```

#include <iostream>
#include <conio.h>

using namespace std;

class Account
{
private:
    double sum;
    const double rate;

public:
    Account(double Rate, double Sum) : rate{ Rate }
    {
        this->sum = Sum;
    }
}

```

```

double getRate() const
{
    return rate;
}

double getIncome() // overloading: the non-const method
{
    return sum * rate / 100;
}

double getIncome() const // overloading: the same but
                          // the constant method
{
    return sum * rate / 100;
}

double getSum() const
{
    return sum;
}

double setSum()
{
    sum += getIncome();
    return sum;
}
};

int main()
{
    Account account1(5, 2000);
    const Account account2(8, 5000); // the constant
                                     // object

    account1.getRate();
    account2.getRate();
    account1.getSum();
    account2.getSum();
}

```

```

account1.getIncome(); // the non-constant double
                        // getIncome() method is called
account2.getIncome(); // the constant double
                        // getIncome() const method
                        // is called

account1.setSum();

_getch();
return 0;
}

```

The program compiles and launches.

Generally, in programs you should make as the constant methods:

- the accessor methods (`getValue() const`);
- methods that display information (`printArray() const`).

All methods that return any characteristics of an object must be declared as the const, without changing the object's value. Otherwise, they cannot be applied to the const objects or even to the standard objects accessed through a constant pointer or a reference. Missing the const in the description of such methods means the wrong programming style in C++.

2. Declaring a Constructor Using the 'explicit' Keyword

2.1. Examples, displaying an implicit object creation

While data processing, it often becomes necessary to convert one data type to another. Type conversion can be performed in the following cases:

- assigning or initializing a variable to a value of another data type:

```
double a(10); // initialization of 'a' variable
              // ('double' type) through the 10 constant
              // ('int' type)

a = 5;        // assign the 5 constant (int type)
              // to the 'a' variable (double type)
```

- passing a value to a function when the type of the passed value differs from the type of the parameter:

```
void doSomething(long number)
{
}

doSomething(5); // passing the 5 values (int type)
               // to the function,
               // with the long type parameter
```

- return from the function when the return type value in the 'return' is different from the return type value in the name function:

```
float doSomething()
{
    return 10.0; // return the 10.0 value (double type)
                // from the function
                // which returns a value of the float type
}
```

- using a binary operator with operands of different types:

```
double d = 5.0 / 4; // division operation with the values
                   // of double and int types
```

Type conversions can also be performed for data types developed by the programmer: conversions between objects of different classes and conversions between the class objects and fundamental types.

Let's give an example of classes for which such transformations would be common and desirable to execute.

It would be convenient to develop the following set of classes to work with date and time:

- `Date`;
- `Time`;
- `DateTime`;
- `TimeSpan`.

In C++, the `time_t` type is used to represent time, which values represent the number of seconds that have elapsed since some base time point (00:00 January 1, 1970). On the whole, the `time_t` values are the integers (`int`).

Thus, conversions between these classes are needed: `Date`, `Time`, `Date`, `Time`, and between these classes data and the data

of the `int` type, for comfortable working with date and time classes and the use of existing functions.

As we know, there are two ways to convert types:

- implicit type conversion, when the compiler automatically converts one data type to another;
- explicit type conversion, when one of the explicit conversion operators is used to convert an object from one data type to another.

By default, the C++ language understands a class constructor with 1 parameter as an implicit conversion operator from the parameter type to the class type.

Consider an example (the program code is in the folder *Lesson03\les03_05*).

```
#include <iostream>
#include <stdio.h>

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

public:
    Date(int day, int month, int year)
        : day{ day }, month{ month }, year{ year }
    {}
    Date(int year) : Date(1, 1, year)
    {}
    friend void displayDate(Date date);
};
```

```

void displayDate(Date date)
{
    std::cout << date.day << "." << date.month << "."
               << date.year << std::endl;
}

Date baseDate()
{
    return 2000;
}


int main()
{
    displayDate(2020);

    Date date = 2010;
    displayDate(date);

    Date date2000 = baseDate();
    displayDate(date2000);
    getchar();
    return 0;
}

```

The program result is shown in Figure 2.



```

1.1.2020
1.1.2010
1.1.2000

```

Figure 2. Implicit creation of the `Date` objects

In the example above, the `Date` constructor has defined with one an integer as a parameter:

```
Date(int)
```

This constructor is called implicitly:

- when calling a function with the `Date` parameter type and actual `int` parameter

```
displayDate(2020)
```

- when performing an assignment an integer to the `Date` type variable

```
Date date = 2010
```

- when returning an integer through a function that returns the `Date`

```
Date date2000 = baseDate()
```

In all cases, implicit conversion of an integer is executed to an object of the `Date` type.

2.2. The 'Explicit' Keyword and Its Usage

2.2.1. Declaring a constructor using the 'explicit' keyword

However, such a conversion is not always desirable, and sometimes it may turn out as a mistake.

Consider the following example (the program code is in the folder *Lesson03\les03_06*).

```
#include <iostream>
#include <conio.h>
using namespace std;

class Array
{
    int size;
```

```
int* array;

public:
    Array(int size = 10);
    ~Array();
    int getSize() const;
    int getValue(int index) const;
    void setValue(int index, int value);
    void display(int index) const;
};

Array::Array(int size)
{
    Array::size = size;
    array = new int[size];
}

Array::~~Array()
{
    delete[] array;
}

int Array::getSize() const
{
    return size;
}

int Array::getValue(int index) const
{
    return array[index];
}

void Array::setValue(int index, int value)
{
    array[index] = value;
}
```

```
void Array::display(int index) const
{
    cout << array[index] << " ";
}

void display(const Array& array)
{
    for (int i = 0; i < array.getSize(); i++)
    {
        array.display(i);
    }
    cout << endl;
}

int main()
{
    cout << "Dynamic integer array" << endl;
    int size = 4;
    Array array(size);

    for (int i = 0; i < size; i++)
    {
        array.setValue(i, size - i);
    }
    display(array);

    cout << "!!!" << endl;
    display(3);

    _getch();
    return 0;
}
```

The program result is shown in Figure 3.

```
Dynamic integer array
4 3 2 1
!!! 🇷🇺
-842150451 -842150451 -842150451
_
```

Figure 3. Implicit the `Array` object creation

The `Array` class is defined with the `Array(int)` constructor. In the example above, the constructor creates an empty dynamic array of the specified size. A global `display(const Array& array)` function prints an array to the screen.

```
int size = 4;
Array array(size);
display(array);
```

And everything would be fine. But... When calling the `display(3)` function, the compiler expects a parameter of the `Array` type as set in the function `display` header (`const Array& array`) and receives a parameter of 3 of the `int` type. As we know, in such a situation (passing a value to a function when the value type being passed differs from the type of the parameter), the value type (3) is converted to the parameter type (`Array`), if such a transformation is possible.

But, it is possible! Since there is the `Array(int)` constructor. Hence, the compiler converts parameter 3 to an object of the `Array` type, that is, creates an `Array[3]` object by calling the `Array(3)` constructor, resulting in a `display(3)` function

parameter becomes an empty `Array[3]`. The data of this empty array is displayed on the screen.

Apparently, this is not at all what was expected from the `display(3)`...

Actually, `display(3)` calling is an error. There is no point in such a function call. But the compiler missed this programmer's mistake, which can be considered a disadvantage of the programming language.

