Classes (Part2)

Prof. Seokin Hong

SUNGKYUNKWAN UNIVERSITY

Agenda

- Friend Function and Friend Class
- Operator Overloading

Friends

Friends

- **Class operations are typically implemented as member functions**
  - Because nonpublic members cannot be accessed by a global function or other classes

```
class a;
a.f();
```

- **Some operations are better implemented as ordinary (nonmember) functions**

```
class a, b, c;
 c=f(a, b);
```

- **A class can allow another class or function to access its nonpublic members by making that class or function a friend**

DayOfYear class

- **The DayOfYear class with an equality function**
  - **equality function**: tests **two objects of type DayOfYear** to see if their values represent the same date

- **The equal() Function**
  - return ***true*** if the dates are the same
  - requires a parameter for each of the two dates to compare
  - **Declaration**

```
bool equal(DayOfYear date1, DayOfYear date2);
```

Notice that equal() is not a member of the class DayOfYear!!

DayOfYear class (Cont.)

- **equal() is not a member function**
  - must use public accessor functions to obtain the day and month from a DayOfYear object

- **equal() can be defined in this way:**

```
bool equal(DayOfYear date1, DayOfYear date2)
 {
   return ( date1.getMonth( ) == date2.getMonth( ) &&
                date1.getDay( ) == date2.getDay( ) );
  }
```

```cpp
//Program to demonstrate the function equal. The class DayOfYear
//is the same as in Self-Test Exercises 23-24 in  Chapter 10.
#include <iostream>
using namespace std;

class DayOfYear
{
public:
    DayOfYear(int theMonth, int theDay);
    //Precondition: theMonth and theDay form a
    //possible date. Initializes the date according
    //to the arguments.

    DayOfYear( );
    //Initializes the date to January first.

    void input( );

    void output( );

    int getMonth( );
    //Returns the month, 1 for January, 2 for February, etc.

    int getDay( );
    //Returns the day of the month.
private:
    void checkDate( );
    int month;
    int day;
};

bool equal(DayOfYear date1, DayOfYear date2);
//Precondition: date1 and date2 have values.
//Returns true if date1 and date2 represent the same date;
//otherwise, returns false.

int main( )
{
    DayOfYear today, bachBirthday(3, 21);

    cout << "Enter today's date:\n";
    today.input( );
    cout << "Today's date is ";
    today.output( );

    cout << "J. S. Bach's birthday is ";
    bachBirthday.output( );

    if (equal(today, bachBirthday))
        cout << "Happy Birthday Johann Sebastian!\n";
    else
        cout << "Happy Unbirthday Johann Sebastian!\n";
    return 0;
}

bool equal(DayOfYear date1, DayOfYear date2)
{
    return ( date1.getMonth( ) == date2.getMonth( ) &&
        date1.getDay( ) == date2.getDay( ));
}

DayOfYear::DayOfYear(int theMonth, int theDay)
    : month(theMonth), day(theDay)
{
    checkDate();
}

int DayOfYear::getMonth( )
{
    return month;
}

int DayOfYear::getDay( )
{
    return day;
}

//Uses iostream:
void DayOfYear::input( )
{
    cout << "Enter the month as a number: ";
    cin >> month;
    cout << "Enter the day of the month: ";
    cin >> day;
}

//Uses iostream:
void DayOfYear::output( )
{
    cout << "month = " << month
         << ", day = " << day << endl;
}
```

Omitted function and constructor definitions are as in Chapter 10, Self-Test Exercises 14 and 24, but those details are not needed for what we are doing here.

DayOfYear class (Cont.)

- **A More Efficient equal()**

```
bool equal(DayOfYear date1, DayOfYear date2)
{
        return (date1.month = = date2.month &&
                date1.day = = date2.day );
}
```

- The code is simpler and more efficient
- Direct access of private member variables is not legal!

