Chapter 8 – Operator Overloading

Basic operators (ex: +, - ,==) are *functions* with a special syntax (syntactic sugar).
You can customize a basic operator to work for a class object by **overloading** the operator.

ex: 3 + 7
3 and 7 are the operands
+ is the operator

Binary operators:	requires 2 operands
Unary operators:	requires 1 operand (ex: negation, such as -7, increment++, decrement--)

Example on Operator Overloading outside of the class:

```
class Money
{
public:
    Money( );
    Money(double amount);
    Money(int theDollars, int theCents);
    Money(int theDollars);
    double getAmount( ) const;
    int getDollars( ) const;
    int getCents( ) const;
    void input( );
    void output( ) const;
private:
    int dollars;
    int cents;
};
```

```
const Money operator +(const Money& amount1, const Money& amount2);

const Money operator -(const Money& amount1, const Money& amount2);

bool operator ==(const Money& amount1, const Money& amount2);

const Money operator -(const Money& amount);
```

```
bool operator ==(const Money& amount1, const Money& amount2)
{
    return ((amount1.getDollars( ) == amount2.getDollars( ))
           && (amount1.getCents( ) == amount2.getCents( )));
}
```

Example on Operator Overloading as members:

```
class Money
{
public:
    Money( );
    Money(double amount);
    Money(int dollars, int cents);
    Money(int dollars);
    double getAmount( ) const;
    int getDollars( ) const;
    int getCents( ) const;
    void input( ); //Reads the dollar sign as well as the amount number.
    void output( ) const;
    const Money operator +(const Money& amount2) const;

    const Money operator -(const Money& amount2) const;

    bool operator ==(const Money& amount2) const;

    const Money operator -( ) const;

private:
    int dollars;
    int cents;
};

bool Money::operator ==(const Money& secondOperand) const
{
    return ((dollars == secondOperand.dollars)
            && (cents == secondOperand.cents));
}
```

Automatic Type Conversion

Example:

```
Money baseAmount(100, 60), fullAmount;
fullAmount = baseAmount + 25;
fullAmount.output();
```

The output prints $125.60.

However, we did not specify a + operator that accepts a Money object and an integer. How did the program manage to compile and produce the correct output?

- C++ checks if there is a valid overloaded operator that matches the operands. Our overloaded + operator accepts two Money objects.
- C++ looks for a constructor that accepts the invalid operand to convert into that object type.
  ```
  Money(int dollars);
  ```
  C++ will automatically convert the integer into a Money object and apply the operation.

Member vs. Nonmember Operator Overloading

```
fullAmount = baseAmount + 25;
```

Will work for both member and nonmember operator overloading. However,

```
fullAmount = 25 + baseAmount;
```

Will only work for the nonmember operator. Since the first operand is an int, it will not call the operator from the Money class.

It is preferable to overload operator functions as a nonmember.

Friend

A **friend function** of a class is not a member function of the class, but it has access to the private members of that class.

Friend functions must be declared inside the class definition. A function can be a friend of multiple classes.

```
class Money
{
public:
    Money( );
    Money(double amount);
    Money(int dollars, int cents);
    Money(int dollars);
    double getAmount( ) const;
    int getDollars( ) const;
    int getCents( ) const;
    void input( );
    void output( ) const;

   friend const Money operator +(const Money& amount1, const Money& amount2);

   friend const Money operator -(const Money& amount1, const Money& amount2);

   friend bool operator ==(const Money& amount1, const Money& amount2);

   friend const Money operator -(const Money& amount);

private:
    int dollars;
    int cents;
};

bool operator ==(const Money& amount1, const Money& amount2)
{
    return ((amount1.getDollars( ) == amount2.getDollars( ))
           && (amount1.getCents( ) == amount2.getCents( )));
}
```

Overloading << and >>

It is possible to overload the << and >> operators from iostream.

```
class Money
{
public:
…
    friend ostream& operator <<(ostream& outputStream, const Money& amount);
    friend istream& operator >>(istream& inputStream, Money& amount);
private:
…
};

ostream& operator <<(ostream& outputStream, const Money& amount)
{
    int absDollars = abs(amount.dollars);
    int absCents = abs(amount.cents);
    if (amount.dollars < 0 || amount.cents < 0)
        //accounts for dollars == 0 or cents == 0
        outputStream << "$-";
    else
        outputStream << '$';
    outputStream << absDollars;

    if (absCents >= 10)
        outputStream << '.' << absCents;
    else
        outputStream << '.' << '0' << absCents;

    return outputStream;
}

//Uses iostream and cstdlib:
istream& operator >>(istream& inputStream, Money& amount)
{
    char dollarSign;
    inputStream >> dollarSign; //hopefully
    if (dollarSign != '$')
    {
        cout << "No dollar sign in Money input.\n";
        exit(1);
    }


    double amountAsDouble;
    inputStream >> amountAsDouble;
    amount.dollars = amount.dollarsPart(amountAsDouble);
    amount.cents = amount.centsPart(amountAsDouble);


    return inputStream;
}
```