Chapter 8

Operator Overloading,
Friends, and References

Learning Objectives

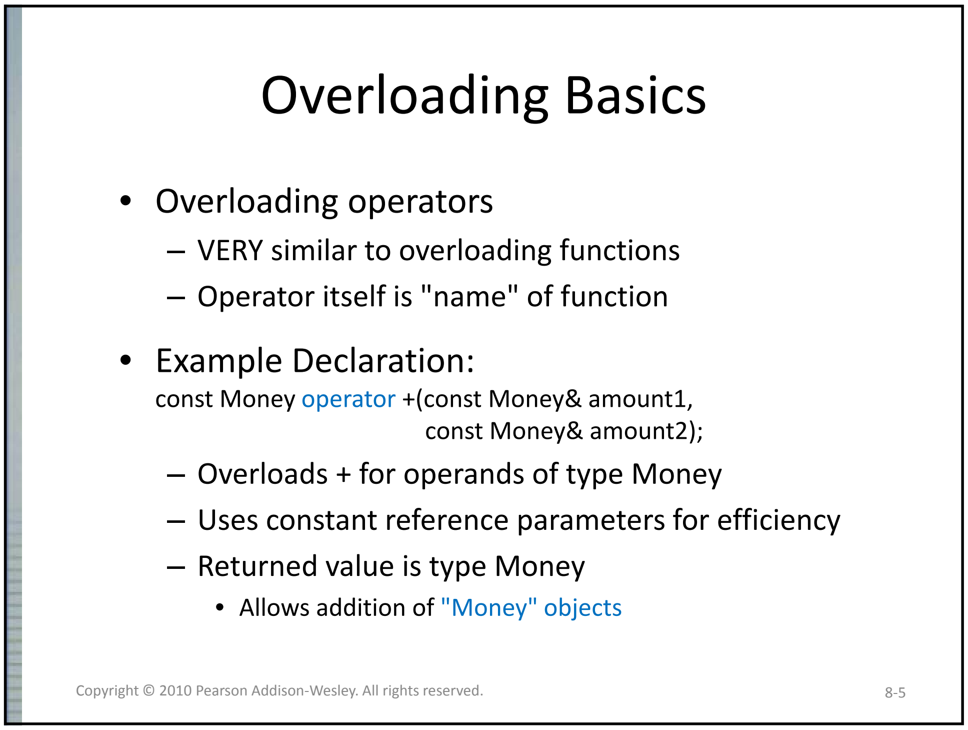
- Basic Operator Overloading
  - Unary operators
  - As member functions
- Friends and Automatic Type Conversion
  - Friend functions, friend classes
  - Constructors for automatic type conversion
- References and More Overloading
  - << and >>
  - Operators: = , [], ++, --

Operator Overloading Introduction

- Operators +, -, %, ==, etc.
  - Really just functions!
- Simply "called" with different syntax:
  x + 7
  - "+" is binary operator with x and 7 as operands
  - We "like" this notation as humans
- Think of it as:
  +(x, 7)
  - "+" is the function name
  - x, 7 are the arguments
  - Function "+" returns "sum" of it's arguments

Operator Overloading Perspective

- Built-in operators
  - e.g., +, -, = , %, ==, /, *
  - 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
- Always overload with similar "actions"!

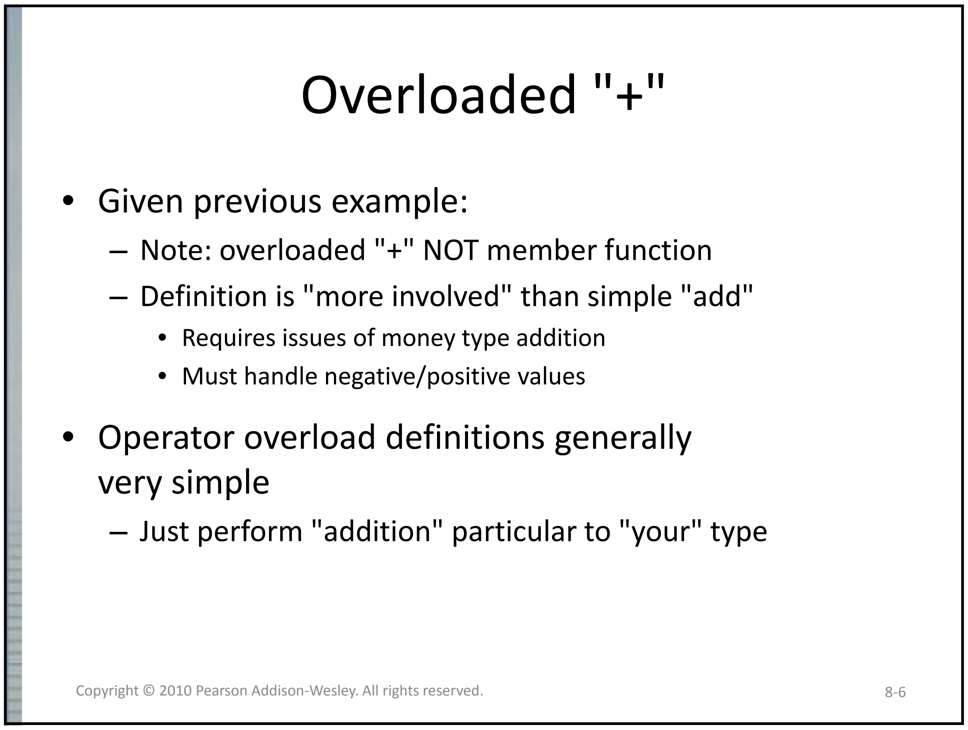


Overloading Basics

- 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

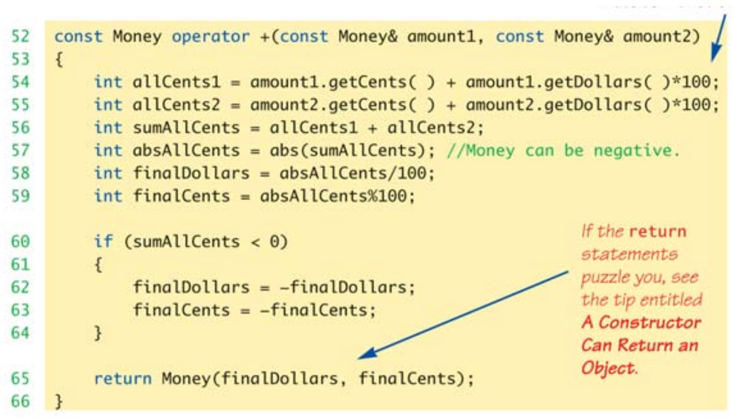


Overloaded "+"

