Object-Oriented progtamming using

C++

Lesson N4

Object-Oriented Programming Using C++

Contents

Increment and decrement overloading

Previous lesson was devoted to operator overloading. We have also considered the example of class implementing the work with string. In this class overloading of the "=" and "+" binary operators has been used.

However, we should also consider unary operator overloading. In particular, we are going to focus on increment and decrement. In addition to the property of unary, these operators have two forms. This is very important when overloading.

It should be noted that in initial versions of C ++ there was no difference between the prefix and postfix forms when overloading the ++ and -- operators. For example:

```
#include <iostream>
using namespace std;
class Digit{
     //Integer number.
     int N;

public:
     //Constructor with a parameter
     Digit(int n)
     {
          N = n;
     }
     //function displaying the number
     void display()
     {
          cout << "\nDigit: N = " << N << "\n";
     }
```

```
      //Member function (the form is not distinguished):
      Digit& operator -- ()
      {
            //Reducing the object content
            //tenfold and returning it
            //to the place of operator calling
            N /= 10;
            return *this;
      }
};
      void main()
      {
      //creating Z object that will be used
      //for experimenting
      Digit Z(100);

      //displaying object in its original form
      cout<<"\nObject Z (before):\n";
      Z.display();

      cout<<"\n-----------------\n";

      //assigning a prefix expression
      //to the Pref object (in this case,
      //Z will be changed first and then
      //the assignment will be executed).
      Digit Pref=--Z;

      //showing the output of
      //the prefix form
      cout<<"\nPrefix\n";
      cout<<"\nObject Pref:\n";
      Pref.display();
      cout<<"\nObject Z (after):\n";
      Z.display();

      cout<<"\n-----------------\n";
      //assigning a postfix expression
      //to the Postf object (unfortunately, in this case
```



```
      //Z will be changed first again
      //and then the assignment
      //will be executed).
      Digit Postf=Z--;

      //showing the output of
      //the postfix form
      cout<<"\nPostfix\n";
      cout<<"\nObject Postf:\n";
      Postf.display();
      cout<<"\nObject Z (after):\n";
      Z.display();
}
```

```
Program output:
Object Z (before):
Digit: N = 100
-----------------
Prefix
Object Pref:
Digit: N = 10
Object Z (after):
Digit: N = 10
-----------------
Postfix
Object Postf:
Digit: N = 1
Object Z (after):
Digit: N = 1
```

In modern version of C++, the following conclusion was adopted:

Overloading the ++ and -- prefix operators does not differ from overloading other unary operators. In other words, functions of a particular class (++ operator and – operator) determine prefix operators for this class.

When determining the "++" and "--" postfix operators, function operators should have an additional parameter of the int type. When a postfix expression is used in the program, the function with a parameter of the int type is called. You !do not need! to pass the parameter. Its value will be null within the function.

```
#include <iostream>
using namespace std;
//A class implementing the work with "a pair of
numbers"
class Pair{
     //Integer number.
     int N;
     //Floating-point number.
     double x;
public:
     //Constructor with parameters
     Pair(int n, double xn)
     {
          N = n;
          x = xn;
     }
     //a function for displaying the data
     void display()
     {
          cout << "\nPair: N = " << N << " x = " << x <<
               "\n";
     }
     //Member function (prefix --):
     Pair& operator -- ()
     {
          //Reducing the object content
          //tenfold and returning it
          //to the place of operator calling
          N /= 10;
          x /= 10;
          return *this;
     }
     //Member function (postfix --):
     Pair& operator -- (int k)
     {
          //persisting the object
          //content to the independent
          //variable of the Pair type meanwhile
          //(Attempting to use
          //the value of the int k additional
          //parameter here
          //confirms that it is equal to 0.)
          Pair temp(0,0.0);
          temp.N=N+k;
          temp.x=x+k;

          //10 times reducing the object
          N /= 10;
          x /= 10;

          //return the previous value of the object.
          //this ploy is used
          //to obtain the effect of postfix
          //form, i.e. if A=B++,
          //current value of B object is recorded
          //to A, whereas B
          //is not changed
          return temp;
     }
};
void main()
{
     //creating Z object that will be used
     //for experimenting
     Pair Z(10,20.2);
     //displaying object in its original form
     cout<<"\nObject Z (before):\n";
     Z.display();
```

```
    cout<<"\n-----------------\n";
    //assigning a prefix expression
    //to the Pref object (in this case,
    //Z will be changed first and then
    //the assignment will be executed).
    Pair Pref=--Z;

    //showing the output of
    //the prefix form
    cout<<"\nPrefix\n";
    cout<<"\nObject Pref:\n";
    Pref.display();
    cout<<"\nObject Z (after):\n";
    Z.display();

    cout<<"\n-----------------\n";

    //assigning a postfix expression
    //to the Postf object (in this case,
    //the assignment will be executed first
    //and then Z will be changed).
    Pair Postf=Z--;

    //showing the output of
    //the postfix form
    cout<<"\nPostfix\n";
    cout<<"\nObject Postf:\n";
    Postf.display();
    cout<<"\nObject Z (after):\n";
    Z.display();
}

_____________________________________________________
Program output:
Object Z (before):
Pair: N = 10 x = 20.2
-----------------
```



```
Prefix
Object Pref:
Pair: N = 1 x = 2.02
Object Z (after):
Pair: N = 1 x = 2.02
Postfix
Object Postf:
Pair: N = 1 x = 2.02
Object Z (after):
Pair: N = 0 x = 0.202
```