In C++ this disadvantage is corrected. An implicit constructor call can be eliminated using the 'explicit' keyword, writing it before the constructor name.

```
explicit Array(int size = 10);
```

`Explicit` is written only in the method prototype. It is not necessary to repeat it in the constructor implementation code, and it remains so

```
Array::Array(int size)
```

Now, having the 'explicit' in the constructor, we get a compilation error message when we run the program:

```
C2664 'void display(const Array &)':  
      cannot convert argument 1  
      from 'int' to 'const Array &'
```

We get a working program without displaying an empty array on the screen by commenting (or removing) the call in error `// display(3)`.

Using an 'explicit' constructor prevents implicit conversions from being performed. Explicit conversions (via explicit

conversion operators) will be allowed. If necessary, we can input a `display(Array(3))`; and get the initial result.

The implicit conversion will also be performed at uniform initialization.

```
//Array array10 = 10;           incorrect – compilation error  
Array array10{ 10 };           // that is allowed
```

Simple but essential rule follows from the above: Constructors with one parameter should be made explicit using the ‘`explicit`’ as a keyword to prevent errors associated with the class objects’ implicit conversion (accidental creation).

3. The Need to Use Operator Overloading

3.1. Code examples (implementation of classes through standard member methods like Sum, Mult, and others)

Human activity is always purposeful. A programmer writes a program to solve a problem.

A programmer creates classes in the program with the purpose of making the program friendly to use and understandable. Then it will have fewer errors. It will be written faster, and, what is most important, — the program will be easier to develop and use to solve other problems.

Let's consider an example. Suppose we have to perform a series of geometric calculations related to the use of coordinates of points and vectors on the coordinate plane. And obviously, we can store the coordinates in the program as two arrays:

```
int count;  
cin >> count;  
  
double* x = new double[count];  
  
double* y = new double[count];
```

But since two coordinates (x, y) represent a single entity — a point on the plane, combining two coordinates into one object in the program will be more convenient.

So, let's create and use the `Point` class

```
class Point
{
private:
    double x;
    double y;
};

int main()
{
    int count;
    cin >> count;
    Point* points = new Point[count];
```

This class, once created, will be helpful to us for solving many problems with points and vectors on the plane. However, you will have to work first to write the functions necessary to solve these problems. Let's name some of these functions.

- Displaying a point on the screen:

```
void display() const
```

- Reading a point:

```
void read()
```

- Comparing the two points (or vectors):

```
static bool isEqual(const Point& point1, const Point& point2)
```

- Adding two vectors:

```
static Point add(const Point& point1, const Point& point2)
```

- Multiplying vector by a number:

```
static Point mult(const Point& point, double value)
```

- Distance between two points:

```
static double distance(const Point& point1, const Point& point2)
```

- Vector length:

```
static double length(const Point& point)
```

The test program for the developed class will be like this (the program code is in the folder *Lesson03\les03_07*):

```
int main()
{
    Point point1(1, 1);
    Point point2;
    Point point3(1, 1);

    if (Point::isEqual(point1, point3))
    {
        cout << "point1 and point3 are equal" << endl;
    }

    cout << "p1: ";
    point1.display();
    cout << endl;

    cout << "Enter point p2 in format x,y (e.g. 12,10) : ";
    point2.read();

    cout << "p2: ";
    point2.display();
    cout << endl;
```

```

    cout << "p2 + p1 = ";
    Point::add(point2, point1).display();
    cout << endl;

    cout << "Distance between ";
    point1.display();
    cout << " and ";
    point2.display();
    cout << " is ";
    cout << Point::distance(point1, point2);
    cout << endl;

    _getch();
    return 0;
}

```

The program result is shown in Figure 4.

```

point1 and point3 are equal
p1: (1,1)
Enter point p2 in format x,y (e.g. 12,10) : 2,2
p2: (2,2)
p2 + p1 = (3,3)
Distance between (1,1) and (2,2) is 1.41421

```

Figure 4. Execution of the test program
for the `Point` class.

We see that the program results are correct, but the same cannot be said about the program's text.

The fact is that the task is mathematical, and the program text looks awkward from the mathematic point. And the program text is not very convenient for creating and editing from a programmer's point of view.

3.2. The Logic of Using Standard Operators (+, — , >, < , etc.)

When writing a program for arithmetical objects, it would be better if the description of arithmetical expressions was used. And the usual arithmetical operators and the usual standard input-output operators were used in this description.

And such a possibility exists.

We can apply the function overloading method to C++ operators. We can define our versions of operators that will work with the data of our classes appropriately. In C++, an example of operator overloading is the << operator, which is used as a stream write operator and a bitwise left shift assignment.

Here is a test program with the same functionality as in the previous example, but with the overloaded operators (the program code is in the folder *Lesson03\les03_08*):

```
int main()
{
    Point point1(1, 1);
    Point point2;
    Point point3(1, 1);

    if (point1 == point3)
    {
        cout << "point1 and point3 are equal" << endl;
    }

    cout << "p1: ";
    point1.display();
    cout << endl;

    cout << "Enter point p2 in format x,y (e.g. 12,10) : ";
    point2.read();
}
```

```

cout << "p2: ";
point2.display();
cout << endl;

point3 = point1 + point2;
cout << "p1 + p2 = ";
point3.display();
cout << endl;

cout << "Distance between ";
point1.display();
cout << " and ";
point2.display();
cout << " is ";
cout << point1 % point2 << endl;

cout << "Vector ";
point1.display();
cout << " length is ";
cout << !point1 << endl;

_getch();
return 0;
}

```

The program result is shown in Figure 5.

```

point1 and point3 are equal
p1: (1,1)
Enter point p2 in format x,y (e.g. 12,10) : 2,2
p2: (2,2)
p1 + p2 = (3,3)
Distance between (1,1) and (2,2) is 1.41421
Vector (1,1) length is 1.41421

```

Figure 5. Test program execution for the `Point` class using overloaded operators

The program has become shorter and more intuitive.

The operators ‘==’ and ‘+’ are used in the usual way in a clear context:

- == — comparison of variables ([Point](#) objects)

```
if (point1 == point3)
```

- + — adding the values of variables ([Point](#) objects)

```
cout << "p2 + p1 = " << point2 + point1 << endl;
```

There are also some unexpected operator overloads in this example:

- % — distance between the points ([Point](#) objects)

```
cout << "Distance between " << point1 << " and "  
    << point2 << ": " << point1 % point2 << endl;
```

- ! — vector length ([Point](#) object)

```
cout << "Vector "  
point1.display();  
  
cout << " length is "  
cout << !point1 << endl;
```

These overload options are shown by example only on how to proceed with those cases.

It would be better to avoid such unusual and incomprehensible constructions. The possibility of overloading operators exists not for the code obfuscation, but on the contrary, to make it a standard look.

4. Operator Overloading

4.1. Fundamentals of operator overloading

4.1.1. Classification of operators based on the number of the operand (binary, unary, triad)

Operator is a character (sometimes several consecutive characters) denoting a specific operation on data. C++ contains the following groups of operators:

- arithmetic (+, -, *, /, %, ++, --);
- relational/comparison (==, !=, >, <, >=, <=);
- logical (&&, ||, !);
- bitwise (&, |, ^, ~, <<, >>);
- assignment operators (=, +=, -=, *=, /=, %=, <<=, >>=, &=, ^=, |=);
- other operators (sizeof, ? x : y, ,(comma), .(point), ->, cast, &, *).

The data, which work with operators, are called operands. By the number of operands, C++ operators are divided into three groups:

- **unary**. Work with one operand, e.g., with a unary minus (-x) or increment (a++). The operand can be to the left or right of the operator;
- **bitwise**. Work with two operands — left and right, e.g., with addition (a + b) or assignment (a += 10);
- **triadic** (*ternary*). They work with three operands. There is only one ternary operator in C++ (x < y ? x : y).