- 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

Money "+" Definition: **Display 8.1** Operator Overloading

- Definition of "+" operator for Money class:



```
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);
}
```

If the return statements puzzle you, see the tip entitled ***A Constructor Can Return an Object.***

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)

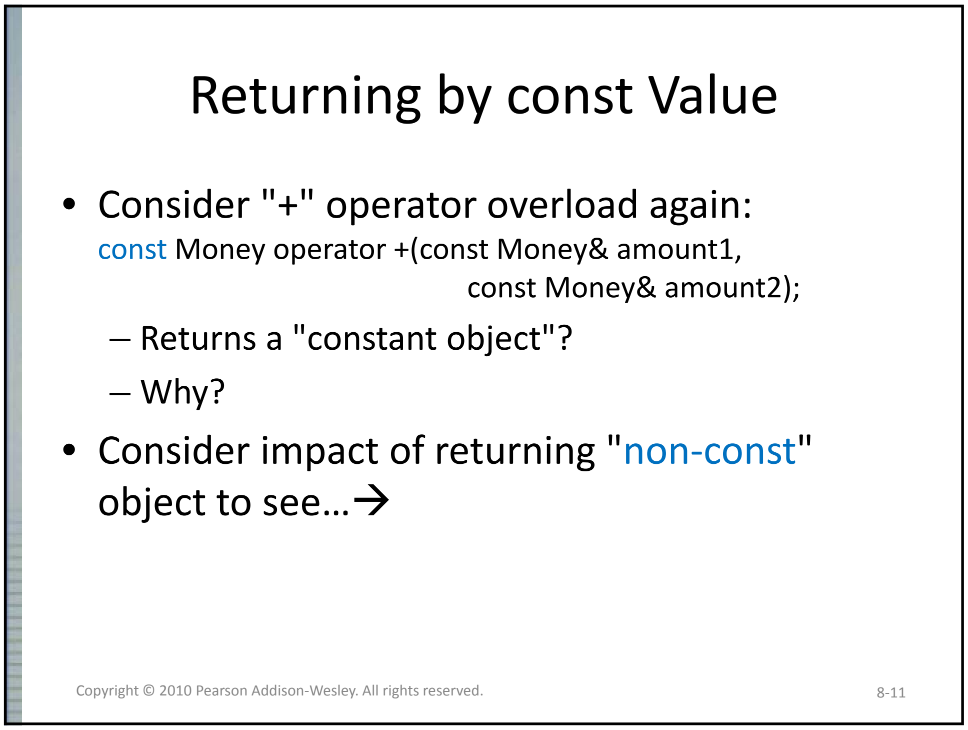
Overloaded "==" for Money: **Display 8.1** Operator Overloading

- Definition of "==" operator for Money class:

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

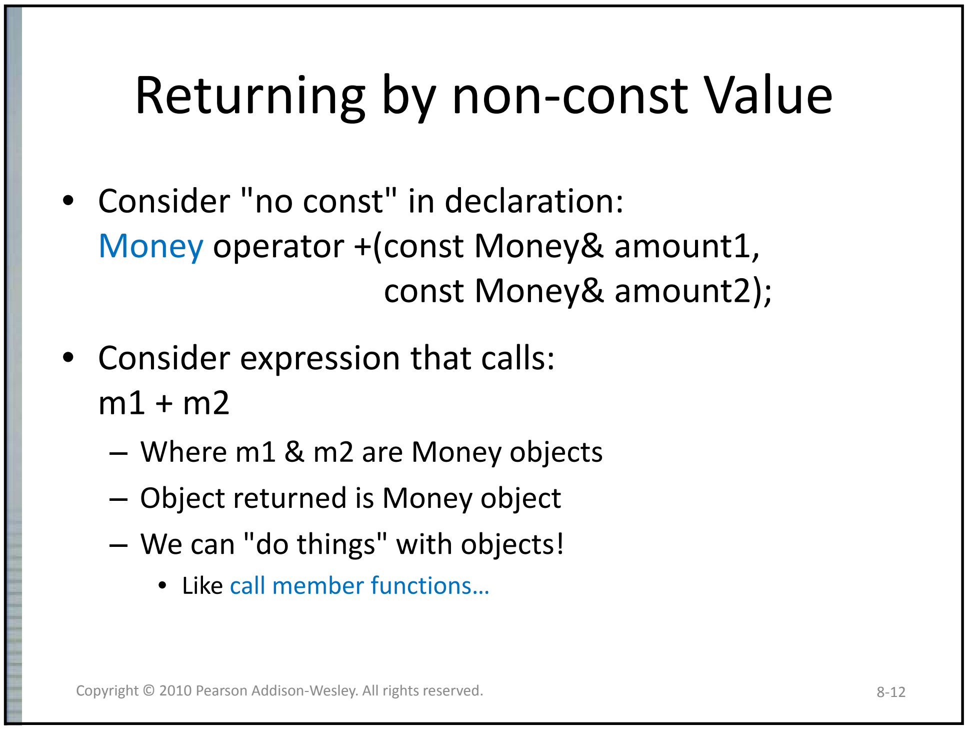
Constructors Returning Objects

- Constructor a "void" function?
  - We "think" that way, but no
  - A "special" function
    - With special properties
    - CAN return a value!
- Recall return statement in "+" overload for Money type:
  - return Money(finalDollars, finalCents);
    - Returns an "invocation" of Money class!
    - So constructor actually "returns" an object!
    - Called an "anonymous object"



