Operator Overloading

CS3081 Program Design and Development

Overloading

- Class Methods

```
void Robot::radius( int r );   // set Radius
int Robot::radius( );          // get Radius
```

- Functions

```
Vector multiply( Vector, Vector );
Vector multiply( Vector, int );
```

- Operators for User-Defined Types (Classes)

```
Vector a, b;
int c;
Vector d = a * b;
Vector e = a * c;
```

Easier to read.
Quicker to type.
But, not necessary.

Unary and Binary Operators

- Unary
  - { + - & ! ~ ++ -- }
- Binary
  - { + - * / % ^ & | << >>}
- Binary Assignment Operators
  - { ++ -- = += -= *= /= %= ^= &= |= >>= <<=}
- Binary Boolean Operators
  - { == != < > <= >= && || }
- And Then Some ...
  - { -> , ->* () [] }

Operator Overloading Syntax

- Defining

```
{return type} operator{symbol}( arg, [arg] ) {}
Integer operator+( const Integer&, const Integer& );
```

- global *friend* or class method

```
public:
  friend Integer operator+(const Integer&, const Integer&);
  Integer operator-( const Integer& );
```

- # arguments

	unary	**binary**
global class friend	1	2
class method	0	1

- argument and return types

Binary Operator. Member Function.

```
class Modulo100 {
private:
   int i;
   const int mod;
public:
   Modulo100(int ii) : i(ii), mod(100) { i = i % mod; }

   Modulo100 operator%( int mod ){
      return Modulo100( i % mod );
   }
```

```
int main() {

  Modulo100 i(25), j(298), k(-350);
  cout << endl << endl << "i(25) j(298) k(-350)" << endl;
  cout << i.getI() << " " << j.getI() << " " << k.getI() << endl;

  Modulo100 m = k % 20;
  cout << "k % 20: " << m.getI() << endl << endl << endl;
```

Binary Operator. Member Function.

```
class Modulo100 {
private:
   int i;
   const int mod;
public:
   Modulo100(int ii) : i(ii), mod(100) { i = i % mod; }

   Modulo100 operator%( int mod ){
      return Modulo100( i % mod );
   }
```

Overloading the modulo operator %.
Binary Operator.
Member Function = 1 argument.

Built-in modulo operator for int's .

Returning a newly constructed Modulo100.

```
int main() {

  Modulo100 i(25), j(298), k(-350);
  cout << endl << endl << "i(25) j(298)
  cout << i.getI() << " " << j.getI() << " " << k.getI() << endl;

  Modulo100 m = k % 20;
  cout << "k % 20: " << m.getI() << endl << endl << endl;
```

```
i(25) j(298) k(-350)
25 98 -50
k % 20: -10
```

Unary as Member. Binary as Global *friend*.

```
const Modulo100 operator-() const {
  return Modulo100(-i);
}

friend const Modulo100 operator-(const Modulo100& left, const Modulo100& right) {
  cout << "operator-" << endl;
  return ( left + -right );
}
```

Unary as Member. Binary as Global *friend*.

```
const Modulo100 operator-() const {
  return Modulo100(-i);
}

friend const Modulo100 operator-(const Modulo100& left, const Modulo100& right) {
  cout << "operator-" << endl;
  return ( left + -right );
}
```

Overloading the sign operator -.
Unary Operator.
Member Function = 0 arguments.

Overloading the subtraction operator -.
Binary Operator.
Global Function = 2 arguments.

Using the sign operator defined above.

Using the add operator (not yet shown).

```
-x(99) = -99
operator-
operator+
x(99) - y(-99) = 98
```

```
Modulo100 x(99), y(-99), z(99), w(-99);
cout << endl << endl << "-x(99) = " << -x << endl;
cout << "x(99) - y(-99) = " << x-y << endl << endl;
```

Binary Member Function. Binary Global *friend*.

```
const Modulo100 operator+(const Modulo100& right) const {
  cout << "operator+" << endl;
  return Modulo100( i + right.i );
}
```

```
friend const Modulo100 operator-(const Modulo100& left, const Modulo100& right) {
  cout << "operator-" << endl;
  return ( left + -right );
}
```

Overloading subtraction operator -.
Binary Operator.
Global Function = 2 arguments.

