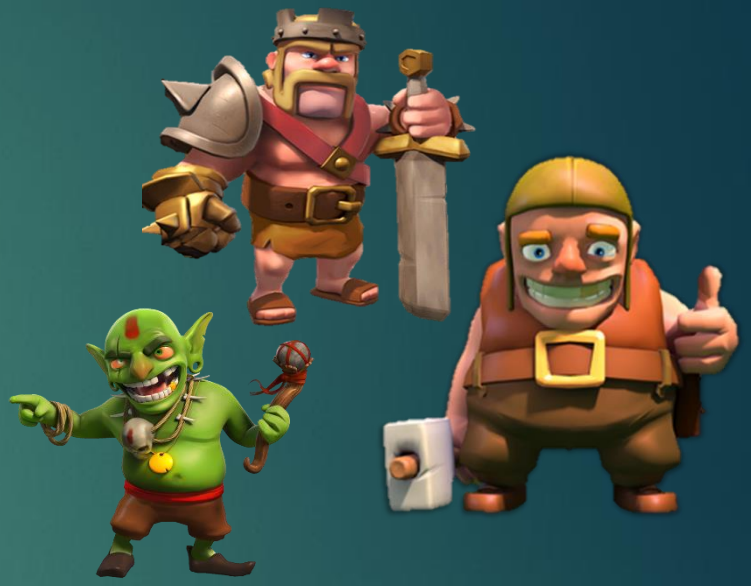
Chapter 8

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: = , [], ++, --

Why operation overloading? Game characters



- Attributes
  - Hp
  - Mp
  - Model file …
- Behavior
  - Set/input hp, mp, etc.
  - Show attributes
  - Merge two character : hp+ hp, mp+mp
  - Split from one: hp- hp, mp- mp
  - Compare: "==", "<"
  - …

Clash of royale trailer:
https://www.youtube.com/watch?v=SXMfete5mrs

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"!

Example: Money class

```
1)  class Money
2)  {
3)  public:
4)      Money( );
5)      Money(double amount);
6)      Money(int theDollars, int theCents);
7)      Money(int theDollars);
8)      double getAmount( ) const;
9)      int getDollars( ) const;
10)     int getCents( ) const;
11)     void input( ); //Reads the dollar sign as well as the amount number.
12)     void output( ) const;
13) private:
14)     int dollars; //A negative amount is represented as negative dollars and
15)     int cents; //negative cents. Negative $4.50 is represented as -4 and -50

16)     int dollarsPart(double amount) const;
17)     int centsPart(double amount) const;
18)     int round(double number) const;
19) };
```

Operator overloading for Money Class

1) const Money operator +(const Money& amount1,
   const Money& amount2);
2) const Money operator -(const Money& amount1,
   const Money& amount2);
3) bool operator ==(const Money& amount1,
   const Money& amount2);
4) const Money operator -(const Money& amount);

Example usage for operator overloaded

```
1)  int main( )
2)  {
3)      Money yourAmount, myAmount(10, 9);
4)       …

5)      Money ourAmount = yourAmount + myAmount;
6)      …
7)      Money diffAmount = yourAmount - myAmount;
8)      …
9)      return 0;
10) }
```

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

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.***

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**

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)

- 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  }
```

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

```
1)  class Money
2)  {
3)  public:
4)      Money( );
5)      Money(double amount);
6)      Money(int dollars, int cents);
7)      Money(int dollars);
8)      double getAmount( ) const;
9)      int getDollars( ) const;
10)     int getCents( ) const;
11)     void input( ); //Reads the dollar sign as well as th    mount number.
12)     void output( ) const;
13)     const Money operator +(const Money& amount2) const;
14)     const Money operator -(const Money& amount2) const;
15)     bool operator ==(const Money& amount2) const;
16)     const Money operator -( ) const;
17) private:
18)     int dollars; //A negative amount is represented as negative dollars and
19)     int cents; //negative cents. Negative $4.50 is represented as -4 and -50

20)     int dollarsPart(double amount) const;
21)     int centsPart(double amount) const;
22)     int round(double number) const;
23) };
```

Member operators