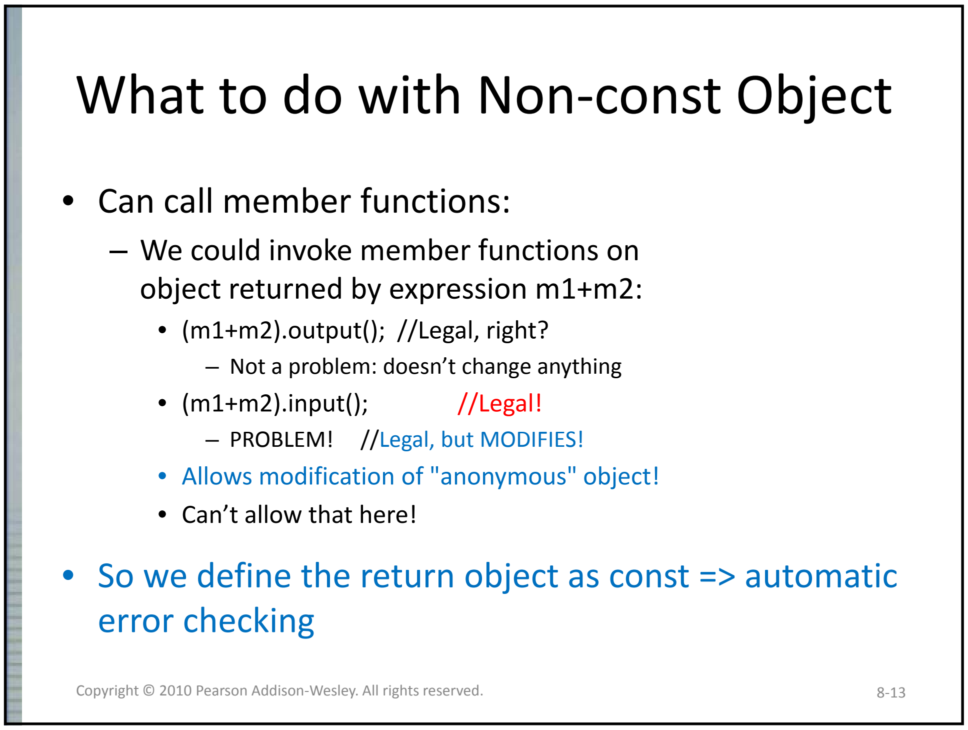
Returning by const Value

- 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 to see…→



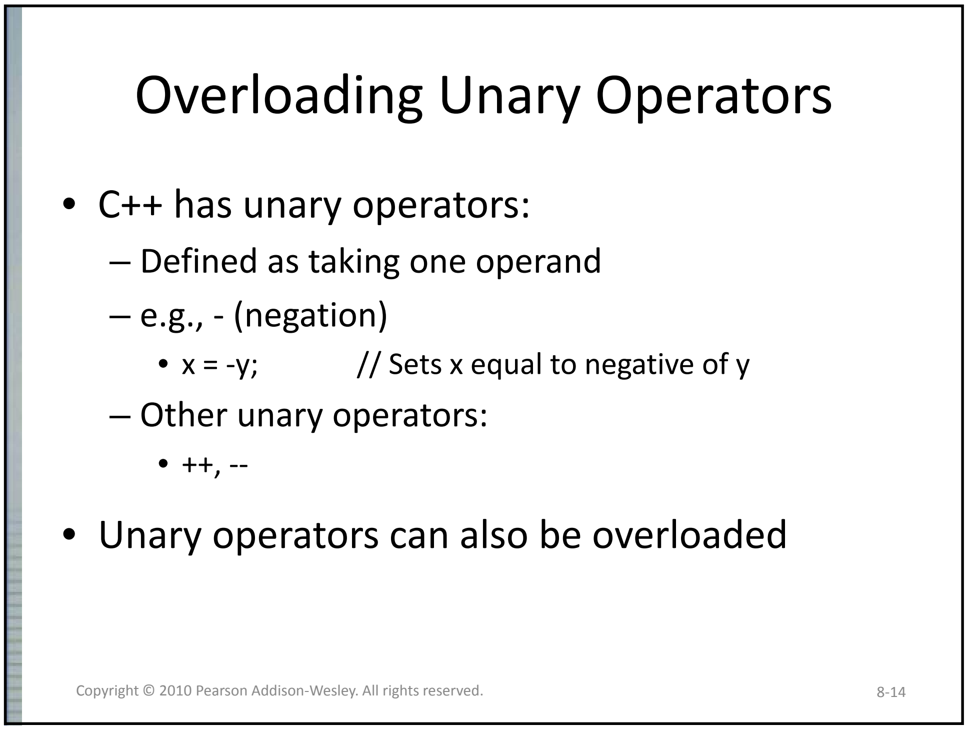
Returning by non-const Value

- Consider "no const" in declaration:
  Money operator +(const Money& amount1,
                                  const Money& amount2);
- Consider expression that calls:
  m1 + m2
  - Where m1 & m2 are Money objects
  - 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 => automatic error checking



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 two operands/arguments (binary)
  - For one operand/argument (unary)
  - Definitions must 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 an 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 as Member Functions

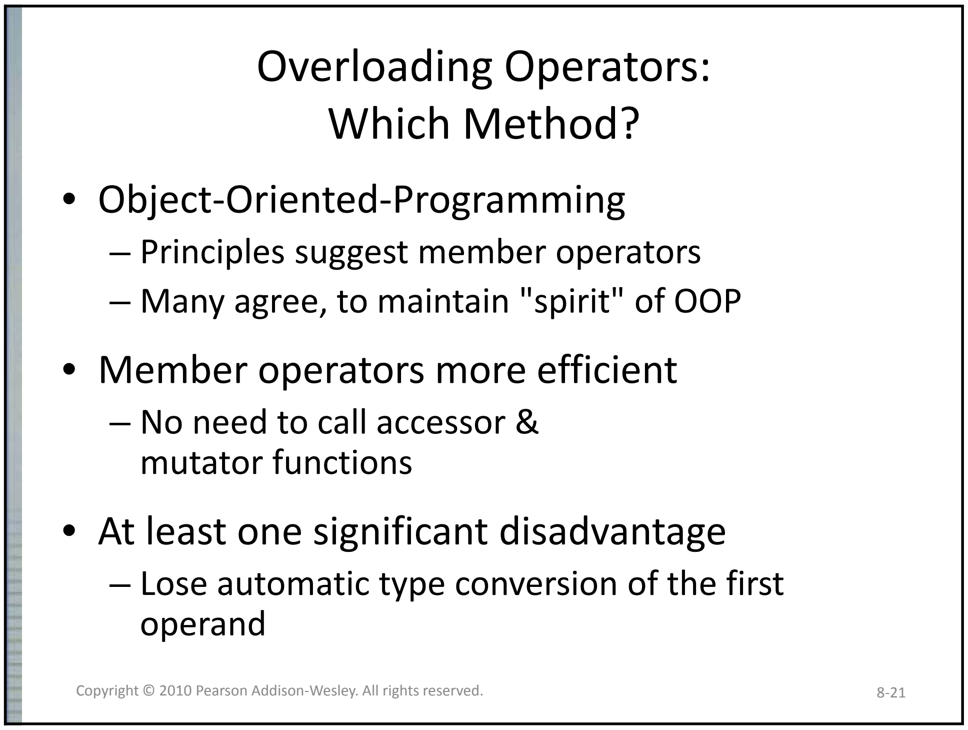
- Previous examples: standalone functions
  - Defined outside a class
- Can overload as "member operator"
  - Considered "member function" like others
- When a binary operator is a member function:
  - Only ONE parameter, not two!
  - Calling object serves as 1st parameter

Member Operator in Action

- 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);
- Declaration of "+" in class definition:
  - const Money operator +(const Money& amount);
  - Notice only ONE argument