Overloading addition operator +.
Binary Operator.
Member Function = 1 argument.

Binary Member Function. Binary Global *friend*.

Binary Member Function. Left operand is “this.”

x + y is analogous to:

```
x.add(y) {
    return x.i + y.i ;
}
```

```
const Modulo100 operator+(const Modulo100& right) const {
  cout << "operator+" << endl;
  return Modulo100( i + right.i );
}
```

```
friend const Modulo100 operator-(const Modulo100& left, const Modulo100& right) {
  cout << "operator-" << endl;
  return ( left + -right );
}
```

Overloading subtraction operator -.
Binary Operator.
Global Function = 2 arguments.

Overloading addition operator +.
Binary Operator.
Member Function = 1 argument.

PreFix / PostFist

```cpp
// prefix (unary). global friend.
friend const Modulo100& operator++(Modulo100& operand) {
  cout << "++Modulo100 " << operand.i << endl;
  operand.i = (operand.i + 1) % operand.mod;
  return operand;
}

// prefix (unary). member function.
const Modulo100& operator--() {
  cout << "--Modulo100 " << i << endl;
  i = (i - 1) % mod;
  return *this;
}

// postfix (binary). global friend.
friend const Modulo100 operator++(Modulo100& operand, int blank) {
  cout << "Modulo100++ " << operand.i << endl;
  Modulo100 before(operand.i);
  operand.i = (operand.i + 1) % operand.mod;
  return before;
}

// postfix (binary). member function.
const Modulo100 operator--(int blank) {
  cout << "Modulo100-- " << i << endl;
  Modulo100 before(i);
  i = (i - 1) % mod;
  return before;
}
```

```cpp
Modulo100 x(99), y(-99), z(99), w(-99);
cout << "++x " << ++x << " x " << x << end
cout << "--y " << --y << " y " << y << end
cout << "z++ " << z++ << " z " << z << end
cout << "w-- " << w-- << " w " << w << end
```

```
++Modulo100 99
++x 0 x 0
--Modulo100 -99
--y 0 y 0
Modulo100++ 99
z++ 99 z 0
Modulo100-- -99
w-- -99 w 0
```

Return Types

```
void operator-=(const Modulo100& right) {
  cout << "operator-=" << endl;
  i = (i - right.i) % mod;
}

Modulo100& operator+=(const Modulo100& right) {
  cout << "operator+=" << endl;
  i = (i + right.i) % mod;
  return *this;
}

const Modulo100& operator*=(const Modulo100& right) {
  cout << "operator*=" << endl;
  i = (i * right.i) % mod;
  return *this;
}

Modulo100* operator/=(const Modulo100& right) {
  cout << "operator/=" << endl;
  i = ((int) (i / right.i)) % mod;
  return this;
}
```

```
cout << ( i -= i ) << endl;
cout << ( j += j ) << endl;
cout << ( k *= k ) << endl;
cout << ( i /= i ) << endl;
```

Which of These
cout Will Work ?

```
friend ostream& operator<<(ostream& os, const Modulo100& rv) {
  return os << rv.i;
}
```

Return Types

```
void operator-=(const Modulo100& right) {
  cout << "operator-=" << endl;
  i = (i - right.i) % mod;
}

Modulo100& operator+=(const Modulo100& right) {
  cout << "operator+=" << endl;
  i = (i + right.i) % mod;
  return *this;
}

const Modulo100& operator*=(const Modulo100& right) {
  cout << "operator*=" << endl;
  i = (i * right.i) % mod;
  return *this;
}

Modulo100* operator/=(const Modulo100& right) {
  cout << "operator/=" << endl;
  i = ((int) (i / right.i)) % mod;
  return this;
}
```

```
cout << ( i -= i ) << endl;
cout << ( j += j ) << endl;
cout << ( k *= k ) << endl;
cout << ( i /= i ) << endl;
```

```
compile error ( j -= j )
operator+= 2
operator*= 4
operator/= 0x7fff5fbff9d0
```

```
friend ostream& operator<<(ostream& os, const Modulo100& rv) {
  return os << rv.i;
}
```

Operator Overloading Syntax

- Defining

friend

```
{return type} operator{symbol}( arg, [arg] ) {}
Integer operator+( const Integer&, const Integer& );
```

