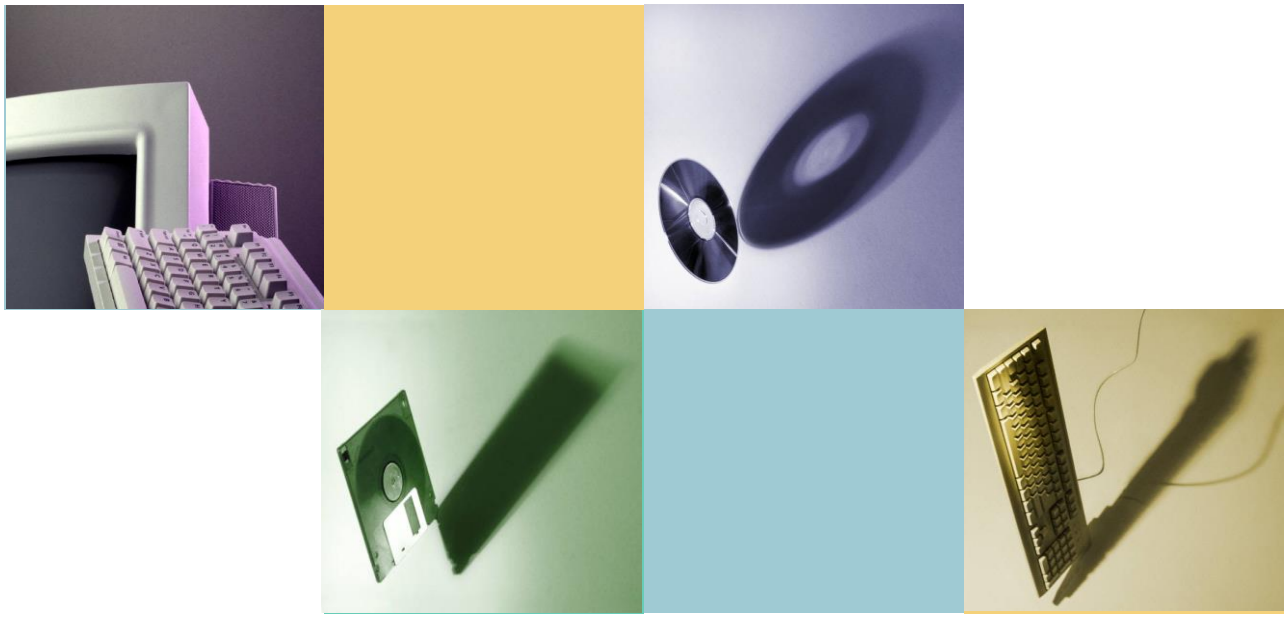
# Object-Oriented Programming



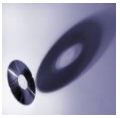
**Chuan-Kang Ting**

Dept of Computer Science and Information Engineering
National Chung Cheng University

Chapter 8

# Operator Overloading, Friends, and References

– Tools to use when defining classes

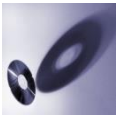




# Outline

- **Basic Operator Overloading**
- **Friend Functions and Automatic Type Conversion**
- **References and More Overloaded Operators**

# Outline

- **Basic Operator Overloading**
  - As nonmember functions
  - As member functions
  - Using friend functions
- **Friend Functions and Automatic Type Conversion**
- **References and More Overloaded Operators**


# Basics

- **Operators? Functions?**
  - The + operator
    - +(x, 7)
    - add(x, 7)
    - x + 7

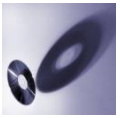
syntactic sugar

more intuitive and comfortable

**Operator** (function)

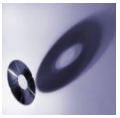
**Operands** (arguments)

→ **Operators are functions with different syntax**

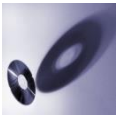

# Perspective

- **Built-in operators**
  - Such as +, -, = , %, ==,  /, *
  - Already work for C++ built-in types
- **We can overload them!**
  - Customization
    - To work with OUR types!
    - As appropriate for our needs
    - In "notation" we're comfortable with