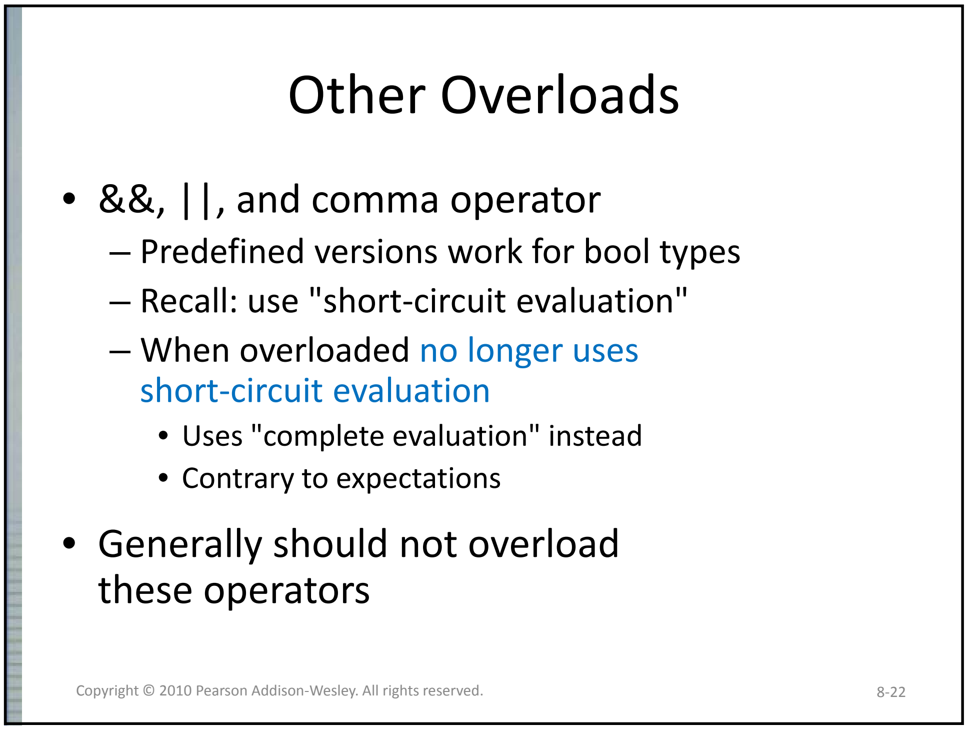
const Functions

- When to make function const?
  - Constant functions not allowed to alter class member data
  - Constant objects can ONLY call constant member functions
- Good style dictates:
  - Any member function that will NOT modify data should be made const
- Use keyword *const* after function declaration and heading



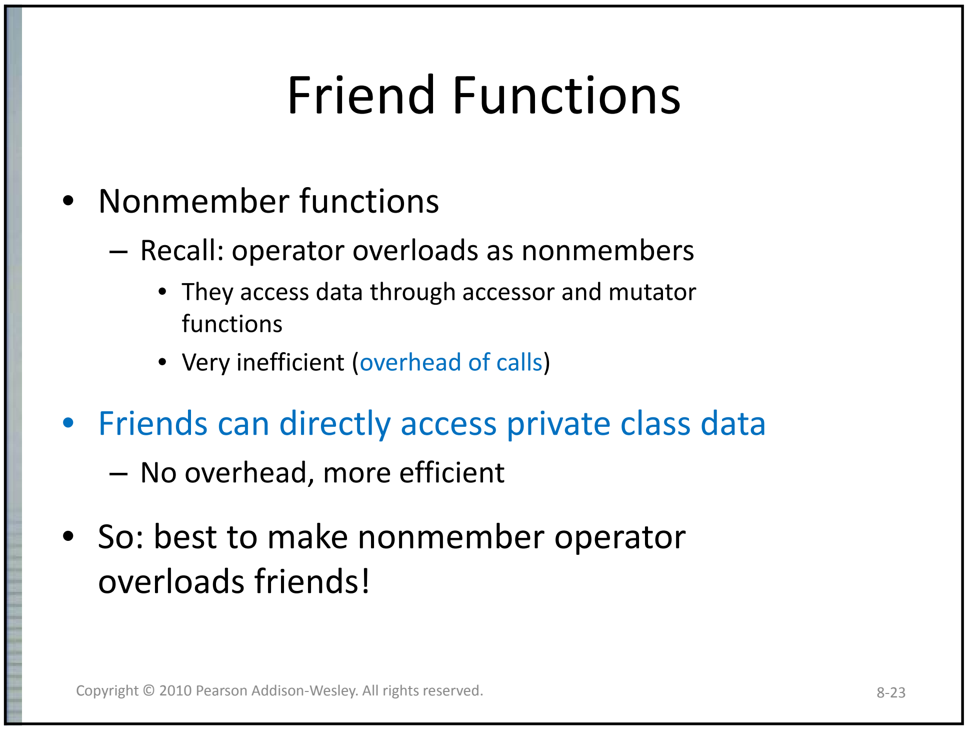
Overloading Operators: Which Method?

- 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
- At least one significant disadvantage
  - Lose automatic type conversion of the first operand



