

Software Design and Analysis for Engineers

by

Dr. Lesley Shannon

Email: lshannon@ensc.sfu.ca

Course Website: <http://www.ensc.sfu.ca/~lshannon/courses/ensc251>



Simon Fraser University

Slide Set: 3

Date: September 14, 2015

What we're learning in this Slide Set:

- More on Using Custom Data Types:
 - Friend Functions
- Constant Parameters
- Overloading Operators

Textbook Chapters:

Relevant to this slide set:

- Sections 11.1-11.2

Coming soon:

- End of Chapter 11
- Reminders from Chapter 5 (Sections 5.3 and 5.4)

Friend Functions

We keep talking about encapsulation and including all functions as members within classes

However, sometimes you might want to support an operation as an “ordinary” non-member function

Let's look at how we define operations on objects as nonmember functions

Recall the day of the year class

```
class DayOfYear
{
public:
    DayOfYear(int the_month, int the_day);
    //Precondition: the_month and the_day form a
    //possible date. Initializes the date according
    //to the arguments.

    DayOfYear( );
    //Initializes the date to January first.

    void input( );

    void output( );

    int get_month( );
    //Returns the month, 1 for January, 2 for February, etc.

    int get_day( );
    //Returns the day of the month.
private:
    void check_date( );
    int month;
    int day;
};
```

Let's add the following nonmember function:

```
bool equal(DayOfYear date1, DayOfYear date2);  
//Precondition: date1 and date2 have values.  
//Returns true if date1 and date2 represent the same date;  
//otherwise, returns false.
```

And update main as follows:

```
int main( )
{
    DayOfYear today, bach_birthday(3, 21);

    cout << "Enter today's date:\n";
    today.input( );
    cout << "Today's date is ";
    today.output( );

    cout << "J. S. Bach's birthday is ";
    bach_birthday.output( );

    if (equal(today, bach_birthday))
        cout << "Happy Birthday Johann Sebastian!\n";
    else
        cout << "Happy Unbirthday Johann Sebastian!\n";
    return 0;
}
```

All of the member functions stay the same, but “equal” is defined as:

```
bool equal(DayOfYear date1, DayOfYear date2)
{
    return ( date1.get_month( ) == date2.get_month( ) &&
            date1.get_day( ) == date2.get_day( ) );
}
```

And you can use it to compare to objects of class type DayofYear:

```
if (equal(today, bach_birthday))
    cout << "Happy Birthday Johann Sebastian!\n";
else
    cout << "Happy Unbirthday Johann Sebastian!\n";
return 0;
```


We could have made `equal` a member function.

However, then one of the objects would have had to call the other object as a parameter using its member function.

-this would not treat the two objects the same way.

Conversely, this implementation of `equal` is inefficient as we have to use the accessor functions to do the comparison (Why does this make it inefficient?):

```
bool equal(DayOfYear date1, DayOfYear date2)
{
    return ( date1.get_month( ) == date2.get_month( ) &&
            date1.get_day( ) == date2.get_day( ) );
}
```

What we really want to do is access or member variables directly, but this is illegal (Why?) and breaks our rules of encapsulation:

```
bool equal(DayOfYear date1, DayOfYear date2)
{
    return (date1.month == date2.month &&
            date1.day == date2.day);
}
```

A friend function of a class is not a member function of the class, but it is allowed to read and write to private member variables (similar to a member function) and use private member functions

Friend functions are listed as part of the class definition, but use the keyword `friend` in front of the declaration:

```
friend bool equal(DayOfYear date1, DayOfYear date2);  
//Precondition: date1 and date2 have values.  
//Returns true if date1 and date2 represent the same date;  
//otherwise, returns false.
```

We make them public because we want them to be usable outside of the class.