Friend Functions

- **Friend functions are not members of a class, but can access private member variables of the class**
  - A friend function is declared using the keyword ***friend*** in the class declaration and definition
  - A friend function is not a member function
  - A friend function is a global function
  - A friend function can access nonpublic members of the class

Declaring A Friend

- **equal() is declared as a friend in the class definition**

```
class DayOfYear
 {
        public:
              friend bool equal(DayOfYear date1, DayOfYear date2);
              // The rest of the public members

        private:
              // the private members
 };
```

Using A Friend Function

- A friend function is declared as a friend in the class definition
- A friend function is defined as a nonmember function without using the scope resolution operator "::"
- A friend function is called without using the '.' operator

```cpp
class DayOfYear
{
public:
    friend bool equal(DayOfYear date1, DayOfYear date2);
    //Precondition: date1 and date2 have values.
    //Returns true if date1 and date2 represent the same date;
    //otherwise, returns false.

    DayOfYear(int theMonth, int theDay);
    //Precondition: theMonth and theDay form a
    //possible date. Initializes the date according
    //to the arguments.

    DayOfYear( );
    //Initializes the date to January first.

    void input( );

    void output( );

    int getMonth( );
    //Returns the month, 1 for January, 2 for February, etc.

    int getDay( );
    //Returns the day of the month.
private:
    void checkDate( );
    int month;
    int day;
};
```

```cpp
bool equal(DayOfYear date1, DayOfYear date2)
{
    return (date1.month == date2.month &&
            date1.day == date2.day);
}
```

```cpp
int main( )
{
    DayOfYear today, bachBirthday(3, 21);

    cout << "Enter today's date:\n";
    today.input( );
    cout << "Today's date is ";
    today.output( );

    cout << "J. S. Bach's birthday is ";

    bachBirthday.output( );

    if (equal(today, bachBirthday))
        cout << "Happy Birthday Johann Sebastian!\n";
    else
        cout << "Happy Unbirthday Johann Sebastian!\n";
    return 0;
}
```

Are Friends Needed?

- use a member function if the task performed by the function involves only one class

- use a nonmember function if the task performed by the function involves more than one classes

- Friend functions are also used in operator overloading.

Program Example: Money Class

- U.S. currency is represented
- Value is implemented as an integer
- Two friend functions, equal() and add(), are used

```
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
```

```cpp
//Program to demonstrate the class Money.
#include <iostream>
#include <cstdlib>
#include <cctype>
using namespace std;

//Class for amounts of money in U.S. currency.
class Money
{
public:
    friend Money add(Money amount1, Money amount2);
    //Precondition: amount1 and amount2 have been given values.
    //Returns the sum of the values of amount1 and amount2.

    friend bool equal(Money amount1, Money amount2);
    //Precondition: amount1 and amount2 have been given values.
    //Returns true if the amount1 and amount2 have the same value;
    //otherwise, returns false.

    Money(long dollars, int cents);
    //Initializes the object so its value represents an amount with the
    //dollars and cents given by the arguments. If the amount is negative,
    //then both dollars and cents must be negative.

    Money(long dollars);
    //Initializes the object so its value represents $dollars.00.

    Money( );
    //Initializes the object so its value represents $0.00.

    double getValue( );
    //Precondition: The calling object has been given a value.
    //Returns the amount of money recorded in the data of the calling object.

    void input(istream& ins);
    //Precondition: If ins is a file input stream, then ins has already been
    //connected to a file. An amount of money, including a dollar sign, has been
    //entered in the input stream ins. Notation for negative amounts is -$100.00.
    //Postcondition: The value of the calling object has been set to
    //the amount of money read from the input stream ins.

    void output(ostream& outs);
    //Precondition: If outs is a file output stream, then outs has already been
    //connected to a file.
    //Postcondition: A dollar sign and the amount of money recorded
    //in the calling object have been sent to the output stream outs.
private:
    long allCents;
};
```

```cpp
int digitToInt(char c);
//Function declaration for function used in the definition of Money::input:
//Precondition: c is one of the digits '0' through '9'.
//Returns the integer for the digit; for example, digitToInt ('3') returns 3.

int main( )
{
    Money yourAmount, myAmount(10, 9), ourAmount;
    cout << "Enter an amount of money: ";
    yourAmount.input(cin);
    cout << "Your amount is ";
    yourAmount.output(cout);
    cout << endl;
    cout << "My amount is ";
    myAmount.output(cout);
    cout << endl;

    if (equal(yourAmount, myAmount))
        cout << "We have the same amounts.\n";
    else
        cout << "One of us is richer.\n";
    ourAmount = add(yourAmount, myAmount);
    yourAmount.output(cout);
    cout << " + ";
    myAmount.output(cout);
    cout << " equals ";
    ourAmount.output(cout);
    cout << endl;
    return 0;
}
Money add(Money amount1, Money amount2)
{
    Money temp;

    temp.allCents = amount1.allCents + amount2.allCents;
    return temp;
}
bool equal(Money amount1, Money amount2)
{
    return (amount1.allCents == amount2.allCents);
}

Money::Money(long dollars, int cents)
{
    if (dollars * cents < 0) //If one is negative and one is positive
```

```
    {
        cout << "Illegal values for dollars and cents.\n";
        exit(1);
    }
    allCents = dollars * 100 + cents;
}

Money::Money(long dollars) : allCents(dollars * 100)
{
    //Body intentionally blank. 96    }

Money::Money( ) : allCents(0)
{
    //Body intentionally blank. 101   }

double Money::getValue( )
{
    return (allCents * 0.01);
}
//Uses iostream, cctype, cstdlib:
void Money::input(istream& ins)
{
    char oneChar, decimalPoint, digit1, digit2;
    //digits for the amount of cents
    long dollars;
    int cents;
    bool negative;//set to true if input is negative.

    ins >> oneChar;
    if (oneChar == '-')
    {
        negative = true;
        ins >> oneChar; //read '$'
    }
    else
        negative = false;
    //if input is legal, then oneChar == '$'

    ins >> dollars >> decimalPoint >> digit1 >> digit2;

    if (oneChar != '$' || decimalPoint != '.'
        || !isdigit(digit1) || !isdigit(digit2))
```

```
    {
        cout << "Error illegal form for money input\n";
        exit(1);
    }
    cents = digitToInt(digit1) * 10 + digitToInt(digit2);

    allCents = dollars * 100 + cents;
    if (negative)
        allCents = -allCents;
}

//Uses cstdlib and iostream:
void Money::output(ostream& outs)
{
    long positiveCents, dollars, cents;
    positiveCents = labs(allCents);
    dollars = positiveCents / 100;
    cents = positiveCents % 100;

    if (allCents < 0)
        outs << "-$" << dollars << '.';
    else
        outs << "$" << dollars << '.';

    if (cents < 10)
        outs << '0';
    outs << cents;
}

int digitToInt(char c)
{
   return ( static_cast<int> ( c ) - static_cast<int>( '0') );
}
```

```
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
```