Note: Despite the fact that one object is initialized by another when creating, we have not used the copy constructor in the two above examples. The reason is that it is not necessary, since bitwise copying is not critical here. So, it makes no sense to overload the code with an excess construction.

Subscript operator overloading

We have just considered peculiarities of the increment and decrement overloading. Now, let's learn about another "special" operator that is a subscript operator ([] — square brackets).

So, it is logical to assume that the expression **A[i]**, where **A** is a class object of abstract type, is represented by the compiler as **A.operator[](i)**. For example:

```
#include <iostream>
using namespace std;
class A{
     //an array of 10 elements
     //of the int type
     int a[10];
     //size of the array
     int size;
public:
     //constructor without parameters
     A(){
          size=10;
          for (int i = 0; i < 10; i++)
     //it is obvious that the [] operator,
     //used here,
     //within the A class constructor, is a standard
     //operator
     //since it is the int type array operator.
               a [i] = i + 1;
     }
     //subscript operator
     //overloading
     //reference return is executed
     //for the OBJECT[i]=VALUE situation
     //the object will return
     //to the place of the indexing call
     int& operator[](int j){
          //specific object return
          return a [j];
     }
     //the function returning
     //the array size
     int Get () const {
          return size;
     }
};

void main () {
     int i,j;
     //Working with one object of the A type
     A object;
     cout<<"\nOne object:\n";
     for(i=0;i<object.Get();i++){
          //the expression array[i] is interpreted as
          //object.operator [](j).
          cout<<object[i]<<" ";
     }
     cout<<"\n\n";
     //Working with the array of A objects
     A array [3];
     cout<<"\nArray of objects:\n";
     for(i=0;i<3;i++){
          for(j=0;j<object.Get();j++){
     //the expression array[i][j] is interpreted as
     //(array [i]).operator [](j).
    //The first of the two [] operators is standard,
```

```
      //since it is executed over the name of the array.
    //The type of its elements does not matter.
  //The second [] operator is overridden,
//since the first [] operator resulted in the A object.
                cout << array [i][j] << " ";
          }
          cout <<"\n\n";
     }
}
```

```
Program output:

One object:
1 2 3 4 5 6 7 8 9 10

Array of objects:
1 2 3 4 5 6 7 8 9 10
1 2 3 4 5 6 7 8 9 10
1 2 3 4 5 6 7 8 9 10
```

Note: Pay attention to the fact that in this example double brackets are not overloaded for the two-dimensional array. We simply create an array of class objects, in which the [] operator is overloaded.

Concluding chapter

... For those who have chosen a specialization that is other than programming. The fact is that starting from the next lesson the distribution by specializations will be carried out. Those students who have chosen design and administration will say good-bye to us. In this regard, today you have the opportunity to make up all your examinations. Namely, you can pass an exam in C. The tasks for this exam were provided in the nineteenth lesson. You can also make the grade in basics of C++, completing all the homework assignments for this course.

In addition, you will find a test in all the studied subjects of C++. There is also a homework assignment for those, who are still with us.

GOOD LUCK!!!

Homework assignment

Create a dynamic array class within which the bound-check is implemented. Overload the following operators: `[]`, `=`, `+`, `-`, `++` (adding an element to the end of the array), `--` (removing an element from the end of the array).