Overloading operators

Overloading functions

In C++ we can have several functions in one program with the same name. The compiler decides which function has to be called depending on the number and type of the parameters and also depending on the type returned.
Example:

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

using namespace std;

int sum(int *matrix, int nb_element)
{
      int result=0;
      int nb;
      for (nb=0; nb<nb_element; nb++)
            result+=matrix[nb];
      return result;
}

float sum(float *matrix, int nb_element)
{
      float result=0;
      int nb;
      for (nb=0; nb<nb_element; nb++)
            result+=matrix[nb];
      return result;
}

void main()
{
      int a[5]={1,2,3,4,5};
      float b[4]={1.11,2.22,3.33,4.44};
      cout<<"the sum of the int values is "<<sum(a,5)<<endl;
      cout<<"the sum of the float values is "<<sum(b,4)<<endl;
}
```

Functions with the same name which return the same data type have to have a different number of parameters, and if the functions (having the same name) have the same number of parameters, then the functions must return a different data type.

What we hide and what we declare public?

As a general rule, the more a program knows about classes the better it is. Therefore you should use private data and methods as often as possible. When using private data and methods, programs must use the object using the object's public methods to access the data object.

Overloading operators

When overloading an operator one has to continue to use that operator in its standard format, namely if one overloads the operator (+), then this has the form *operand+operand.*

We can overload only existing operators, C++ does not allow us to define our own operators. The overloaded operator is applied only to instances of the specified class. For example, if we have a class String and we overload the operator + such that it will concatenate two strings *newstr=str1+str2*; then if you use the operator + for two integers, then the overloading will not be applied.

Operators which **CANNOT** be overloaded are **. .* :: ?:**
We **cannot use friend functions** for overloading the following operators = [ ] ( ) ->

In most of the cases the functions operator return an object of the class on which it is applied. When you overload a unary operator the list of arguments has to be empty, and when you overload a binary operator the list of parameters has to contain only one single parameter.

Syntax:

```
Return_type class_name operator #(list_of_parameters){
                    //operations
            }
```

The overloading can be done by using:
· nonstatic member functions
· friend functions

Exercises:

1. Overloading the operator +

Let us use the class Complex for overloading the operators + and –

```
class Complex
{
int re, im;
public:
Complex(int re=0, int im=0);
~Complex(void);
void display();
};

Complex::Complex(int re, int im)
{
this->re=re;
this->im=im;
}
Complex::~Complex(void)
{
}
void Complex::display()
{
cout << re << "+" << im << "i" << "\n";
}
```

We add in the class Complex the following prototype of declaring the functions:

```
friend Complex operator +(Complex, Complex);
```

And the definition of the function:

```
Complex operator +(Complex c1, Complex c2)
{
Complex tmp;
tmp.re = c1.re + c2.re;
tmp.im = c1.im + c2.im;
return tmp;
}
```

The corresponding function main:

```
void main()
{
Complex c1(1,1), c2(2,2);
Complex r1, r2;
cout<<"c1="; c1.display();
```

```
cout<<"c2=";c2.display();
r1 = c1 + c2;
cout<<"r1=";
r1.display();
r2 = c1 + c2 + r1; //it evaluates (c1 + c2) + r1
// by taking into account the associativity of the operators
cout<<"r2=";
r2.display();
}
```

Overloading operators by using member functions:

2. Overloading the operator -

Add in the previous program in the class Complex the following function prototype:

```
Complex operator- (Complex);
```

The definition of the function is

```
Complex Complex::operator- (Complex c)
{
Complex tmp;
tmp.im = this->im - c.im;
tmp.re = this->re - c.re;
return tmp;
}
```

If we have two Complex variables x and y the call x-y can be transformed into x.operator-(y)

Restriction: in this case the type of the first operand is always the type of the class.

Properties of operators which cannot be modified:
· plurality
· precedence and associativity

3. Overloading the operator =

For example we use the Stack class:

```
class Stack
{
int dim;
int Currentposs;
int* tab;
public:
Stack(int dim=0);
Stack (Stack &);
~ Stack (void);
void add(int el);
void display(void);
};

Stack:: Stack (int dim)
{
cout<<"Explicit constructor "<<endl;
this->dim = dim;
Currentposs = 0;
tab = new int[dim];
}

Stack::~Stack (void)
{
cout<<"Object destroyed "<<endl;
delete tab;
}

void Stack::display(void)
{
cout << " the stack contains: " ;
for (int i=0; i < Currentposs; i++){
cout << tab[i] << " ";
}
cout << endl;
}

