Operator Overloading

Computer Programming for Engineers
(DASF003-41)

Instructor:

Sungjae Hwang (jason.sungjae.hwang@gmail.com)

Today

- **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
- **We can overload them!**
  - To work with OUR types!
  - To add "Chair types" or "Money types"
    - As appropriate for our needs
    - In "notation" we're comfortable with
  - Cannot define new operators
  - Cannot overload operators of built in data type such as int, char

- **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 "+"

```
class Money {
 public:
   getDollars();
   getCents();
private:
   int dollars;
   int cents;
   …
};
const Money operator+(const Money& amount1, const Money& amount2);
```

- **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

```
class Money {
 Money();
 Money(int dollar, int cents);
 …
};
```

- **Is constructor a "void" function?**
  - We "think" that way, but no:
  - A "special" function with special properties
  - CAN return a value (i.e., an object of that class)!

- **Recall return statement in "+" overload for Money type:**

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

  - Returns an "invocation" of Money class!
  - So constructor actually "returns" an object!
  - Remind the "**anonymous object**".

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

Returning by const Value

- 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

- 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**)
  - For two operands/arguments (**binary**)
  - Definitions can exist for both

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

DEMO

```cpp
#include <iostream>
using namespace std;
class Money{
  public:
    Money(int d, int c);
    int getDollars() const;
    int getCents() const;
  private:
    int dollars;
    int cents;
};
Money::Money(int d, int c){
  dollars = d;
  cents = c;
}
int Money::getDollars() const {
  return dollars;
}
int Money::getCents() const {
  return cents;
}
const Money operator-(const Money& amount)
{
  return Money(-amount.getDollars(), -amount.getCents());
}
/*
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;
}
*/
int main(){
  Money m1(10, 9);
  cout << "doller: " << m1.getDollars() << " cents: " << m1.getCents() << endl;
  Money m2 = -m1;
  cout << "doller: " << m2.getDollars() << " cents: " << m2.getCents() << endl;
  return 0;
}
```

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:**
  - Only ONE parameter, not two!
  - Calling object serves as the first parameter
    - ‘`*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
  - Allows use of class object like a function
  - Can overload for all possible numbers of arguments

- **Example:**

```
Aclass anObject;
anObject(42);
```

  - If ( ) overloaded → calls overload

Pitfall

- **&&, ||, and comma operator**
  - Predefined versions work for bool types
  - Recall: use “short-circuit” evaluation
  - When overloaded no longer uses short-circuit
  - Comma operator guarantees left-to-right evaluations
- **Generally should not overload these operators**
  - When overloaded, short-circuit evaluation is not guarantees
  - Left-to-right evaluation is not guarantees
  - It is not most programmers expectation.

■Overloading as Member Functions

```cpp
#include <iostream>
using namespace std;
class Money{
  public:
    Money(int d, int c);
    int getDollars() const;
    int getCents() const;
    const Money operator-();
    void operator()(int d);
    void operator()(int d, int c);
  private:
    int dollars;
    int cents;
};
Money::Money(int d, int c){
  dollars = d; cents = c;
}
int Money::getDollars() const {
  return dollars;
}
int Money::getCents() const {
  return cents;
}
const Money Money::operator-(){
  return Money(-dollars, -cents);
}
void Money::operator()(int theDollars) {
  dollars = theDollars; }
void Money::operator()(int theDollars, int theCents) {
  dollars = theDollars;  cents = theCents;
}
int main(){
  Money m1(10, 9);
  cout << "doller: " << m1.getDollars() << " cents: " << m1.getCents() << endl;
  Money m2 = -m1;
  cout << "doller: " << m2.getDollars() << " cents: " << m2.getCents() << endl;
  m1(20);
  cout << "doller: " << m1.getDollars() << " cents: " << m1.getCents() << endl;
  m2(5,50);
  cout << "doller: " << m2.getDollars() << " cents: " << m2.getCents() << endl;
  return 0;
}
```

DEMO

MORE OVERLOADING

Overloading << and >>

- **Enables input and output of our objects**
  - Similar to other operator overloads
  - New subtleties
- **Improves readability**
  - Like all operator overloads do
  - Enables:

```
std::cout << myObject;
myObject.output();

std::cin >> myObject;
myObject.input();
```

  - 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
    - From library `<iostream>`
    - It makes `operator<<` **not** to be a member of a class.
  - 2nd operand is literal string "Hello“

Overloading <<

- **Operands of <<**
  - cout object, of class type ostream
  - Our class type
- **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 << Return Value

```
Money amount(100);
cout << amount;
```

