Laboratory lesson №6
(The spring session)

Operator Overloading 2.

Overloading Operation []

The indexing operation [] is overloaded in order to use the standard C ++ form of this operation to get an access the class members. Operation "[]" is overloaded as a binary operation.

It can be overloaded only for the class and only with the help of functions – class members.

However, since this operation is usually used to the left of the "=" sign, an overloaded function has to return its own value by reference.

The ***i-th*** element of the integer array ***beg*** is returned by overloading the indexing operation

```
int Vector::operator [](int i)
{
    if(i<0) cout<<"index <0";
    if(i>=size) cout<<"index>size";
    return beg[i];
}
```

The parameter of the function **operator [] ()** can have any data type: ***char, real, string***.

Overloading Operation ( )

C ++ allows you to overload ***the function-call operator ().*** In this case the function with an arbitrary number of parameters is created.

In the general, when ***operator ()*** is overloaded, parameters are determined that must be passed to operator () (). And the arguments set for the ***operator ( )*** are copied to these parameters. The object that generates the call to the operator function is addressed by the pointer ***this***. For example, for the ***Vector*** class:

```
void Vector:: operator()(int n)
{
    for(int i=0; i<size; i++)
        beg[i]=n*beg[i];
}
```

Overloading input >> and output << operators

The input and output of standard data types is provided in C++, using operators put into stream >> and take from stream <<. These operators are already overloaded in the <iosream> library to work with various standard data types, including dates and addresses. But these operators can be overloaded to input and output user-defined data types.

The operator functions for the input and output operators overloading cannot be the members of a class . They can be overloaded as friendly functions to have access to the class elements.

```
ostream &operator<<(ostream &output, const Vector &v)
{
    if(v.size==0) out<<"Empty\n";
    else
    {
        for (int i=0; i<v.size; i++)
```

```
            output<<v.beg[i]<<" ";
        out<<endl;
    }
    return output;
}
istream &operator >>(istream &input, Vector &v)
{
    for(int i=0; i<v.size; i++)
    {
        cout<<">";
        input>>v.beg[i];
    }
    return input;
}
```

Note, that these functions return a reference to an object of ***ostream*** or ***istream*** type. This allows you to combine several output operators in one.

Operator functions in this case have two parameters. The first is a link to the stream, which is used on the left side of the operator. The second represents the object that is on the right side of the operator. If necessary, the second parameter may also have a reference to the object. The body of the function itself, for this example, consists of instructions for outputting or inputting an array of the ***Vector*** class.

Overloading new and delete operators

The ***new*** and ***delete*** operators can also be overloaded in C ++. For various tasks, it may be necessary to create your own version of these operators.

The following format is used for overloading:

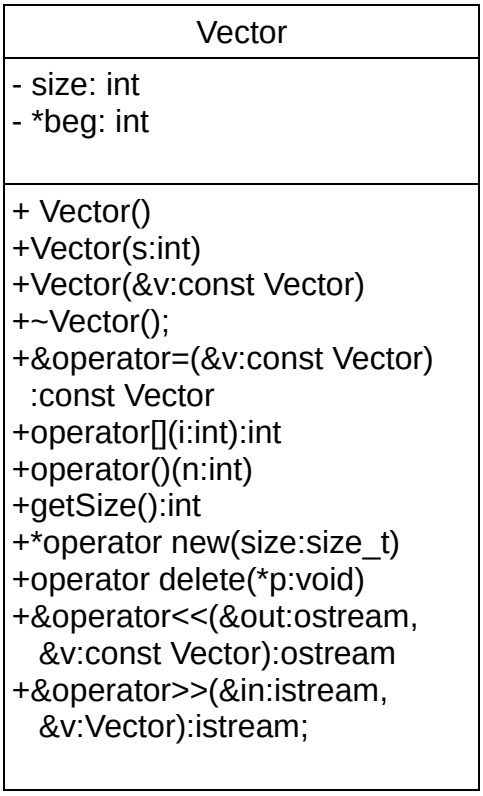
```
// Memory allocation for an object.
void *operator new(size_t size)
 {
// The constructor is called automatically
.
  return pointer_to_memory;
 }
// delete object
void operator delete(void *p)
 {//Memory deallocation. The constructor is called automatically
 }
```

The ***size_t*** type is specifically defined to provide the maximum required size of memory that can be allocated for an object. The type ***size_t*** is an unsigned integer type. The ***size_t*** parameter determines the number of bytes of memory required to store the object.

```
// Виділення пам'яті для масиву об'єктів.
void *operator new[](size_t size)
 {// Кожний конструктор викликається автоматично
   return pointer_to_memory;
 }
// Видалення масиву об'єктів.
void operator delete[](void *p)
 {//Деструктор для кожного об'єкту масиву викликається автоматично
 }
```

It is necessary to check that the memory allocation is correct.

