Operator Overloading

Computer Programming for Engineers (DSAF003-42)

Instructor:

Youngjoong Ko (nlp.skku.edu)

This Week

- Operator Overloading Basics
  - Globally overloaded "+" and "=="
  - Unary operators
  - As member functions

- More Overloading
  - Operators: << and >>
  - Operators: =
  - Operators: ++, --
  - Operators: []

OPERATOR OVERLOADING BASICS

Operator Overloading Introduction

- Operators +, -, %, ==, etc.
  - are really just functions!

- Simply "called" with different syntax: `x + 7`
  - "+" is binary operator with x and 7 as operands
  - Human-friendly notations

- Function-like notation: `+(x,7)`
  - "+" is the function name (later, we call "operator+")
  - x, 7 are the arguments
  - Function "+" returns "sum" of it’s arguments

Operator Overloading Perspective

- Built-in operators
  - +, -, = , %, ==, /, *, ...
  - Already work for C++ built-in types
  - In standard "binary" notation with two operands
- We can overload them!
  - To work with OUR types!
  - To add "Chair types" or "Money types"
    - As appropriate for our needs
- Always overload with similar "actions" (meaning)!
  - An entire different meaning can lead to confusion for users.

Overloading Basics

- Overloading operators
  - VERY similar to overloading functions
  - Operator itself is "name" of function

- Example Declaration:

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

  - Overloads + for operands of type Money
  - Uses constant reference parameters for efficiency
  - Returned value is type Money
    - Allows addition of "Money" objects

(global) Overloaded "+"

- Given previous example:
  - Note: overloaded "+" NOT member function
  - Definition is "more involved" than simple "add"
    - Requires issues of money type addition
    - Must handle negative/positive values

- Operator overload definitions generally very simple
  - Just perform "addition" particular to "your" type

Overloaded "+" for Money

- In Display 8.1 Operator Overloading

```
const Money operator+(const Money& amount1, const Money& amount2)
{
    int allCents1 = amount1.getCents( ) + amount1.getDollars( )*100;
    int allCents2 = amount2.getCents( ) + amount2.getDollars( )*100;
    int sumAllCents = allCents1 + allCents2;
    int absAllCents = abs(sumAllCents); //Money can be negative.
    int finalDollars = absAllCents / 100;
    int finalCents = absAllCents % 100;
    if (sumAllCents < 0)
    {
        finalDollars = -finalDollars;
        finalCents = -finalCents;
    }
    return Money(finalDollars, finalCents);
}
```

The return statements puzzle you.
A Constructor can Return an Object.

(global) Overloaded "=="

- Equality operator, ==
  - Enables comparison of Money objects
  - Declaration:

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

    - Returns bool type for true/false equality
  - Again, it’s a non-member function (like "+" overload)

Overloaded "==" for Money

- In Display 8.1 Operator Overloading

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

Constructors Returning Objects

- Recall return statement in "+" overload for Money type
  - Returns an "invocation" of Money class!
  - So constructor actually "returns" an object!
  - Remind the "**anonymous object**". 익명객체 : 한번 쓰고 cut!

```
cout << 1 + 2 << endl; // 3 is placed in an anonymous object.
```

```
return Money(finalDollars, finalCents);
```

Returning by const Value

반환되는 값 상수

B+C에서 C상수

const : B+C에서 B 상수

- Consider "+" operator overload again:

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

  - Returns a "constant object"? Why?

- Consider impact of returning "non-const" object.
  - Consider "no const" in declaration:

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

  - Consider expression that calls:

```
m1 + m2
```

  - Object returned is Money object
  - We can "do things" with objects! Like call member functions…

What to do with Non-const Object

- Can call member functions:
  - We could invoke member functions on object returned by expression m1+m2:

```
(m1+m2).output();  // Legal, right?
```

  - Not a problem: doesn't change anything

```
(m1+m2).input();  // Legal!
```

  - PROBLEM! // Legal, but MODIFIES!
  - Allows modification of "anonymous" object!
  - Can't allow that here!