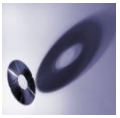

# Outline

- **Basic Operator Overloading**
  - As nonmember functions
  - As member functions
  - Using friend functions
- **Friend Functions and Automatic Type Conversion**
- **References and More Overloaded Operators**


# Outline

- **Basic Operator Overloading**
  - As nonmember functions
  - As member functions
  - Using friend functions
- **Friend Functions and Automatic Type Conversion**
- **References and More Overloaded Operators**


# Overloading as Nonmember Function

- **General Rule**
  - Operator symbol represents the function name
  - Similar to function overloading

z = **add**(x, y)

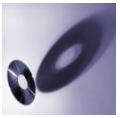
```
int    function add(int x, int y);
```

↓

```
Money function add(Money x, Money y);
```

↓

```
const Money function add(const Money& x,
                         const Money& y);
```

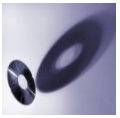




# Overloading as Nonmember Function (1)

- **An example**

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

  - Keyword operator with symbol + (function name)
  - Nonmember overloading:
    - The overloaded operators are NOT a member operators
    - Two operands
      - Restriction: At least one is a class type
      - Use constant reference parameter for **efficiency**
  - Returns a value of type Money
    - Allow addition of “money” objects


# Overloading as Nonmember Function (2)

```
#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;
    void input( ); //Reads the dollar sign as well as the amount number.
    void output( ) const;
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.

    int dollarsPart(double amount) const;
    int centsPart(double amount) const;
    int round(double number) const;
};

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

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

Nonmember!

# Overloading as Nonmember Function (3)

- **Money "+" operator**
  - Note: overloaded + is NOT a member operator
  - Definition is "more involved" than simple "add"

```
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);
}
```

need to use accessor and mutator functions → Not member fn.

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



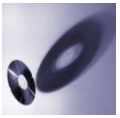
# Overloading as Nonmember Function (4)

- **Money "==" operator**
  - Compare two Money objects
  - Returns a bool type as truth value
  - Again: overloaded == is NOT a member operator

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

need to use accessor and mutator functions → Not member fn.

return truth value (true/false)


# A Constructor Can Return an Object

- **A constructor is viewed**
  - Mostly, as if a void function
  - Sometimes, as returning a value

    → anonymous object

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

```
return int(3);
```

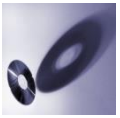
is equivalent to

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

```
int temp;
temp = 3;     //int(3)
return temp;
```

  - Full-fledged

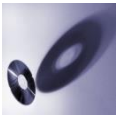
```
Money(finalDollars, finalCents).getDolloars();     //finalDollars
```


# const

## To be or not to be?

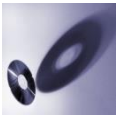
```
const Money operator +(const Money& amount1,
                       const Money& amount2);
```


# Returning by `const` Value

- **Consider (m1 + m2)**
  - m1 and m2 are Money objects
  - Returned object is Money object
  - We can do "anything" with the returned object
- **The cases without `const`**
  - m1, m2, and (m1+m2) are of type Money
    - `(m1 + m2).output();` ← legal and make sense
    - `(m1 + m2).input();` ← legal but make no sense
      - Modifying anonymous object?!
        → Disallowed
  - So we define the returned objects as const for "read-only"

```
return Money(fDollars,
             fCents);
```

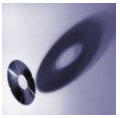

# Returning by `const` Value (cont'd)

- **Returning by `const` value**
  - The **returned** object CANNOT be changed

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

As a result,

- `(m1 + m2).output();` ← legal
- `(m1 + m2).input();` ← **illegal**
  - Disallow modification of anonymous object

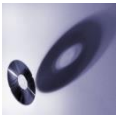




# More about Assignment

- **Assignment**