- << should return some value
- To allow cascades:
  `cout << "I have " << amount;`
  `(cout << "I have ") << amount;`
  - Two are equivalent

- **What to return?**

Overloaded << Return Value

```
Money amount(100);
cout << amount;
```

- << should return some value
- To allow cascades:
  ```
  cout << "I have " << amount;
  (cout << "I have ") << amount;
  ```
  - Two are equivalent
  1. cout << “I have ”;
  2. cout << amount;

- **What to return?**
  - a reference to cout object!
    - Returns its first argument type, ostream

Overloaded << and >> Example

- 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 <
    //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 >> (6 of 7)

```
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.dolla
    amount.cents = amount.centsPart(amountAsDouble);

    return inputStream;
}
```

In the main function, cin is plugged in for inputStream.

Since this is not a member operator,
you need to specify a calling object
for member functions of Money.

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**
  - 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)

Increment and Decrement

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

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

■Overloading = and ++

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?
}
```

■Overloading = and ++

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);
}

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

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

  return 0;
}
```

SUPPLEMENTARY SLIDES
< FRIEND >

Friend Functions

- **Nonmember functions**
  - Recall: operator overloads as nonmembers
    - They access data through accessor and mutator functions
    - Very inefficient (overhead of calls)
- **Friends can directly access private class data**
  - No overhead, more efficient
- **So: best to make nonmember operator overloads friends!**

Friend Functions

- **Friend function of a class**
  - Not a member function
  - Has direct access to private members
    - Just as member functions do
- **Use keyword *friend* in front of function declaration**
  - Specified IN class definition
  - But they're NOT member functions!

■friend function

DEMO

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

class A{
  private:
    int num;
  public:
    A(): num(10){}
    friend void printNum(A);
};

void printNum(A a){
  cout << "num: " << a.num << endl;
}

/*
void printNum2(A a){
  cout << "num: " << a.num << endl;
}
*/

int main(){
  A obj1;
  printNum(obj1);
  return 0;
}
```

Friend Function Uses

- Operator Overloads
  - Most common use of friends
  - Improves efficiency
  - Avoids need to call accessor/mutator member functions
- Advantageous?
  - For operators: very!
  - Still encapsulates: friend is in class definition
  - Improves efficiency
  - Allows automatic type conversion

```
Money baseAmount(100,60), fullAmount;
fullAmount = baseAmount + 25; //legal
fullAmount = 25 + baseAmount; //illegal
```

■Automatic Type Conversion

DEMO

```
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);
}
```

```
int main()
{
    Money baseAmount(100,60), fullAmount;
    fullAmount = baseAmount + 25;
    cout << fullAmount << endl;

    fullAmount = 30 + baseAmount;
    cout << fullAmount << endl;

    return 0;
}
```

Friend Classes

- **Entire classes can be friends**
  - Similar to function being friend to class
  - Example:
    class F is friend of class C
    - All class F member functions are friends of C
    - NOT reciprocated
    - Friendship granted, not taken
- **Syntax: friend class F**
  - Goes inside class definition of "authorizing" class

■friend class

DEMO

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

class A{
  private:
    int num;
  public:
    A(): num(10){}
    friend class B;
};

class B{
  public:
    void printNum(A a){
      cout << "num: " << a.num << endl;
    }
};

/*
void printNum2(A a){
  cout << "num: " << a.num << endl;
}
*/
```

```
int main(){
  A obj1;
  B obj2;
  obj2.printNum(obj1);
  return 0;
}
```

Overload Array Operator, [ ]

- **Can overload [ ] for your class**
  - To be used with objects of your class
  - Operator must return a reference!
  - Operator [ ] must be a member function!
    - a[2] : a is the calling object, 2 is the second argument

■Array operator

DEMO

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

class CharPair{
  public:
    CharPair(){}
    CharPair(int first_val, int second_val) : first(first_val), second(second_val) {}
    char& operator[](int index);
  private:
    char first;
    char second;
};

int main(){
  CharPair a;
  a[1] = 'A';
  a[2] = 'B';
  cout << a[1] << " " << a[2] << endl;
}

char& CharPair::operator[](int index)
{
  if(index == 1)
    return first;
  else if(index == 2)
    return second;
  else
    cout << "Illegal index value" << endl;
    exit(1);
}
```

Summary

- Operator Overloading
- +, -, ++, (), [], =, <<, >>
- Non-Member vs Member Operator
- Friend Function
- Friend Class