The operator overloading technique is implemented depending on the given operator's groups.

4.1.2. Defining the Operator Overloading

Operator overloading is an implementation of the polymorphism principle for execution with operators. Overloaded operators have the same name (operator's character) but operate on different operands.

When creating a class, operator overloading allows defining the actions a given operator will perform on objects of that class.

Overloading means creating a function that contains the 'operator' and an overloaded operator character, e.g., `operator+(Point p1, Point p2)`. An operator function can be defined as a class member and outside the class.

You can only overload operators that are already defined in C++. You cannot create new operators.

The precedence and associativity of all operators are fixed in C++ and cannot be changed by overloading. An overload also cannot change the number of operands' operators. You cannot overload operands of built-in C++ types.

Each operator is overloaded separately. For example, overloading the `+` operator does not mean automatically overloading the `+=` operator.

Suppose the operator function is defined as a separate function, and it is not a class member. In that case, the parameters number of such a function is the same as the number of the operator operands.

Function with unary operator (e.g. unary minus: `-x`) will contain one parameter: `<T> operator-(T op)`.

Function with binary operator (e.g., addition: $x + y$) will contain two parameters: `<T> operator+(T op1, T op1)`.

And upon that, the first operand (x) is passed to the first parameter of the function (`op1`), and the second operand (y) is given to the second parameter (`op2`).

At least one of the operator function parameters must represent a class type.

Let's look at an example:

```
#include <iostream>
#include <conio.h>

using namespace std;

class Point
{
public:
    double x;
    double y;

    Point(double x, double y) : x{ x }, y{ y }
    {
    }

    void display() const
    {
        cout << "(" << x << "," << y << ")";
    }
};

Point operator+(const Point& point1, const Point& point2)
{
    return Point(point1.x + point2.x, point1.y + point2.y);
}
```

```

int main()
{
    Point p1(1,1);
    Point p2(2, 2);
    Point p3 = p1 + p2;
    p3.display();    // (3, 3)

    _getch();
    return 0;
}

```

The `+` operator is overloaded for adding the objects of the `Point` class (vectors on the plane).

Overload is determined by a function:

```
Point operator+(const Point& point1, const Point& point2)
```

The function name is the `operator+`. It indicates the purpose of the function — i.e., overloading the addition operator (`+`). The function is defined outside the class, and it has two parameters and a return value of the `Point` type.

When performing the `p1 + p2` operation, the first `p1` summand appears as the first parameter (`const Point& point1`) of the `operator+` function. And the second `p2` summand becomes the second parameter (`const Point& point2`) of this function.

A new object is created in the `Point` function body with coordinates equal to the sum of parameter coordinates, and this object is returned to the call point. Thus, the adding of the vectors is implemented by using the `+` operator: `(p1 + p2)` is a vector that is the sum of the `p1` and `p2` vectors.

4.1.3. Different kinds of overloading (member method, friend function, global function)

When processing an expression containing an operator, the compiler does the following:

- if all operands are operands of the built-in data types, then the appropriate version of the operator is called

```
int a = 1;
int b = 2;
int c = a % b; // operator % is executed (getting
               // the remainder of the division)
               // for the integers;
```

- if there are no such operator overloads, the compiler will generate an error;

```
string a = "1";
string b = "2";
string c = a % b;    // compilation error
```

The compiler outputs the message: *Error C2676 binary ‘%’: ‘std::string’ does not define this operator or a conversion to a type acceptable to the predefined operator.*

- If any of the operands is a class object, the compiler will find the operator’s version working with such a data type.

```
Point p1 (1, 1);
double a = 10;
Point p2 = p1 * a; // the * operator is executed
                  // (multiplying a vector by a number),
                  //overloaded in the Point class
```

- If there is no such overloading, the compiler will convert user-defined data types into inline data. If this is also impossible, the compiler will output an error.

```
Point p10(1, 1);
double a = 10;
Point p20 = p10 % a; // compilation error
```

Error C2679 binary ‘%’: no operator found which takes a right-hand operand of type ‘double’ (or there is no acceptable conversion)

There are three ways to overload operators in C++:

- via friend functions;

```
friend Point operator+(const Point& point1,
                      const Point& point2)
{
    return Point(point1.x + point2.x, point1.y + point2.y);
}
```

- through the standard functions;

```
Point operator+(const Point& point1, const Point& point2)
{
    return Point(point1.x + point2.x, point1.y + point2.y);
}
```

- using the class methods.

```
Point operator+(const Point& point)
{
    return Point(this->x + point.x, this->y + point.y);
}
```

Some operators can be overloaded in any of these techniques, while others can only be overloaded in a specific way.

Consider various overloading options using the example of a binary '+' operator and the unary '-' operator for the `Point` class. When implementing any option, the vectors can be added and the negation operation can be performed on them like this:

```
Point p1(1,1);
Point p2(2,2);
Point p3 = p1 + p2;

p3.display();           // (3, 3)
(-p1).display();        // (-1, -1)
```

Standard function:

```
Point operator+(const Point& point1, const Point& point2)
{
    return Point(point1.x + point2.x, point1.y + point2.y);
}

Point operator-(const Point& point)
{
    return Point(-point.x, -point.y);
}
```

When overloading an operator through a standard function, we set the function name in the following way: `operator-Operator'sCharacter`. Operands represent the function parameters. The first parameter is the left operand for the bitwise operation, and the second parameter is the right operand. For

the unary operation: a single parameter is a single operand. The return value represents the operation result.

The disadvantage of the global overloading means if to implement it, the class member data must be public, which is highly objectionable in many cases.

Friend function:

```
friend Point operator+(const Point& point1,
                      const Point& point2)
{
    return Point(point1.x + point2.x, point1.y + point2.y);
}

friend Point operator-(const Point& point)
{
    return Point(-point.x, -point.y);
}
```

Operator overloading through a friend function is almost identical to global overloading, but it is devoid of its main disadvantage. The friend function has access to all the class fields (x and y in this example can be private). Friendly overloading is the correct overloading option in many cases.

Class method:

```
Point operator+(const Point& point2)
{
    return Point(this->x + point2.x, this->y + point2.y);
}

Point operator-()
{
    return Point(-this->x, -this->y);
}
```

As we can see, the syntax for overloading via a class method is different from the syntax for global and friend overloading.

The left operand becomes the implicit object pointed to by the hidden pointer `*this` when overloading a bitwise operator, and the right is a single parameter. There are no parameters overloading the unary operation, and the implicit object becomes the operand by pointer `*this`.

Operator overloading via class methods is not possible if the left operand is not a class (like `int`) or a class that we can't change (like `std::ostream`).

4.2. Operator Overloading Examples

4.2.1. Arithmetic Operators Overloading

4.2.1.1. Overloading of +, -, * Operators and Others Operators

Let's consider operator overloading using an example with the `Point` class (point or vector on the coordinate plane).

Binary operators that do not modify the left operand are the best overloaded via friend functions on the grounds of:

- overloading through class methods requires the first parameter to be the class object, which does not always correspond to the peculiarities of the operation. For instance,

```
Point point1(1, 1);  
Point point2 = 10 * point1;
```

it is impossible to overload operation `10 * point1` through the class methods since the first operand (`10`) is an integer, not the `Point` object;

- overloading via an ordinary function requires the public class fields, which goes against the OOP encapsulation principle.

Sometimes it's good to have the ability to perform operations both through the use of operators and through function calls with the same purpose.

