Laboratory lesson №5
(The spring session)

Key pointer this.

The key pointer **this** is a pointer to an object that calls the class method. The key pointer **this** is automatically set to a method of object, when the method is called. The pointer **this** is an implicit parameter accepted by all methods of the class. Accordingly, in any method, the pointer **this** can be used for reference to the object that calls it.

Friend functions

In C ++, it is possible to allow access to private class members to functions that are not members of this class. To do this, it is enough to declare these functions “friendly” (or “friends”) with respect to the class that is being considered. In order to make a function as “friend” of a class, you must include its prototype in the public-section of the class declaration with the keyword **friend.** Function may be a "friend" of several classes.

```
class someClass {
//...
public:
     friend void frnd (someClass obj);
//...
};
```

The keyword **friend** allows function, that is not a member of class, an access to its private members.

```
#include <iostream>
using namespace std;
class myclass {
int a, b;
public:
    myclass(int i, int j) { a=i; b=j; }
    friend int sum(myclass x);
};
int sum(myclass x)
{
    return x.a + x.b;
}
int main ()
{
  myclass n (3, 4);
  cout << sum(n);
  return 0;
}
```

Operator Overloading.

C ++ allows you to overload most operations so that standard operations can be used for objects created by the user classes.

Overloading operations allows you to use your own data types as standard, and turn complex and not clear program text into intuitive one. Designations of own operations cannot be entered.

You can overload any operations that exist in C++, with the exception of:

.   .*   ?   :   ::   #   ##   sizeof

Operations are overloaded using methods of a special form (functions-operations) and by using the following rules:

• the number of arguments, operation priorities and association rules (from left to right or from right to left) are used, which are used in standard data types;

- ➢ operations cannot be redefined for standard data types;
- ➢ functions-operators cannot have default arguments;
- ➢ functions-operators are inherited (except for "=");
- ➢ functions-operators cannot be defined as **static**.

Operation function can be defined in three ways:

- ✓ as a class method;
- ✓ as a "friendly" class function;
- ✓ as a simple function.

In the last two cases, the function must have at least one argument that has the class type, pointer, or class reference.

The operation function contains the keyword **operator**, followed by the sign of the operation to be redefined:

```
<type> operator <operation> (<parameter list>)
{ <function body> }
```

This syntax tells the compiler that if the operand belongs to a user-defined class, it is necessary to call a function with the same name in the program text.

Overloading operators using class methods.

It is possible to overload both binary operators (for example, "+", "="), and unary (for example, "++", "-").

When overloading a binary operation, the operator function has one parameter. In all cases, the object activates the operator function, passed implicitly using the this pointer. The object located to the right of the operation sign is transferred to the method as a parameter.

```
class three_d
{
    int x,y,z;
public:
    three_d operator+ (three_d op2);
};
three_d three_d::operator+(three_d op2)
{
    three_d temp;
    temp.x=x+op2.x;
    temp.y=y+op2.y;
    temp.z=z+op2.z;
    return temp;
}
```

Note:

`temp.x=x+op2.x;`

`x` corresponds to `this ->x` , where `x` is associated with the object that calls the operator function.

When unary operations are overloaded, the operator function has no parameters, so no object is transferred explicitly. The operation is executed on the object, which generates a call to this method through an implicit pointer.

Consider an example in which an increment operation is defined for objects of type ***three_d.***

Let's look at an example in which for the objects of class ***three_d*** the increment operation is defined

```
     class three_d
     {
         int x,y,z;
     public:
         three_d operator--();
     };
       three_d three_d::operator--()
       {
           x--;
           y--;
           z--;
       }
```

The increment and decrement operators have both prefix and postfix forms.

In the above example, the operator ++ function defines the prefix form of the “++” operator for the three_d class. If it is necessary to overload the postfix form of the “++” operator, this can be done as follows:

```
       three_d three_d::operator++(int notused)
```

The ***notused*** parameter is not used by the function itself. It is an indicator for the compiler, which allows to distinguish the prefix form of the increment operator from postfix. It is also used for the decrement operator (“--”).

Operator overloading with "friendly" functions

Since "friendly" functions are not members of a class, they cannot have an implicit argument **this**.

Therefore, when the binary function of the operator is overloaded, both operands are transferred to the function, and when the unary operators are overloaded, one operand is transferred.

The following operators **cannot be overloaded** using "friendly" functions:

= **( ) [ ] ->.**

```
class three_d {
    int x, y, z;
 public:
    friend three_d operator*(three_d op1, three_d op2);
};
three_d operator*(three_d op1, three_d op2)
{
    three_d temp;
    temp.x = op1.x * op2.x;
    temp.y = op1.y * op2.y;
    temp.z = op1.z * op2.z;
    return temp;
}
```

If it is necessary to overload the increment or decrement operators using the "friendly" function, you need to transfer its pointer to the object.

Since the parameter in this form is an implicit pointer to the argument, the changes that will be made to the parameter also affect the argument.

So, if function uses the pointer to the object as a parameter, it can increment or decrement an object that is transferred as an operand.