```
m3 = (m1 + m2);
m3.input();
```

  - m3 and (m1+m2) are different objects
  - The *default* assignment operator does
    - Not make the two objects the same object
    - Copy values of member variables from one object to another
      - **Member-by-member** copy (recall: copy of vectors)
    - We will further discuss it later on

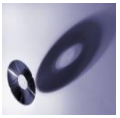




# Overloading Unary Operators

- **Unary operator**
  - An operator that takes only one operand (one argument)
  - Binary operator: two operands
  - Example: The "–" operator
    - Subtraction: binary, e.g. m1 – m2
    - Negation: unary, e.g. –m2

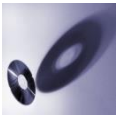
    → Overload twice

```
const Money operator -(const Money& amount1,
                       const Money& amount2);
const Money operator -(const Money& amount);
```


# Outline

- **Basic Operator Overloading**
  - As nonmember functions
  - As member functions
  - Using friend functions
- **Friend Functions and Automatic Type Conversion**
- **References and More Overloaded Operators**

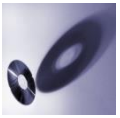

# Overloading as Member Functions (1)

- **Overloading Operators as**
  - Nonmember functions
    - Standalone functions (defined *outside* a class)
    - Two operands, two parameters
  - Member functions
    - Just as member functions (defined *within* a class)
    - Two operands, **one** parameter
    - **Calling object:** 1st parameter

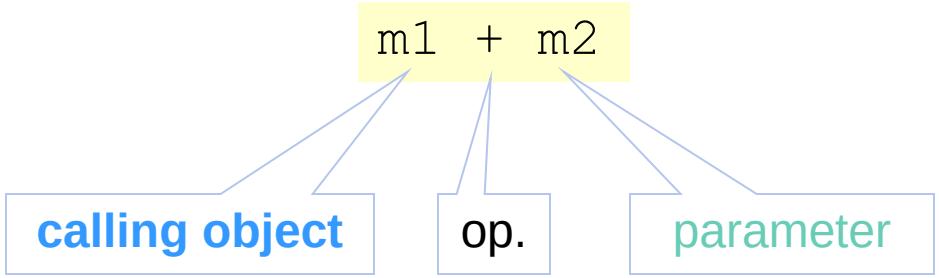
Member of what?

```
m1 + m2
```

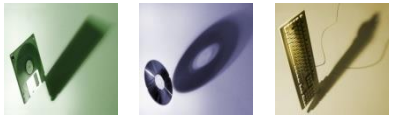
**calling object** | op. | parameter


# Overloading as Member Functions (2)



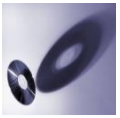

```
class Money
{
  public:
    ...
          Money operator +(      Money& amount2);
          Money operator -(      Money& amount2);
          Money operator -();
    bool operator ==(const Money& amount2);
  private:
    ...
};
```



# const

## To be or not to be?

```
class Money
{
  public:
    ...
    const Money operator +(const Money& amount2) const;
    const Money operator -(const Money& amount2) const;
    const Money operator -() const;
    bool operator ==(const Money& amount2) const;
  private:
    ...
};
```

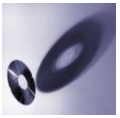




# `const` Returned Values & Functions

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

- **`const` returned values**
  - The returned object CANNOT be changed
- **`const` functions**
  - When to make function **`const`**?
    - Constant functions disallowed to alter class member data
      - Protect calling objects
    - Constant objects can ONLY call constant member functions
      - Cascading protection
  - Good style
    - Add the **`const`** *whenever* the operator invocation does not change the calling object (the 1st operand)

also member function


# Nonmember vs. Member

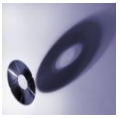
```
m3 = m1 + m2
```

- **As nonmember function**

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

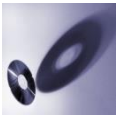
- **As member function**

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

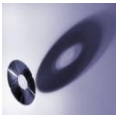

# All You Can Overload ?

- **You can, but do better NOT to, overload**
  - Function application `()`
  - And `&&`
  - Or `||`
  - Comma `,`
- **You CANNOT overload**
  - Dot operator `.`
  - Scope resolution operator `::`
  - `sizeof`
  - `?:`


# Outline

- Basic Operator Overloading
- Friend Functions and
  **Automatic Type Conversion**
- References and More Overloaded Operators


# Automatic Type Conversion (1)

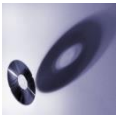
- **The case**