A call-by-reference parameter for class type

- **Call-by-reference parameters would be more better when the parameters are class type**

```
//A function declaration with constant parameters
friend Money add(Money& amount1, Money& amount2);

//A function definition with constant parameters
Money add(Money& amount1, Money& amount2)
{
   …
}
```

- **Need to use the modifier "const" when using a call-by-reference parameter**
  - If the function does not change the value of the parameter, mark the parameter with **the modifier "const"** so the compiler knows it should not be changed

const Parameter Modifier

- **To mark a call-by-reference parameter so it cannot be changed:**
  - Use the modifier **const** before the parameter type
  - The parameter becomes a ***constant parameter***
    - Example

```
//A function declaration with constant parameters
friend Money add(const Money& amount1, const Money& amount2);

//A function definition with constant parameters
Money add(const Money& amount1, const Money& amount2)
{
   ...
}
```

const Parameter Modifier (Cont.)

- **Will the compiler accept an accessor function call from the constant parameter?**

```
Money add(const Money& amount1,  const Money& amount2)
 {          …
            amount1.output(cout);
}
```

- The compiler will not accept this code
  - Because there is no guarantee that output() will not change the value of the parameter

Function Declarations With const

- **If a constant parameter makes a member function call…**
  - The invoked member function must be marked so the compiler knows it will not change the parameter
  - **const** is used to mark functions that will not change the value of an object
    - Use **const** after the parameter list and just before the semicolon

- **Example**

```
class Money
 {
      public:
                …
            void output (ostream& outs) const ;
                …
```

```
void Money::output(ostream& outs) const
{
      // output statements
}
```

Friend Class

- A **friend class** can access the non-public (private and protected) members of the class in which it is declared as a friend.

```
#include <iostream>
class A {
private:
    int a;

public:
    A() { a = 0; }
    friend class B;
};
```

```
class B {
private:
    int b;

public:
    void showA(A& x)
    {
        std::cout << "A::a=" << x.a;
    }
};
```

```
int main()
{
    A a;
    B b;
    b.showA(a);
    return 0;
}
```