Therefore, we will write the necessary functions for arithmetic operations with vectors, and then we will implement the overloading of arithmetic operands through calls to the corresponding functions (the program code is in the folder *Lesson03\les03_10*):

```
// ++functions of arithmetic operations
static const Point add(const Point& point1,
                      const Point& point2)
{
    return Point(point1.x + point2.x, point1.y + point2.y);
}

static const Point subtract(const Point& point1,
                           const Point& point2)
{
    return Point(point1.x - point2.x, point1.y - point2.y);
}

static const Point mult(const Point& point, double value)
{
    return Point(point.x * value, point.y * value);
}

static const Point divide(const Point& point, double value)
{
    return Point(point.x / value, point.y / value);
}
```

```

// --functions of arithmetic operations
// ++operands of arithmetic operations
friend const Point operator+(const Point& point1,
                             const Point& point2)
{
    return add(point1, point2);
}

friend const Point operator-(const Point& point1,
                             const Point& point2)
{
    return subtract(point1, point2);
}

friend const Point operator*(const Point& point,
                             double value)
{
    return mult(point, value);
}

friend const Point operator*(double value, const Point& point)
{
    return mult(point, value);
}

friend const Point operator/(const Point& point, double value)
{
    return divide(point, value);
}

// --operands of arithmetic operations
// ++unary minus
const Point operator-()
{
    return Point(-x, -y);
}
// --unary minus

```

Unary minus (one operand) implemented via a class method.

`Point` is a structure. It should be passed as a parameter by reference. Since the function does not change the value of this parameter, the reference must be constant (`const Point& point`).

The return value must also be constant (`const Point`). Otherwise, the compiler will accept the expressions as valid in which the overloaded operator will appear to the left of the assignment sign, like in `point1+point3=point2`.

4.2.1.2. Increment and Decrement Overloading

4.2.1.2.1. Purpose and Objectives of Increment and Decrement Overloading

The increment (by 1 value) and decrement (by 1) operators are so commonly used in programming that C++ has special operators for them (`++` and `--`). Moreover, these operators have two versions: prefix and postfix.

In the postfix version (`x++`, `y--`), the value of the expression is evaluated first with the contained operand, and then the value of the operand is changed.

In prefix version (`++x`, `--y`) is on the contrary: the operand value is changed first, and then the expression is evaluated.

Indeed, when overloading these operators, the features of the postfix and prefix forms must be kept.

4.2.1.2.2. Syntax of Overloading and Differences Between Postfix and Prefix Forms

The increment and decrement operators are unary, and they change the values of their operands. Therefore, overloading must be done with the class methods.

Overloading the prefix increment and decrement operators is implemented in the same way as overloading any other unary operators:

```
Point& operator++()
{
    ++x; ++y; return *this;
}

Point& operator--()
{
    --x; --y; return *this;
}
```

For postfix operators in C++ is used a special dummy parameter. This dummy integer parameter is not used in the function code. However, the dummy parameter distinguishes between the postfix and prefix increment/decrement operators.

```
const Point operator++(int)
{
    Point point{ x, y }; // a temporary object is created
                        // with the current values
    ++(*this);           // prefix overloading is performed
    return point;        // returns a temporary object
                        // with the current values
}

const Point operator--(int)
{
    Point point{ x, y };
    --(*this);
    return point;
}
```

The prefix increment/decrement operator is used to implement the overloading, but the initial value is returned, which is not modified yet.

And during the subsequent access to the object, the access will be passed to the assigned changed value.

4.2.2. Overloading Comparison and Logical Operators

Comparison operator overloading is similar to arithmetic operators overloading, and it is also implemented via a friend function.

```
friend bool operator==(const Point& point1,
                       const Point& point2)
{
    return point1.x == point2.x && point1.y == point2.y;
}

friend bool operator!=(const Point& point1,
                       const Point& point2)
{
    return !(point1.x == point2.x &&
             point1.y == point2.y);
}

friend bool operator>(const Point& point1,
                      const Point& point2)
{
    return length(point1) > length(point2);
}
```

The operators `==` and `!=` almost always have logic for objects of new classes. The operators `<`, `>`, `<=`, `>=` often do not have such a meaning, so they should not be overloaded.

Logical operators `!`, `&&`, `||` are even less likely to have any reasonable use for objects, and they are therefore reloaded even less frequently.

Overloading of `&&`, `||` operators are carried out through a friendly function, similar to arithmetic operators overloading.

Overloading of the `!` operator is similar to the unary minus overloading, and it is implemented through a class member function.

4.2.3. Returning by Reference. Operator overloading `<<`, `>>`

Some operations on objects can take the form of a chain (for bitwise operations):

```
operand1 <operator> operand2 <operator> operand3 . . .
```

or (for the unary operations):

```
operand <operator> operand <operator> operand . . .
```

The prefix increment operation is a clear example of such an operation:

```
int x = 1;
cout << ++(++x) << endl;    // result: 3
```

Overloading of this operation in the `Point` class should work similarly:

```
Point point1(1, 1);
cout << ++(++point1) << endl;    // result: (3,3)
```

For such a (correct) implementation, the result of the operation must be the same object, which is the object of the operation.

It means that the overload function must return the result by a reference (not a constant!).

```
Point& operator++()
{
    ++x; ++y; return *this;
}
```

The same situation occurs when performing input / output operations when:

```
cout << point1 << " " << point2 << " " << point3 << endl;
```

The above statement outputs the values of multiple points (`point1`, `point2`, `point3`) to the output stream.

Stream is an abstract object through which the program communicates with user. We can imagine stream as a sequence of characters that appear at one end and move to the other end as the program progresses.

There are two types of streams.

The data source is an external device (keyboard, file) at the input stream, and a program is a receiving object.

On the contrary, in the output stream, the source is the program, and the receiving object is an external device (monitor, file, printer).

C++ has several standard input-output streams provided to the program immediately after its launch (by including the `<iostream>` header file). The most commonly used are:

- `cin` is a class associated with standard input (usually the keyboard);
- `cout` is a class related to standard output (usually a monitor).

These classes allow you to work with all built-in data types. To handle custom input/output operations, we need to use the additional tools for working with streams.

These tools are the `istream` and `ostream` classes — basic input/output classes (see Figure. 6).

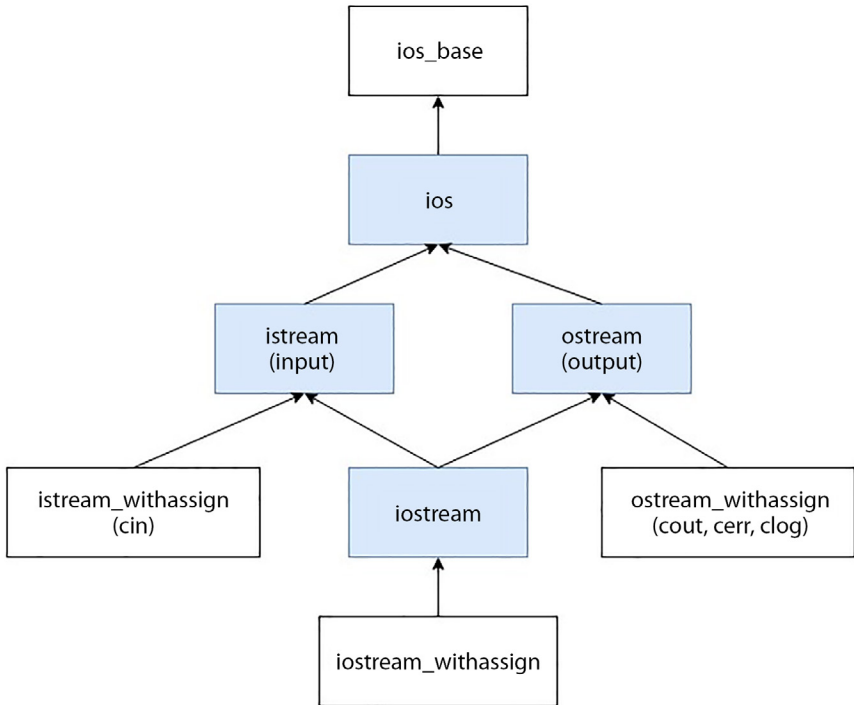


Figure 6. Input/output classes of the iostream library

The `ostream` class is responsible for data output, and it uses an overridden left shift operation (`<<`), called as the streaming operation.