```
Money baseAmount(100, 60), fullAmount;
fullAmount = baseAmount + 25;
fullAmount.output();                //125.60
```

```
class Money
{
  public:
    Money();
    Money(int dollars);
    Money(int dollars,
          int cents);
    void output const;
    ...
  private:
    int dollars;
    int cents;
    ...
};
```

  - The system does
    - check if operator "+" has been overloaded for **`Money + int`**
      (No, it lacks such overloading)
    - check if there is a constructor that takes a single argument of type `int`
      (Bingo! `Money(25)` makes `int` → `Money`)

Automatic type conversion


# Automatic Type Conversion (2)

- **However, the case**

```
fullAmount = 25 + baseAmount;
```

Overloading + as **nonmember** operator

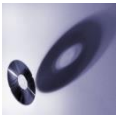
```
const Money operator +(const Money& amount1,
                       const Money& amount2);
```

→ Works well to deal with `int + Money`

Overloading + as **member** operator

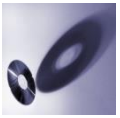
```
const Money operator +(const Money& amount2) const;
```

→ The expression `25 + baseAmount` is illegal because 25 cannot be a calling object

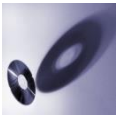

# Outline

- **Basic Operator Overloading**
  - As nonmember functions
  - As member functions
  - Using friend functions
- **Friend Functions and Automatic Type Conversion**
- **References and More Overloaded Operators**


# Friend Functions (1)

- **Nonmember functions**
  - Cannot access member variables directly
  - Through accessor and mutator functions
    → Inefficient
- **Friend functions**
  - Are not member functions
  - Have access to the private members (variables and functions)
  - No overhead, more efficient!
  - Ideal to overload operators as **friend** functions (Why?)


# Friend Functions (2)

- **Use friend functions**
  - Place **`friend`** in front of function declaration
    - Specified in class definition
      - Do not place `friend` in function definition
    - Friend functions are NOT member functions
      - Do not use the dot operator to call a friend function
      - Do not use a type qualifier (::) in the function definition

**Friend:**
a guy who can access your private matters

```
class Money
{
  public:
    ...
    friend const Money operator +(const Money& m1, const Money& m2);
    ...
};
...
const Money operator +(const Money& m1, const Money& m2)
{
  int allCents1 = m1.cents + m1.dollars*100;
  ...
}
```

NOT a member function (no **::** in fn. definition)

Directly access member variables of Money

# Friend Functions (3)

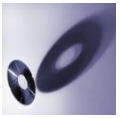
- **Friends not pure?**
  - The true spirit of OOP: *All operators and functions should be member functions*
    - Friend functions are NOT member functions
    - Many believe friends violate basic OOP principles
  - However, the friend operator declaration is inside the class definition
    - Provides at least a bit more encapsulation than nonmember, non-friend operators

# Friend Classes

- **A class can be a friend of another class**
  - Similar to function being friend to class
  - e.g.
    class F is friend of class C
    - All class F member functions are friends of C
    - NOT reciprocated
    - Friendship granted (by C), not taken (by F)
      - The one (F) who got friendship can directly access the members (of C)

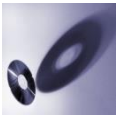
forward declaration

```
class F;

class C
{
  public:
    ...
    friend class F;
    ...
};

class F
{
  ...
```


# Outline

- Basic Operator Overloading
- Friend Functions and
  Automatic Type Conversion
- **References** and More Overloaded Operators


# References (1)

- **Reference**
  - The name of a storage location
  - Essentially an **alias** for the variable

- **Standalone reference**

```
int robert;
int& bob = robert;
```

  - *bob* is a reference to the storage location for *robert*
  - Makes *bob* an alias for *robert*
    - Changes made to *bob* will affect *robert*

  → Confusing and useless

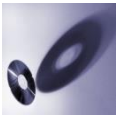
# References (2)

- **Use of references**
  - Call-by-reference
    - Parameter as an alias of argument
  - Returning a reference
    - Can be viewed as returning an alias to a variable
    - e.g.

