Overloading Operators

- Last time we discussed the string class
  - Strings can be concatenated using +, and assigned using =
  - This is done with operator overloading

- Like function overloading, operator overloading lets you redefine how (most) operators work with arguments as operands.

Operators

- Operators are a nice convention that lets you write code a little easier:
  - 5 + 6 is essentially the same as calling +(5,6).
  - The arguments are the operands, and the function is the operator.

- C++ provides a way to override these operators and allow them to work with classes.

Example

- Assume you had a class called Money which represented monetary values:

```
class Money{
   public:
      ...
   private:
      int dollars;
      int cents;
};
```

- It would be possible to define the member function add(Money value) which would add two Money objects together...

Example

- But why not make a way to let you do this?

```
Money value1(1,2), value2(2,3), value3;
value3 = value1 + value2;
```

- With operator overloading, you can.

- The basic prototype syntax to start with:

```
const Money operator +(const Money value1,
                       const Money value2);
```

- In this case, we are passing in Money by reference, and returning another Money. (Ignore the const for now). However, call-by-value can be used, any type can be used for arguments (as long as one is a class), and any type can be returned.

Overloading Operators

- Note that to overload the operator, we have given “operator +” in the prototype, showing that we are defining a new version of a particular operator.
  - You cannot define new operators!
  - This version of + is not a member of the Money class, so to use it we will need Money to have mutators and accessors.
    - Later we will look at how to make a member operator.
  - We can also overload unary operators (i.e. taking a single argument) like -, and overload the comparators.

Overloading Operators – Non-member version

- Most but not all operators can be overloaded.
- You cannot redefine operators for basic types – at least one of your operands must be a class type.
- You must use the keyword **operator** to overload an operator.
- Syntax for binary operator overloading:

```
<return_type> operator <symbol>(<argument 1>,
                                <argument 2>);
```

- Syntax for unary operator overloading:

```
<return_type> operator <symbol>(<argument>);
```

An example of how to do non-member operator overloading

- A shortened Money class and overloadings:

```
class Money{
public:
   Money(int newDollars, int newCents);
   int getDollars() const;
   int getCents() const;
   void output() const;
private:
   int dollars;
   int cents;
};
const Money operator +(const Money & value1,
                       const Money & value2);
const Money operator -(const Money & value1,
                       const Money & value2);
bool operator ==(const Money & value1,
                 const Money & value2)
const Money operator -(const Money & value);
```

Defining the + overload

- Overloading addition of Money:

```
const Money operator +(const Money & value1,
                       const Money & value2){
   int allCents1 = value1.getDollars()*100 +
                    value1.getCents();
   int allCents2 = value2.getDollars()*100 +
                    value2.getCents();
   int sumAllCents = allCents1 + allCents2;
   int absAllCents = abs(sumAllCents);
   int finalDollars = absAllCents/100;
   int finalCents = absAllCents % 100;
   if(sumAllCents < 0){
      finalDollars = -finalDollars;
      finalCents = -finalCents;
   }
   return Money(finalDollars, finalCents);
}
```

Defining the == and unary - overloads

- Overloading equivalence/negative of Money:

```
const Money operator -(const Money & value){
   return Money(-value.getDollars(),
                -value.getCents());
}

bool operator ==(const Money & value1,
                 const Money & value2){
   return ((value1.getDollars() == value2.getDollars()
         && (value1.getCents() == value2.getCents()));
}
```

Overloading is done!

- It is now possible to use +,-,==, and – with Money in a program.
- Since they are non-member, we needed accessor and mutator functions.
- You probably noticed **const** a lot.
  - This is to prevent the Money returned from the addition from being changed immediately!
  - Generally, use const for operator overloading unless you have a good reason not to.
- Unary operators like ++,-- can also be overloaded.

Overloading operators as a member

- Very similar to overloading as a non-member: you place the prototype within your class definition, and omit the first argument.
  - The first argument is now the calling object.
  - Since it is a member function, private variables can be called directly!
  - While it is more efficient and more Object-Oriented to use member operators, this version does have drawbacks.

An example of how to do member operator overloading

- A shortened Money class and overloadings:

```
class Money{
public:
   Money(int newDollars, int newCents);
   int getDollars() const;
   int getCents() const;
   void output() const;
   const Money operator +(Money & value2) const;
   const Money operator -(Money & value2) const;
   bool operator ==(const Money & value2) const;
   const Money operator -() const;
private:
   int dollars;
   int cents;
};
```

Defining the + overload as a member

- Overloading addition of Money:

```
const Money operator +(Money & value2){
   int allCents1 = dollars*100 + cents;
   int allCents2 = value2.dollars*100 +
                    value2.cents;
   int sumAllCents = allCents1 + allCents2;
   int absAllCents = abs(sumAllCents);
   int finalDollars = absAllCents/100;
   int finalCents = absAllCents % 100;
   if(sumAllCents < 0){
      finalDollars = -finalDollars;
      finalCents = -finalCents;
   }
   return Money(finalDollars, finalCents);
}
```

Defining the == and unary – overloads as members

- Overloading equivalence/negative of Money:

```
const Money operator -(){
   return Money(-dollars, -cents);
}

bool operator ==(const Money & value2){
   return ((dollars == value2.dollars)
          && (cents == value2.cents));
}
```

Defining as Member Operators

- Making the definitions is very similar
  - You can now access without accessors.
  - The first argument is now just the calling object.
  - Remember to add const at the end of the prototype and definition to make sure the calling object can't be changed.
  - Also remember to add the scope to your definition now:

```
bool Money::operator ==(const Money & value2){
   return ((dollars == value2.dollars)
           && (cents == value2.cents));
}
```

Let's work on doing overloading as a class

- I've deleted the prototypes and definitions that I wrote for a class called Fraction.
- I'll hand out this blanked version of the code.
- Work in groups of 2 or 3 and try to fill it out on your own, and we'll see how well we can get it to work.
- After you finish, think about how you might change it to use member functions.