Overloading Operators

```
string s1 = “good ”;
string s2 = “morning!”;

string s3 = s1+s2;
```

Overloading Operators

- **In the Money class, function add() was used to add two objects of type Money**

- **We will see how to use the '+' operator to make this code legal:**

```
Money total, cost, tax;
                …
total = cost + tax;
// instead of  total = add(cost, tax);
```

Operator Overloading

- **Operators can be overloaded**
- **The definition of + operator for the Money class is nearly the same as member function add()**

- **To overload the + operator for the Money class**
  - Use the name **+** in place of the name add()
  - Use keyword ***operator*** in front of the +
  - Example:

```
Money operator+ (const Money& amount1…
```

a3 =operator+(a1, a2);

a3 = a1 + a2;

Operator Overloading Rules

- **An overloaded operator can be a class member function or a global function**
  - If the overloaded operator is a global function,
    - At least one argument of an overloaded operator must be of a class type
    - If the overloaded operator accesses nonpublic members of a class, it should be declared as a friend of the class
- **New operators cannot be created**
- **The number of arguments for an operator cannot be changed**
- **The return type of an overloaded operator**
  - The logical and relational operators should return **bool**
  - The arithmetic operators should return a value of the class type
  - Assignment (=) and the compound-assignment (+=) operators should return a reference of the left-hand operand. Ex) a=b
- **The precedence of an operator cannot be changed**

[Operators that can be overloaded]

+	-	*	/	%	^	&	|	~
!	=	<	>	+=	-=	*=	/=	%=
^=	&=	|=	<<	>>	<<=	>>=	==	!=
<=	>=	&&	||	++	--	,	->*	->
()	[]	new	delete	new[]	delete[]			

Choosing Member or Nonmember Implementation

- **We must decide whether to make the operator either**
  - a class member
  - an ordinary nonmember (global) function
- **Guidelines**
  - The assignment (=), subscript ([]), call (()), and member access arrow (->) operators must be defined as **members**
  - The compound-assignment operators (e.g., +=) ordinarily ought to be **members**
  - Operators that change the state of their object usually should be **members**
  - Symmetric operators, such as the arithmetic, equality, relational, and bitwise operators, usually should be defined as ordinary **nonmember functions**
    - Example) int a; int b; double c;

      a=b+c; a=c+b;

Nonmember Implementation Example

```
using namespace std;

//Class for amounts of money in U.S. currency.
class Money
{
public:
    friend Money operator +(const Money& amount1, const Money& amount2);
    //Precondition: amount1 and amount2 have been given values.
    //Returns the sum of the values of amount1 and amount2.

    friend bool operator ==(const Money& amount1, const Money& amount2);
    //Precondition: amount1 and amount2 have been given values.
    //Returns true if amount1 and amount2 have the same value;
    //otherwise, returns false.

    Money(long dollars, int cents);

    Money(long dollars);

    Money( );

    double getValue( ) const;

    void input(istream& ins);
    void output(ostream& outs) const;
private:
    long allCents;
};
```

Some comments from Display 11.4 have been omitted to save space in this book, but they should be included in a real program.

<Any extra function declarations from Display 11.3 go here.>

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

    cout << "cost = ";
    cost.output(cout);
    cout << endl;
    cout << "tax = ";
    tax.output(cout);
    cout << endl;
    cout << "total bill = ";
    total.output(cout);
    cout << endl;
```

```
    if (cost == tax)
        cout << "Move to another state.\n";
    else
        cout << "Things seem normal.\n";
    return 0;
}

Money operator +(const Money& amount1, const Money& amount2)
{
    Money temp;
    temp.allCents = amount1.allCents + amount2.allCents;
    return temp;
}

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

```
cost = $1.50
tax = $0.15
total bill = $1.65
Things seem normal.
```

Overloading –

- Overloading the – operator with two parameters allows us to subtract Money objects as in

```
Money  amount1, amount2, amount2;
        …
       amount3 = amount1 – amount2;
```

- Overloading the – operator with one parameter allows us to negate a money value like this

```
amount3 =  -amount1;
```