The `istream` class is responsible for data input, and it uses an overridden right shift operation (`>>`), called the extraction operation from the stream.

Streaming and extraction operations allow calls in the same operator because they return the value of a stream reference.

The `istream` and `stream` classes overload the streaming and extraction operations for all built-in data types. Such overloading allows you to use a single syntax for input and output of characters, strings, integers, and real numbers.

Input/output operations can easily be extended to custom data types.

Output operation (`cout << point1`) is a bitwise operation with two operands:

- the first operand (`cout`) is a reference to the output stream,
- the second operand (`point`) is a constant reference to an object outputted to the stream.

The operation result (`cout << point1`) should be a stream reference (`cout`).

```
(cout << point) << point
| |
cout      << point
```

This is obligatory for the next object in the output chain can be outputted to this stream. So, input/output overloading should be implemented like this:

```
// ++input/output operators
friend ostream& operator<< (ostream& output,
                           const Point& point)
{
    output << "(" << point.x << "," << point.y << ")";
    return output;
}
```

```
friend istream& operator>> (istream& input, Point& point)
{
    input >> point.x;
    input.ignore(1);
    input >> point.y;
    return input;
}
// --input/output operators
```

The `operator<<` function outputs the elements of the point object. This function returns an output object (output stream) by reference so that this function return value can be the first parameter of the next output.

Input operator (`>>`) implemented similarly.

4.2.4. Assignment Operator Overloading

Assignment operator (`=`), unlike other operators, is generated automatically by the compiler in each developed class. This operator performs a member-by-member copy of class member variables by default.

And if there are no references to other objects in the class object that must also be copied, then there is no need to overload the assignment operator. The `Point` class does not require assignment operator overloading.

However, if the class has pointers to dynamic memory, then the default assignment will not work correctly and will require overloaded.

The assignment operator does much the same thing as the copy constructor: copying the elements of one object into another one. But, some differences exist:

- it is recommended to avoid self-assignment (`a = a`);

- it is required to return a reference (pointer) to the current object to be able to chain assignments;
- the object needs to be appropriately cleaned to which the new value is assigned to avoid the memory leak.

Let's consider an example of overloading an assignment operator in a class with dynamic memory (the program code is in the folder *Lesson03\les03_11*).

```
#include <iostream>
#include <conio.h>
using namespace std;

class Name
{
    char* firstName;
    char* secondName;

    void setCharArray(char*& dest, const char* source)
    {
        int strSize = strlen(source) + 1;
        dest = new char[strSize];
        strcpy_s(dest, strSize, source);
    }

    void remove()
    {
        if (firstName != nullptr)
        {
            delete[] firstName;
        }
        if (secondName != nullptr)
        {
            delete[] secondName;
        }
    }
}
```

```

public:
    Name()
    {
        firstName = nullptr;
        secondName = nullptr;
    }

    Name(const char* fName, const char* sName)
    {
        setCharArray(firstName, fName);
        setCharArray(secondName, sName);
    }

    Name(const Name& name)
    {
        setCharArray(firstName, name.firstName);
        setCharArray(secondName, name.secondName);
    }
    ~Name()
    {
        remove();
    }

    void write()
    {
        cout << firstName << " " << secondName << endl;
    }
};

void writeLine(Name name)
{
    name.write();
}

int main()
{
    setlocale(LC_ALL, "");

```

```

char firstName[10] = "John";
char secondName[10] = "Smith";
{
    Name name(firstName, secondName);
    cout << "name: object constructor executed ";
    name.write();
}
cout << "name: object destructor executed "
    << endl << endl;

{
    Name name(firstName, secondName);
    cout << "name: object constructor executed ";
    writeLine(name);
    cout << "name: object was copied ";
    name.write();
}
cout << "name: object destructor executed "
    << endl << endl;

Name aName;
{
    Name name(firstName, secondName);
    cout << "name: object constructor executed ";
    name.write();
    aName = name;
    cout << "aName: object assigned ";
    aName.write();
}
cout << "name object destructor executed"
    << endl << endl;

cout << "Accessing to aName: object ";
aName.write();
_getch();
return 0;
}

```

The program result is shown in Figure 7.

The program outputs an exception when it terminates its processing, as shown in Figure 8.

```
name: John Smith object constructor executed  
name object destructor executed  
  
name: John Smith object constructor executed  
name: John Smith object was copied  
name object destructor executed  
  
name: John Smith object constructor executed  
aName: John Smith object assigned  
name object destructor executed  
  
Accessing to aName: object EEEEEEEEE EEEEEEEEEEEEEEE
```

Figure 7. Test program execution for the `Name` class using dynamic memory



Figure 8. The test program completion for the `Name` class

The program uses the `Name` class containing pointers to the dynamic character arrays.

```
class Name
{
    char* firstName;
    char* secondName;
}
```

The `main` function creates, processes, and deletes objects of this class.

```
{
    Name name(firstName, secondName); // the name object
                                     // is created
    cout << "The name: object constructor executed ";
    writeLine(name); // the name object is copied
                   // when passing a parameter
    cout << "The name: object was copied ";
    name.write();
}                                     // the name object deleted
                                     // this is how the block ends
                                     // where it was defined

cout << "The name: object destructor executed "
     << endl << endl;
```

As we see, the class destructor and the copy operation work correctly. But after assigning an object to another object

```
aName = name;
```

errors occur in the program's operation, and the program's termination displays an exception (a sure sign that the program does not work correctly with dynamic memory).

And the following is found: after copying (on assigning) the `name` object, the pointers (`*firstName`, `*secondName`) of the `name` and `aName` objects point to one memory location. And after the deletion of the `name` object, this memory is complete, and the use of `aName` object pointers results in an error (see Figure 9).

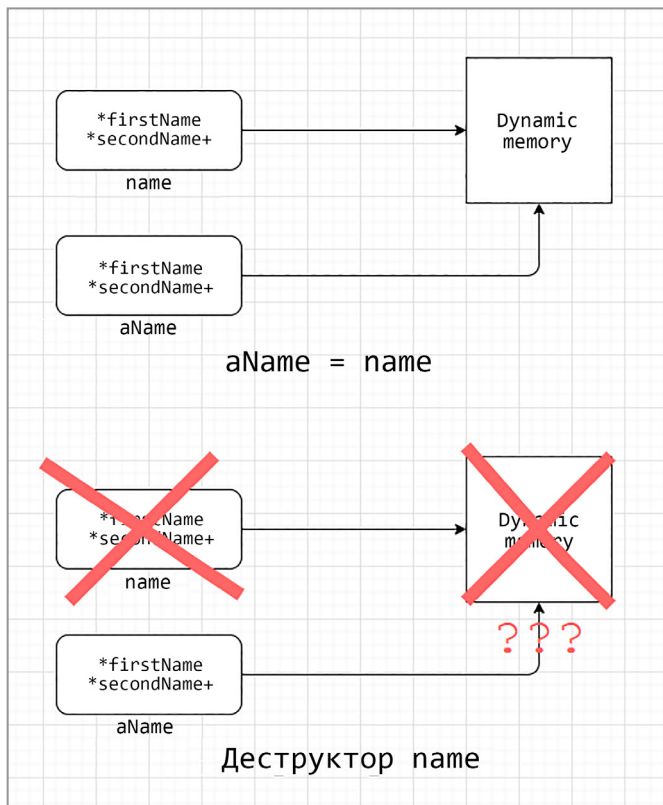


Figure 9. Error element-by-element assignment of the objects with pointers to dynamic memory

