Friend Functions
Operator Overloading
in C++
Friend functions and classes

- Friend functions and classes get to access private and protected data in the class that friends them
  - A way to selectively break encapsulation
- As we will see, it can sometimes be essential to implementing operator overloading
class X { // see Friend for code
    int m;
    int n;
public:
    X(int mm, int nn) {m = mm; n = nn;}
    friend class Y;
    friend void print(X*);
};

class Y {
private:
    X* x;
    int t;
public:
    Y(X* xobj) {x = xobj; t = x->m + x->n; }
    virtual int get_t() { return t;}
};

void print(X* ptr) {
    cout << ptr->m << " " << ptr->n << endl;
}

int main( ) {
    X* ptr = new X(100, 200);
    Y y(ptr);
    cout << y.get_t() << endl;
    print(ptr);
    return 0;
}
Operator overloading

You have been using operator overloading since you began programming
• 3+5: Integer addition, bitwise+carries
• 3.+ 0.0059: Floating point addition
• “hello” + “ world” // concatenate two strings

Operator overloading is not essential in object-oriented programming. Java does not allow programmer-defined operator overloading (but, of course, does allow overloading functions)
Java and C: Two different philosophies

James Gosling, leader of the original Java language team
I left out operator overloading as a fairly personal choice because I had seen too many people abuse it in C++.

http://www.gotw.ca/publications/c_family_interview.htm

Bjarne Stroustrup, designer of C++
Many C++ design decisions have their roots in my dislike for forcing people to do things in some particular way [...] Often, I was tempted to outlaw a feature I personally disliked, I refrained from doing so because I did not think I had the right to force my views on others

The Design and Evolution of C++ (1.3 General Background)
How and what operators can be overloaded

• We have seen overloaded operator for << >> and =
  • cout << "overload <<" << endl;

• at least one operand (for binary operators) must be an object or enumeration type (i.e. not a built-in type)
  • this prevents overloading integer +, for example

• precedence is not changed

• arity not changed (! always unary, % always binary)

• argument(s) may be passed by value (copy) or by reference, not by pointer

• default argument value(s) illegal

• cannot overload :: .* ?. ?: sizeof

• Cannot overload sizeof and typeid functions
Binary operators

• If the *first operand is an object*, a binary operator can be implemented in two forms
  • a member function, the first operand is the object pointed to by the *this* pointer
  • a “free” function (not a member of any class), usually declared as a *friend* to access private attributes
    • but not both (C++ needs one function to unambiguously choose)
• If the first operand is not an object (such as int), the operator must be a free (typically a *friend*) function.
• In most cases, the operand arguments should use *reference* to prevent calling a copy constructor.
class MyComplex {
    double re, im;

public:
    MyComplex(double r, double i);
    double getReal() const;
    double getImag() const;
};

MyComplex operator+(const MyComplex&, const MyComplex&);
MyComplex operator-(const MyComplex&, const MyComplex&);
std::ostream& operator<<(std::ostream&, const MyComplex&);
MyComplex::MyComplex(double r, double i) : re(r), im(i) { }

double MyComplex::getReal( ) const {return re; }

double MyComplex::getImag( ) const {return im; }

MyComplex operator+ (const MyComplex& arg1, const MyComplex& arg2) {
    double d1 = arg1.getReal( ) + arg2.getReal( );
    double d2 = arg1.getImag( ) + arg2.getImag( );
    return MyComplex(d1, d2);
}

Operator+ is not part of the MyComplex class
MyComplex operator-(const MyComplex& arg1, const MyComplex& arg2) {
    double d1 = arg1.getReal() - arg2.getReal();
    double d2 = arg1.getImag() - arg2.getImag();
    return MyComplex(d1, d2);
}

Operator- is analogous to operator+, except it performs the operation \( \text{arg1 - arg2} \).