Overloading –

```
//Class for amounts of money in U.S. currency.
class Money
{
public:
    friend Money operator +(const Money& amount1, const Money& amount2);

    friend Money operator -(const Money& amount1, const Money& amount2);
    //Precondition: amount1 and amount2 have been given values.
    //Returns amount1 minus amount2.

    friend Money operator -(const Money& amount);
    //Precondition: amount has been given a value.
    //Returns the negative of the value of amount.

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

    Money(long dollars, int cents);

    Money(long dollars);

    Money( );

    double getValue( ) const;

    void input(istream& ins);
    void output(ostream& outs) const;
private:
    long allCents;
};
```

This is an improved version of the class Money *given in Display 11.5.*

We have omitted the include *directives and some of the comments, but you should include them in your programs.*

<Any additional function declarations as well as the main part of the program go here.>

```
Money operator -(const Money& amount1, const Money& amount2)
{
    Money temp;
    temp.allCents = amount1.allCents - amount2.allCents;
    return temp;
}

Money operator -(const Money& amount)
{
    Money temp;
    temp.allCents = -amount.allCents;
    return temp;
}
```

Overloading <<

- **The insertion operator << is a binary operator**
  - The first operand is the output stream
  - The second operand is the value following <<

```
cout << "Hello out there.\n";
```

Operand 1

Operand 2

Operator

Overloading <<

- **Overloading the << operator allows us to use << instead of Money's output() function**
  - Given the declaration:

```
Money amount(100);
amount.output( cout );
```

can become

```
cout << amount;
```

Overloading <<

- **What Does << Return?**

  - Because << is a binary operator
    cout << "I have " << amount << " in my purse.";

    seems as if it could be grouped as
    ((cout << "I have") << amount) << "in my purse.";

  - To provide cout as an argument for << amount,
    (cout << "I have") must return cout

To provide cout as an argument for << amount

```
ins >> dollars >> decimalPoint >> digit1 >> digit2;
```

```
cout << "I have " << amount << " in my purse.\n";
```

means the same as

```
((cout << "I have ") << amount) << " in my purse.\n";
```

and is evaluated as follows:

First evaluate `(cout << "I have ")`, which returns `cout`:

```
((cout << "I have ") << amount) << " in my purse.\n";
```

and the string "I have" is output.

```
(cout << amount) << " in my purse.\n";
```

Then evaluate `(cout << amount)`, which returns `cout`:

```
(cout << amount) << " in my purse.\n";
```

and the value of amount is output.

```
cout << " in my purse.\n";
```

Then evaluate `cout << " in my purse.\n"`, which returns `cout`:

```
cout << " in my purse.\n";
```

and the string "in my purse.n" is output.

```
cout;
```

Since there are no more << operators, the process ends.

Overloading <<

- **Declaration**

```
class Money
{
   public:
            …
      friend ostream& operator << (ostream& outs,
                                   const Money& amount);
           …
```

- **Definition**

```
ostream& operator <<(ostream& outs, const Money& amount)
{
      ….
      return outs;
 }
```

