Operator Overloading

A Game of Cards

Let's say you and I are playing a (very simple) game of cards:

```
// create a shuffled deck of cards
Deck the_deck;


// declare some cards for the two of us
Card my_card, your_card;


// deal some cards
my_card = the_deck.deal_card();
your_card = the_deck.deal_card();
```

A Game of Cards

Let's say you and I are playing a (very simple) game of cards:

```
// display my card's value
cout << "My card:   ";
my_card.output(cout);
cout << endl;

// display your card's value
cout << "Your card: ";
your_card.output(cout);
cout << endl;
```

A Game of Cards

Let's say you and I are playing a (very simple) game of cards:

```
// I hope my card isn't the old hag...
if (my_card.equals("Queen of Spades")) {
    cout << "DOH!" << endl;
} else {
    cout << "Phew..." << endl;
}
```

A Game of Cards

Let's say you and I are playing a (very simple) game of cards:

```
// see who won (pretty sure I did...)
if (my_card.get_value() == your_card.get_value())
    cout << "It's a tie!" << endl;
} else if (my_card.get_value() > your_card.get_value()) {
    out << "I win! Hooray!" << endl;
} else {
    cout <<"You win. =(" << endl;
}
```

Not bad...

But here's a different version:

A Game of Cards (Version 2)

Let's say you and I are playing a (very simple) game of cards:

```
// create a shuffled deck of cards
Deck the_deck;


// declare some cards for the two of us
Card my_card, your_card;


// deal some cards
the_deck >> my_card >> your_card;
```

A Game of Cards (Version 2)

Let's say you and I are playing a (very simple) game of cards:

```
// display my card's value
cout << "My card: " << my_card << endl;


// display your card's value
cout << "Your card: " << your_card << endl;
```

A Game of Cards (Version 2)

Let's say you and I are playing a (very simple) game of cards:

```
// I hope my card isn't the old hag...
if (my_card == "Queen of Spades") {
    cout << "DOH!" << endl;
} else {
    cout << "Phew..." << endl;
}
```

A Game of Cards (Version 2)

Let's say you and I are playing a (very simple) game of cards:

```
// see who won (pretty sure I did...)
if (my_card == your_card)
    cout << "It's a tie!" << endl;
} else if (my_card > your_card) {
    out << "I win! Hooray!" << endl;
} else {
    cout <<"You win. =(" << endl;
}
```

That one felt fairly natural, right?

Dealing a Pair of Cards

Version 1:

```
// deal some cards

my_card = the_deck.deal_card();

your_card = the_deck.deal_card();
```

Version 2:

```
// deal some cards

the_deck >> my_card >> your_card;
```

Displaying the Cards

Version 1:

```
// display my card's value
cout << "My card: ";
my_card.output(cout);
cout << endl;
```

Version 2:

```
// display my card's value
cout << "My card: " << my_card << endl;
```

Comparing a Card to a String

Version 1:

```
// is my card the queen of spades?
if (my_card.equals("Queen of Spades")) { ... }
```

Version 2:

```
// is my card the queen of spades?
if (my_card == "Queen of Spades") { ... }
```

Are Two Cards Equal?

Version 1:

```
// are our cards the same?
if (my_card.get_value() == your_card.get_value()) { ... }
```

Version 2:

```
// are our cards the same?
if (my_card == your_card) { ... }
```

Is One Card Bigger Than Another?