- So we define the return object as const

Overloading Unary Operators

단항 연산자

- C++ has unary operators:
  - Defined as taking one operand
  - e.g., - (negation)
    - `x = -y; // Sets x equal to negative of y`
  - Other unary operators:
    - `++, --`
- Unary operators can also be overloaded

Overload "-" for Money

- Overloaded "-" function declaration
  - Placed outside class definition:

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

  - Notice: only one argument: since only 1 operand (unary)

- "-" operator is overloaded twice!
  - For one operand/argument (**unary**) 단일
  - For two operands/arguments (**binary**) 두개
  - Definitions can exist for both

) 인자 달리 하여 둘 다 선언 ㅇㅋ

Overloaded "-" Definition

- Overloaded "-" function definition:

```
const Money operator-( const Money& amount )
{
    return Money(-amount.getDollars(), -amount.getCents());
}
```

- Applies "-" unary operator to built-in type
  - Operation is "known" for built-in types
- Returns anonymous object again

Overloaded "-" Usage

- Consider:

```
Money  amount1(10), amount2(6), amount3;
amount3 = amount1 – amount2;
//Calls binary "-" overload
amount3.output();   //Displays $4.00
amount3 = -amount1;
//Calls unary "-" overload
amount3.output();      //Displays -$10.00
```

- Overloading Basics (1_basic_oper_over.cpp, 1_int~.cpp)
  - We will add unary operator -.
  - Error case: operator overloading with primitive types

DEMO

```
class Money
{
  public:
    …
  private:
    int dollars;
    int cents;
};
```

amount3 = -amount1. 얘는 - 오버로딩.

```
const Money operator-(const Money& amount)
{
  return Money(-amount.getDollars(), -amount.getCents());
}

// What happens with the following code?
const int operator+(const int num1, const int num2)
{
  return num1 * num2;
}
```

안돼~

OVERLOADING
AS MEMBER FUNCTIONS

Overloading as Member Functions

- Previous examples: standalone global functions
  - Defined outside a class
- Can overload as "member operator"
  - Considered "member function" like others
- When operator is member function: 멤버함수인 경우
  - Only ONE parameter, not two! 파라미터 1개!
  - Calling object serves as the first parameter 호출 개체가 첫 파라미터로 쓰임
    - ‘*this’ is the first parameter implicitly.

Member Operator in Action

- Consider:

```
Money  cost(1, 50), tax(0, 15), total;
total = cost + tax;
```

  - If "+" overloaded as member operator:
    - Object "cost" is calling object
    - Object "tax" is single argument
  - Think of as: total = cost.+(tax);
    - Actually, total = cost.operator+(tax)
- Declaration of "+" in class definition:

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

  - Notice only ONE argument

Overloading Operators: Which Method?

- Object-Oriented-Programming
  - Principles suggest member operators
  - Many agree to maintain "spirit" of OOP

- Member operators more efficient
  - No need to call accessor & mutator functions

Overloading Function Application ()

- Function call operator: ( )
  - Must be overloaded as member function 멤버함수로 무조건 overload!
  - Allows use of class object like a function
  - Can overload for all possible numbers of arguments

- Example:

```
class anObject;
anObject(42);
```

  - If ( ) overloaded → calls overload

- 2. Overloading as Member Functions (2_member, 2_paren)
  - We will move unary operator- from global to member.
  - We will add the function call operator ().

DEMO

```
class Money {
  public:
    const Money operator-();
    void operator()(int theDollars);
    void operator()(int theDollars, int theCents);
  private:
    int dollars;
    int cents;
};

const Money Money::operator-(const Money& amount) {
  return Money(-amount.getDollars(), -amount.getCents());
}

void Money::operator()(int theDollars) { dollars = theDollars; }

void Money::operator()(int theDollars, int theCents) {
  dollars = theDollars;  cents = theCents;
}
```

↳ member func. 선언

본론은 amount3 = amount.operator-(amount2);

↳ 같은 연산자.

amount3(100, 99); 했는데 에러가 없으면 X!

MORE OVERLOADING

Overloading << and >>

- Enables input and output of our objects
  - Similar to other operator overloads

입출력 활성화

- Improves readability
  - Like all operator overloads do
  - Enables:

```
std::cout << myObject;
std::cin >> myObject;
```

  - Instead of need for: `myObject.output();` …

Overloading <<

- Insertion operator, <<
  - Used with cout
  - A binary operator
- Example:

```
std::cout << "Hello";
```

  - Operator is <<
  - 1st operand is predefined object cout 피연산자는 cout
    - From library <iostream>
    - It makes operator<< **not** to be a member of a class. class 멤버 아님!
  - 2nd operand is literal string "Hello“

Overloading <<

- Recall Money class
  - Used member function output()
  - Nicer if we can use << operator:

```
Money amount(100);
cout << "I have " << amount << endl;
```