Therefore, overloading of the assignment operation is required in the `Name` class. During overloading, the dynamic

memory areas of the objects will be copied as well (the program code is in the folder *Lesson03\les03_12*).

```
Name& operator= (const Name& name)
{
    // omit the self-copying
    if (this == &name)
        return *this;

    // if there is a value, remove it
    remove();

    // deep copying
    setCharArray(firstName, name.firstName);
    setCharArray(secondName, name.secondName);

    // Return the current object
    return *this;
}
```

The assignment operator overloading is implemented through a class member function. The function parameter is a constant reference to the object being assigned. The return value refers to the current object that received the value.

If an object is assigned to itself (`this == &name`), the pointer will be returned to that object.

After that, the `remove()` function will be executed. This function checks the dynamic memory usage in the object to which the value is being assigned, and if so, the memory is deallocated.

Then, a deep copy (creating and copying of the dynamic arrays) of the source elements into the current object will be executed (`setCharArray` function).

The pointer returns to the current updated object (`return *this;`).

And now, the program runs without errors (see Figure 10).

```
name: John Smith object constructor executed
name object destructor executed

name: John Smith object constructor executed
name: John Smith object was copied
name object destructor executed

name: John Smith object constructor executed
aName: John Smith object assigned
name object destructor executed

Accessing to aName: John Smith
```

Figure 10. Test program execution for the Name class with assignment operator overloading

4.2.5. How do we choose an appropriate overloading method?

The following exceptions and limitations should be considered when overloading operators:

- You cannot define a new operator, e.g., as `operator**`.
- You cannot redefine operators for the standard type operands.
- The number of operands, execution order and operator associativity cannot be changed.
- At least one operand must be the class type for defining the overloading.

Operators that cannot be overloaded:

- `?:` (ternary operator);
- `::` (access to the nested names);

- `.` (access to the fields);
- `.*` (access to the fields by a pointer);
- `sizeof`, `typeid`, and `cast` operators.

Operators that can only be overloaded as class methods:

- `=` (assignment);
- `->` (access to the fields by a pointer);
- `()` (calling a function);
- `[]` (access by the index);
- `->*` (pointer-to-field access by a pointer);
- conversion and storage control operators.

In cases where the operator overloading method can be chosen, you need to pay attention to the following.

Working with binary operators that do not change the left operand (e.g., `operator+()`), overloading via regular or friend function is used. Since, this overload works for all parameter data types (even if the left operand is not an object of a class, or it is a class object that cannot be changed). Overloading via a regular/friend function has “symmetry” additional benefit. All operands become the explicit parameters (rather than overloading via a class method where the left operand becomes the implicit object pointed by the `*this` pointer).

Working with the binary operators that change the left operand (e.g., `operator+=()`), overloading through the class methods is usually used. In such cases, the left operand is always the class object pointed by the hidden `*this` pointer.

Unary operators are mostly overloaded via the class methods since the parameters are not used in such cases.

To sum it up:

- For assignment (`=`), index (`[]`), function call (`()`), or member selection (`->`) operators, use overloading through the class methods.
- Use overloading through the class methods for the unary operators.
- For overloading the binary operators that modify the left operand (e.g., `operator+=()`), use overloading through the class methods if possible.
- For overloading the binary operators that don't change the left operand (e.g., `operator+()`), use standard/friendly function overloading.

It is recommended that the purpose and meaning of the overloaded operator be close to its standard usage.

4.2.6. Example of a Class with Operator Overloading (Dynamic Array)

When working with the static C++ arrays, some functional operations are missing: there is no array comparison, and you cannot assign one array to another, an array does not define its size, and so on.

Using operator overloading, you can implement these valuable properties (the program code is in the folder *Lesson03\les03_13*).

Array.h: header file of the `Array` class

```
#include <iostream>

class Array {
    friend std::ostream& operator<<(std::ostream&, const Array&);
```

```

    friend std::istream& operator>>(std::istream&, Array&);

public:
    explicit Array(int = 10);
    Array(const Array&);
    ~Array();
    int length() const;
    const Array& operator=(const Array&);
    bool operator==(const Array&) const;
    bool operator!=(const Array& a) const {
        return !(*this == a);
    }

    int& operator[](int);
    int operator[](int) const;

private:
    int size;
    int* arr;
};

```

The class contains an array constructor with an array size, a default constructor, a copy constructor.

Such the operators have been overloaded: comparison (`==`, `!=`), assignment (`=`), indexing (`[]`), input/output (`<<`, `>>`).

There are two overload versions for indexing: standard and constant.

Input/output is implemented via friend functions.

Array.cpp: File with Array class implementation

```

#include <iostream>
#include <iomanip>
#include <stdexcept>
#include "Array.h"

```

```

using namespace std;

// default constructor with 10 elements
Array::Array(int aSize)
    : size{aSize}, arr{new int[size]{} }
{
}

// copy constructor
Array::Array(const Array& a)
    : size{a.size}, arr{new int[size]}
{
    for (int i = 0; i < size; ++i)
    {
        arr[i] = a.arr[i];
    }
}

// destructor
Array::~~Array()
{
    delete[] arr;
}

// file size
int Array::length() const
{
    return size;
}

// assignment operator
const Array& Array::operator=(const Array& a)
{
    if (&a != this)
    {
        if (size != a.size)
        {

```

```

        delete[] arr;
        size = a.size;
        arr = new int[size];
    }

    for (int i = 0; i < size; ++i)
    {
        arr[i] = a.arr[i];
    }
}

return *this;
}

// comparison operator
bool Array::operator==(const Array& a) const
{
    if (size != a.size)
    {
        return false;
    }

    for (int i{0}; i < size; ++i) {
        if (arr[i] != a.arr[i]) {
            return false;
        }
    }

    return true;
}

// index operator
int& Array::operator[](int index)
{
    if (index < 0 || index >= size)
    {
        cout << "Out of range" << endl;
    }
}

```

```

        _getch();
        exit(1);
    }

    return arr[index];
}

// 'get' index operator
int Array::operator[](int index) const
{
    if (index < 0 || index >= size)
    {
        cout << "Out of range" << endl;
        _getch();
        exit(1);
    }

    return arr[index];
}

// input operator
istream& operator>>(istream& input, Array& a)
{
    for (size_t i{0}; i < a.size; ++i) {
        input >> a.arr[i];
    }

    return input;
}

// output operator
ostream& operator<<(ostream& output, const Array& a)
{
    for (int i{0}; i < a.size; ++i) {
        output << a.arr[i] << " ";
    }
}

```



```

    output << endl;
    return output;
}

```

The `Array` class operators are overloaded with the standard methods.

