CS 31: Introduction To Computer Science I

Howard A. Stahl

Agenda

- Class Example : File Streams
- Classes and Friends
- `const` Parameters
- Operator Overloading

File Streams

- Streams Are System Classes That Handle I/O
  - require `#include <fstream>`
  - `cin` and `cout` are pre-defined streams
- Streams Let You Persist Data (Input or Output) Beyond One Program Execution
  - `ifstream` input file stream like `cin`
  - `ofstream` output file stream like `cout`
  - otherwise known as "files"

File Streams

Read Chapter 5 For More Complete Details On File Streams

ifstream
void open(string) bool fail( ) void close( ) bool eof( )

ofstream
void open(string) bool fail( ) void close( ) void precision(int) void setf(int)

setf flags include	ios::fixed ios::scientific ios::showpoint ios::showpos

Using `ifstream` Class

- `#include <fstream>`
- `ifstream in_stream;`
- `in_stream.open( “file.dat” );`
- `if (in_stream.fail()) ...`
- `in_stream >> var1 >> var2;`
- `while (in_stream >> next) ...`
- `while (!in_stream.eof()) ...`
- `in_stream.close();`

Using `ofstream` Class

- `#include <fstream>`
- `ofstream out_stream;`
- `out_stream.open(“file.dat”);`
- `if (out_stream.fail()) ...`
- `out_stream << var1 << endl;`
- `out_stream.close();`

An Important Note!

- If You Pass Streams As Parameters To Functions, You Must Always Use Pass-By-Reference Techniques
  - with the `&` operator

`<cstdlib>` System Library

- Often When Using Files, Your Code Will Want To Fail Immediately When File Errors Occur
  - without the desired input or output, what is the point of continuing execution?
- Example:

```
#include <cstdlib>
exit( 1 ); // denotes failure
```

Time For Our Second Demo!

- IO.cpp

Summarizing Our Second Demo!

- Working With DataFiles Prevents DataEntry Errors
- Always Pass Streams By Reference
  - use `&` operator with stream parameters
- The “Magic Formula” Applies To Streams As Well As To `cin` And `cout`

Revisiting Classes

Number
void setValue(int) int getValue( ) void printRomanNumeral( )
int value

```
class Number {
public:
  Number( );
  Number( int initValue );
  void setValue( int v );
  int getValue( );
  void printRomanNumeral();
private:
  int value;
};
```

Revisiting Classes

```
Number::Number() {
  value = 0;
}
Number::Number( int initValue ) {
  value = initValue;
}
int Number::getValue( ) {
  return( value );
}
void Number::setValue( int newValue ) {
  value = newValue;
}
```

Revisiting Classes

```
Number four = Number( 4 );
Number five = Number( 5 );
```

Revisiting Classes

```
Number four = Number( 4 );
Number five = Number( 5 );
```

Wouldn’t be great to...

```
Number nine = add( four, five );
```

Revisiting Classes

```
Number four = Number( 4 );
Number five = Number( 5 );


Number nine = add( four, five );


Number add( Number left, Number right ) {
  Number temp=Number(left.value+right.value);
  return( temp );
}
```

Revisiting Classes

```
Number four = Number( 4 );
Number five = Number( 5 );


Number nine = add( four, five );


Number add( Number left, Number right ) {
  Number temp=Number(left.value+right.value);
  return( temp );
}
```

Trouble Is:
It's ILLEGAL!

`friend` Functions

- `friend` Function Of A Class Is **<u>NOT</u>** A Member Function But Has Access To Private Members Of That Class
  - `friend` functions must be named inside the class definition
- `friend` Functions Are Always `public`
  - regardless of where they are placed in the class definition

`friend` Functions

```
class Number {
public:
  digit( );
  digit( int initValue );
  void setValue( int v );
  int getValue( );
  void printRomanNumeral();
  friend Number add(Number left, Number right);
private:
  int value;
};
```

`friend` Functions

```
class Number {
public:
  ...
  friend Number add(Number left,
                    Number right);
  ...
}

Number add( Number left, Number right ) {
  Number t=Number( left.value + right.value );
  return( t );
}
```

`friend` Functions

```
class Number {
public:
  ...
  friend Number add(Number left,
                    Number right);
  ...
}

Number add( Number left, Number right ) {
  Number t=Number( left.value + right.value );
  return( t );
}
```

There is no :: operator

Time For Our Next Demo!

- Number.cpp

Summarizing Our Next Demo!