```
double& sampleFunction(double& var);
```

      - double& is a **different** type from double
      - Must use the & in both function declaration and heading
    - Returned value must have a "reference"
      - Cannot be a expression, e.g. (x+5)
      - Cannot be a local variable

      → Both of the above have no place in memory to "refer to"

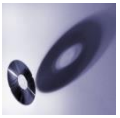

# L-Values and R-Values (1)

```
m1 = m2;
```

L-value

R-value

- **L-value**
  - Something that can appear on the left-hand side of an assignment operator
- **R-value**
  - Something that can appear on the right-hand side of an assignment operator


# L-Values and R-Values (2)

- **General rule**
  - If you want the returned object to be an ***l-value***, it must be returned by reference
  - Why?

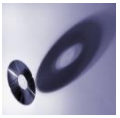
```
m1 = m2;
```

| Variable | L-value | | R-value | Value or Variable |
|---|---|---|---|---|

  - Example

```
 x = 5;
 y = x;
15 = y;     //error
15 = 2;     //error
```

**L-value** must be a **variable**
→ return a reference (alias) to a variable


# Returning a Reference

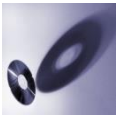
- **A trivial example**
  - To show the concept

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

```
double m = 99;
cout << sampleFunction(m) << endl;      //99
sampleFunction(m) = 42;
cout << m << endl;                      //42
```

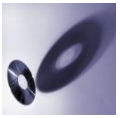
What will happen if NOT returning a reference?

**L-value**
→ return a reference


# Outline

- Basic Operator Overloading
- Friend Functions and Automatic Type Conversion
- References and **More Overloaded Operators**

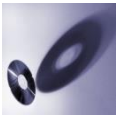

# Overloading << and >>

- **Enables input and output of our objects**
  - Similar to other operator overloads
  - New subtleties
- **Improve readability**
  - Which style is preferable?

```
Money amount(100);
cout << "I have ";
amount.output();
cout << " in my purse. \n";
```

OR

```
Money amount(100);
cout << "I have " << amount << " in my purse. \n";
```

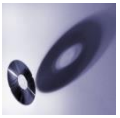




# Overloading << (1)

- **Insertion operator <<**
  - Used with `cout`
  - Binary operator
- **Example:**

```
cout << "Hello";
```

  - Operator is **<<**
  - 1st operand is predefined object `cout`
    - From library iostream
    - Class type *ostream*
  - 2nd operand is literal string "Hello"


# Overloading << (2)

- **What should be returned?**

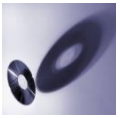
```
cout << amount
```

  – Considering the following case

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

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

```
cout << "I have "
```


# Overloading << (3)

- **<< returns a stream**
  - Type ostream for cout
  - Therefore,

```
friend ostream& operator <<(ostream& outs,
                           const Money& amount);
```

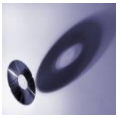
# Example (1)

**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.
```

# Example (2)

```
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;
```


# Example (3)

**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.*

# Example (4)

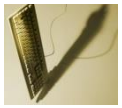
```
    if (absCents >= 10)
        outputStream << '.' << absCents;
    else
        outputStream << '.' << '0' << absCents;

    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`*.*





# Example (5)

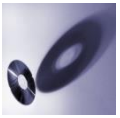
**Display 8.5 Overloading << and >>**

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

    return inputStream;
}
```