Friend functions may look like member functions (because they are part of the class definition):

```
class DayOfYear
{
public:
    friend bool equal(DayOfYear date1, DayOfYear date2);
    //Precondition: date1 and date2 have values.
    //Returns true if date1 and date2 represent the same date;
    //otherwise, returns false.

    DayOfYear(int the_month, int the_day);
    //Precondition: the_month and the_day form a
    //possible date. Initializes the date according
    //to the arguments.

    DayOfYear( );
    //Initializes the date to January first.

    void input( );

    void output( );

    int get_month( );
    //Returns the month, 1 for January, 2 for February, etc.


    int get_day( );
    //Returns the day of the month.
private:
    void check_date( );
    int month;
    int day;
};
```

However, they are really normal functions

- They are defined the same way normal functions are (the `DayOfYear::` qualifier is not included in the function heading); they just have special access to the class member variables.

```
bool equal(DayOfYear date1, DayOfYear date2)
{
    return (date1.month == date2.month &&
           date1.day == date2.day);
}
```

*Note that the private member variables **month** and **day** can be accessed by name.*



- And they are called the same way (just as we had previously shown in main):

```
main()
{
    ...
    if (equal(today, bach_birthday))
        cout << "Happy Birthday Johann Sebastian!\n";
    else
        cout << "Happy Unbirthday Johann Sebastian!\n";
    return 0;
}
```

You could make all of your public functions friend functions (giving you access to private variables), but this is not very clean.

You still want to use accessor and mutator functions:

```
DayOfYear today;
cout << "enter today's date: \n";
today.input();
cout << "There are " << (12 - today.get_month())
    << " months left in this year.\n";
```

They provide the cleaner interface to “get” and “set” variables.


Instead, you want to use friend functions to make your code more efficient and easier to read

Notes: you can have more than one friend function:

SYNTAX (of a class definition with friend functions)

```
class Class_Name
{
public:
    friend Declaration_for_Friend_Function_1
    friend Declaration_for_Friend_Function_2
        .
        .
        .
    Member_Function_Declarations
private:
    Private_Member_Declarations
};
```

You need not list the friend functions first. You can intermix the order of these function declarations.



And remember it is not a member function; so you don't use the dot operator to call it and you don't use the type qualifier to define it.

```
if (equal(today, bach_birthday))
    cout << "Happy Birthday Johann Sebastian!\n";
else
    cout << "Happy Unbirthday Johann Sebastian!\n";
return 0;
```

How to decide if your function should be a member/non-member function:

- Make it a member function: if the function performs a task involving only one object
- Make it a non-member function: if the function performs a task involving more than one object
 - Additional qualifier: and the objects are used symmetrically

When you make a non-member function, you can use your accessor/mutator functions or you can make it a friend.

- Just remember if it is a friend, you may have to change its implementation if you change the class implementation

An example from the text:

```
//Class for amounts of money in U.S. currency.
class Money
{
public:
    friend Money add(Money amount1, Money amount2);
    //Precondition: amount1 and amount2 have been given values.
    //Returns the sum of the values of amount1 and amount2.

    friend bool equal(Money amount1, Money amount2);
    //Precondition: amount1 and amount2 have been given values.
    //Returns true if the amount1 and amount2 have the same value;
    //otherwise, returns false.

    Money(long dollars, int cents);
    //Initializes the object so its value represents an amount with the
    //dollars and cents given by the arguments. If the amount is negative,
    //then both dollars and cents must be negative.

    Money(long dollars);
    //Initializes the object so its value represents $dollars.00.

    Money( );
    //Initializes the object so its value represents $0.00.

    double get_value( );
    //Precondition: The calling object has been given a value.
    //Returns the amount of money recorded in the data of the calling object.
}
```

An example from the text (cont'd):

```
void input(istream& ins);  
//Precondition: If ins is a file input stream, then ins has already been  
//connected to a file. An amount of money, including a dollar sign, has been  
//entered in the input stream ins. Notation for negative amounts is -$100.00.  
//Postcondition: The value of the calling object has been set to  
//the amount of money read from the input stream ins.  
  
void output(ostream& outs);  
//Precondition: If outs is a file output stream, then outs has already been  
//connected to a file.  
//Postcondition: A dollar sign and the amount of money recorded  
//in the calling object have been sent to the output stream outs.  
private:  
    long all_cents;  
};  
  
int digit_to_int(char c);  
//Function declaration for function used in the definition of Money::input:  
//Precondition: c is one of the digits '0' through '9'.  
//Returns the integer for the digit; for example, digit_to_int ('3') returns 3.
```

```

int main( )
{
    Money your_amount, my_amount(10, 9), our_amount;
    cout << "Enter an amount of money: ";
    your_amount.input(cin);
    cout << "Your amount is ";
    your_amount.output(cout);
    cout << endl;
    cout << "My amount is ";
    my_amount.output(cout);
    cout << endl;

    if (equal(your_amount, my_amount))
        cout << "We have the same amounts.\n";
    else
        cout << "One of us is richer.\n";
    our_amount = add(your_amount, my_amount);
    your_amount.output(cout);
    cout << " + ";
    my_amount.output(cout);
    cout << " equals ";
    our_amount.output(cout);
    cout << endl;
    return 0;
}