Note that operator+ and operator- could have been (and should have been) implemented as part of the MyComplex class.
int main() {
    MyComplex first(3,4);
    MyComplex second(2,9);
    std::cout << first;          (3, 4)
    std::cout << second;         (2, 9)
    std::cout << first + second;  (5, 13)
    std::cout << first - second;  (1, -5)
    return 0;
}
Operator overloading using member functions

(class MyComplex)

class MyComplex {
    double re, im;

public:
    MyComplex(double r, double i);
    double getReal( ) const;
    double getImag( ) const;
    MyComplex operator+(const MyComplex&);
    MyComplex operator-(const MyComplex&);
};

friend std::ostream& operator<<(std::ostream&, const MyComplex&);

+ is still a binary operator, but only one argument is declared. Why?

first + second;

can be thought of as performing the invocation first.operator+ (second); with the this pointer pointing to the first object.
The + code (- is similar, << is as with global functions)

MyComplex MyComplex::operator+ (const MyComplex& rightOp) {
  double d1 = getReal( ) + rightOp.getReal( );
  double d2 = getImag( ) + rightOp.getImag( );
  return MyComplex(d1, d2);
}

The use of getter functions is optional since this are part of the MyComplex class. We could say re+rightOp.getReal( ) and im+rightOp.getImag
And we get the same answer as with the global functions

```c++
int main( ) {
    MyComplex first(3,4);
    MyComplex second(2,9);
    std::cout << first;  (3, 4)
    std::cout << second; (2, 9)
    std::cout << first + second; (5, 13)
    std::cout << first - second; (1, -5)
    return 0;
}
```
std::ostream& operator<<(std::ostream& os, const MyComplex& arg) {
    double d1 = arg.getReal();
    double d2 = arg.getImag();
    os << "(" << d1 << ", " << d2 << ")" << std::endl;
    return os;
}

This case is more interesting. This function allows us to say things like
std::cout << myCmplxObj << std::endl; where myCmplxObj is a MyComplex
object.

It cannot, however, be implemented as part of the MyComplex class!
Let’s look at our code a little closer

We’ll start with trying to make this a Member function

class MyComplex {
    double re, im;

public:
    MyComplex(double r, double i);
    double getReal( ) const;
    double getImag( ) const;
};

MyComplex operator+(const MyComplex&, const MyComplex&);
MyComplex operator-(const MyComplex&, const MyComplex&);
std::ostream& operator<<(std::ostream&, const MyComplex&);
Let’s try and make the ostream function a member

class MyComplex {
    double re, im;

public:
    MyComplex(double r, double i);
    double getReal() const;
    double getImag() const;
    MyComplex operator+(const MyComplex&);
    MyComplex operator-(const MyComplex&);
    std::ostream& operator<<(const MyComplex&);
    std::ostream& operator<<(std::ostream&);
};
This is what the member code would look like if it could work

```cpp
std::ostream& MyComplex::operator<<(const MyComplex& arg) {
    double d1 = arg.getReal( );
    double d2 = arg.getImag( );
    this << "(" << d1 << ", " << d2 << ")" << std::endl;
    return os;
}

std::ostream& MyComplex::operator<<(std::ostream& os) {
    double d1 = getReal( );
    double d2 = getImag( );
    os << "(" << d1 << ", " << d2 << ")" << std::endl;
    return os;
}
```
We need a ostream this pointer

std::cout << myComplexObj;

std::ostream& MyComplex::operator<<(const MyComplex& arg) {
    double d1 = arg.getReal();
    double d2 = arg.getImag();
    this << "(" << d1 << "", " << d2 << ")" << std::endl;
    return os;
}
We need a complex explicit argument

std::cout << myComplexObj;

std::ostream& MyComplex::operator<<(std::ostream& os) {
    double d1 = getReal();
    double d2 = getImag();
    os << "(" << d1 << ", " << d2 << ")" << std::endl;
    return os;
}

The same this pointer problems as before, but compounded by the MyComplex, which will be the first named argument to the function, is passed to an ostream parameter.

What if we need to access private fields, i.e., if re and im were declared as private? Make std::ostream& MyComplex::operator<<(const MyComplex& arg) a friend function
Let’s look at those references a little closer
(GlobalNoRefComplex)

class MyComplex {
    double re, im;

public:
    MyComplex(double r, double i);
    double getReal( ) const;
    double getImag( ) const;
};