- global *friend* or class method

public:

```
friend Integer operator+(const Integer&, const Integer&);
Integer operator-( const Integer& );
```

- # arguments

	unary	binary
global class friend	1	2
class method	0	1

- argument and return types

Overloading "Rules"

Operator	Recommended Use
All unary operators	member
= () [] -> ->*	MUST be member
+= -= /= ...	member
All other binary operators	non-member (global friend)

Arguments and Return Values (Recommendations Only)

- argument type when read-only pass as const reference, else reference.
- return type when value is expected pass back const new object.
- return type when (possibly) used as lvalue pass back nonconst reference.
- return type when logical operators pass back bool

Your Turn

```
class Position {
private:
   int x;
   int y;
public:

   Position( ) : x(0), y(0) {}
   Position( int inX, int inY ) : x(inX), y(inY) {}
```

You write
- Binary Addition operator+
- Unary Sign operator-

```
cout << "pos1 " << pos1 << "  pos2 " << pos2 << endl;
cout << "pos1 + pos2 = " << ( pos1 + pos2 ) << endl;
cout << "-pos2 = " << ( -pos2 ) << endl;
```

```
pos1 [100,100]  pos2 [-50,-25]
pos1 + pos2 = [50,75]
-pos2 = [50,25]
```

```
// unary member operators (0 arguments)
const Position operator-() const {
  return Position( -x, -y );
}

// binary member operators (1 argument)
const Position operator+(const Position& rp) const {
  return Position( x+rp.x, y+rp.y );
}
```

Mixed Type Operands. Global friends.

```
// global binary mixed-type operands (2 arguments)
friend ostream& operator<<(ostream& os, const Position& rp) {
  return os << "[" << rp.x << "," << rp.y << "]";
}
```

If there is 1 operator you are going to overload, << is it!

cout << myClass ;

Note that this is not ...

ostream& operator<<(ostream& os);

because, that would mean ...

myClass << cout;

Mixed Operands

```
class Position {
private:
  int x;
  int y;
public:

  Position( ) : x(0), y(0) {}
  Position( int inX, int inY ) : x(inX), y(inY) {}
```

```
class PositionFloat {
private:
  float x;
  float y;
public:
  PositionFloat(float inX=0.0, float inY=0.0 ) : x(inX), y(inY) {}
```

Wouldn’t it be nice to …

```
Position pos(10,10);
PositionFloat posF(2.3, 5.8);
cout << ( pos + posF ) << endl;
cout << ( posF + pos ) << endl;
```

Writing Reciprocal Overloaded Operators

Defined Inside of class Position

Position on Left. PositionFloat on Right

```
PositionFloat operator+( PositionFloat& pf);
```

```
PositionFloat Position::operator+( PositionFloat& pf ) {
  PositionFloat pif( x, y );
  return (pf + pif) ;
}
```

```
cout << ( pos + posF ) << endl;
```

```
Position pos(10,10);
PositionFloat posF(2.3, 5.8);
```

```
cout << ( posF + pos ) << endl;
```

Defined Inside of class PositionFloat

```
PositionFloat operator+( Position& pi);
```

```
PositionFloat PositionFloat::operator+( Position& pi ) {
  PositionFloat pif( x+pi.getX(), y+pi.getY() );
  return pif;
}
```

PositionFloat on Left. Position on Right

Writing Reciprocal Overloaded Operators

Defined Inside of class Position

```
PositionFloat operator+( PositionFloat& pf);
```

```
PositionFloat Position::operator+( Positio
  PositionFloat pif( x, y );
  return (pf + pif) ;
}
```

Defined Inside of cla

```
PositionFl                    ion& pi);
```

```
PositionF              t::operator+( Position& pi ) {
  PositionF             +pi.getX(), y+pi.getY() );
  return pif
}
```

That's a pain. There is an easier way.