```

```
Money add(Money amount1, Money amount2)
```

```
{
```

```
    Money temp;
```

```
    temp.all_cents = amount1.all_cents + amount2.all_cents;
```

```
    return temp;
```

```
}
```

```
bool equal(Money amount1, Money amount2)
```

```
{
```

```
    return (amount1.all_cents == amount2.all_cents);
```

```
}
```

```
Money::Money(long dollars, int cents)
```

```
{
```

```
    if (dollars * cents < 0) //If one is neaative and one is positive
```

```
    {
```

```
        cout << "Illegal values for dollars and cents.\n";
```

```
        exit(1);
```

```
    }
```

```
    all_cents = dollars * 100 + cents;
```

```
}
```

Constructors with initializers:

```
Money::Money(long dollars) : all_cents(dollars * 100)
{
    //Body intentionally blank.
}
```

```
Money::Money( ) : all_cents(0)
{
    //Body intentionally blank.
}
```

```
double Money::get_value( )
{
    return (all_cents * 0.01);
}
```

```

//Uses iostream, ctype, cstdlib:
void Money::input(istream& ins)
{
    char one_char, decimal_point, digit1, digit2;
    //digits for the amount of cents
    long dollars;
    int cents;
    bool negative;//set to true if input is negative.

    ins >> one_char;
    if (one_char == ' ')
    {
        negative = true;
        ins >> one_char; //read '$'
    }
    else
        negative = false;
    //if input is legal, then one_char == '$'

    ins >> dollars >> decimal_point >> digit1 >> digit2;

    if (one_char != '$' || decimal_point != '.'
        || !isdigit(digit1) || !isdigit(digit2))
    {
        cout << "Error illegal form for money input\n";
        exit(1);
    }
    cents = digit_to_int(digit1) * 10 + digit_to_int(digit2);

    all_cents = dollars * 100 + cents;
    if (negative)

```

Note: we made digit1 and digit2 chars to read the two digits after the decimal.

What would happen if we made them numbers and digit1 was a zero?

(i.e. what happens to leading zeroes)?

```

void Money::input(istream& ins)
{
    char one_char, decimal_point, digit1, digit2;
    //digits for the amount of cents
    long dollars;
    int cents;
    bool negative;//set to true if input is negative.

```

```

ins >> one_char;
if (one_char == ' ')
{
    negative = true;
    ins >> one_char; //read '$'
}
else
    negative = false;
//if input is legal, then one_char == '$'

ins >> dollars >> decimal_point >> digit1 >> digit2;

```

```

if (one_char != '$' || decimal_point != '.'
    || !isdigit(digit1) || !isdigit(digit2))
{
    cout << "Error illegal form for money input\n";
    exit(1);
}
cents = digit_to_int(digit1) * 10 + digit_to_int(digit2);

all_cents = dollars * 100 + cents;
if (negative)
    all_cents = -all_cents;
}

```

Note: we made digit1 and digit2 chars to read the two digits after the decimal.

What would happen if we made them numbers and digit1 was a zero?

(i.e. what happens to leading zeroes)?

The remaining functions...

```
//Uses cstdlib and iostream:
void Money::output(ostream& outs)
{
    long positive_cents, dollars, cents;
    positive_cents = labs(all_cents);
    dollars = positive_cents / 100;
    cents = positive_cents % 100;

    if (all_cents < 0)
        outs << "-$" << dollars << '.';
    else
        outs << "$" << dollars << '.';

    if (cents < 10)
        outs << '0';
    outs << cents;
}

int digit_to_int(char c)
{
    return (static_cast<int>(c) - static_cast<int>('0'));
}
```

This code could be more robust (needs more error checking)

A quick note...

Recall:

```
Money your_amount, my_amount(10, 9), our_amount;
```

Replacing '9' with "09" shouldn't be a problem, but it could be if your compiler doesn't follow the ANSI standard

- In that case a leading zero might be interpreted as an octal number and "09" is not a valid octal number.