Stack:: Stack (Stack &s)
{
cout<<"The copy constructor "<<endl;
this->dim = s.dim;
this-> Currentposs = s.Currentposs;
for (int i=0; i < Currentposs; i++){
this->tab[i] = s.tab[i];
}
}
```

```
void Stack::add(int el)
{
cout<<"Adding an element "<<endl;
tab[Currentposs ++] =el;
}
```

Add the prototype in the class Stack

```
Stack & operator= (Stack &);
```

The function definition:

```
Stack & Stack::operator= (Stack &s)
{
cout<<" overloading the operator = "<<endl;
if (this != &s){
this->dim = s.dim;
this-> Currentposs = s.Currentposs;
for (int i=0; i < Currentposs; i++){
this->tab[i] = s.tab[i];
}
}
return *this;
}
```

Example:

```
int main()
{
Stack s1(10), s2(20), s3(30);
s1 = s2;
s3 = s2 =s1;
getch();
return 0;
}
```

4. The operator []

Is a binary operator and we show the overloading of this operator in order to access the members of the stack.

Add the following prototype in the class Stack:

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

the function definition is:

```
int & Stack::operator[](int i)
{
cout<<"Overloading the operator [] "<<endl;
return tab[i];
}

int main()
{
Stack s1(10), s2(20), s3(30);
s1.add(3);
s1.add(4);
s1.add(8);
s1.display();
cout << "Stack [0]: " <<s1[0] << endl;
getch();
return 0;
}
```

5. Study the following example, run the program and explain:

```
#include <stdafx.h>
#include <iostream>
#include <stdlib.h>

using namespace std;

class Loc
{
     int longitude;
     int latitude;
public:
     Loc(){}
     Loc(int lg, int lt)
     {
          longitude=lg;
          latitude=lt;
     }
     ~Loc()
     {
          //cout<<"The object is destroyed "<<endl;
     }
     void print()
     {
          cout<<"longitude "<<longitude<<" ";
          cout<<"latitude "<<latitude<<endl;
     }
     Loc operator=(Loc op2);
```

```
     friend  Loc  operator++(Loc  &op1);  //Overloading  the
// operator by using a friend function
     friend Loc operator--(Loc &op1); // -||-

     void *operator new(size_t size);
     void operator delete(void *p);
};

Loc Loc::operator=(Loc op2)
{
     longitude=op2.longitude;
     latitude=op2.latitude;
     return *this;
}

Loc operator++(Loc &op1)
{
     op1.longitude++;
     op1.latitude++;
     return op1;
}

Loc operator--(Loc &op1)
{
     op1.longitude--;
     op1.latitude--;
     return op1;
}

void *Loc::operator new(size_t size)
{
     cout<<"Our function new. "<<endl;
     return malloc(size);
}

void Loc::operator delete(void *p)
{
     cout<<" Our function delete. "<<endl;
     free(p);
}

void main()
{
     Loc ob1(10,20), ob2;
     ob1.print();
     ++ob1;
     ob1.print();

     ob2=++ob1;
     ob2.print();
```

```
      --ob2;
      ob2.print();

      Loc *p1, *p2;
      p1=new Loc(40,50);
      if (!p1)
      {
            cout<<"Allocation Error ! "<<endl;
            exit(1);
      }
      p2=new Loc(-40,-50);
      if (!p2)
      {
            cout<<" Allocation Error !"<<endl;
            exit(1);
      }
      p1->print();
      p2->print();
      delete p1;
      delete p2;
}
```

HOMEWORK

1. Define a class PhoneAgenda which contains a list of pairs of the form name, phone number. The program can do the following operations:
· the union of two phone agendas
· the difference of two phone agendas
· the direct access of a phone number by introducing the name (overloading the operator [])
· print, add/erase pairs name, phone number.

2. Implement the class Ratio which allows working with rational (the addition of two rational numbers, the subtraction, multiplication, division):
-the class has two atributes: numerator, denominator, three constructors (default, explicit, copy constructor), a destructor
-add member functions in order to access the numerator, respectively the denominator of a rational number;
-If two rational numbers are equal, then this will be printed on the screen.

3. Modify the problem 2 (class Ratio) such that all the necessary operators are overloaded.

Extra Homework:

4. Implement a class Vector of real numbers and a class Matrix of real numbers. Implement at leas the following:
a) the classes, the corresponding constructors and destructors
b) the scalar product of two vectors
c) the addition, the subtraction, the multiplication of two vectors
d) the addition, the subtraction, the multiplication of two matrices, the invere of a matrix
e) the multiplication of a vector and of a matrix with one constant
f) the product of a vector with a matrix
Carefully choose the friend functions.