```
1) const Money Money::operator +(const Money& secondOperand) const
2)     {
3)         int allCents1 = cents + dollars*100;
4)         int allCents2 = secondOperand.cents +
                                        secondOperand.dollars*100;
5)         int sumAllCents = allCents1 + allCents2;
6)         int absAllCents = abs(sumAllCents); //Money can be negative.
7)         int finalDollars = absAllCents/100;
8)         int finalCents = absAllCents%100;

9)         if (sumAllCents < 0)    {
10)            finalDollars = -finalDollars;
11)            finalCents = -finalCents;
12)        }

13)        return Money(finalDollars, finalCents);
14)    }
```

```
1)  int Money::dollarsPart(double amount) const
2)  {
3)      return static_cast<int>(amount);
4)  }

5)  int Money::centsPart(double amount) const
6)  {
7)      double doubleCents = amount*100;
8)      int intCents = (round(fabs(doubleCents)))%100;
        //% can misbehave on negatives
9)      if (amount < 0)
10)         intCents = -intCents;
11)     return intCents;
12) }

13) int Money::round(double number) const
14) {
15)     return static_cast<int>(floor(number + 0.5));
16) }
```

What have you noticed a special keyword?

The syntax for Constant Functions

```
<return-value> <class>::<mbr-fun>(<args>) const
{
    // ...
}
```

put the const:
at the end of
the function

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**

- Const functions can always be called
- Non-const functions can only be called by non-const objects

By the way,

- What's the difference between
  (1) "const X* p",
  (2) "X* const p" and
  (3) "const X* const p"?

- const X* p : "p points to an X that is const": the X object can't be changed via p.
- X* const p : "p is a const pointer to an X that is non-const": you can't change the pointer p itself, but you can change the X object via p.
- const X* const p : "p is a const pointer to an X that is const": you can't change the pointer p itself, nor can you change the X object via p.

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
  - Automatic type conversion problem!

Automatic Type Conversion

- Overload a binary operator as a member operator
  - 1st argument is a calling object, and
  - only the second "argument" is a true argument
  - So,
    Any automatic type conversion will only apply to the **second argument**
- **Ex.**
  1) Money baseAmount(100, 60), fullAmount;
  2) fullAmount = baseAmount + 25;

Legal!

  - Money has a constructor with one argument of type int , and so the value 25 will be automatically converted to a value of type Money

Automatic Type Conversion

- What if
  1) Money baseAmount(100, 60), fullAmount;
  2) fullAmount = 25 + baseAmount ;
- 25 cannot be a calling object

illegal!

  - Conversion of int values to type Money works for arguments but **not** for calling objects

Overloading an operator as a nonmember gives you **automatic type conversion** of all arguments

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

Overloading an operator as a nonmember gives you automatic type conversion of all arguments

- There is a way to overload an operator that offers both of advantages:
  - Efficiency
  - Auto type conversion

- YES, overloading as a friend function

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!

```
1) class Money
2) {
3) public:
4)     Money( );
5)     Money(double amount);
6)     …
7)     void output( ) const;
8)     friend const Money operator +(const Money& amount1,
                                      const Money& amount2);
9)     friend const Money operator -(const Money& amount1,
                                      const Money& amount2);
10)    friend bool operator ==(const Money& amount1,
                               const Money& amount2);
11)    friend const Money operator -(const Money& amount);
12)private:
13)    int dollars;
14)    int cents;
15)   …
16)    int round(double number) const;
17)};
```

```
1)  int main( )
2)  {
3)      Money yourAmount, myAmount(10, 9);
4)       …
5)      if (yourAmount == myAmount)
6)          cout << "We have the same amounts.\n";
7)      else
8)          cout << "One of us is richer.\n";

9)      Money ourAmount = yourAmount + myAmount;
10)     yourAmount.output( ); cout << " + "; myAmount.output( );
11)     cout << " equals "; ourAmount.output( ); cout << endl;

12)     Money diffAmount = yourAmount - myAmount;
13)     yourAmount.output( ); cout << " - "; myAmount.output( );
14)     cout << " equals "; diffAmount.output( ); cout << endl;