The `const` Parameter modifier

A call-by-reference (using a pointer) is more efficient than a call-by-value

Why? (Remember scope)

However, remember that if you call-by-reference be careful, any changes you make to the object will be permanent.

... Unless ...

The `const` Parameter modifier

Using `const` in conjunction with a call-by-reference tells the compiler that you want a constant parameter and that it should not be changed by the function call.

You need to use `const` in both the function declaration:

```
class Money
{
public:
    friend Money add(const Money& amount1, const Money& amount2);
    //Precondition: amount1 and amount2 have been given values.
    //Returns the sum of the values of amount1 and amount2.
    ...
}
```

And in the heading of the function definition:

```
Money add(const Money& amount1, const Money& amount2)
{
    ...
}
```

If you try to change a constant parameter in a function, then the compiler will issue an error message.

You can use the modifier `const` with any kind of parameter, but normally used with call-by-reference parameters

What about declaring objects?

```
Money m;  
m.input(cin);
```

Although not explicit, by default a member function behaves like a call-by-reference as you can change the value of the calling object.

If you have a member function that should not change the value of a calling object, it can also be marked as a `const`

```
class Money  
{  
public:  
    ...  
    void output(ostream& outs) const;  
    ...
```

The compiler will issue an error message if your function's code changes the value of the calling object.

Again, the `const` modifier should be used in both the function declaration:

```
class Money
{
  public:
    ...
    void output(ostream& outs) const;
    ...
}
```

and the function definition:

```
void Money::output(ostream& outs) const
{
  ...
}
```

Be warned:

Because of embedded function calls:

```
void guarantee(const Money& price)
{
    cout << "If not satisfied, we will pay you\n"
         << "double your money back.\n"
         << "That's a refund of $"
         << (2 * price.get_value()) << endl;
}
```

This will give an error on most compilers.

The compiler will note that `get_value` is a member function of class `Money`, and because it does not have a `const` in its definition, it **may** change the value of `Money`.

To make this work, you need to use the `const` modifier to the `get_value` member function.

Summary:

If you use the `const` parameter for one parameter of a particular type, you should use it for every other parameter of that type that is not changed by the function call.

If you use the `const` parameter with a class type, all member functions that do not modify the calling object should be used with every member function

In short, you have to use `const` everywhere or nowhere to ensure that your code will compile as you may end up embedding functions in functions.

So the two ways to use `const` are:

```
class Sample
{
public:
    Sample();
    friend int compare(const Sample& s1, const Sample& s2);
    void input();
    void output() const;
private:
    int stuff;
    double more_stuff;
};
```

Our updated version of the class Money would look like:

```
//Class for amounts of money in U.S. currency.
class Money
{
public:
    friend Money add(const Money& amount1, const Money& amount2);
    //Precondition: amount1 and amount2 have been given values.
    //Returns the sum of the values of amount1 and amount2.

    friend bool equal(const Money& amount1, const Money& amount2);
    //Precondition: amount1 and amount2 have been given values.
    //Returns true if amount1 and amount2 have the same value;
    //otherwise, returns false.

    Money(long dollars, int cents):
    //Initializes the object so its value represents an amount with the
    //dollars and cents given by the arguments. If the amount is negative,
    //then both dollars and cents must be negative.

    Money(long dollars);
    //Initializes the object so its value represents $dollars.00.

    Money( );
    //Initializes the object so its value represents $0.00.
```

...

Our updated version of the class Money (cont'd):

...

```
double get_value( ) const;
//Precondition: The calling object has been given a value.
//Returns the amount of money recorded in the data of the calling object.
void input(istream& ins);
//Precondition: If ins is a file input stream, then ins has already been
//connected to a file. An amount of money, including a dollar sign, has been
//entered in the input stream ins. Notation for negative amounts is -$100.00.
//Postcondition: The value of the calling object has been set to
//the amount of money read from the input stream ins.
void output(ostream& outs) const;
//Precondition: If outs is a file output stream, then outs has already been
//connected to a file.
//Postcondition: A dollar sign and the amount of money recorded
//in the calling object have been sent to the output stream outs.
private:
    long all_cents;
};
```

Overloading Operators

Although we can create functions called “add” and “subtract,” it would be nice to be able to declare objects and use the normal operators ‘+’ and ‘-’:

```
Money total, cost, tax;  
cout << "Enter cost and tax: ";  
cost.input(cin);  
tax.input(cin);  
total = cost + tax;
```

(Instead of: `total = add(cost, tax);`)

But operators are actually functions - with a slightly different syntax of use.