Overloading >>

- >> could be defined this way for the Money class

```
istream& operator >>(istream& ins, Money& amount);
{
        ….
        return ins;
}
```

```cpp
//Program to demonstrate the class Money
#include <iostream>
#include <fstream>
#include <cstdlib>
#include <cctype>
using namespace std;

//Class for amounts of money in U.S. currency.
class Money
{
public:
    friend Money operator +(const Money& amount1, const Money& amount2);

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

    friend Money operator -(const Money& amount);

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

    Money(long dollars, int cents);

    Money(long dollars);

    Money( );

double get_value( ) const;

    friend istream& operator >>(istream& ins, Money& amount);
    //Overloads the >> operator so it can be used to input values of type Money.
    //Notation for inputting negative amounts is as in -$100.00.
    //Precondition: If ins is a file input stream, then ins has already been
    //connected to a file.

    friend ostream& operator <<(ostream& outs, const Money& amount);
    //Overloads the << operator so it can be used to output values of type Money.
    //Precedes each output value of type Money with a dollar sign.
    //Precondition: If outs is a file output stream,
    //then outs has already been connected to a file.
private:
    long all_cents;
};
int digit_to_int(char c);
//Used in the definition of the overloaded input operator >>.
//Precondition: c is one of the digits '0' through '9'.
//Returns the integer for the digit; for example, digit_to_int('3') returns 3.
```

```cpp
int main( )
{
    Money amount;
    ifstream in_stream;
    ofstream out_stream;

    in_stream.open("infile.dat");
    if (in_stream.fail( ))
    {
        cout << "Input file opening failed.\n";
        exit(1);
    }

    out_stream.open("outfile.dat");
    if (out_stream.fail( ))
    {
        cout << "Output file opening failed.\n";
        exit(1);
    }

    in_stream >> amount;
    out_stream << amount
               << " copied from the file infile.dat.\n";
    cout << amount
         << " copied from the file infile.dat.\n";

    in_stream.close( );
    out_stream.close( );

    return 0;
}
```

```cpp
//Uses iostream, cctype, cstdlib:
istream& operator >>(istream& ins, Money& amount)
{
    char one_char, decimal_point,
        digit1, digit2; //digits for the amount of cents
    long dollars;
    int cents;
    bool negative;//set to true if input is negative.

    ins >> one_char;
    if (one_char == '-')
    {
        negative = true;
        ins >> one_char; //read '$'
    }
    else
        negative = false;
    //if input is legal, then one_char == '$'

    ins >> dollars >> decimal_point >> digit1 >> digit2;

    if (one_char != '$' || decimal_point != '.'
        || !isdigit(digit1) || !isdigit(digit2))
    {
        cout << "Error illegal form for money input\n";
        exit(1);
    }

    cents = digit_to_int(digit1) * 10 + digit_to_int(digit2);

    amount.all_cents = dollars * 100 + cents;
    if (negative)
        amount.all_cents = -amount.all_cents;
    return ins;
}
```

```cpp
int digit_to_int(char c)
{
    return ( static_cast<int>(c) - static_cast<int>('0') );
}

//Uses cstdlib and iostream:
ostream& operator <<(ostream& outs, const Money& amount)
{
    long positive_cents, dollars, cents;
    positive_cents = labs(amount.all_cents);
    dollars = positive_cents/100;
    cents = positive_cents%100;

    if (amount.all_cents < 0)
        outs << "- $" << dollars << '.';
    else
        outs << "$" << dollars << '.';

    if (cents < 10)
        outs << '0';
    outs << cents;

    return outs;
}
```

Automatic Type Conversion

- **With the right constructors, the system can do type conversions for your classes**

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

  - The integer 25 is converted to type Money so it can be added to baseAmount!
  - How does that happen?

Automatic Type Conversion (Cont.)

- **When the compiler sees "baseAmount + 25", it first looks for an overloaded *+ operator* to perform "MoneyObject + integer"**
  - If it exists, it might look like

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

  - If the appropriate version of + is not found, the compiler looks for a constructor that takes a single integer
  - The Money constructor that takes a single parameter of type long will work
    - The constructor Money(long dollars) converts 25 to a Money object so the two values can be added!

baseAmount + 25 → OK!

baseAmount + 25.67 → Error!

Automatic Type Conversion (Cont.)

- **To permit baseAmount + 25.67,**
  - the following constructor should be declared and defined

```
class Money
{
    public:
            …
            Money(double amount);
            // Initialize object so its value is $amount
            …
```

Overloaded Operator as a Class Member

- **Declare the overloaded operators as member functions of a class**
- **The first (left-hand) operand is the object in which the overloaded operator is invoked.**
  - So, the member operator function has one parameter less than the operator has operands
  - Example :

1st operand (object) → **a = b+c;**

1st operand (object) → **a += b;**

=

a.operator+=(b);

this

A hidden **pointer variable** that holds the address of current object

- **Example**

```
class Money
 {
      public:
           Money operator+(const Money& amount2);
              ……
```

```
Money Money::operator+(const Money& amount2)
{
      Money temp;
      temp.all_cents = this->all_cents + amount2.all_cents;
      return temp;
}
```

Operator +=

- The compound-assignment operators (e.g., +=) ordinarily ought to be **members**

1st operand

a += b;

=

a.operator+=(b);

- **Example**

```
Money Amount1(10,10), Amount2(10, 9);
 Amount1 += Amount2;
```

```
class Money
 {
  public:
     Money operator+=(const Money& amount2);
              ……
```

```
Money& Money::operator+=(const Money& amount2)
{
        this->all_cents = this->all_cents + amount2.all_cents;
        return *this;
}
```

Why return reference?

Amount1+= Amount2 += Amount3;

Amount1 = Amount2 = Amount3;

NEXT ?

Classes (Part3)

Arrays and Classes