Version 1:

```
// is my card bigger than yours?

if (my_card.get_value() > your_card.get_value()) { ... }
```

Version 2:

```
// is my card bigger than yours?

if (my_card > your_card) { ... }
```

Version 2 used overloaded operators

Operator Overloading

C++ comes with predefined behaviors for operators…

- the addition operator (+) adds two numbers and returns the result
- the << and >> operators actually do bitwise shifting of numbers by default

We can add additional behavior via operator overloading

- many times, operations are naturally described using symbols such as == and +

The string class overloads + to concatenate (join) two strings:

```
string s1 = "Foo", s2 = "Bar";
string s3 = s1 + s2; // joins s1 and s2 using +
```

Operator Overloading

C++ comes with predefined behaviors for operators…

- the addition operator (+) adds two numbers and returns the result
- the << and >> operators actually do bitwise shifting of numbers by default

We can add additional behavior via operator overloading

- many times, operations are naturally described using symbols such as == and +

The ostream class overloads << to do output:

```
// does output using overloaded <<
cout << "Hi!";
```

Operator Overloading

Operators are implemented in C++ as functions

- you can think of writing `5 + 4` as actually saying *add*`(5, 4)`

Operator functions have a few differences from other functions:

- their names follow a slightly different syntax (`operator X`)
- they are not called using parentheses, but by being placed between / by their arguments (`e.g. text + 23`)
- at least one of the arguments to the operator MUST be of a class type

Other than that, these are just like any other function you've written

- actually, they're a whole lot cooler. :)
- we'll also see them all the time in the STL (standard template library)

Operator Overloading

Assume we have the following class declaration:

```
class Point {
  public:
    Point(int X = 0, int Y = 0) : x(X), y(Y) { }
    int get_x() const { return x; }
    int get_y() const { return y; }

  private:
    int x, y;
};
```

Operator Overloading

Let's say we want to be able to add two Points together:

```
Point p1, p2;
Point p3 = p2 + p3;
```

C++ doesn't magically know how to do this...

- we have to tell it how by overloading operator +

We can implement this operator in a few different ways:

- as a global function, that uses the public methods provided by the class
- as a friend global function, that has access to private properties of Point objects
- as a member function

Operator Overloading

The function name for a hypothetical binary operator X:

```
operator X  // X is the operator you want to overload
```

General prototype syntax:

```
return_type operator X(arg_type LHS, arg_type RHS);
```

Using the operator:

```
lhs X rhs  // translates to:  operator X(LHS, RHS)
```

General Just so you know:

- `lhs` means 'left-hand side' of the operator (will be used as first argument)
- `rhs` means 'right-hand side' of the operator (will be used as second argument)

Global Operators

Global non-friend operators must use the public interface of a class

- they are able to access only what the class makes publicly available

Example implementation:

```
// global operator; must use getters to access x and y
Point operator +(const Point& p1, const Point& p2) {
    return Point(
        p1.get_x() + p2.get_x(), // use getters
        p1.get_y() + p2.get_y()  // to access x and y
    );
}
```

friend operators

friend operators:

- are written just like global operators (friend operators *are* global operators)
- must be prototyped in the class declaration, prefixed with the friend keyword
- even though they're prototyped in the class declaration, they do not actually belong to the class (NO `Classname::` prefix when implementing)
- can access private properties and methods of objects belonging to that class
- the implementation of the operator should not have the friend keyword

Be aware:

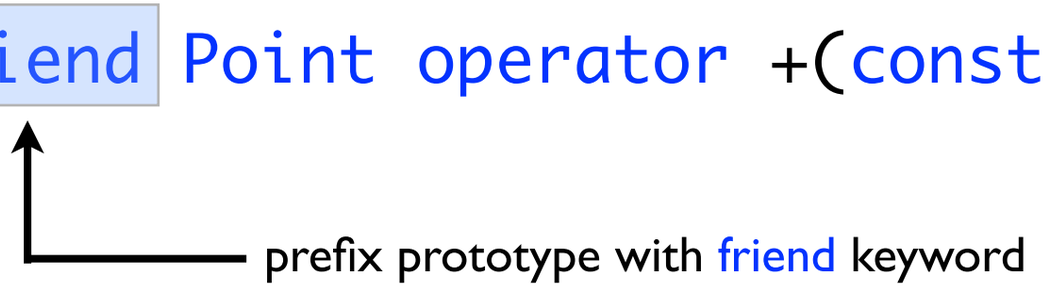
- there is no 'calling object' in the context of a friend operator function
- you can only access properties and methods of a class by using the dot operator on an object of the class (either passed as an argument or created locally in the function)