MyComplex operator+(const MyComplex&, const MyComplex&);
MyComplex operator-(const MyComplex&, const MyComplex&);
std::ostream& operator<<(std::ostream&, const MyComplex&);
Let’s make these arguments not be references and add a copy constructor so we can see what is going on.

```cpp
MyComplex operator+(const MyComplex, const MyComplex);
MyComplex operator-(const MyComplex, const MyComplex);
```
class MyComplex {
  double re, im;
public:
  MyComplex(double r, double i) : re(r), im(i) { }
  MyComplex(const MyComplex& orig) {
    re = orig.getReal();
    im = orig.getImag();
    cout << "Called copy constructor on (" << re << ", " << im << ")" << endl;
  }
  double getReal() const {return re; }
  double getImag() const {return im; }
};
int main() {
    MyComplex first(3,4);
    MyComplex second(2,9);
    cout << first;
    cout << second;
    cout << first + second;
    cout << first - second;
    return 0;
}

(3, 4)
(2, 9)
Called copy constructor on (2, 9)
Called copy constructor on (3, 4)
(5, 13)
Called copy constructor on (2, 9)
Called copy constructor on (3, 4)
(1, -5)

Using reference parameters eliminates the copies.
Member functions for operator overloading using a reference argument

MyComplex MyComplex::operator+(const MyComplex& arg) const {
    double d1 = re + arg.re;
    double d2 = im + arg.im;
    return MyComplex(d1, d2);
}

MyComplex MyComplex::operator-(const MyComplex& arg) const {
    double d1 = re - arg.re;
    double d2 = im - arg.im;
    return MyComplex(d1, d2);
}
Unary operator, friend/global function

#include <iostream>
using namespace std;
class MyComplex {
private:
    double re, im;
public:
    MyComplex (double r, double i) : re(r), im(i) {} 
    double getReal( ) const {return re;}
    double getImag( ) const {return im;}
    friend MyComplex operator-(const MyComplex& arg);
    friend ofstream& operator<<(ofstream& os, const MyComplex& arg);
};
// global, non-class member overload definition
MyComplex operator-(const MyComplex& arg) {
  return MyComplex(-arg.getReal(), -arg.getImag());
}

// friend overload definition for "<<" as a
// binary operator
ostream& operator<<(ostream& os, const MyComplex& arg) { ... }
#include <iostream>
using namespace std;

class MyComplex {
private:
    double re, im;
public:
    MyComplex (double r, double i) : re(r), im(i) {}  
    double getReal() const {return re;}
    double getImag() const {return im;}
    MyComplex operator-( ) {
        return MyComplex(-re; -im);
    }
};

int main( ) {  
    MyComplex c(3, 4);
    cout << c; // (3,4);
    cout << -c; // (-3, -4);
    return 0;
}
Start with overloadBinaryLocal.cpp, and add the bold function to class MyComplex.

MyComplex operator+(const MyComplex& arg);
MyComplex operator-(const MyComplex& arg);

MyComplex operator-( );

Define the unary operator- as in overloadUnaryLocal.cpp

MyComplex MyComplex::operator-( ) {
    return MyComplex(-re, -im);
}

The new definition of operator- overloads the previous definition of operator-.

The compiler knows which to call based on the signature, i.e. the function name + the arguments to the function.
int main( ) {
    MyComplex first(3,4);
    MyComplex second(2,9);
    cout << first + -first; // (0, 0)
    cout << first + first;  // (6, 8)
    return 0;
}

Operator precedence follows the rules for the built-in operator
Comparison operators < and ==

class Student {
private:
    string sName;
public:
    Student (string s) : sName(s) { }
    bool operator== (const Student& std2) const {
        return (sName == std2.sName);
    }
    bool operator< (const Student& std2) const {
        return sName < std2.sName;
    }
};
int main( ) {
    Student s1("John");
    Student s2("Amy");
    cout << (s1 < s2) << endl; // false 0
    cout << (s2 < s1) << endl; // true 1
    cout << (s2 < s2) << endl; // false 0
    cout << (s2 == s2) << endl; // true 1
    return 0;
}
Summary

• Most binary and unary operators in C++ can be overloaded
  • Specify the function name as operator<op>

• Declaring the arguments as const Type& will keep the object argument from being copied and from being changed (so the function is as safe as if the the object was copied).
• Don't try and change them or even look like you are trying to change them.
  • Should not change the input operator if you do this
  • you will get errors like error: passing ‘const Student’ as ‘this’ argument of ‘void Student::set(std::string)’ discards qualifiers