  - // instead of:

```
cout << "I have ";
amount.output()
```

Overloaded << and >> Example (3_io~)

- Display 8.5 Overloading << and >> (1 of 7)

```
#include <iostream>
#include <cstdlib>
#include <cmath>
using namespace std;
//Class for amounts of money in U.S. currency
class Money
{
public:
    Money( );
    Money(double amount);
    Money(int theDollars, int theCents);
    Money(int theDollars);
    double getAmount( ) const;
    int getDollars( ) const;
    int getCents( ) const;
```

Overloaded << and >> Example

- Display 8.5 Overloading << and >> (2 of 7)

```
    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);
    friend ostream& operator<<(ostream& outputStream,
                    const Money& amount);
    friend istream& operator>>(istream& inputStream,
                    Money& amount);
```

Overloaded << and >> Example

- Display 8.5 Overloading << and >> (3 of 7)

```
private:
//A negative amount is represented as negative dollars
//and negative cents. Negative $4.50 is represented as
//-4 and –50.
    int dollars, cents;
    int dollarsPart(double amount) const;
    int centsPart(double amount) const;
    int round(double number) const;
};

int main()
{
    Money yourAmount, myAmount(10, 9);
    cout << "Enter an amount of money: ";
```

Overloaded << and >> Example

- Display 8.5 Overloading << and >> (4 of 7)

```
    cin >> yourAmount;
    cout << "Your amount is " << yourAmount << endl;
    cout << "My amount is " << myAmount << endl;
    if (yourAmount == myAmount)
    cout << "We have the same amounts.\n";
    else cout << "One of us is richer.\n";

    Money ourAmount = yourAmount + myAmount;
    cout << yourAmount << " + " << myAmount
        << " equals " << ourAmount << endl;

    Money diffAmount = yourAmount - myAmount;
    cout << yourAmount << " - " << myAmount
        << " equals " << diffAmount << endl;
    return 0;
}
```

Overloaded << and >> Example

- Display 8.5 Overloading << and >> (5 of 7)

```
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;
}
```

In the main function, cout is plugged in for outputStream.

Returns a reference

Overloaded << and >> Example

- Display 8.5 Overloading << and >> (7 of 7)

```
Enter an amount of money: $123.45
Your amount is $123.45
My amount is $10.09.
One of us is richer.
$123.45 + $10.09 equals $133.54
$123.45 - $10.09 equals $113.36
```

Assignment Operator: =

- Must be overloaded as member operator 멤버로 오버로드
  - Simultaneously with copy constructor
- Automatically overloaded
  - Default assignment operator:
    - Member-wise copy (i.e., shallow copy)
    - Member variables from one object → corresponding member variables from other
- Default is OK for simple classes
  - But with pointers → must write our own!
  - Implement deep copy (allocating new memory and copying the content)

Overloading = (4_assign~.cpp)

DEMO

```
class Money {
  public:
    const Money& operator=(const Money& theMoney);
    Money operator++(); // preifx
  // Make members public just for the example
  //private:
    int dollars;
    int cents;
};

const Money& Money::operator=(const Money& theMoney)
{
  // Just for example
  dollars = theMoney.getDollars() - 1;
  return *this; // Why should we return *this?
}
```

```
Money a1(10), a2(2);
a2=a1;
```

출력하면 10, 9 나옴.

Increment and Decrement

- Each operator has two versions
  - Prefix notation: ++x;
  - Postfix notation: x++;
- Must distinguish in overload
  - Standard overload method → Prefix 표준방법 → 전위
  - Add the second parameter of (dummy) type int → Postfix int 인수 추가 → 후위
    - Just a marker for compiler! It's dummy.
    - Specifies postfix is allowed

```
Money& operator++(){ ... }     // prefix
Money operator++(int){ ... }  // postfix
```

Overloading ++ (5_icrement~.cpp)

DEMO

```
// Postfix version, not a member
Money operator++(Money& theMoney, int ignoreMe) {
  // We need range checks for cents. This is just for an example.
  int dollars = theMoney.dollars++, cents = theMoney.cents++;
  return Money(dollars, cents);
}
    &
    ∨
Money Money::operator++() {
  // We need range checks for cents. This is just for an example.
  return Money(++dollars, ++cents); (X) → ++dollars; ++cents; return *this;
}

int main()
{
  Money   amount(10);
  Money a = amount++;
  a.output();  amount.output();

  amount = Money(10);
  a = ++amount;
  a.output();  amount.output();

  return 0;
}
```