friend operators

To make operator + a friend, add its prototype to the class declaration:

```
class Point {
  public:
    // make operator + a friend
    friend Point operator +(const Point&, const Point&);

  private:
    int x, y;
};
```



friend operators

Example implementation:

```
// friend operator; has direct access to x and y
Point operator +(const Point& p1, const Point& p2) {
    return Point(
        p1.x + p2.x, // direct access to properties!
        p1.y + p2.y  // no need to use getters
    );
}
```

Notice how this function can directly access x and y!

- though not a member of the Point class, it still has access private properties of class Point
- this is a global function, so must use `p1.x` and `p1.y` (rather than x and y on their own)

Two Versions of the Same Operator

As a non-friend global operator:

```
Point operator +(const Point& p1, const Point& p2) {
    return Point(p1.get_x() + p2.get_x(), p1.get_y() + p2.get_y());
}
```

As a friend global operator:

```
Point operator +(const Point& p1, const Point& p2) {
    return Point(p1.x + p2.x, p1.y + p2.y);
}
```

The only difference is that the second was declared as a friend

- this is as simple as placing its prototype in the class declaration, prefixed with 'friend'

Two Versions of the Same Operator

In either case, we use the operator the same way:

```
Point p1, p2;
Point p3 = p2 + p3;
```

Generally, make binary operators friends (unless you can't)

- if you're the author of a class, no problem—make 'em friends
- if you can't change the class declaration, you're restricted to non-friend operators and whatever public interface is provided

Think of it like a
friend request on Facebook...

Your friends get to see stuff about you
that the rest of the world doesn't!

A Facebook analogy?
Wow... I feel dirty.

Still, if the glove fits...
(you must acquit???)

I/O Operators

Let's say we want to add input and output operators to this class:

```
class Point {
  public:
    Point(int X = 0, int Y = 0) : x(X), y(Y) { }
  private:
    int x, y;
};
```

We want our points to be in this format: `(x,y)`

- the output operator (<<) should print this
- and the input operator (>>) should read it

I/O Operators

Add friend-ed prototypes to the class declaration:

```
class Point {
  public:
    Point(int X = 0, int Y = 0) : x(X), y(Y) { }
    // overloaded input and output operators
    friend ostream& operator <<(ostream&, const Point&);
    friend istream& operator >>(istream&, Point&);

  private:
    int x, y;
};
```

I/O Operators

The output operator's prototype:

```
friend ostream& operator <<(ostream&, const Point&);
```

An example implementation:

```
ostream& operator <<(ostream& out, const Point& p) {
    return out << "(" << p.x << "," << p.y << ")";
}
```

Notice:

- the output operator should accept an `ostream` object by reference and return the same stream (again, by reference)
- because this is a friend function, we have direct access to `p`'s `x` and `y` coordinates
- the friend keyword only appears in the prototype!

I/O Operators

The input operator's prototype:

```
friend istream& operator >>(istream&, Point&);
```

An example implementation:

```
istream& operator >>(istream& in, Point& p) {
    char trash; // for parentheses and comma
    return in >> trash >> p.x >> trash >> p.y >> trash;
}
```

Notice:

- even though we don't need the parentheses and the comma, we still have to read past them (since we want our input to be in the form of `(x,y)`)

I/O Operators

A more *robust* implementation for the input operator:

```
// reads a Point in the form (x,y). Assume that x and y cannot be negative
istream& operator >>(istream& in, Point& p) {
    int x, y;    // temporary variables to hold the coordinates
    char trash;  // for the comma and parentheses, which aren't needed
    // reads in "(x,y)" and ensures that x and y are not negative
    if (!(in >> trash >> x >> trash >> y >> trash) || x < 0 || y < 0) {
        in.setstate(ios::failbit); // if x or y are negative, set failure state
    } else {
        p.x = x; // x and y are valid, so directly modify
        p.y = y; // the x and y values of p
    }
    return in;   // always return the stream object
}
```

I/O Operators

Some "best practice" steps to follow for your input operators:

- first read the appropriate values from the input stream into temporary local variables (you may need some extra variables to store 'trash' values, such as the parentheses and comma in the previous example)
- if the input operation succeeded, ensure that the values make sense according to the rules of your class. If not, set the input stream into failure state
- only after you have checked that the input succeeded and that the values make sense should you modify the object
- always return the stream object!