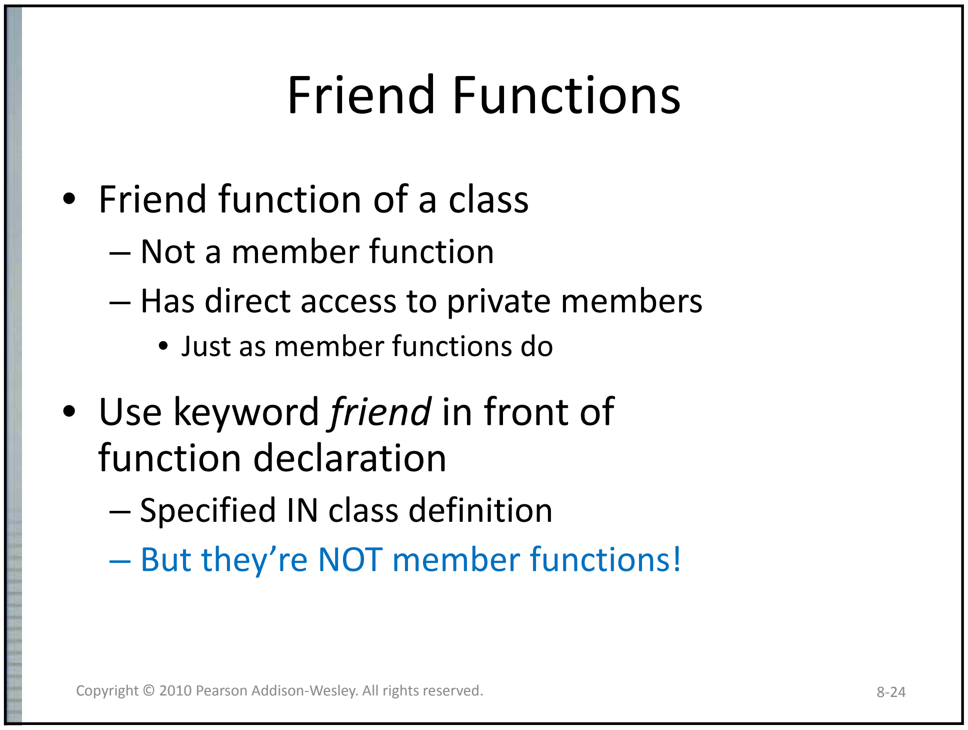
Other Overloads

- &&, ||, and comma operator
  - Predefined versions work for bool types
  - Recall: use "short-circuit evaluation"
  - When overloaded no longer uses short-circuit evaluation
    - Uses "complete evaluation" instead
    - Contrary to expectations
- Generally should not overload these operators



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

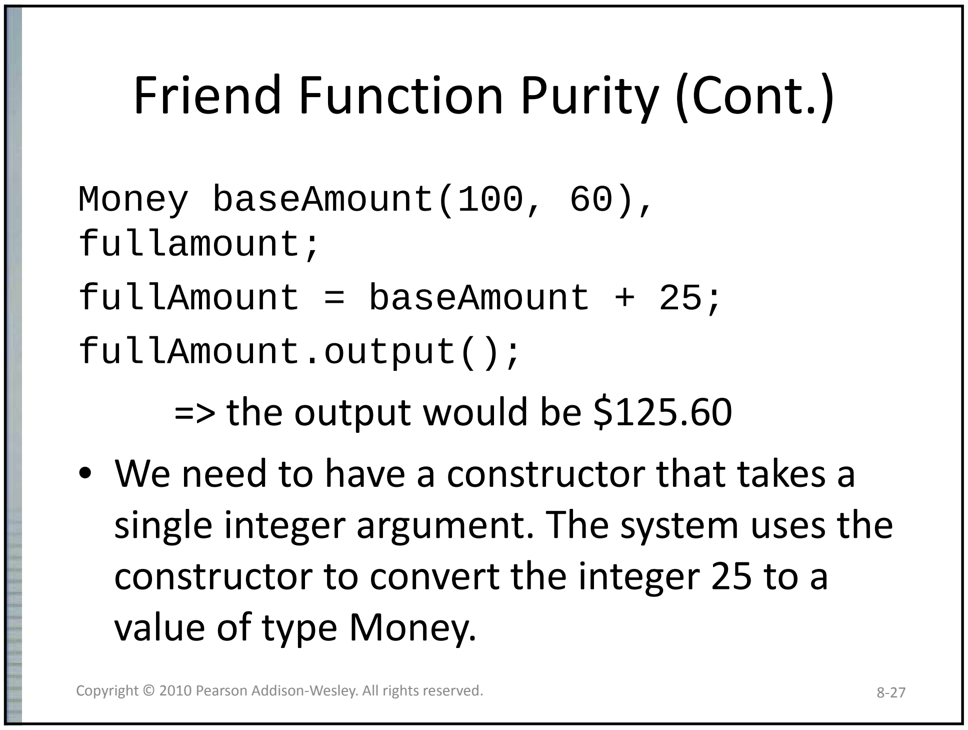
- Friend function of a class
  - Not a member function
  - Has direct access to private members
    - Just as member functions do
- Use keyword *friend* in front of function declaration
  - Specified IN class definition
  - But they're NOT member functions!

Friend Function Uses

- Operator Overloads
  - Most common use of friends
  - Improves efficiency
  - Avoids need to call accessor/mutator member functions
  - Operator must have access anyway
    - Might as well give full access as friend
- Friends can be any function

Friend Function Purity

- Friends not pure?
  - "Spirit" of OOP dictates all operators and functions be member functions
  - Many believe friends violate basic OOP principles
- Advantageous?
  - For operators: very!
  - *Allows automatic type conversion* (with the appropriate constructors)
  - Still encapsulates: friend is in class definition
  - Improves efficiency

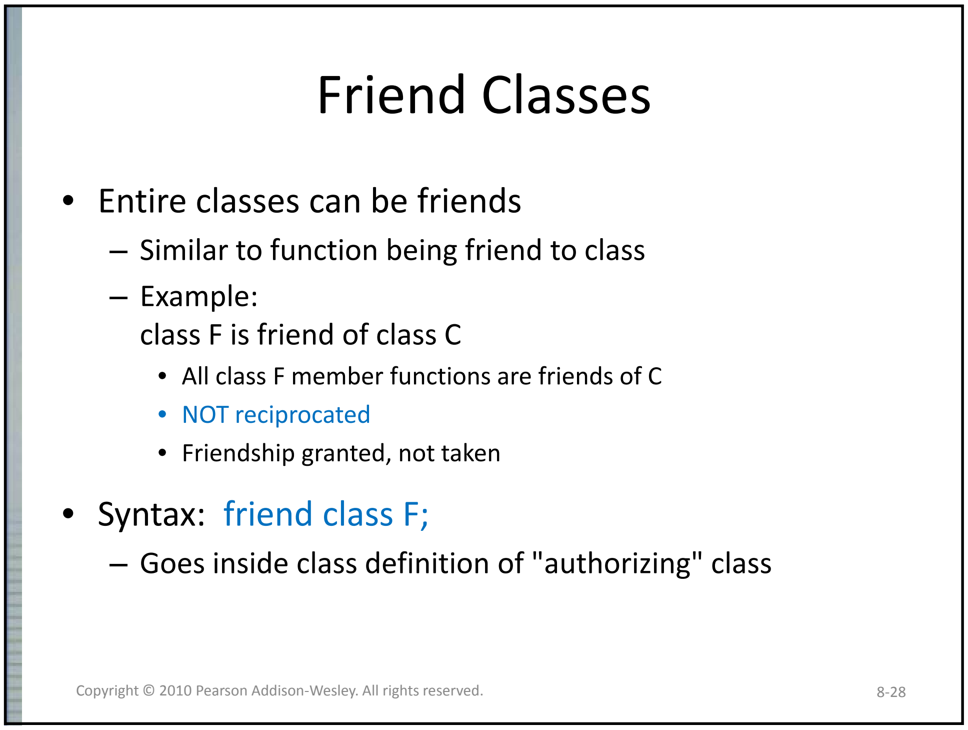


Friend Function Purity (Cont.)