If the friend function is used to overload the increment and decrement operators, its prefix form takes one parameter (which is the operand), and the postfix form takes two parameters (the second is an integer value that is not used).

```
friend three_d operator++(three_d &op1)
{
     op1.x++;
     op1.y++;
     op1.z++;
     return op1;
}
friend three_d operator++(three_d &op1, int notused)
{
     three_d temp = op1;
     op1.x++;
     op1.y++;
     op1.z++;
     return temp;
}
```

Operator overloading using simple functions

The difference between operator overloading through simple and “friendly” functions is that they have different levels of access to private members of the class. Moreover, the “friendly” function must be declared in the class, and the simple function outside the class body.

If you use the usual function to overload operators, access to private class members must be organized through **getters.**

```
class three_d
{
    int x,y,z;
public:
    int getX(){return x;}
    int gexY(){return y;}
    int getZ() {return z;}
};
three_d operator+ (three_d &op1,three_d &op2)
{
     return
three_d(op1.getX()+op2.getX(),op1.gexY()+op2.gexY(),op1.getZ()+op2
.getZ());
}
```

Example.

Operator overloading on the example of **three_d** class.

Overload:

• operators "+", "=" and prefix and postfix decrements as class methods;

• operators "*", prefix and postfix increments as "friendly" functions;

• “-" operator as a simple function.

Header file **three_d.h**

```
#ifndef THREE_D_H_INCLUDED
#define THREE_D_H_INCLUDED
class three_d
{
    int x,y,z;
public:
    three_d(){x=y=z=0;}
    three_d(int i,int j,int k) {x=i;y=j;z=k;}
    int getX(){return x;}
    int gexY(){return y;}
    int getZ() {return z;}
    three_d operator+ (three_d op2);
    three_d operator= (three_d op2);
    three_d operator--();
    three_d operator--(int notused);
    friend three_d operator++(three_d &op1);
    friend three_d operator++(three_d &op1, int notused);
    friend three_d operator* (three_d &op1, three_d &op2);
    void show();
};
three_d operator- (three_d &op1,three_d &op2);

#endif // THREE_D_H_INCLUDED
```

File with class implementation **three_d.cpp**

```
#include "three_d.h"
#include <iostream>

using namespace std;
three_d operator- (three_d &op1,three_d &op2)
    {
        return three_d(op1.getX()-op2.getX(),op1.gexY()-
op2.gexY(),op1.getZ()-op2.getZ());
    }
three_d three_d::operator+(three_d op2)
  {
    three_d temp;
    temp.x=x+op2.x;
    temp.y=y+op2.y;
    temp.z=z+op2.z;
    return temp;
  }
three_d three_d::operator= (three_d op2)
  {
      x=op2.x;
      y=op2.y;
      z=op2.z;
      return *this;
  }
three_d three_d::operator--()
  {
      x--;
      y--;
      z--;
      return *this;
  }
three_d three_d::operator--(int notused)
  {
      three_d temp = *this;//storing an initial value
      x--;
      y--;
      z--;
      return temp;
  }
three_d operator* (three_d &op1, three_d &op2)
  {
      three_d temp;
      temp.x = op1.x *op2.x;
      temp.y = op1.y * op2.y;
      temp.z = op1.z * op2.z;
      return temp;
  }
three_d operator++(three_d &op1)
  {
      op1.x++;
      op1.y++;
      op1.z++;
      return op1;
  }
```

```
three_d operator++(three_d &op1, int notused)
  {
      three_d temp = op1;
      op1.x++;
      op1.y++;
      op1.z++;
      return temp;
  }
void three_d::show()
  {
      cout<<x<<", "<<y<<", "<<z<<"\n";
  }
```

File **main.cpp**

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

int main(){
     three_d a(1,2,3), b(10,10,10), c;
     cout<<"c=a+b \n";
     c=a+b;
     a.show();  b.show();  c.show();
     cout<<"c=b=a \n";
     c=b=a;
     a.show();  b.show();  c.show();
     cout<<"a=--c \n";
     a=--c;  // prefix form of decrement - the object receives value c after its decrement
     a.show(); c.show();
     cout<<"a=c-- \n";
     a=c--;// postfix form of decrement - the object receives value c after its decrement
     a.show(); c.show();
     cout<<"c=a*b \n";
      c=a*b;
      c.show();
      cout<<"a=++c; \n";
      a=++c; a.show(); c.show();
      cout<<"a=c++ \n";
      a=c++; a.show(); c.show();
      cout<<"c=a-b \n";
      c=a-b;   c.show();
  }
```

Laboratory work 5.
Task progress:
1. Model a class for a given concept (according to a variant).
2. Overload operator according to your variant.
It is necessary to use three methods of overloading:
- as a class method,
- as a "friendly" function,
- as a simple function.

3. Write a program in which the user will be able to use overloaded operators.

The variants:

Variant 1.

For class “**fraction”**.

Overload operators:

- addition (+)
- comparison (== ,! =),
- multiplication (*),
- decrement (-),
- assignment (=).

Variant 2.

For class “**line**” (defined through the coordinates of two points).

Overload operators:

- determining the parallelism of two lines (||),
- determining the angle between two lines (%),
- assignment (=),
- decrement (-).

Variant 2.

For class “parabola”. ($y = ax^2 + bx + c.$)

Overload operators:

- assignment (=),
- check for coincidence of parabolas (||),
- check for intersection of parabolas (/).