Comparison Operators

Using this class declaration...

```
class Point {
  public:
    Point(int X = 0, int Y = 0) : x(X), y(Y) { }
    // distance from this point to the given coordinates
    double distance(double, double) const;
  private:
    int x, y;
};
```

Comparison Operators

Let's say we want to compare Point objects like this:

```
Point p1, p2;


p1 == p2; // are p1 and p2 equal?
p1 != p2; // not equal?
p1 > p2;  // is p1 greater than p2?
p1 < p2;  // less than p2?
p1 >= p2; // is p1 greater than or equal to p2?
p1 <= p2; // less than or equal to p2?
```

Comparison Operators

Prototypes in class declaration:

```
friend bool operator ==(const Point&, const Point&);
friend bool operator !=(const Point&, const Point&);
```

Example implementations:

```
bool operator ==(const Point& LHS, const Point& RHS)
{
    return LHS.x == RHS.x && LHS.y == RHS.y;
}

bool operator !=(const Point& LHS, const Point& RHS)
{
    return !(LHS == RHS);
}
```

Comparison Operators

Prototypes in class declaration:

```
friend bool operator >(const Point&, const Point&);
friend bool operator <(const Point&, const Point&);
```

Example implementations (distance to origin):

```
bool operator <(const Point& LHS, const Point& RHS)
{
    return LHS.distance(0,0) < RHS.distance(0,0);
}

bool operator >(const Point& LHS, const Point& RHS)
{
    return LHS.distance(0,0) > RHS.distance(0,0);
}
```

Comparison Operators

Prototypes in class declaration:

```
friend bool operator >=(const Point&, const Point&);
friend bool operator <=(const Point&, const Point&);
```

Example implementations (distance to origin):

```
bool operator <=(const Point& LHS, const Point& RHS)
    return !(LHS > RHS);
}

bool operator >=(const Point& LHS, const Point& RHS)
    return !(LHS < RHS)
}
```

Operators as Member Functions

Operators can also be implemented as member functions

- this can be somewhat confusing, since binary operators implemented as member functions accept only one argument!
- the calling object (since it's not a global function) is used as the left-hand-side argument

Operators as Member Functions

Example implementation as a member function:

```
bool Point::operator <(const Point& p) const {
    return (x * x + y * y) < (p.x * p.x + p.y * p.y);
}
```

Corresponding class declaration:

```
class Point {
   public:
     bool operator <(const Point&) const;
   private:
     int x, y;
};
```

Operators as Member Functions

Example implementation as a member function:

```
bool Point::operator <(const Point& p) const {
    return (x * x + y * y) < (p.x * p.x + p.y * p.y);
}
```

Some things to note:

- x and y can be use directly (they belong to the calling object)
- the right-hand-side is still accessed using an argument and the dot operator
- the implementation gets prefixed with `Point::`
- the method is declared as const, since comparing two points shouldn't change the calling object

Operators as Member Functions

Most of the time, try to avoid member operators...

- however, some operators (the array-access operator, `operator []`, and the grouping operator, `operator ()`) must be implemented as member functions
- since the left-hand-side must be an object, you could not implement an operator where the LHS is a primitive (like an int) as a member function

This wouldn't work implemented as a member operator:

```
5 < my_card; // 5 is not an object!
```

Operators aren’t scary.

They’re really useful, and often really easy to implement.