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

=> the output would be $125.60

- We need to have a constructor that takes a single integer argument. The system uses the constructor to convert the integer 25 to a value of type Money.



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

References

- Reference defined:
  - Name of a storage location
  - Similar to "pointer"
- Example of standalone reference:
  - int a;
    int& b = a;
    - *b* is reference to storage location for *a*
    - Changes made to *b* will affect *a*
- Confusing?

References Usage

- Seemingly dangerous
- Useful in several cases:
- Call-by-reference
  - Often used to implement this mechanism
- Returning a reference
  - Allows operator overload implementations to be written more naturally
  - Think of as returning an "alias" to a variable

Returning Reference

- Syntax:
  double& sampleFunction(double& variable);
  - double& and double are different
  - Must match in function declaration and heading
- Returned item must "have" a reference
  - Like a variable of that type
  - Cannot be an expression like "x+5"
    - *Has no place in memory to "refer to"*

Returning Reference in Definition

- Example function definition:

```
double& sampleFunction(double& variable)
{
	return variable;
}
```

- Trivial, useless example
- Shows concept only
- Major use:
  - Certain overloaded operators

Returning Reference in Definition (Cont.)

```
double m = 99;
cout << sampleFunction(m) << endl;
sampleFunction(m) = 42;
Cout << m << endl;
    => output 99 and then 42
```

Overloading >> and <<

- Enables input and output of our objects
  - Similar to other operator overloads
- Improves readability
  - Like all operator overloads do
  - Enables:
    cout << myObject;
    cin >> myObject;
  - Instead of need for:
    myObject.output(); …

Overloading <<

- Insertion operator, <<
  - Used with cout
  - A binary operator
- Example:
  cout << "Hello";
  - Operator is <<
  - 1st operand is predefined object *cout*
    - From library iostream
  - 2nd operand is literal string "Hello"

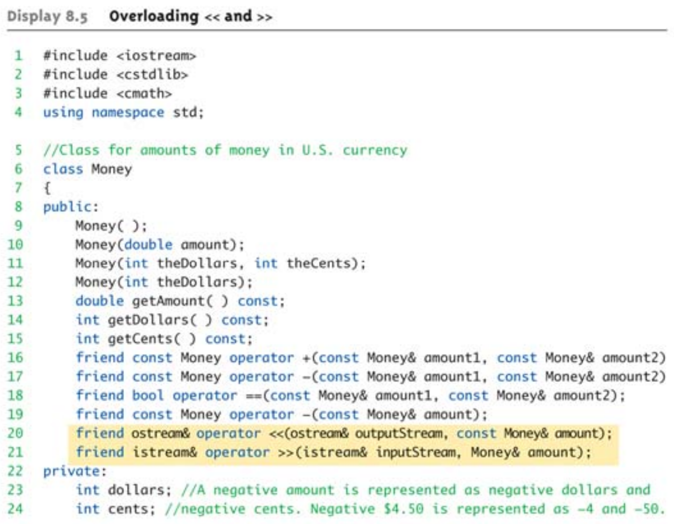
Overloading <<

- 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?
  - *cout object*!
    - Returns it's first argument type, ostream

Overloaded >> Example:
Display 8.5 Overloading << and >> (1 of 5)


Display 8.5 **Overloading << and >>**

```
#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;
    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);
private:
    int dollars; //A negative amount is represented as negative dollars and
    int cents; //negative cents. Negative $4.50 is represented as -4 and -50.
```

Overloaded >> Example: **Display 8.5** Overloading << and >> (2 of 5)

```
    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: ";
    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;
```

Overloaded >> Example: **Display 8.5** Overloading << and >> (3 of 5)

Display 8.5 **Overloading << and >>**

```
    cout << yourAmount << " + " << myAmount
         << " equals " << ourAmount << endl;

    Money diffAmount = yourAmount - myAmount;
    cout << yourAmount << " - " << myAmount
         << " equals " << diffAmount << endl;

    return 0;
}
```

Since << returns a reference, you can chain << like this. You can chain >> in a similar way.

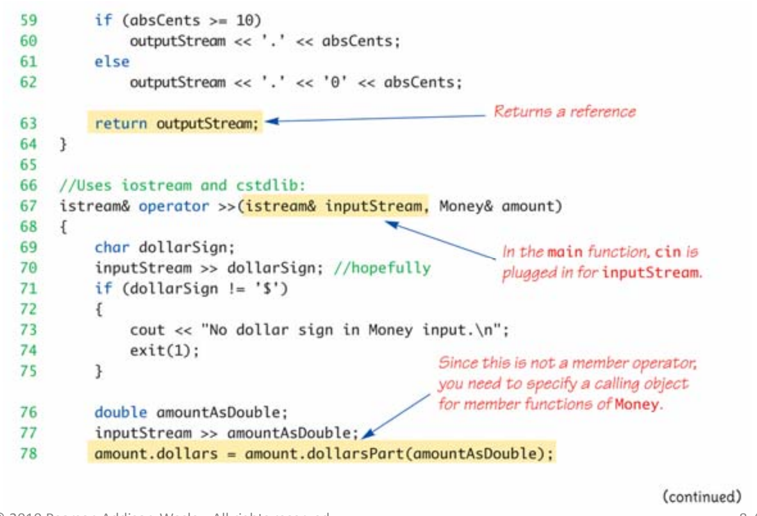
<Definitions of other member functions are as in Display 8.1. Definitions of other overloaded operators are as in Display 8.3.>

```
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;
```

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

For an alternate input algorithm, see Self-Test Exercise 3 in Chapter 7.

Overloaded >> Example:
Display 8.5 Overloading << and >> (4 of 5)



```
    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);
```

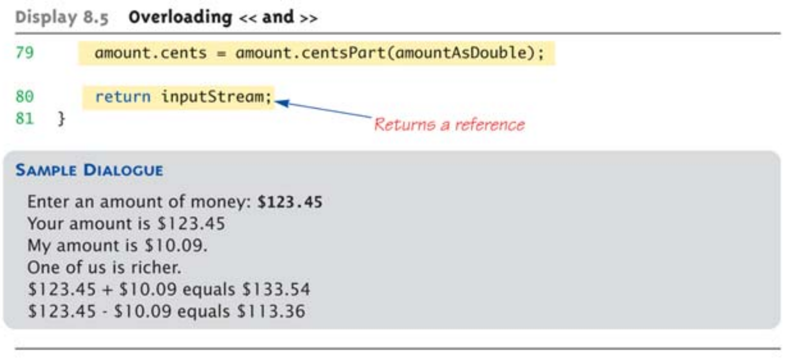
Returns a reference

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**.