Automatic Type Conversion with Special Constructor

```
class Pos {
private:
  int x;
  int y;
public:
  Pos( ) : x(0), y(0) {}
  Pos( int inX, int inY ) : x(inX), y(inY) {}
```

Specialized constructor for PosFloat. This creates a new temporary PosFloat object based on passed Pos object.

```
class PosFloat {
private:
  float x;
  float y;
public:
  PosFloat(float inX=0.0, float inY=0.0 ) : x(inX), y(inY) {}
  PosFloat( Pos p );
```

```
PosFloat::PosFloat( Pos p ) {
  cout << "in conversion from posFloat to pos" << endl;
  x = p.getX();
  y = p.getY();
}
```

Automatic Type Conversion with Special Constructor

```
PosFloat::PosFloat( Pos p ) {
  cout << "in conversion from posFloat to pos" << endl;
  x = p.getX();
  y = p.getY();
}
```

```
friend const PosFloat operator+(const PosFloat& left, const PosFloat& right) {
  return PosFloat( left.x+right.x, left.y+right.y );
}
```

```
Pos pos1(10,20);
PosFloat posF1(2.8,3.6);
cout << pos1 << " + " << posF1 << " = " << ( pos1 + posF1 ) << endl;
cout << posF1 << " + " << pos1 << " = " << ( posF1 + pos1 ) << endl;
```

```
in conversion from posFloat to pos
Pos::operator<< [10,20] + [2.8,3.6] = [12.8,23.6]
in conversion from posFloat to pos
[2.8,3.6] + Pos::operator<< [10,20] = [12.8,23.6]
```

Automatic Type Conversion with Special Operator

```
PosFloat::PosFloat( Pos p ) {
  cout << "in conversion from posFloat to pos" << endl;
  x = p.getX();
  y = p.getY();
}
```

Special constructor. Defined in final type.

```
friend const PosFloat operator+(const PosFloat& left, const PosFloat& right) {
  return PosFloat( left.x+right.x, left.y+right.y );
}
```

```
class Vector {
private:
  int x;
  int y;
public:

  Vector(int xx, int yy) : x(xx), y(yy) {}

  operator Pos() const {
    cout << "in conversion from Vector to Pos" << endl;
    return Pos(x,y); }
};
```

Special operator. Defined in type being cast.

Creates a new Position object using Vector object values.

Automatic Type Conversion with Special Operator

```
PosFloat::PosFloat( Pos p ) {
  cout << "in conversion from posFloat to pos" << endl;
  x = p.getX();
  y = p.getY();
}
```

Special constructor. Defined in final type.

```
in conversion from Vector to Pos
pos2 + v = Pos::operator<< [17,29]
in conversion from Vector to Pos
v + pos2 = Pos::operator<< [17,29]
```

```
class Vector {
private:
   int x;
   int y;
public:

   Vector(int xx, int yy) : x(xx), y(yy) {}

   operator Pos() const {
     cout << "in conversion from Vector to Pos" << endl;
     return Pos(x,y); }
};
```

Special operator. Defined in type being cast.

```
Vector v( 7, 9);
Pos pos2( 10, 20 );

cout << "pos2 + v = " << ( pos2 + v ) << endl;
cout << "v + pos2 = " << ( v + pos2 ) << endl;
```

Automatic Type Conversion with Special Operator

```
PosFloat::PosFloat( Pos p ) {
  cout << "in conversion from posFloat to pos" << endl;
  x = p.getX();
  y = p.getY();
}
```

This requires adding a constructor to an existing class.
(Violates open to extension, closed to change.)

```
class Vector {
private:
  int x;
  int y;
public:

  Vector(int xx, int yy) : x(xx), y(yy) {}

  operator Pos() const {
    cout << "in conversion from Vector to Pos" << endl;
    return Pos(x,y); }
};
```

This requires adding operator to class you are adding.
(Open to extension, closed to change.)

Copy Constructor

- Operator overloading of = is related to copy constructor.
- Passing arguments by value requires a copy.
- Compiler writes a copy constructor that makes a bitwise copy.
- Bitwise copy doesn’t always work. You probably want to write your own logical copy.