(e.g. `cost + tax` instead of `add(cost, tax)`)

Since we can overload functions, we can also overload operators.

Note: '+' is a binary operator, with the arguments before and after it (as opposed to having its function parameters listed in parentheses after the function name).

Overloaded operators look like this:

```
class Money
{
public:
    friend Money operator +(const Money& amount1, const Money& amount2);
    //Precondition: amount1 and amount2 have been given values.
    //Returns the sum of the values of amount1 and amount2.

    friend bool operator ==(const Money& amount1, const Money& amount2);
    //Precondition: amount1 and amount2 have been given values.
    //Returns true if amount1 and amount2 have the same value;
    //otherwise, returns false.

    Money(long dollars, int cents);

    Money(long dollars);

    Money( );

    double get_value( ) const;

    void input(istream& ins);
    void output(ostream& outs) const;
private:
    long all_cents;
};
```

Some comments from Display 11.4 have been omitted to save space in this book, but they should be included in a real program.

You don't have to make them friend functions, but note the operator keyword.

When you use an overloaded operator, it looks like normal:

```
int main( )
{
    Money cost(1, 50), tax(0, 15), total;
    total = cost + tax;

    cout << "cost = ";
    cost.output(cout);
    cout << endl;
    cout << "tax = ";
    tax.output(cout);
    cout << endl;
    cout << "total bill = ";
    total.output(cout);
    cout << endl;
    if (cost == tax)
        cout << "Move to another state.\n";
    else
        cout << "Things seem normal.\n";
    return 0;
}
```

When you define an overloaded operator, it looks like:

```
Money operator +(const Money& amount1, const Money& amount2)
{
    Money temp;
    temp.all_cents = amount1.all_cents + amount2.all_cents;
    return temp;
}

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

Again note the keyword `operator` is used here.

7 Rules for Overloading Operators

1. At least one argument of the overloaded operator must be of a class type.
2. Overloaded operators can be friend functions, member functions, or ordinary (non-friend functions).
3. You cannot create a new operator.
4. You cannot change the number of arguments that an operator uses (e.g. you can't change "*" to a unary operator and you can't change "++" to a binary operator).

7 Rules for Overloading Operators

5. You cannot change the precedence of an operator (e.g. $a*b-c$ always means $(a*b)-c$ even if a , b , and c are objects).
6. You cannot overload the following operators: the dot operator ($.$), the scope resolution operator ($::$), or the operators $.*$ and $?:$
7. If you want to overload the overloaded assignment operator, this is done differently (we'll talk about it later). Some other operators (e.g. $[]$ and $->$) are also overloaded differently. We might talk about these later, but they are in your text book.

```
int integer
float floating_pt:
...
floating_pt = integer;
```

Similar to when an int is automatically cast to a float, if you have the right constructors, the compiler will perform certain types of conversion for your classes automatically.

The output of:

```
Money base_amount(100, 60), full_amount;  
full_amount = base_amount + 25;  
full_amount.output(cout);
```

is: \$125.60

How?

When the compiler goes to evaluate: `base_amount + 25`, it first checks to see that '+' is overloaded for an object of type Money and an integer.

If there isn't, it looks for a constructor with a single parameter of type integer (which there is) and converts it to type Money.

After the conversion, the overloaded '+' can be used to add the two parameters of class type Money together.

As such, this assignment won't work (Why?):

```
full_amount = base_amount + 25.67;
```

To fix it:

```
class Money
{
public:
    . . .
    Money(double amount);
    //Initializes the object so its value represents $amount.
    . . .
```

You can write the function definition yourself for practice.

Remember you can also overload unary operators so:

```
Money amount1(10), amount2(6), amount3;  
amount3 = -amount1;
```

So `amount3` would equal `-$10`.

If you overload `++` and `--`, by default you are overloading the prefix version (with that order of precedence). The postfix version is done differently.

If you are feeling comfortable with programming, you should look it up.

This is an improved version of our Money class:

```
//Class for amounts of money in U.S. currency.
class Money
{
public:
    friend Money operator +(const Money& amount1, const Money& amount2);

    friend Money operator -(const Money& amount1, const Money& amount2);
    //Precondition: amount1 and amount2 have been given values.
    //Returns amount1 minus amount2.

    friend Money operator -(const Money& amount);
    //Precondition: amount has been given a value.
    //Returns the negative of the value of amount.

    friendbool operator ==(const Money& amount1, const Money& amount2);

    Money(long dollars, int cents);

    Money(long dollars);

    Money( );

    double get_value( ) const;

    void input(istream& ins);
    void output(ostream& outs) const;
private:
    long all_cents;
};
```

This is an improved version of the class Money given in Display 11.5.

We have omitted the include directives and some of the comments, but you should include them in your programs.

This is an improved version of our Money class (cont'd):

```
Money operator -(const Money& amount1, const Money& amount2)
{
    Money temp;
    temp.all_cents = amount1.all_cents - amount2.all_cents;
    return temp;
}
```

```
Money operator -(const Money& amount)
{
    Money temp;
    temp.all_cents = -amount.all_cents;
    return temp;
}
```

REMEMBER: comments in the class definition are part of your interface, so be sure to clearly define all member and friend functions.

You can also overload the insertion operators, but there are a few more details to consider.

Because it is cleaner, ideally, what we want is:

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

instead of:

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

But what is the return type of “<<“?

Remember the following assignment in C++:

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

is equivalent to:

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

So the return type of “<<” is the stream (cout)

```
class Money
{
public:
    . . .
    friend ostream& operator <<(ostream& outs, const
                               Money& amount);
    //Precondition: If outs is a file output stream, then outs
    //has already been connected to a file.
    //Postcondition: A dollar sign and the amount of money
    //recorded in the calling object have been sent to the output
    //stream outs.
    . . .
}
```

We no longer need the output member function as part of the class `Money` and the replacement function looks very similar

```
ostream& operator <<(ostream& outs, const Money& amount)
{
    <This part is the same as the body of Money::output
    that is given in Display 11.3 (except that all_cents
    is replaced with amount.all_cents).>
    return outs;
}
```

Think: Why do we need to replace the references to `all_cents` with `amount.all_cents`.

Also, why is the return value `ostream&`?

Why is the return value `ostream&`?

```
ostream& operator <<(ostream& outs, const Money& amount)
{
    <This part is the same as the body of Money::output
    that is given in Display 11.3 (except that all_cents
    is replaced with amount.all_cents).>

    return outs;
}
```

If you returned `ostream`, you would be returning, the entire file/keyboard/screen (that doesn't make sense)

The '`&`' means you are returning a reference (a pointer) as opposed to the value.

This means you are returning the actual object and not simply its value

The extraction operator is similar to the insertion operator

```
istream& operator >>(istream& ins, Money& amount)
{
    <This part is the same as the body of
    Money::input given in Display 11.3 (except that
    all_cents is replaced with amount.all_cents).>
    return ins;
}
```

It returns a pointer to the input stream (incremented through the file/keyboard input to just after the last extracted value)

This might make it a little easier to understand why we need to use pointers instead of values.

Summary of Insertion and Extraction operator function declarations and definitions:

FUNCTION DECLARATIONS

```
class Class_Name
{
    public:
        . . .
        friend istream& operator >>(istream& Parameter_1,
                                     Class_Name& Parameter_2);
        friend ostream& operator <<(ostream& Parameter_3,
                                     const Class_Name&
                                     Parameter_4);
        . . .
```

Parameter for the stream →

Parameter for the object to receive the input ↓

DEFINITIONS

```
istream& operator >>(istream& Parameter_1,
                      Class_Name& Parameter_2)
{
    . . .
}

ostream& operator <<(ostream& Parameter_3,
                      const Class_Name& Parameter_4)
{
    . . .
```

Here's an example:

```
//Program to demonstrate the class Money
#include <iostream>
#include <fstream>
#include <cstdlib>
#include <cctype>
using namespace std;

//Class for amounts of money in U.S. currency.
class Money
{
public:
    friend Money operator +(const Money& amount1, const Money& amount2);
    friend Money operator -(const Money& amount1, const Money& amount2);
    friend Money operator -(const Money& amount);
    friend bool operator ==(const Money& amount1, const Money& amount2);

    Money(long dollars, int cents);
    Money(long dollars);
    Money( );

    double get_value( ) const;
```

*This is an improved version of the class **Money** that we gave in Display 11.6.*

Although we have omitted some of the comments from Displays 11.5 and 11.6, you should include them.

Here's an example (rest of the class def'n and non-member function):

.....