15)     return 0;
16) }
```

```
1)  const Money operator +(const Money& amount1,
                           const Money& amount2)
2)  {
3)      int allCents1 = amount1.cents + amount1.dollars*100;
4)      int allCents2 = amount2.cents + amount2.dollars*100;
5)      int sumAllCents = allCents1 + allCents2;
6)      int absAllCents = abs(sumAllCents); //Money can be negative.
7)      int finalDollars = absAllCents/100;
8)      int finalCents = absAllCents%100;

9)      if (sumAllCents < 0)
10)     {
11)         finalDollars = -finalDollars;
12)         finalCents = -finalCents;
13)     }

14)     return Money(finalDollars, finalCents);
15) }
```

Compare to overloaded as
(1) member functions, p24
(2) functions, p10

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

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

```
1)  const Money operator +(const Money& amount1, const Money& amount2)
2)  {
3)     int allCents1 = amount1.getCents( ) + amount1.getDollars( )*100;
4)     int allCents2 = amount2.getCents( ) + amount2.getDollars( )*100;
5)     int sumAllC
6)     int absAllCents = abs(sumAllCents); //Money can be negative
7)     int finalDol
8)     int finalCe
9)     if (sumAllC
10)    {
11)       finalDoll
12)       finalCen
13)    }
14)    return Mon
15) }
```

const Money operator +(const Money& amount2) const;

```
1)  const Money Money::operator +(const Money& secondOperand) const
2)  {
3)     int allCents1 = cents + dollars*100;
4)     int allCents2 = secondOperand.cents + secon
5)     int sumAllCents = allCents1 + allCents2;
6)     int absAllCents = abs(sumAllCents); //Money
7)     int finalDollars = absAllCents/100;
8)     int finalCents = absAllCents%100;

9)     if (sumAllCents < 0)
10)    {
11)       finalDollars = -finalDollars;
12)       finalCents = -finalCents;
13)    }
14)    return Money(finalDollars, finalCents);
15) }
```

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

```
1)  const Money operator +(const Money& amount1, const Money& amount2)
2)  {
3)     int allCents1 = amount1.cents + amount1.dollars*100;
4)     int allCents2 = amount2.cents + amount2.dollars*100;
5)     int sumAllCents = allCents1 + allCents2;
6)     int absAllCents = abs(sumAllCents); //Money can be negative.
7)     int finalDollars = absAllCents/100;
8)     int finalCents = absAllCents%100;

9)     if (sumAllCents < 0)
10)    {
11)       finalDollars = -finalDollars;
12)       finalCents = -finalCents;
13)    }
14)    return Money(finalDollars, finalCents);
15) }
```

Friend Classes

See Chapter 17 for detail

- 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;  // as contrast to (x+5)
}
```

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

Example of returning a reference

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

1) double m = 99;
2) cout << sampleFunction(m) << endl;
3) **sampleFunction(m) = 42;**
4) cout << m << endl;

L-Values and R-Values

The term **l-value** is used for something that can appear on the left-hand side of an assignment operator. The term **r-value** is used for something that can appear on the right-hand side of an assignment operator.

If you want the object returned by a function to be an l-value, it must be returned by reference.

output 99 and then 42

What Mode of Returned Value to Use?

- A function can return a **value** of type **T** in four different ways:
  1. By plain old value, as in the function declaration **T** f( );
  2. By constant value, as in the function declaration const **T** f( );
  3. By **reference**, as in the function declaration **T&** f( );
  4. By const **reference**, as in the function declaration const **T&** f( );