*Returns a reference*

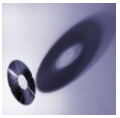
**SAMPLE DIALOGUE**

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 =

- **Overloading the assignment operator**
  - Must overload it as a member operator
  - If you don't overload it, then you get one automatically
- **Default assignment operator**
  - Automatically overloaded by system
  - Do member-wise copy:
    - Member variables from one object → corresponding member variables from other
  - Works well with simple classes
    - but not for those using pointers
      - The effect of member-wise copy depends on the base types of member variables


# Increment ++ and Decrement –– (1)

- **Each operator has two versions**
  - Prefix notation: **++**`x`
  - Postfix notation: `x`**++**
- **When overloading, you must distinguish**
  - **Prefix** ← Regular overloading
    - Nonmember function with *one* parameter
    - Member function with *no* parameters
  - **Postfix** ← Add a *2nd parameter* of type `int`
    - Just a marker for the compiler
- **Simply return by value**

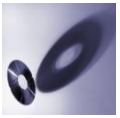
# Increment ++ and Decrement –– (2)

- **Example**

```
class Money
{
  public:
    Money(int dollars, int cents);
    ...
    Money operator++();     //Prefix
    Money operator++(int); //Postfix
  private:
    int dollars;
    int cents;
}
...
```

```
Money Money::operator++()
{
  dollars++;
  return Money(dollars,
               cents);
}
```

```
Money Money::operator++(int)
{
  int temp = dollars;
  dollars++;
  return Money(temp,
               cents);
}
```


# Array Operator `[]` (1)

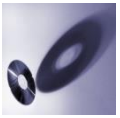
- **Overloading array operator**
  - Must be a member function
  - Should support *l-value*
    - → Return by **reference**
  - e.g. **`a[2]`**
    - a is the calling object
    - 2 is the argument

# Array Operator [] (2)

- **Example**

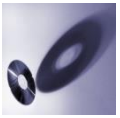
```
class Money
{
  public:
    ...
    int& operator[](int index);
  ...
}

int main()
{
  Money a;        //$100.16
  a[1] = 100;
  a[2] = 16;
  ...
```

```
int& Money::operator[](int index)
{
  if (index == 1)
    return dollars;
  else if (index == 2)
    return cents;
  else {
    cout << "Illegal index \n";
    exit(1);
  }
}
```

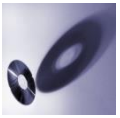

# What Mode of Returned Value to Use?

- **Four ways**
  - By plain value, as in `T f( );`
  - By constant value, as in `const T f( );`
  - By reference, as in `T& f( );`
  - By const reference, as in `const T& f( );`

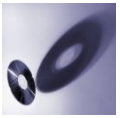

# What Mode of Returned Value to Use?

- **General rules**
  - Simple type (int, char, …)
    - Typically NOT use **const**
    - Allow an l-value → return by **reference**
    - Otherwise → return by **plain** value
  - **Class** type
    - Allow a l-value → return by **reference**
    - const T vs. const T&
      - Very similar
      - Both disallow modifying the returned object by some mutator function, e.g. f( ).mutator( );
      - Both can be copied to another variable with assignment and that variable can be modified
      - Just use **const T**, if you cannot decide between them


# Summary (1)

- **Operators are really just functions**
- **Overloading operators**
  - As nonmember functions
    - Two operands (arguments)
    - Supports automatic type conversion of all arguments
  - As member functions
    - 1st operand: calling object
    - 2nd operand: argument
    - More efficient but cannot converse the 1st operand
  - Using friend functions
    - Have access to private members → Efficient
    - Two operands → Supports automatic type conversion of all arguments


# Summary (2)

- **Reference**
  - An alias for the variable
  - `const` → Make it "read-only"
  - **L-value** vs. R-value
    - If you want the returned object to be an *l-value*, it must be returned by **reference**
- **When overloading operators**
  - Just as overloading functions
  - As nonmember / member / friends?
  - To be or not to be const?
  - Return by reference? L-value?
  - Special care to `<<`, `>>`, `=`, `++`, `--`, `[]`