OPERATOR OVERLOADING

CHAPTER 19

prepared by Senem Kumova Metin
modified by İlker Korkmaz

```
#include <iostream>
using namespace std;

class Money
{
   private:
        int lira;
        int kurus;
   public:
        Money() {
        }
        Money(int l, int k) {
                lira=l+k/100;
                kurus=k%100;
        }
        Money sum(const Money & x) {
                int tempLira, tempKurus;
                tempLira=x.lira+lira;
                tempKurus =x.kurus+ kurus;
                Money total( tempLira, tempKurus);
                return total;
        }
        void print() {
                cout << lira << " Lira and " << kurus << " Kurus"<< endl;
        }
} ;
int main()
{
        Money m1(3,550);
        Money m2(19,600);
        Money x= m1.sum(m2);            /*   Money m3 = m1 + m2;      gives a SYNTAX ERROR!! */
        x.print();                      /*    cout << x;              gives a SYNTAX ERROR!!*/
        return 0;
}
```

Function Overloading && Operator Overloading

- C++ supports function overloading; so we can use the same name for two or more functions in the same program

- C++ language supports operator overloading as well. By overloading an operator,

  – We can define our own meaning to the operator whenever we use that operator together with an object of our own defined type.

Operator Overloading

- Operators that can be overloaded:
  - arithmetic operators: **+, -, *, /, %, ++, --, etc.**
  - relational operators: **<, <=, ==, !=, >, >=, etc.**
  - assignment operator: **= ; +=,*=,/=,%=, etc.**
  - logical operators: **&&, ||, !,|, ~, &,^, etc.**
  - input/output operators: **<<, >>**
- Operators that can not be overloaded !!!!!!!!!!!!!!

  - Member selection: **.**
  - Member selection with pointer-to-member: **.***
  - Scope resolution: **::**
  - Conditional: **?:**

Operator Overloading :

1. Member function (Class method)

2. Nonmember (Top level-Global) function
   a. non-friend function
   b. friend function

1. Operator overloading with Member functions (a method of the class)

- Declare methods in the class with names
  - operator+()
  - operator-()
  - operator<()
  - operator>()
  - operator!()
  - …

- For binary operations (operations which require two operands) you need only one input parameter (like +,-,%,<,>,=)

```
Money m1,m2;
m1+m2;
```

- For unary operations you need no parameters (like !, -(negation), &(address of))

```
Money m1,m2;
!m1; &m2; -m1;
```

EXAMPLE: Member Function Operator + Overloaded

```
class Money
{    private:        int lira;
                     int kurus;
     public:         Money(){};
                     Money(int  l, int k) { lira=l+k/100; kurus=k%100;  }

                     Money  operator+ (const Money & x)  // MEMBER FUNCTION
                     {          int l, k;
                                l=x.lira+lira;
                                k =x.kurus+ kurus;
                                Money total( l, k);
                                return total;       }

                     void print_Money()
                     {cout<<lira << “YTL" <<kurus<< " KURUS“<< endl;}
};
main()
{    Money m1(3, 550);      Money m2(19, 600);
     Money x = m1 + m2;  //m1.operator+(m2);
     x.print_Money();  }
```

Operator Overloading with Member Functions "+" and "–"

```
Money Money:: operator+ (const Money & x) // A MEMBER FUNCTION
                        {       int l, k;
                                l=x.lira+lira;          k=x.kurus+ kurus;
                                Money result( l, k );
                                return result;          }

Money Money::operator- (const Money & x)  // A MEMBER FUNCTION
                        {       int l, k;
                                l=-x.lira-lira;         k=-x.kurus- kurus;
                                Money result ( l, k );
                                return result;          }
main()
{   Money m1(3, 550);
    Money m2(19, 600);

    Money x = m1 + m2;
    Money y= m1 - m2; //m1.operator-(m2)
    //cout << x;        // SYNTAX ERROR!! Operator << is not defined for Money
    x.print_Money();        }
```

Operator Overloading with Member Functions Binary and Unary "–"

```
Money Money ::operator- (const Money & x)  // BINARY OPERATOR NEEDS TWO OPERANDS
                        {           int l, k;
                                    l=lira-x.lira;          k=kurus- x.kurus;
                                    Money result( l, k);
                                    return result;          }

Money  Money ::operator- ()  //  UNARY OPERATOR NEEDS ONLY ONE OPERAND
                              //  WILL GET THE NEGATIVE  : x → -x
                        {           int l, k;
                                    l=-lira;  k=- kurus;
                                    Money result( l, k);
                                    return result;          }
main()
{    Money m1(3, 550);
     Money m2(19, 600);
     Money x = m1 - m2;  //m1.operator-(m2);
     Money y= -m1; //m1.operator-();
     x.print_Money();
     y.print_Money();           }
```

Operator Overloading with Member Functions : Relational operator ==

```
// Money m1(…), m2(….);
// if(m1==m2) {…..}
// else {…..}

bool  Money:: operator> (const Money & x)
                    {       int total_this, total_x;

                            total_x=x.lira*100+x.kurus;
                            total_this=lira*100+kurus;

                            if(total_this>total_x) return true;
                            else     return false;    }
```

Operator Precedence

- Operator precedence and syntax can not be changed through overloading !!!