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 >> and <<

- 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 >> and <<

- 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

Overload operator << and >>

```
1)  class Money
2)  {
3)  public:
4)      Money( );
5)      …
6)      friend const Money operator +(const Money& amount1, const Money& amount2);
7)      friend const Money operator -(const Money& amount1, const Money& amount2);
8)      friend bool operator ==(const Money& amount1, const Money& amount2);
9)      friend const Money operator -(const Money& amount);
10)     friend ostream& operator <<(ostream& outputStream, const Money& amount);
11)     friend istream& operator >>(istream& inputStream, Money& amount);
12) private:
13)     int dollars; //A negative amount is represented as negative dollars and
14)     int cents; //negative cents. Negative $4.50 is represented as -4 and -50

15)     int dollarsPart(double amount) const;
16)     int centsPart(double amount) const;
17)     int round(double number) const;
18) };
```

```
1)  ostream& operator <<(ostream& outputStream, const Money& amount)
2)  {
3)      int absDollars = abs(amount.dollars);
4)      int absCents = abs(amount.cents);
5)      if (amount.dollars < 0 || amount.cents < 0)
6)          //accounts for dollars == 0 or cents == 0
7)          outputStream << "$-";
8)      else
9)          outputStream << '$';
10)     outputStream << absDollars;

11)     if (absCents >= 10)
12)         outputStream << '.' << absCents;
13)     else
14)         outputStream << '.' << '0' << absCents;

15)     return outputStream;
16) }
```

```
1)  istream& operator >>(istream& inputStream, Money& amount)
2)  {
3)      char dollarSign;
4)      inputStream >> dollarSign; //hopefully
5)      if (dollarSign != '$')    {
6)          cout << "No dollar sign in Money input.\n";
7)          exit(1);
8)      }

9)      double amountAsDouble;
10)     inputStream >> amountAsDouble;
11)     amount.dollars = amount.dollarsPart(amountAsDouble);
12)     amount.cents = amount.centsPart(amountAsDouble);

13)     return inputStream;
14) }
```

```
1)  int main( )
2)  {
3)      Money yourAmount, myAmount(10, 9);
4)      cout << "Enter an amount of money: ";
5)      cin >> yourAmount;
6)      cout << "Your amount is " <<
                yourAmount << endl;
7)      cout << "My amount is " << myAmount << endl;

8)      if (yourAmount == myAmount)
9)          cout << "We have the same amounts.\n";
10)     else
11)         cout << "One of us is richer.\n";

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

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

16)     return 0;
17) }
```

```
cout << "Enter an amount of money: ";
yourAmount.input( );

cout << "Your amount is ";
yourAmount.output( );
```

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.

Can we overload >> and << as member operators?

P. 337, CALLING OBJECT!

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

Increment and Decrement (Contd.)

- The increment and decrement operator on simple types, such as int and char , return
  - by **reference** in the **prefix** form and
  - by **value** in the **postfix** form.
- Simply return **by value** for all versions of the increment and decrement operators.

```
1)  class IntPair
2)  {
3)  public:
4)      IntPair(int firstValue, int secondValue);
5)      IntPair operator++( ); //Prefix version
6)      IntPair operator++(int); //Postfix version
7)      void setFirst(int newValue);
8)      void setSecond(int newValue);
9)      int getFirst( ) const;
10)     int getSecond( ) const;
11) private:
12)     int first;
13)     int second;
14) };
```

```
1)  int main( )
2)  {
3)      IntPair a(1,2);
4)      cout << "Postfix a++: Start value of object a: ";
5)      cout << a.getFirst( ) << " " << a.getSecond( ) << endl;
6)      IntPair b = a++;
7)      cout << "Value returned: ";
8)      cout << b.getFirst( ) << " " << b.getSecond( ) << endl;
9)      cout << "Changed object: ";
10)     cout << a.getFirst( ) << " " << a.getSecond( ) << endl;

11)     a = IntPair(1, 2);
12)     cout << "Prefix ++a: Start value of object a: ";
13)     cout << a.getFirst( ) << " " << a.getSecond( ) << endl;
14)     IntPair c = ++a;
15)     cout << "Value returned: ";
16)     cout << c.getFirst( ) << " " << c.getSecond( ) << endl;
17)     cout << "Changed object: ";
18)     cout << a.getFirst( ) << " " << a.getSecond( ) << endl;
19)     return 0;
20) }
```

```
1) IntPair IntPair:: operator++( ) //Prefix version
2) {
3)         first++;
4)         second++;
5)         return IntPair(first, second);
6) }
```

```
1) IntPair IntPair:: operator++(int ignoreMe)
   //Postfix version
2) {
3)         int temp1 = first;
4)         int temp2 = second;
5)         first++;
6)         second++;
7)         return IntPair(temp1, temp2);
8) }
```