- Use `friend` Functions With Care
  - defeats encapsulation
- Use Member Functions When Working With Only One Object Instance
- Use `friend` Functions When Working With More Than One Object Instance

Revisiting `const` Modifier

- Named Constants Improve Readability

```
const int DAYS_IN_WEEK = 7;

for (int i = 0; i < DAYS_IN_WEEK; i++) {
    read_textbook_chapter();
    study();
}
```

`const` Modifier

- `const` Modifier Also Applies To Function Parameters
  - member functions or normal functions
- `const` Modifier Is Unnecessary With Call-By-Value Parameters
  - any changes made are never seen by the caller
- `const` Modifier Can Be Applied To Call-By-Reference Parameters

`const` Modifier

- Recall That Call-By-Value Results In Argument Copies
  - can be expensive when working with large object graphs
- Call-By-Reference Is Preferred When Passing Objects
- If You Know No Changes Are Made, Mark That Parameter With The `const` Modifier
  - compiler will complain if you alter its value

`const` Modifier

- `const` Modifier Can Also Apply To Member Functions
  - informs the compiler that a member function does not update the `this` pointer of the object being referenced

`const` Modifier

```
class Number {
public:
  digit( );
  digit( int initValue );
  void setValue( int v );
  int getValue( ) const;
  void printRomanNumeral() const;
  friend Number add(const Number& left,
                    const Number& right);
private:
  int value;
};
```

`const` Modifier

```
class Number {
public:
  ...
  friend Number add(const Number& left,
                    const Number& right);
  ...
}

Number add( const Number& left,
            const Number& right ) {
  Number t=Number( left.value + right.value );
  return( t );
}
```

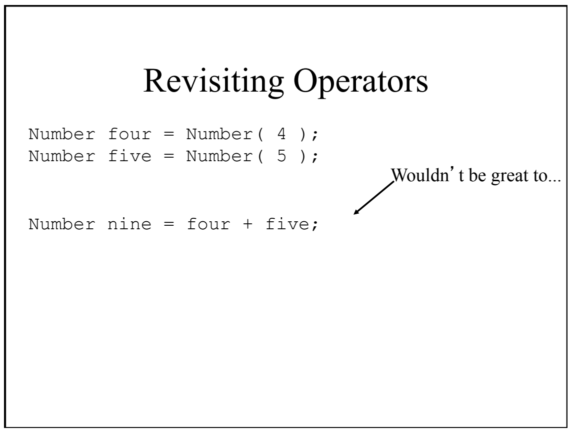
`const` Modifier

```
class Number {
public:
  ...
  friend Number add(const Number& left,
                    const Number& right);
  ...
}

Number add( const Number& left,
            const Number& right ) {
  Number t=Number( left.value + right.value );
  return( t );
}
```

const Modifier
In Both Declaration
And Definition

Understanding The Effect Of `const`

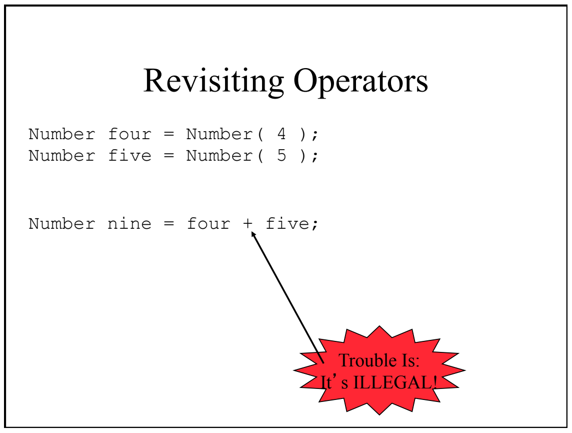
- Using `const` Modifier Is An All-Or-Nothing Proposition
- Due To Function Calls Within Functions, The Compiler Will Cascade `const` Modifier Requirements



Revisiting Operators

```
Number four = Number( 4 );
Number five = Number( 5 );


Number nine = four + five;
```

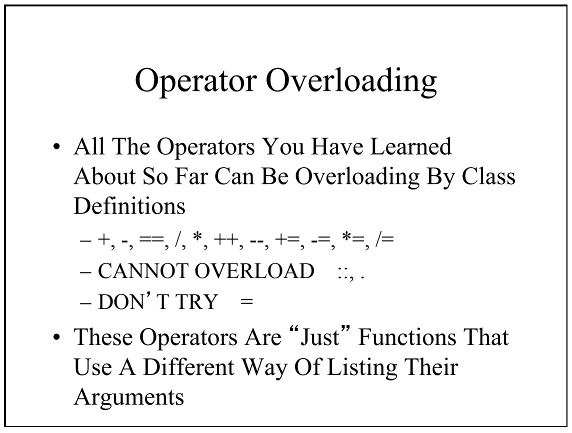
Wouldn’t be great to...



Revisiting Operators

```
Number four = Number( 4 );
Number five = Number( 5 );


Number nine = four + five;
```

Trouble Is:
It’s ILLEGAL!