(continued)

Overloaded >> Example:
Display 8.5 Overloading << and >> (5 of 5)



Display 8.5 **Overloading << and >>**

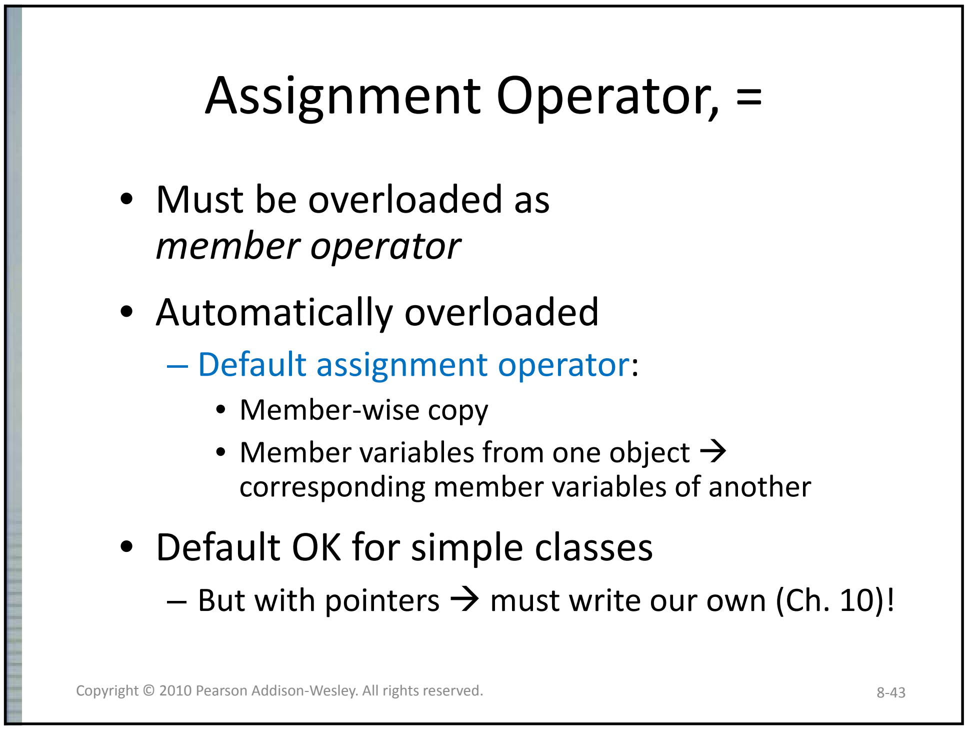
```
    amount.cents = amount.centsPart(amountAsDouble);

    return inputStream;
}
```

Returns a reference

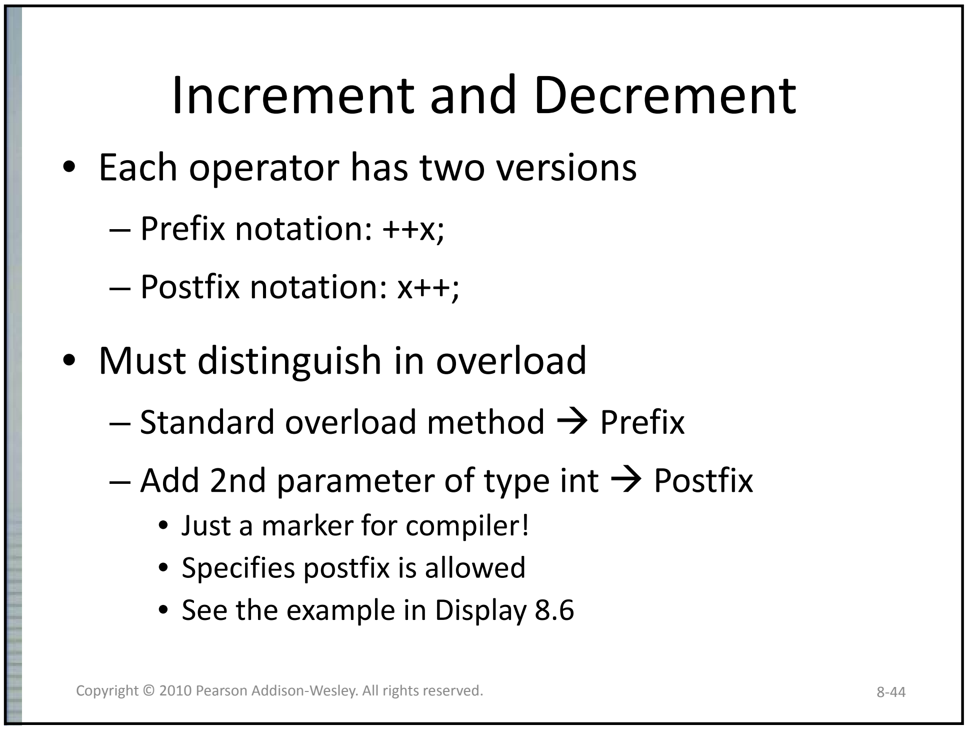
SAMPLE DIALOGUE

```
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
```



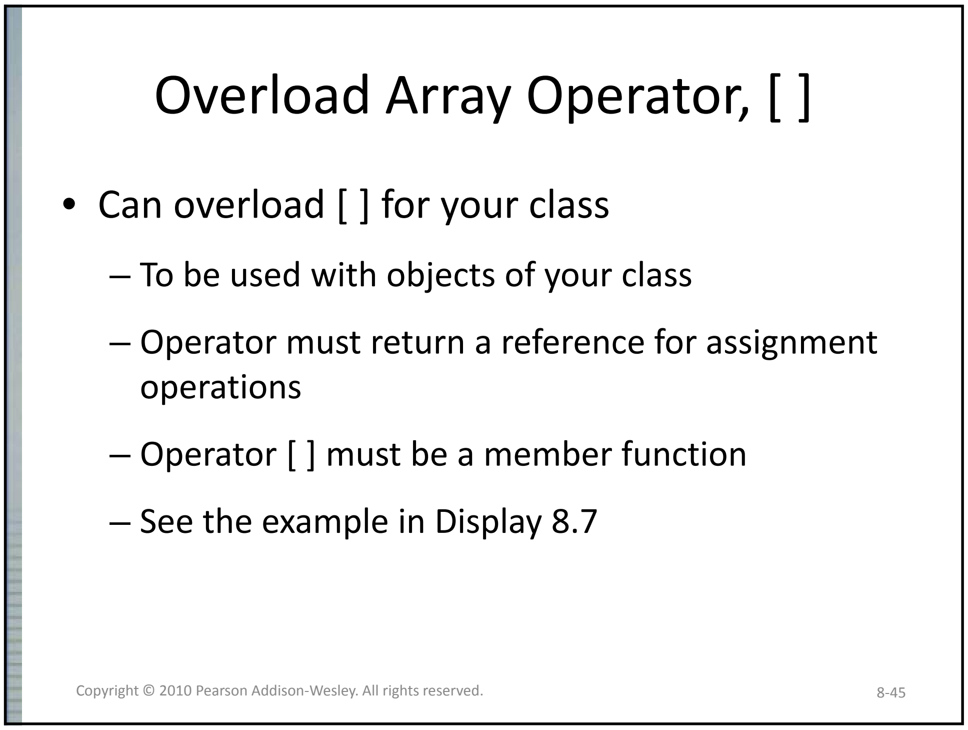
Assignment Operator, =

- Must be overloaded as *member operator*
- Automatically overloaded
  - Default assignment operator:
    - Member-wise copy
    - Member variables from one object → corresponding member variables of another
- Default OK for simple classes
  - But with pointers → must write our own (Ch. 10)!