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**!

```
1)  class CharPair
2)  {
3)  public:
4)      CharPair( ){/*Body intentionally empty*/}
5)      CharPair(char firstValue, char secondValue)
6)                      : first(firstValue), second(secondValue)
7)      {/*Body intentionally empty*/}

8)      char& operator[](int index);
9)  private:
10)     char first;
11)     char second;
12) };
```

```
1)  char& CharPair::operator[](int index)
2)  {
3)      if (index == 1)        return first;
4)      else if (index == 2)        return second;
5)      else
6)      {
7)          cout << "Illegal index value.\n";
8)          exit(1);
9)      }
10) }
```

```
1)  int main( )
2)  {
3)      CharPair a;
4)      a[1] = 'A';
5)      a[2] = 'B';
6)      cout << "a[1] and a[2] are:\n";
7)      cout << a[1] << a[2] << endl;

8)      cout << "Enter two letters (no spaces):\n";
9)      cin >> a[1] >> a[2];
10)     cout << "You entered:\n";
11)     cout << a[1] << a[2] << endl;

12)     return 0;
13) }
```

So, must return a reference

Rules on Overloading Operators

- When overloading an operator, at least one parameter (one operand) of the resulting overloaded operator must be of a class type.
- Most operators can be overloaded as a member of the class, a friend of the class, or a nonmember, nonfriend.
- The following operators can only be overloaded as (nonstatic) members of the class: `=`, `[]`, `->`, and `()`.
- You cannot create a new operator. All you can do is overload existing operators such as `+`, `-`, `*`, `/`, `%`, and so forth.
- You cannot change the number of arguments that an operator takes. For example, you cannot change `%` from a binary to a unary operator when you overload `%`; you cannot change `++` from a unary to a binary operator when you overload it.
- You cannot change the precedence of an operator. An overloaded operator has the same precedence as the ordinary version of the operator. For example, `x*y + z` always means `(x*y) + z`, even if `x`, `y`, and `z` are objects and the operators `+` and `*` have been overloaded for the appropriate classes.
- The following operators cannot be overloaded: the dot operator (.), the scope resolution operator (`::`), `sizeof`, `?:`, and the operator `.*`, which is not discussed in this book.
- An overloaded operator cannot have default arguments.

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

Reference declaration

- A reference is required to be initialized to refer to a valid object or function
  - Because references are not objects, there are
    - no arrays of references,
    - no pointers to references, and
    - no references to references:

```
int& a[3]; // error
int&* p;   // error
int& &r;   // error
```

- Lvalue references
  - Lvalue references can be used to alias an existing object

```
1)  #include <iostream>
2)  #include <string>
3)
4)  int main()
5)  {
6)      std::string s = "Ex";
7)      std::string& r1 = s;
8)      const std::string& r2 = s;
9)
10)     r1 += "ample";          // modifies s
11) //  r2 += "!";    // error: cannot modify through reference to const
12)     std::cout << r2 << '\n'; // prints s, which now holds "Example"
13) }
```

- Rvalue references
- Rvalue references can be used to extend the lifetimes of temporary objects

```
1)  #include <iostream>
2)  #include <string>
3)
4)  int main()
5)  {
6)      std::string s1 = "Test";
7)  //  std::string&& r1 = s1;          // error: can't bind to lvalue
8)
9)      const std::string& r2 = s1 + s1; // okay: lvalue reference to const extends lifetime
10)     //  r2 += "Test";               // error: can't modify through reference to const
11)
12)     std::string&& r3 = s1 + s1;     // okay: rvalue reference extends lifetime
13)     r3 += "Test";                   // okay: can modify through reference to non-const
14)     std::cout << r3 << '\n';
15) }
```