When an index takes out of the array range, the message is shown “Out of range” and the program stops.

les03_13.cpp: file contains the test program for the `Array` class

```

#include <iostream>
#include <conio.h>
#include "Array.h"

using namespace std;

int main() {
    Array array1{ 5 };
    Array array2;

    for (int i = 0; i < 5; i++)
    {
        array1[i] = i;
    }

    for (int i = 0; i < 10; i++)
    {
        array2[i] = i + 11;
    }

    cout << "Size of array1: " << array1.length() << endl;
    cout << "Array1: " << array1 << endl;
    cout << "Size of array2: " << array2.length() << endl;
    cout << "Array2: " << array2 << endl;
}

```

```

cout << "(array1 == array2) ?" << endl;
if (array1 == array2)
{
    cout << "array1 == array2" << endl << endl;;
}
else
{
    cout << "array1 != array2" << endl << endl;;
}

cout << "Array array3{ array1 }; // copy constructor"
    << endl;
Array array3{ array1 }; // copy constructor
cout << "Size of array3: " << array3.length() << endl;
cout << "Array3: " << array3 << endl;

cout << "array1 = array2; // assignment operator"
    << endl;
array1 = array2; // assignment operator
cout << "Array1: " << array1;
cout << "Array2: " << array2 << endl;

cout << "array2[5] = 1000;" << endl;
array2[5] = 1000;
cout << "Array2: " << array2 << endl;

cout << "array2[15] = 1000;" << endl;
array2[15] = 1000; // error: index is out
                  // of the array range

_getch();
return 0;
}

```

The result of the test program for the `Array` class is shown in Figure 11.

```
Size of array1: 5
Array1: 0  1  2  3  4

Size of array2: 10
Array2: 11  12  13  14  15  16  17  18  19  20

(array1 == array2) ?
array1 != array2

Array array3{ array1 }; // copy constructor
Size of array3: 5
Array3: 0  1  2  3  4

array1 = array2;          // assignment operator
Array1: 11  12  13  14  15  16  17  18  19  20
Array2: 11  12  13  14  15  16  17  18  19  20

array2[5] = 1000;
Array2: 11  12  13  14  15  1000  17  18  19  20

array2[15] = 1000;
Out of range
```

Figure 11. Test program execution for the `Array` class

5. Summary

Section 1. The Const Method

- When defining an object, the `const` keyword indicates that the object is immutable, and any attempt to modify that object will be an error.
- A value cannot be assigned to a constant object; therefore, such an object can only receive a value by initialization.
- Calling a method that changes the object value on `const` objects results in a compilation error.
- After the closing parenthesis in the header of the function name, the `const` keyword, for example, `int getDay() const`, means that the function does not change the object value. Such a function is called the `const` class member function or the `const` method. `Const` methods (and only they) can be called for the `const` objects.
- Any object modification in the code of a constant method leads to a compilation error.
- The method is specified as constant in the prototype and the method definition.
- It is possible to overload a function in a class in such a way as to have a constant and a non-`const` version of this function at one stroke. It makes sense, and it is even necessary, e.g., when overloading the operator `[]` where you need both a non-`const` overload for reading and writing (`item[2] = 10;`), and the `const` overloading for the `const` objects (`cout << itemC[2];`).

- Constructors and destructors cannot be declared as the `const`.
- Constant class fields and fields that are references must be initialized with the initializers of these fields.
- All methods that return any characteristics of an object must be declared as the `const`, without changing the object's value. Otherwise, they cannot be applied to the `const` objects or even to the standard objects accessed through a constant pointer or a reference. Missing the `const` in the description of such methods means the wrong programming style in C++.

Section 2. Declaring a constructor using the 'explicit' keyword

- The compiler can call a constructor with one parameter to perform an implicit conversion.
- For instance, when calling the `displayDate(2020)` function, an object of the `Date` type is automatically created, instead of an integer value (2020), by executing the `Date(2020)` constructor.
- In some cases, calling a constructor for an implicit conversion is an error.

Like this:

```
Array array(size); // array constructor of the size
display(array);    // array output function
display(3);        // array(3) constructor is called
                  // implicitly

// and an empty array of 3 elements displays on the screen
```

- The `explicit` keyword does not allow implicit conversions using the one-parameter constructors. Constructor declared as `explicit` cannot be used in an implicit conversion.
- Declaring a constructor using the `explicit` keyword should be executed for constructors with one parameter.

Section 3. The Need to Use Operator Overloading

- We can apply the function overloading method to C++ operators. We can define our versions of operators that will work with the data of our classes appropriately.
- In C++, an example of operator overloading is operator `<<`, which is used as a stream write operator and a bit-wise left shift assignment.
- The actions performed by overloaded operators can be implemented by creating and calling the corresponding functions. However, writing the implementation using appropriate operators would be more explicit and shorter.
- Operator is a character (sometimes several consecutive characters) denoting a specific operation on data. That data is called operands which work with operators.
- By the number of operands, C++ operators are divided into three groups:
 - ▷ unary,
 - ▷ bitwise,
 - ▷ ternary.
- Unary operators interact with one operand, such as unary minus (`-x`) or increment (`a++`). The operand can be to the left or right of the operator;

- Binary operators interact with two operands — left and right, i.e., with addition (`a+b`) or assignment (`a += 10`);
- Ternary operators interact with three operands. There is only one ternary operator in C++ (`x < y ? x : y`).
- The operator overloading technique is implemented depending on these groups (binary or unary) the given operator belongs to.
- Overloading means creating a function that contains the 'operator' and an overloaded `operator` character, e.g., `operator+(Point p1, Point p2)`. An operator function can be defined as a class member and outside the class.
- You can only overload operators that are already defined in C++. You cannot create new operators.
- The precedence and associativity of all operators are fixed in C++, and cannot be changed by overloading.
- An overload also cannot change the number of operands' operators.
- You cannot overload operands of built-in C++ types.
- Each operator is overloaded separately. For example, overloading the operator `+` does not mean automatically overloading the `+=` operator.
- Suppose the operator function is defined as a separate function, and it is not a class member. In that case, the parameters number of such a function is the same as the number of the operator operands.
- Function with unary operator (e.g. unary minus: `-x`) will contain one parameter: `<T> operator!(T op)`
- Function with binary operator (e.g., addition: `x + y`) will contain two parameters: `<T> operator+(T op1, T op1)`.

- And upon that, the first operand (**x**) is passed to the first function parameter (**op1**), and the second operand (**y**) is passed to the second parameter (**op2**).
- At least one of the operator function parameters must represent a class type.
- There are three ways to overload operators in C++:
 - ▷ using standard functions;
 - ▷ using friend functions;
 - ▷ using the class methods.
- When overloading an operator through a standard function, we set the function name in the following way: **operatorOperator'sCharacter**. Operands represent the function parameters.
- The first parameter is the left operand for the bitwise operation, and the second parameter is the right operand.
- For the unary operation: a single parameter is a single operand. The return value represents the operation result.
- The disadvantage of the global overloading means if to implement it, the class member data must be public, which is highly objectionable in many cases.
- Class member function accesses to the left operand through the implicit 'this' pointer when overloading a bitwise operation, and the right operand accesses via this function parameter.
- Operator overloading through a friend function is almost identical to global overloading, but it is devoid of its main disadvantage. The friend function has access to all the class fields. Friendly overloading is the correct overloading option in many cases.

- The syntax for overloading through a class method is different from the syntax for global and friend overloading.
- The left operand becomes the implicit object pointed to by the hidden pointer `*this` when overloading a bitwise operator, and the right is a single parameter.
- There are no parameters overloading the unary operation, and the implicit object becomes the operand by the pointer `*this`.
- Operator overloading via class methods is not possible if the left operand is not a class (like `int`) or a class that we can't change (like `std::ostream`).
- Object parameters to an overload function are passed by a reference or by a constant reference.
- The return value must also be constant (`const Point`). Otherwise, the compiler will accept the expressions as valid in which the overloaded operator will appear to the left of the assignment sign, like in `point1 + point3 = point2;`
- The increment and decrement operators are unary, and they change the values of their operands. Therefore, overloading must be done with the class methods.
- Overloading the prefix increment and decrement operators is implemented in the same way as overloading any other unary operators.
- For postfix operators in C++ is used a special dummy parameter: `operator++(int)`. This dummy integer parameter is not used in the function code. However, the dummy parameter is there to distinguish between the postfix and prefix increment/decrement operators.