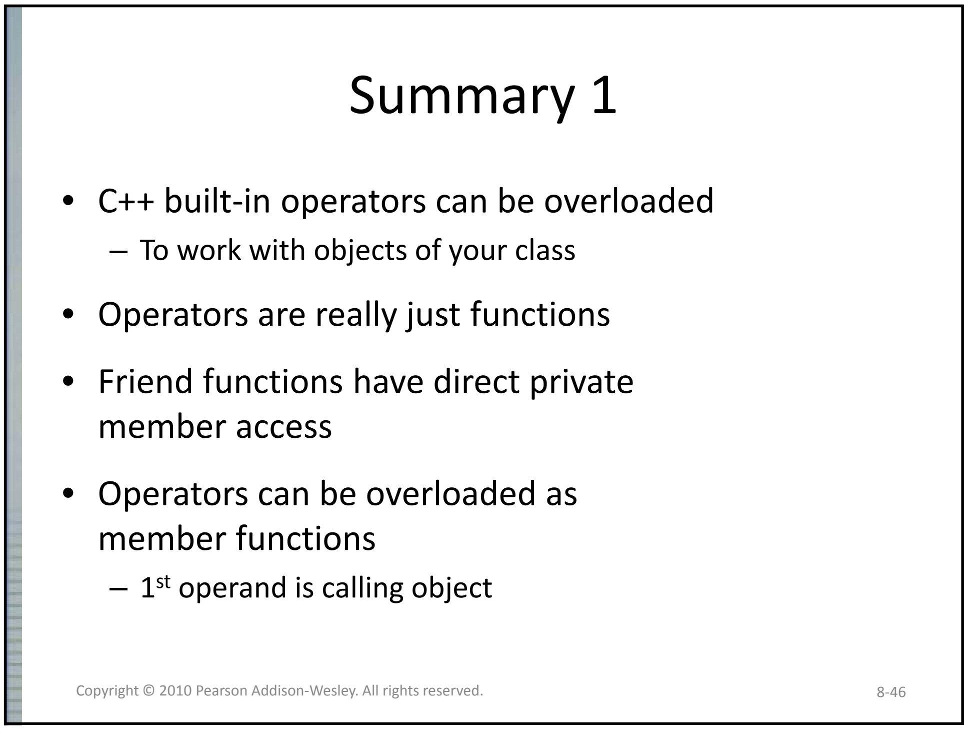
Increment and Decrement

- Each operator has two versions
  - Prefix notation: ++x;
  - Postfix notation: x++;
- Must distinguish in overload
  - Standard overload method → Prefix
  - Add 2nd parameter of type int → Postfix
    - Just a marker for compiler!
    - Specifies postfix is allowed
    - See the example in Display 8.6



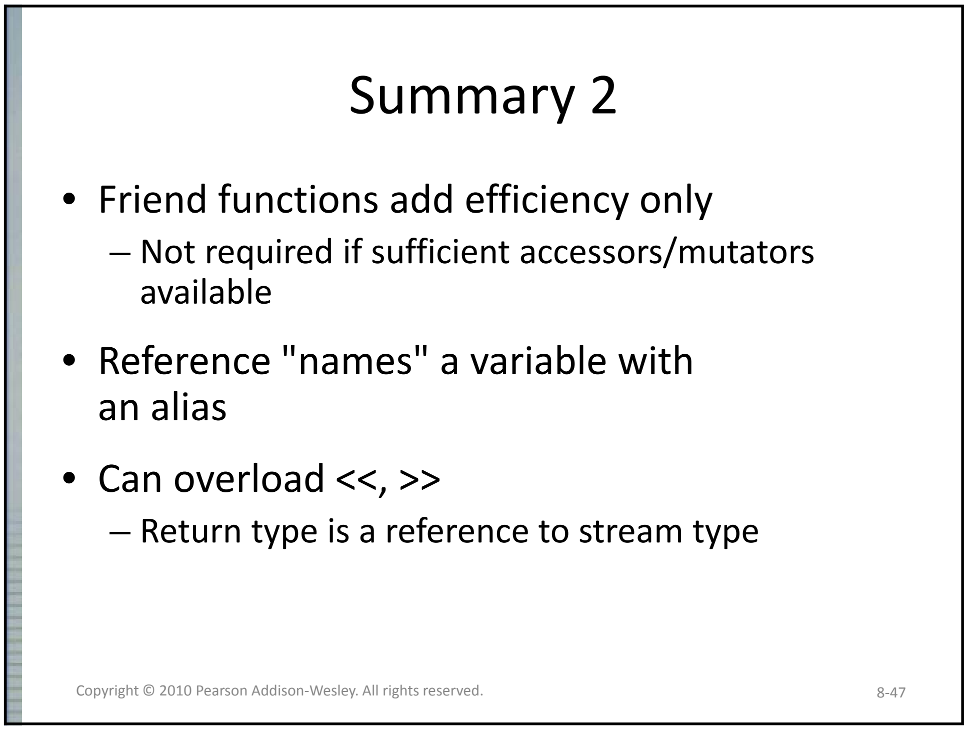
Overload Array Operator, [ ]

- Can overload [ ] for your class
  - To be used with objects of your class
  - Operator must return a reference for assignment operations
  - Operator [ ] must be a member function
  - See the example in Display 8.7



Summary 1

- C++ built-in operators can be overloaded
  - To work with objects of your class
- Operators are really just functions
- Friend functions have direct private member access
- Operators can be overloaded as member functions
  - 1st operand is calling object



Summary 2

- Friend functions add efficiency only
  - Not required if sufficient accessors/mutators available
- Reference "names" a variable with an alias
- Can overload <<, >>
  - Return type is a reference to stream type