- when a function has both rvalue reference and lvalue reference overloads, the rvalue reference overload binds to rvalues (including both prvalues and xvalues), while the lvalue reference overload binds to lvalues:

```
1)  void f(int& x) {
2)      std::cout << "lvalue reference overload f(" << x << ")\n";
3)  }
4)  void f(const int& x) {
5)      std::cout << "lvalue reference to const overload f(" << x << ")\n";
6)  }
7)  void f(int&& x) {
8)      std::cout << "rvalue reference overload f(" << x << ")\n";
9)  }
10)
11) int main() {
12)     int i = 1;
13)     const int ci = 2;
14)     f(i);  // calls f(int&)
15)     f(ci); // calls f(const int&)
16)     f(3);  // calls f(int&&)
17)            // would call f(const int&) if f(int&&) overload wasn't provided
18)     f(std::move(i)); // calls f(int&&)
19)
20)     // rvalue reference variables are lvalues when used in expressions
21)     int&& x = 1;
22)     f(x);            // calls f(int& x)
23)     f(std::move(x)); // calls f(int&& x)
24) }
```

reference initialization

- http://en.cppreference.com/w/cpp/language/reference_initialization

A reference to T can be initialized with an object of type T, a function of type T, or an object implicitly convertible to T. Once initialized, a reference cannot be changed to refer to another object

1) When a named lvalue reference variable is declared with an initializer

- T & ref = object ;
- T & ref = { arg1, arg2, ... };
- T & ref (object) ;
- T & ref { arg1, arg2, ... } ;

2) When a named rvalue reference variable is declared with an initializer

- T && ref = object ;
- T && ref = { arg1, arg2, ... };

- T && ref (object) ;

- T && ref { arg1, arg2, ... } ;

3) In a function call expression, when the function parameter has reference type

- given R fn (T & arg); or R fn (T && arg);
- fn (object)

- fn ({ arg1, arg2, ... })

4) In the return statement, when the function returns a reference type

- given T & fn () { or T && fn () {
- return object ;

5) When a non-static data member of reference type is initialized using a member initializer

- Class::Class(...) : refmember(expr) {...}

```
#include <utility>
#include <sstream>
struct S {
 int mi;
 const std::pair<int,int>& mp; // reference member
};

void foo(int) {}

struct A {};
struct B : A {
  int n;
  operator int&() { return n; };
};

B bar() {return B(); }

//int& bad_r; // error: no initializer
extern int& ext_r; // OK

int main()
{
 // lvalues
    int n = 1;
    int& r1 = n;  // lvalue reference to the object n
    const int& cr(n); // reference can be more cv-qualified
    volatile int& cv{n}; // any initializer syntax can be used
    int& r2 = r1; // another lvalue reference to the object n
//    int& bad = cr; // error: less cv-qualified
    int& r3 = const_cast<int&>(cr); // const_cast is needed

    void (&rf)(int) = foo; // lvalue reference to function
    int ar[3];
    int (&ra)[3] = ar; // lvalue reference to array

    B b;
    A& base_ref = b; // reference to base subobject
    int& converted_ref = b; // reference to the result of a conversion

// rvalues
//  int& bad = 1; // error: cannot bind lvalue ref to rvalue
    const int& cref = 1; // bound to rvalue
    int&& rref = 1; // bound to rvalue

    const A& cref2 = bar(); // reference to A subobject of B temporary
    A&& rref2 = bar();      // same

    int&& xref = static_cast<int&&>(n); // bind directly to n
//  int&& copy_ref = n; // error: can't bind to an lvalue
    double&& copy_ref = n; // bind to an rvalue temporary with value 1.0

// restrictions on temporary lifetimes
    std::ostream& buf_ref = std::ostringstream() << 'a'; // the ostringstream temporary
                    // was bound to the left operand of operator<<, but its lifetime
                    // ended at the semicolon: buf_ref is now a dangling reference.

    S a { 1, {2,3} }; // temporary pair {2,3} bound to the reference member
                    // a.mp and its lifetime is extended to match a
                    // (Note: does not compile in C++17)
    S* p = new S{ 1, {2,3} }; // temporary pair {2,3} bound to the reference
                    // member p->mp, but its lifetime ended at the semicolon
                    //  p->mp is a dangling reference
    delete p;
}
```