- The prefix increment/decrement operator is used to implement the overloading, but the initial value is returned, which is not modified yet.
- And during the subsequent access to the object, the access will be passed to the assigned changed value.
- Comparison operator overloading is similar to arithmetic operators overloading, and it is also implemented via a friend function.
- The operators `==` and `!=` almost always have logic for objects of new classes. The operators `<`, `>`, `<=`, `>=` often do not have such a meaning, so they should not be overloaded.
- Logical operators `!`, `&&`, `||` are even less likely to have any reasonable use for objects, and they are therefore reloaded even less frequently.
- Overloading of `&&`, `||` operators are carried out through a friendly function, similar to arithmetic operators overloading.
- Overloading of the `!` operator is similar to the unary minus overloading, and it is implemented through a class member function.
- Some operations on objects can take the form of a chain (for bitwise operations):

```
operand1 <operator> operand2 <operator> operand3 . . .
(cout << a << b<<c)
```

or (for the unary operations):

```
operand <operator> operand <operator> operand. . .
++(++x)
```

To implement such operations, the operation result must be the same object, which is the object of the operation. It means that the overload function must return the result by a reference (not a constant!).

- Output operation (`cout << object`) is a bitwise operation with two operands:
 - ▷ the first operand (`cout`) is a reference to the output stream,
 - ▷ the second operand (`object`) is a constant reference to an object outputted to the stream.
 - ▷ The operation result (`cout << object`) should be a stream reference (`cout`).

```
(cout << point) << object
  |  |
  cout    << object
```

This is obligatory for the next object in the output chain can be outputted to this stream.

So, input/output overloading should be implemented like this:

```
friend ostream& operator<< (ostream& output, const T& object)
{
    output << object.x << "," << object.y << endl;
    return output;
}
```

- Assignment operator (`=`), unlike other operators, is generated automatically by the compiler in each developed class. This operator performs a member-by-member

copy of class member variables by default. And if there are no references to other objects in the class object that must also be copied, then there is no need to overload the assignment operator.

- However, if the class has pointers to dynamic memory, then the default assignment will not work correctly and will require overloaded.
- The assignment operator does much the same thing as the copy constructor: copying the elements of one object into another one. But, some differences exist:
 - ▷ it is recommended to avoid self-assignment (`a = a`);
 - ▷ it is required to return a reference (pointer) to the current object to be able to chain assignments;
 - ▷ the object needs to be appropriately cleaned to which the new value is assigned to avoid the memory leak.
- The assignment operator overloading is implemented through a class member function. The function parameter is a constant reference to the object being assigned. The return value refers to the current object that received the value.
- The following exceptions and limitations should be considered when overloading operators:
 - ▷ You cannot define a new operator, e.g., as `operator**`.
 - ▷ You cannot redefine operators for the standard type operands.
 - ▷ The number of operands, execution order and operator associativity cannot be changed.
 - ▷ At least one operand must be the class type for defining the overloading.

- Operators that cannot be overloaded:
 - ▷ `?:` (ternary operator);
 - ▷ `::` (access to the nested names);
 - ▷ `.` (access to the fields);
 - ▷ `.*` (access to the fields by a pointer);
 - ▷ `sizeof`, `typeid`, and cast operators.
- Operators that can only be overloaded as class methods:
 - ▷ `=` (assignment);
 - ▷ `->` (access to the fields by a pointer);
 - ▷ `()` (calling a function);
 - ▷ `[]` (access by the index);
 - ▷ `->*` (pointer-to-field access by a pointer);
 - ▷ conversion and storage control operators.
- A class member function can overload a bitwise operator with one parameter or a global function with two parameters (one must be a class object or a reference to a class object).
- Member functions that implement operator overloading cannot be static because they need access to the non-static class data.
- The indexing operator (`operator[]`) must contain two overload versions: `const` (const method) and non-`const`.

6. Homework

Create the “Set of integers” class

A set is one of the fundamental concepts in mathematics.

A set is a collection of any objects called the set elements. Generally, each element in a set occurs only once, and these elements are not ordered.

A set is regularly defined just by listing its elements.

A set of integers (A) can be given, like this:

$$A = \{3, 8, 46, 5, 11\}.$$

And another set of integers (B) can be given as follows:

$$B = \{18, 8, 90, 11, 2\}.$$

A set is a collection of elements. Other collection sets in C++ are arrays, vectors, stacks. Like other types of collections, the set has its own set of operations.

Such operations include:

- addition elements to and deallocation elements out of a set

$$\{3, 8, 46, 5, 11\} + 4 = \{3, 8, 46, 5, 11, 4\}$$

$$\{3, 8, 46, 5, 11\} + 3 = \{3, 8, 46, 5, 11\} \text{ — (set elements are not repeated!)}$$
- set comparison (sets are equal if they contain the same set of the elements)
- basic operations on sets:
 - ▷ union of sets (mathematical notation: $A \cup B$)

$$\{3, 8, 46, 5, 11\} \cup \{18, 8, 90, 11, 2\} =$$

$$\{3, 8, 46, 5, 11, 18, 90, 2\} \text{ — (all elements of A + all elements of B)}$$

- ▷ the intersection of sets (mathematical notation: $A \cap B$)
 $\{3, 8, 46, 5, 11\} \cap \{18, 8, 90, 11, 2\} = \{8, 11\}$ — (elements are members of both A and B)
- ▷ set-theoretic difference (mathematical notation: $A \setminus B$)
 $\{3, 8, 46, 5, 11\} \setminus \{18, 8, 90, 11, 2\} = \{3, 46, 5\}$ — (elements belonging to A but not to B).

In the created “Set of integers” class:

- place the set elements into a dynamic array;
- each element of the set is unique (elements are not repeated);
- elements are not ordered.

Implement the following methods:

- constructors (default, with parameters, copy);
- destructor;
- checking elements for belonging to a set.

Implement the following operations:

- addition elements to a set (+, +=);
- union of two sets (+, +=);
- deallocation elements out of a set (-, -=);
- set-theoretic difference (-, -=);
- intersection of the sets (*, *=);
- assignment (=);
- set comparisons (==);
- input/output stream (<<, >>).

7. Terminology

- `*this`
- `const`
- `explicit`
- `istream`
- `operator`
- `operator!`
- `operator!=`
- `operator()`
- `operator[]`
- `operator+`
- `operator++`
- `operator++(int)`
- `operator+=`
- `operator-`
- `operator--`
- `operator-=`
- `operator<`
- `operator<=`
- `operator=`
- `operator==`
- `operator>`
- `operator>=`
- `operator>>`
- `operator<<`
- `ostream`

- stream
- bitwise operator
- const class member function
- const method
- const object
- copy constructor
- conversion constructor
- constructor with one parameter
- implicit constructor call
- function call operator ()
- conversion operator
- overloaded operator
- friend function operator overloading
- global function operator overloading
- operator overloading by a class member function
- triadic (ternary) operator
- unary operator



Lesson 3.

Const Methods. Explicit constructor. Operator Overloading

© STEP IT Academy, www.itstep.org

© Evgeniy Lactionov

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

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

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

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