EXAMPLE:

```
Money m1, m2, m3, ans;
ans=m1+m2* m3;
// is similar with
ans=m1+(m2*m3);
```

2. Operator Overloading with Toplevel (Nonmember-Global) Functions

- Use functions to overload operators

- Declare functions like
  - operator+() //Money operator+(Money&x, Money&y);
  - operator-()
  - operator<()
  - operator!() //Money operator!(Money&x);
  - …
- For unary operators function must have one input parameter
  →!x

- For binary operators function must have two input parameters
  →x+y

- If top level functions tries to access private members , they have to call public methods that return private members (get methods..)

2. Operator Overloading with Toplevel (Global) Functions

```
class Money
{    private:  int lira, kurus;
     public:          Money(){};
                      Money(int  l, int k)  { lira=l+k/100; kurus=k%100;  }
                      int getLira() { return lira;} /* toplevel functions cannot access private
                                                       members so we have to define methods
                                                       to return private members */
                      int getKurus(){ return kurus;}                          };

Money  operator+ (Money & x, Money &y)
                      {          int l=x.getLira()+y.getLira();
                                 int k =x.getKurus()+ y.getKurus();
                                 Money result( l, k);
                                 return result;          }

Money  operator- (Money & x) // UNARY MINUS-NEGATION
                      {          int l=-x.getLira();  int k =-x.getKurus();
                                 Money result( l, k);
                                 return result;          }
main()
{    Money m1(3, 550), m2(19, 600);
     Money x = m1 + m2;        Money y= -m1;        }
```

Operator Overloading with Friend Functions

A class's private members are accessible to

- its methods
- and also its friend functions!!!

Operator Overloading with Friend Functions

```
class Money
{    private:          int lira, kurus;
     public:           Money(){};
                       Money(int  l, int k)  { lira=l+k/100; kurus=k%100;  }
                       void print_Money(){ cout<<lira << " Lira "<<kurus <<" Kurus";}


                       friend Money operator+(const Money & , const Money &);
                       friend Money operator-(const Money & );
};
Money  operator+ (const Money & x, const Money &y)
                       {          int l=x.lira+y.lira;
                                  int k=x.kurus+ y.kurus;
                                  Money result( l, k);
                                  return result;        }

Money  operator- (const Money & x) // UNARY MINUS-NEGATION
                       {          int l=-x.lira;  int k =-x.kurus;
                                  Money result( l, k);
                                  return result;        }
main()
{    Money m1(3, 550), m2(19, 600);
     Money x = m1 + m2;        Money y= -m1;
     x.print_Money();          y.print_Money();    }
```

Some Important Remarks

- For all different ways of use of operator overloading , they are all called in the same way!!

```
Money m1, m2,m3;
m3=m1+m2;
```

- Remember that parameters to the operators are POSITIONAL→the order of the 1st and 2nd parameter does matter

```
Money  operator- (Money & x)
                    {       int l,k;
                            l=lira-x.lira;   k=kurus- x.kurus;
                            //l=x.lira-lira;  k=x.kurus- kurus
                            Money result( l, k);
                            return result;      }

Money m1, m2,m3;
m3=m1-m2;  // m1.operator-(m2);
```

OVERLOADING THE INPUT AND OUTPUT OPERATORS

```
int i;
Money y;

cin>>i;
cin>>y;
```

are interpreted as

```
operator>>(cin,i);
operator>>(cin,y);
```

- Operand >> belongs to a system class (e.g istream)
- cin is an istream object

- If the leftmost operand is an object of a different class or a fundamental data type , this operator should be implemented as non-member function or friend function

OVERLOADING THE INPUT AND OUTPUT OPERATORS

– To overload >> to read a two integers for Money class (lira and kurus) we can write a top level function

```
istream& operator>>(istream & in,Money &m)
{       in>>m.lira>>m.kurus;
        return  in;      }
```

Use this function as

```
Money m1;
cin>>m1; // is equivalent to operator >>(cin, m1)
```

– If you want to use a method to overload >> you will need to modify the system class istream. Instead, overload >> as a top-level function (either friend or not).