```
2014/0 friend istream& operator >>(istream& ins, Money& amount);  
//Overloads the >> operator so it can be used to input values of type Money.  
//Notation for inputting negative amounts is as in -$100.00.  
//Precondition: If ins is a file input stream, then ins has already been  
//connected to a file.  
  
friend ostream& operator <<(ostream& outs, const Money& amount);  
//Overloads the << operator so it can be used to output values of type Money.  
//Precedes each output value of type Money with a dollar sign.  
//Precondition: If outs is a file output stream,  
//then outs has already been connected to a file.  
  
private:  
    long all_cents;  
};  
int digit_to_int(char c);  
//Used in the definition of the overloaded input operator >>.  
//Precondition: c is one of the digits '0' through '9'.  
//Returns the integer for the digit; for example, digit_to_int('3') returns 3.
```

```

int main( )
{
    Money amount;
    ifstream in_stream;
    ofstream out_stream;

    in_stream.open("infile.dat");
    if (in_stream.fail( ))
    {
        cout << "Input file opening failed.\n";
        exit(1);
    }

    out_stream.open("outfile.dat");
    if (out_stream.fail( ))
    {
        cout << "Output file opening failed.\n";
        exit(1);
    }

    in_stream >> amount;
    out_stream << amount
        << " copied from the file infile.dat.\n";
    cout << amount
        << " copied from the file infile.dat.\n";

    in_stream.close( );
    out_stream.close( );

    return 0;
}

```

```

//Uses iostream, ctype, cstdlib:
istream& operator >>(istream& ins, Money& amount)
{
    char one_char, decimal_point,
        digit1, digit2; //digits for the amount of cents
    long dollars;
    int cents;
    bool negative;//set to true if input is negative.

    ins >> one_char;
    if (one_char == '-')
    {
        negative = true;
        ins >> one_char; //read '$'
    }
    else
        negative = false;
    //if input is legal, then one_char == '$'

    ins >> dollars >> decimal_point >> digit1 >> digit2;

    if (one_char != '$' || decimal_point != '.'
        || !isdigit(digit1) || !isdigit(digit2))
    {
        cout << "Error illegal form for money input\n";
        exit(1);
    }

    cents = digit_to_int(digit1) * 10 + digit_to_int(digit2);

    amount.all_cents = dollars * 100 + cents;
    if (negative)
        amount.all_cents = -amount.all_cents;
    return ins;
}

```

Remaining code describing the insertion operator:

```
int digit_to_int(char c)
{
    return ( static_cast<int>(c) - static_cast<int>('0') );
}

//Uses cstdlib and iostream:
ostream& operator <<(ostream& outs, const Money& amount)
{
    long positive_cents, dollars, cents;
    positive_cents = labs(amount.all_cents);
    dollars = positive_cents/100;
    cents = positive_cents%100;

    if (amount.all_cents < 0)
        outs << "- $" << dollars << '.';
    else
        outs << "$" << dollars << '.';

    if (cents < 10)
        outs << '0';
    outs << cents;

    return outs;
}
```

Review Questions for Slide Set 3

- What are friend functions? Why are they valuable/needed?
- Should friend functions be public or private?
- Are friend functions member functions?
- Do friend functions break our rules of encapsulation?
- When should you use friend functions?
- If you change your class implementation, would you have to change the friend functions?
- When and why is a call by reference more efficient than a call by value?
- What does the const parameter do? When should you use the const parameter?

Review Questions for Slide Set 3

- Do you use the const parameter in the function declaration, function definition, or both?
- Will a compiler allow you to change a parameter marked with the const label?
- Why is the const parameter important in the context of member functions?
- What is operator overloading? What keyword do you need? Are they friend, member or non-member functions?
- Why do you return ostream& when overloading the insertion operator (<<) as opposed to just ostream?
- What is the extraction operator?