HowMany ?

```
class HowMany {
private:
  static int objectCount;
public:
  HowMany() { objectCount++; }

  static void print(const string& msg = "") {
    if (msg.size() != 0) cout << msg << ": ";
    cout << "objectCount = " << objectCount << endl;
  }

  ~HowMany() {
    --objectCount;
    print("~HowMany()");
  }
};
```

Constructor : objectCount++
Destructor : objectCount--

Passing by Value.

```
HowMany f( HowMany x ) {
  x.print("x arg inside f()");
  return x;
}

int main() {
  HowMany h;
  HowMany::print("after construction of h");
  HowMany h2 = f(h);
  HowMany::print("after call to f()");
}
```

HowMany? Constructor is not being called.

```
class HowMany {
private:
  static int objectCount;
public:
```

Constructor : objectCount++
Destructor : objectCount--

```
after construction of h: objectCount = 1
x arg inside f(): objectCount = 1
~HowMany(): objectCount = 0
after call to f(): objectCount = 0
~HowMany(): objectCount = -1
~HowMany(): objectCount = -2
```

```
dl;
```

```
  ~HowMany() {
    --objectCount;
    print("~HowMany()");
  }
};
```

```
HowMany f( HowMany x ) {
  x.print("x arg inside f()");
  return x;
}

int main() {
  HowMany h;
  HowMany::print("after construction of h");
  HowMany h2 = f(h);
  HowMany::print("after call to f()");
}
```

HowMany? Using Copy-Constructor

Constructor : objectCount++
Copy-Constructor : objectCount++
Destructor : objectCount--

```
class HowMany {
private:
  static int objectCount;
public:
  HowMany() { objectCount++; }

  static void print(const string& msg = "") {
    if (msg.size() != 0) cout << msg << ": ";
    cout << "objectCount = " << objectCount << endl;
  }

  ~HowMany() {
    --objectCount;
    print("~HowMany()");
  }
```

```
HowMany(const HowMany& h) {
  ++objectCount;
  print("HowMany(const HowMany&");
}
```

```
after construction of h: objectCount = 1
HowMany(const HowMany&: objectCount = 2
x arg inside f(): objectCount = 2
HowMany(const HowMany&: objectCount = 3
~HowMany(): objectCount = 2
after call to f(): objectCount = 2
~HowMany(): objectCount = 1
~HowMany(): objectCount = 0
```

Copy-Constructor syntax:
ClassName(const ClassName& varName);

Copy-Constructor and Pointers

```
class Pointers {
private:
  int* a;
  int* b;
public:
  Pointers( int A, int B );
  void setA( int A ) { *a = A; }
```

```
Pointers::Pointers( int A, int B ) {
  a = new int(A);
  b = new int(B);
}
```

Passing by Value BUT problem is the assignment.

```
void myFunc( Pointers p2 ) {
  p2.setA(12);
}

int main() {
  Pointers p1(10,10);
  Pointers p2 = p1;
  cout << p1 << endl;
  myFunc(p2);
  cout << p1 << endl;
}
```

```
p1: a 10 b 10
After myFunc(p2)...
p1: a 12 b 10
```

Compiler defined this.

Copy Constructor and Pointers

```
class Pointers {
private:
  int* a;
  int* b;
public:
  Pointers( int A, int B );
  void setA( int A ) { *a = A; }
```

```
Pointers::Pointers( int A, int B ) {
  a = new int(A);
  b = new int(B);
}
```

```
void myFunc( Pointers p2 ) {
  p2.setA(12);
}

int main() {
  Pointers p1(10,10);
  Pointers p2 = p1;
  cout << p1 << endl;
  myFunc(p2);
  cout << p1 << endl;
}
```

```
p1: a 10 b 10
in copy-constructor a 10 b 10
After myFunc(p2)...
p1: a 10 b 10
```

```
Pointers(const Pointers& p);
```

```
Pointers::Pointers( const Pointers& p ) {
  a = new int(*p.a);
  b = new int(*p.b);
  cout << "in copy-constructor ";
  cout << p << endl;
}
```

Polymorphism

Polymorphism: generally defined as "the ability to create a variable, a function, or an object that has more than one form." The result is that you get different behavior (i.e. different pieces of code are executed) depending on the type of object or objects that are being acted upon.

- **Operator Overloading**: One operator can be applied to different types.
- **Method Overriding (Ad-hoc polymorphism)**: Derived class redefining base class method.
- **Method Overloading (Ad-hoc polymorphism)**: Multiple function definitions with different parameter lists.
- **Subtype Polymorphism**: Upcasting – derived class object can be used in place of base class object.
- **Parametric Polymorphism**: Templates – one function with same behavior across multiple types. (Stack of ints, strings, ClassA, ...)

cite: Wikipedia and http://www.catonmat.net/blog/cpp-polymorphism/