EXAMPLE:

Operator >> overloaded

```
#include<iostream>
using namespace std;

class Money
{    friend istream& operator>>(istream &,Money &);

     private:
          int lira; int kurus;

     public:          Money(int  l, int k) { lira=l+k/100; kurus=k%100;  }
                      void print_Money(){ cout<<lira << " " <<kurus<<endl;}        };

istream& operator>>(istream &in,Money &m)
{    in>>m.lira;
     in>>m.kurus;
     return in;          }

main()
{    Money m2(0, 0);
     cin>>m2;
     //cout << x;        // SYNTAX ERROR!! Operator << is not defined for Money
     m2.print_Money();          }
```

OVERLOADING THE INPUT AND OUTPUT OPERATORS

– To overload << to print out a two integers for Money class (lira and kurus) we can write a top level function

```
ostream& operator<<(ostream & out,Money &m)
{       out<<m.lira<<"Lira "<<m.kurus<<" Kurus"<<endl;
        return out;     }
```

Use this function as

```
Money m1;
cout<<m1; // is equivalent to operator <<(cout, m1)
```

– If you want to use a method to overload << you will need to modify the system class ostream. Instead, overload << as a top-level function (either friend or not).

EXAMPLE:

Operator << overloaded

```
#include<iostream>
#include<iomanip>
using namespace std;

class Money
{    friend istream& operator>>(istream &,Money &);
     friend ostream& operator<<(ostream &,Money &);

     private:                    int lira; int kurus;
     public:                     Money(int  l, int k) { lira=l+k/100; kurus=k%100;  }  };

istream& operator>>(istream &in,Money &m)
{    in>>m.lira>>m.kurus;
     return in;          }

ostream& operator<<(ostream &out,Money &m)
{    out<<m.lira<<" Lira "<<m.kurus<<" Kurus"<<endl;
     return out;         }

main()
{    Money m2(0, 0);
     cin>>m2;
     cout<<m2;          }
```

Overloading Assignment Operator =

- Compiler provides default copy constructor and assignment operator if the author does not implement his/her own alternatives.

```
class Money {.... };
main()
{
  Money m1(10, 1000);
  Money m2(100, 1000);
  m1=m2;
}
```

Overloading Assignment Operator =

```
class Money
{   private:       int lira; int kurus;
    public:
    Money(int  l, int k) { lira=l+k/100; kurus=k%100;  }
    Money & operator=(const Money & x) // will return a reference to m1
    {       lira=x.lira; kurus=x.kurus;
            return *this;
    }
};
main()
{   Money m1(100, 200), m2(30,430);
    m1=m2;      // this is m1 and x is m2
    cout<<m1;  // << has been overloaded in previous slides
 }
```

Overloading Increment and Decrement operators

- The increment ++ and decrement - - operators come in a *prefix* and *a postfix flavor*

```
int x = 6;
++x // prefix ++
x++ // postfix ++
```

- We can overload the prefix ++, the postfix ++, the prefix - -, and the postfix - -.

Overloading Increment and Decrement operators

- The declaration **operator++()** refers to the prefix operator ++x
- The declaration **operator++(int)** refers to the postfix operator x++
- If the prefix increment operator has been overloaded, the expression **++obj** is equivalent to **obj.operator++()**
- If the postfix increment has been overloaded, the expression **obj++** is equivalent to **obj.operator++(0)**

EXAMPLE: Overloading Increment

- A Clock class will be implemented.
- If **c** is a Clock object, **c++** advances the time one minute. The value of the expression c++ is the original Clock
- Executing **++c** also advances the time one minute, but the value of the expression ++c is the *updated* Clock.

EXAMPLE :
OVERLOADING
INCREMENT OPERATOR

```
/* Clock.h*/
class Clock
{ public :
        friend ostream & operator <<(ostream &out, Clock &p) ;
        Clock ( int =12, int =0);
        Clock tick();
        Clock operator++(); //++c
        Clock operator++(int); // c++
    private:
        int hour;
        int min;
};
ostream & operator <<(ostream &out, Clock &p)
{       out<<setfill('0')<<setw(2)<<p.hour<<":" ;
        out<<setw(2)<<p.min<<endl;
        return out;     }
```

EXAMPLE :
OVERLOADING INCREMENT OPERATOR

```
#include  “Clock.h”
Clock::Clock ( int h, int m)
    { hour=h; min=m;}

Clock Clock::tick()
{   ++min;
    if (min>=60) {        hour= hour+min/60;
                          min= min%60;}
                          return *this;              }

Clock Clock::operator ++()  {return tick ();}              //PREFIX

Clock Clock::operator ++(int n)                            //POSTFIX
{   Clock c=*this;
    tick();          // this will be incremented
    return c;      // previous value of this will be returned
}
```

```
main(){
                Clock c( 12, 50);      Clock d;
                cout <<"Clock c: "<<c;
                cout <<"Clock d: "<<d;
                d= c++;
                cout<<"Clock c: "<<c;
                cout<<"Clock d: "<<d;
                cout<<"........"<<endl;

                cout <<"Clock c: "<<c;
                cout <<"Clock d: "<<d;
                d= ++c;
                cout<<"Clock c: "<<c;
                cout<<"Clock d: "<<d; }
```

EXAMPLE :
OVERLOADING INCREMENT OPERATOR