Operator Overloading

- All The Operators You Have Learned About So Far Can Be Overloading By Class Definitions
  - +, -, ==, /, *, ++, --, +=, -=, *=, /=
  - CANNOT OVERLOAD ::, .
  - DON’T TRY =
- These Operators Are “Just” Functions That Use A Different Way Of Listing Their Arguments

Operator Overloading

- Operator Functions Are Typically Defined As `friend` Functions With `const` Parameter Arguments
  - be sure to use the `operator` keyword

```
friend Number operator +(const Number& left,
                         const Number& right);

friend bool operator ==(const Number& left,
                        const Number& right);
```

Operator Overloading

```
class Number {
public:
  ...
  friend Number operator +(const Number& left,
                           const Number& right);
  friend bool operator ==(const Number& left,
                          const Number& right);
  ...
}
```

Operator Overloading

```
Number operator +(const Number& left,
                  const Number& right) {
  Number t=Number( left.value + right.value );
  return( t );
}

bool operator ==(const Number& left,
                 const Number& right) {
  return( left.value == right.value );
}
```

Operator Overloading

```
Number operator +(const Number& left,
                  const Number& right) {
  Number t=Number( left.value + right.value );
  return( t );
}

bool operator ==(const Number& left,
                 const Number& right) {
  return( left.value == right.value );
}
```

operator Modifier In Both Declaration & Definition And No :: Operator

Time For Our Next Demo!

- Operators.cpp

(See Handout For Example 4)

Overloading << and >>

- The Insertion And Extraction Operators Can Also Be Overloaded By A Class Definition
- These Operators Must Be Friends

```
friend ostream& operator <<(ostream& outs,
                            const Number& n);

friend istream& operator >>(istream& ins,
                            Number& n);
```

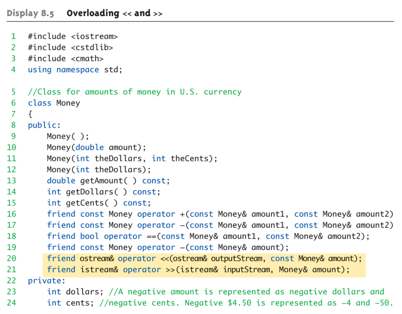
Time For Our Next Demo!

- NumberWithOperators.cpp

Summarizing The Demo!

- Overloading << And >> Let A Class' Author Determine How A Class Should Be Dumped To And From A File Stream

Textbook Example : Money



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.
```

Textbook Example : Money

```
    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;
```

Textbook Example : Money

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.

Textbook Example : Money

```
    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);
```

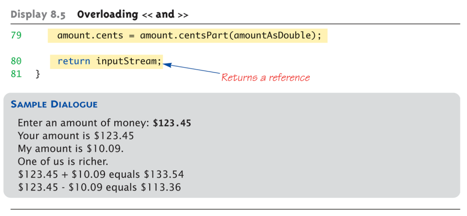
Returns a reference

*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)

Textbook Example : Money



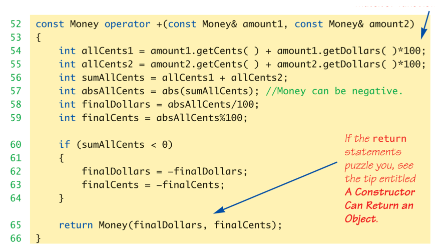
Display 8.5 Overloading << and >>

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

    return inputStream;
}
```

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
```

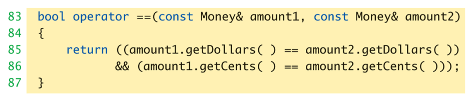
Textbook Example : Money



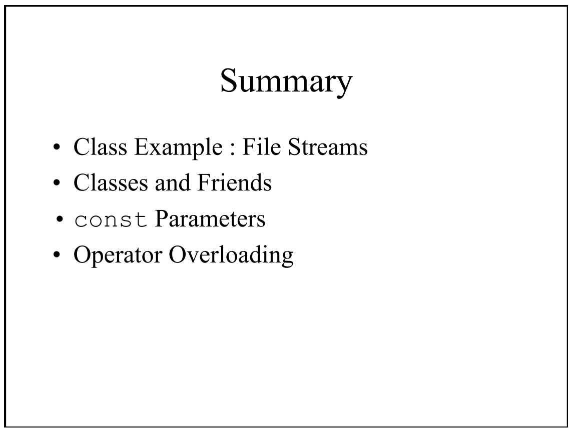
```
const Money operator +(const Money& amount1, const Money& amount2)
{
    int allCents1 = amount1.getCents( ) + amount1.getDollars( )*100;
    int allCents2 = amount2.getCents( ) + amount2.getDollars( )*100;
    int sumAllCents = allCents1 + allCents2;
    int absAllCents = abs(sumAllCents); //Money 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.***

Textbook Example : Money



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



Summary

- Class Example : File Streams
- Classes and Friends
- `const` Parameters
- Operator Overloading