The UML presentation of developed class:



The header file «Vector.h»

```
#ifndef THREE_D_H_INCLUDED
#define THREE_D_H_INCLUDED
#include <iostream>
using namespace std;
const int MAX_SIZE=20;
class Vector
{
    int size;
    int *beg;
public:
    Vector()
    {
        size=0;
        beg=0;
    }
    Vector(int s);
    Vector(const Vector &v);
    ~Vector();
//overloading operation =
    const Vector& operator=(const Vector&v);
//overloading operation[]
    int operator[](int i);
// overloading operator ()
    void operator()(int n);
    int getSize();
// overloading new and delete operators
    void *operator new(std::size_t size);
    void operator delete(void* p);
// overloading output and input operators
```

```
    friend ostream& operator<<(ostream&out,const Vector&v);
    friend istream& operator>>(istream& in, Vector&v);
};
#endif // THREE_D_H_INCLUDED
```

Class implementation file «Vector.cpp»

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

using namespace std;
Vector::Vector(int s)
{
    if(s>MAX_SIZE)
        cout<<"Vector length more than MAXSIZE\n";

    size=s;
    beg=new int [s];
    for(int i=0; i<size; i++)
        beg[i]=i;
}
Vector::Vector(const Vector &v)
{
    size=v.size;
    beg=new int [size];
    for(int i=0; i<size; i++)
        beg[i]=v.beg[i];
}
Vector::~Vector()
{
    if (beg!=0)
        delete[]beg;
}

const Vector& Vector::operator =(const Vector &v)
{
    if(this==&v)
        return *this;
    if(beg!=0)
        delete []beg;
    size=v.size;
    beg=new int [size];
    for(int i=0; i<size; i++)
        beg[i]=v.beg[i];
    return*this;
}
ostream& operator<<(ostream&output, const Vector&v)
{
    if(v.size==0) output<<"Empty\n";
    else
    {
        for (int i=0; i<v.size; i++)
            output<<v.beg[i]<<" ";
        output<<endl;
    }
```

```
    return output;
}
istream& operator >>(istream &input, Vector &v)
{
    for(int i=0; i<v.size; i++)
    {
        cout<<">";
        input>>v.beg[i];
    }
    return input;
}
int Vector::operator [](int i)
{
    if(i<0) cout<<"index <0";
    if(i>=size) cout<<"index>size";
    return beg[i];
}
int Vector::getSize ()
{
    return size;
}
void Vector:: operator()(int n)
{
    for(int i=0; i<size; i++)
        beg[i]=n*beg[i];
}
void *Vector::operator new(std::size_t size)
{
    cout<<"\n New memory \n";
    Vector *p=new Vector[size];
    if(!p)
        cout<<"Not memory \n";
    return p;
}
void Vector:: operator delete(void* p)
{
    cout << "Delete memory\n";
    return::operator delete(p);
}
```

File «main.cpp» with the e[ample of overloading operators

```
#include "Vector.h"
#include <iostream>
using namespace std;
int main()
{
        Vector x(15);
        Vector y(x);
        cout<<"Massiv x \n";
        cout<<x;
        cout<<"Nomer?\n";
        int i;
        cin>>i;
```

```
        cout<<x[i]<<endl;
        cout<<"Massiv y \n";
        cout<<y<<endl;
        x(3);
        cout<<"Massiv x*3 \n";
        cout<<x;
        Vector *p1=new Vector(5);
        cout<<"Size="<<p1->getSize();
        cout<<"\nel_3="<<(*p1)[3];
        cout<<"\n"<<*p1<<endl;
        (*p1)(5);
        cout<<"\n"<<*p1<<endl;
         *p1=y;
        cout<<"Massiv p1=y \n";
        cout<<"\n"<<*p1<<endl;
        delete p1;
    return 0;
}
```

Laboratory work №6.

Task progress:

1. Create the class for a given concept.
   Class should include:
   a. Constructors;
   b. Destructor;
   c. Overloading operators:
      - Indexing [];
      - Function-call ();
      - Input >> and output <<;
      - New and delete for dynamic memory. It should be organized for the array of objects.
2. Describe the class in UML notation.
3. Write a program in which the user will be able to manipulate the created class. Organize work with the array of objects.
   The Variants.
   - 1. Data
   - 2. Time
   - 3. Student