Using Constructors for Automatic Type Conversion

- Recall how we defined Money and its overloaded operators earlier. How would this code work?

```
Money original(100,60), fullAmount;
fullAmount = original + 25;
fullAmount.output();
```

- This outputs $125.60, but how?
  - 25 is not appropriate. We didn't overload + to take a Money and an int.
  - If we have a constructor that converts a single int into Money, though...

How the system knows what to do

- Let's say you pass in

```
    original + 25;
```

- The system starts by looking for an overload of + that has a Money for argument 1 and an int for argument 2.
- When it doesn't find that, it will try to make it fit the only overload we made: Money and Money.
- So, it uses (if it exists) the single int constructor to automatically convert the int to Money!

When doesn't automatic type conversion work?

- If you don't have an appropriate constructor defined.
- Keep in mind, this is just like with other overloaded functions. Remember the matching rules from the first time we talked about overloading!
- Note: Member operators will behave oddly.
  - If you try to do `25 + original;`, the non-member will handle this fine.
  - But, member operators MUST have the class type as the first operand – an int like 25 cannot make a call!
  - This is that previously mentioned pitfall of member operators.

Friend Functions

- When we previously defined a non-member overloaded operator, we needed to use accessors to define it properly.
- While this is sufficient, it is also inefficient and harder to read.
- How can we eliminate this intermediate step? (Hint: Look at the slide title!)
- Yes, friend functions!
- A **friend function** of a class is not a member function, but has access to private members of that class.

Using friend functions

- To make a function a friend of a class, you give its prototype in the class definition with the keyword **friend** in front.
- You can then define it normally, but any objects of that class used in the function can access the private members of the class.

```
class Money{
public:
   ...
   friend const Money operator +(const Money &
              value 1, const Money & value 2);
   ...
}
const Money operator +(const Money & value 1,
                 const Money & value 2){....}
```

Friend functions

- Any function can be a friend, but overloaded operators are most common.
- You can make a function a friend of as many classes as you like – just put the friend prototype in each class that applies.
- Depending on who you ask, friend functions are not “pure” object-oriented functions. They break the spirit of encapsulation. (Though not as much as a non-member, non-friend overload, perhaps.)
- More on the use of friends later.

Compilers without friends

- Some older compilers don't handle friends properly. Be warned!

Rules for operator overloading

- At least one argument must be of a class type.
- Most operators can be overloaded as a no-member, a member, or a friend.
- Some operators can only be done as members: =,[],->,().
- You cannot make new operators.
- You cannot change the number of arguments an operator takes!
- Precedence does not change.
- .,::,sizeof,?:, and .* cannot be overloaded.
- Overloaded operators have no default arguments.

Return by Reference

- A reference is the name of a storage location.

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

- This may be familiar from call-by-reference – a reference is effectively an alias for a variable.
- These references can be returned:

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

- This simple example you let you do this:

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

Return by reference

- The previous example would output 99, then 42.
- Note: never return the reference to a local variable!
- L-value and R-value:
  - L-value is something that can appear on the left side of an assignment operator.
  - R-value is something that can appear on the right side of an assignment operator.
  - In order to use the object returned by a function as an l-value, it must be returned by reference.
- We will use this return by reference to overload certain operators.

Overloading << and >>

- You may remember from our discussion of streams that << and >> return the first variable when they complete.
  - This allows us to string << or >> together:
    - cout << x << “ is equal to “ << y;
- So, to overload these, the first parameter is an appropriate stream type (output for <<, input for >>) and the second is your class type.

-
  ```
  ostream& operator <<(ostream& out,
                       const Money& value);
  ```

Overloading <<

- How might you do this?

```
ostream& operator <<(ostream& out, const
Money& value){
// As a regular non-member overload
    value.print(out);
    return out;
}
{ // As a friend function
    out << value.variable;
    return out;
}
```

Two schools of thought

- The book recommends overloading << and >> as friend functions.
- Most professors I have spoken with do not recommend this.
- Many professors and professionals use friend rarely if at all!

Against friend functions

- Friend functions have several knocks against them:
  - They violate the security/encapsulation principles of Object Oriented Programming.
  - They tend to become overly complex to program and use, and lead to bad programming practices.
    - Yes, you can use an overloaded +, but you can also write a member add() function or some other function that replicates the functionality you need.
      - For example, rather than overload <<, make a char* toString() member function, and send that to the stream.
  - They aren't actually needed in most cases!
  - Depending on who you ask, even operator overloading isn't needed.

Overloading increment/decrement

- To make these work properly, you must define two overloads:
  - One with no argument (for prefix)
  - One with a single int argument (for postfix)
- The single int argument is irrelevant – it just lets the compiler know which version to call.
- This should be a member operator.

Overloading array operator

- You can define your own bracket functionality so that you can access a class with x[0] if you like.
- It must be a member operator, and it must take a single int parameter.

Overloading Assignment Operators

- We will discuss this later.
- Just know that doing this lets you override how to do copying and assignment.

Friend classes

- We will not use this. Should you ever see it:

```
class X;

class Y{
   public:...
      friend X;...
   private:
      ...
}

class X{...
```

- This lets objects of class X access private variables of objects of class Y.