



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 & 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:

```
52  const Money operator +(const Money& amount1, const Money& amount2)
53  {
54      int allCents1 = amount1.getCents( ) + amount1.getDollars( )*100;
55      int allCents2 = amount2.getCents( ) + amount2.getDollars( )*100;
56      int sumAllCents = allCents1 + allCents2;
57      int absAllCents = abs(sumAllCents); //Money can be negative.
58      int finalDollars = absAllCents/100;
59      int finalCents = absAllCents%100;

60      if (sumAllCents < 0)
61      {
62          finalDollars = -finalDollars;
63          finalCents = -finalCents;
64      }

65      return Money(finalDollars, finalCents);
66  }
```

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:

```
83 bool operator ==(const Money& amount1, const Money& amount2)
84 {
85     return ((amount1.getDollars( ) == amount2.getDollars( ))
86             && (amount1.getCents( ) == amount2.getCents( )));
87 }
```

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`

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 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 operator is 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:
 - Variable/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
 - (Later in chapter...)

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

Other Overloads

- &&, ||, and comma operator
 - Predefined versions work for bool types
 - Recall: use "short-circuit evaluation"
 - When overloaded no longer uses short-circuit
 - 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
 - Still encapsulates: friend is in class definition
 - Improves efficiency

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 stand alone reference:
 - `int robert;`
`int& bob = robert;`
 - *bob* is reference to storage location for *robert*
 - Changes made to *bob* will affect *robert*
- 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 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

Overloading >> and <<

- Enables input and output of our objects
 - Similar to other operator overloads
 - New subtleties
- 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 its first argument type, ostream

Overloaded >> Example:

Display 8.5 Overloading << and >> (1 of 5)

Display 8.5 Overloading << and >>

```
1  #include <iostream>
2  #include <cstdlib>
3  #include <cmath>
4  using namespace std;

5  //Class for amounts of money in U.S. currency
6  class Money
7  {
8  public:
9      Money( );
10     Money(double amount);
11     Money(int theDollars, int theCents);
12     Money(int theDollars);
13     double getAmount( ) const;
14     int getDollars( ) const;
15     int getCents( ) const;
16     friend const Money operator +(const Money& amount1, const Money& amount2)
17     friend const Money operator -(const Money& amount1, const Money& amount2)
18     friend bool operator ==(const Money& amount1, const Money& amount2);
19     friend const Money operator -(const Money& amount);
20     friend ostream& operator <<(ostream& outputStream, const Money& amount);
21     friend istream& operator >>(istream& inputStream, Money& amount);
22 private:
23     int dollars; //A negative amount is represented as negative dollars and
24     int cents; //negative cents. Negative $4.50 is represented as -4 and -50.
```

Overloaded >> Example:

Display 8.5 Overloading << and >> (2 of 5)

```
25     int dollarsPart(double amount) const;
26     int centsPart(double amount) const;
27     int round(double number) const;
28 };

29 int main( )
30 {
31     Money yourAmount, myAmount(10, 9);
32     cout << "Enter an amount of money: ";
33     cin >> yourAmount;
34     cout << "Your amount is " << yourAmount << endl;
35     cout << "My amount is " << myAmount << endl;
36
37     if (yourAmount == myAmount)
38         cout << "We have the same amounts.\n";
39     else
40         cout << "One of us is richer.\n";

41     Money ourAmount = yourAmount + myAmount;
```

Overloaded >> Example:

Display 8.5 Overloading << and >> (3 of 5)

Display 8.5 Overloading << and >>

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

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

47     return 0;
48 }
```

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.>

```
49 ostream& operator <<(ostream& outputStream, const Money& amount)
50 {
51     int absDollars = abs(amount.dollars);
52     int absCents = abs(amount.cents);
53     if (amount.dollars < 0 || amount.cents < 0)
54         //accounts for dollars == 0 or cents == 0
55         outputStream << "$-";
56     else
57         outputStream << '$';
58     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)

```
59     if (absCents >= 10)
60         outputStream << '.' << absCents;
61     else
62         outputStream << '.' << '0' << absCents;
63     return outputStream;
64 }
65
66 //Uses iostream and cstdlib:
67 istream& operator >>(istream& inputStream, Money& amount)
68 {
69     char dollarSign;
70     inputStream >> dollarSign; //hopefully
71     if (dollarSign != '$')
72     {
73         cout << "No dollar sign in Money input.\n";
74         exit(1);
75     }
76
77     double amountAsDouble;
78     inputStream >> amountAsDouble;
79     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 >>

```
79     amount.cents = amount.centsPart(amountAsDouble);  
  
80     return inputStream;  
81 }
```

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 from other
- Default OK for simple classes
 - But with pointers → must write our own!

Increment and Decrement

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

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!

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