Operator Overloading

Operators in C++ may be overloaded in the same way that functions are overloaded. In C, the + (plus) operator is "overloaded" to work for int or float values. In C++, this concept is extended to include class types.

Notes:

- You may overload the following operators:

  ```
  +   -   *   /   %   ^   &   |
  ~   !   ,   =   <   >   <=  >=
  ++  --  <<  >>  ==  !=  &&  ||
  +=  -+  /=  %=  ^=  &=  |=  *=
  <<= >>= []  ()  ->  ->* new  delete
  ```

- Most operators may be overloaded, both binary and unary operators. The following operators may not be overloaded:

  ```
  .     direct member
  .*    direct pointer to member
  ::    scope resolution
  ? :   ternary
  ```

- To overload an operator, create a function called operator@ where @ is the operator symbol you wish to overload.

- Operator precedence is still in effect for overloaded operators and may not be changed.

- Default arguments are not allowed in overloaded operator functions.

- For an expression involving binary operators, A + B means:

  ```
  A.operator+(B)        if operator+() is a class member function
  operator+(A,B)        if operator+() is a non-class member function
  ```

- An overloaded operator function may be defined as a class member function, a friend function, or even a non-friend function.

- You may not overload an operator (redefine) for the built-in primitive types. In other words, if a and b are ints, then a+b will always be (int) a+b.

- You may not create any new operator symbols

Example 6-3 - Fraction class with overloaded + and ! operators

```
1  // File: ex6-3.cpp overloaded + and ! operator for fraction class
2
3  #include <iostream>
4  using namespace std;
5
6  class fraction
7  {
8    private:
9        int numer;
10       int denom;
11   public:
12       fraction(int n = 0, int d = 1);
13       void operator!(void) const;
14       fraction operator+(const fraction&);
15 };
16
17 fraction::fraction(int n, int d)
18 {
19   numer = n;
20   denom = d;
21 }
22
23 void fraction::operator!(void) const
24 {
25   cout << numer << '/' << denom << endl;
26   return;
27 }
28
29 fraction fraction::operator+(const fraction& f2)
30 {
31   fraction temp(0,0);
32   temp.numer = numer * f2.denom + f2.numer * denom;
33   temp.denom = denom * f2.denom;
34   return temp;
35 }
36
37 int main(void)
38 {
39   fraction f(3,4);
40   fraction g(2,3);
41   fraction h = f + g;         // Do you need a default ctor here?
42   !h;                                                  // prints 17/12
43
44   return 0;
45 }
```

✔ In this example operator+ returns a fraction by value. Is it possible or appropriate to have the function return by reference or have a void return?

- Line 41: What is the difference between ***fraction h = f + g;*** and ***fraction h(f+g);***

In this example, operator+ is defined as a friend function.

Example 6-4 - A friendly overloaded +

```
1  #include <iostream>
2  using namespace std;
3
4  class fraction {
5    private:
6       int numer, denom;
7    public:
8      fraction(int n = 0, int d = 1);
9      void operator!(void) const;
10    friend  fraction operator+(const fraction&,const fraction&);
11 };
12
13
14 fraction::fraction(int n, int d) {
15   numer = n;
16   denom = d;
17 }
18
19 void fraction::operator!(void) const {
20   cout << numer << '/' << denom << endl;
21 }
22
23 // fraction friend function
24 fraction operator+(const fraction& f1,const fraction& f2) {
25   fraction temp(f1.numer *f2.denom + f2.numer * f1.denom,
26      f1.denom * f2.denom);
27   return temp;
28 }
29
30
31 int main(void) {
32   fraction f(3,4);
33   fraction g(2,3);
34   fraction h = f + g;
35   !f;
36   !g;
37   !h;
38   return 0;
39 }
```

```
******  Output  ******

3/4
2/3
17/12
```

✔ What's the better approach, example 6-3 or example 6-4?

This example demonstrates a more "complete" set of overloaded operators for the fraction class. Notice that all operators are specified as member functions

Example 6-5 - The Overloaded fraction class

```
1  // File: ex6-5.cpp
2  #include <iostream>
3  #include <cassert>
4  using namespace std;
5
6  class fraction {
7        int numer, denom;
8    public:
9        fraction(int = 0, int = 1);
10       void operator!(void) const;       // print the fraction
11       fraction& operator~(void);        // reduce the fraction
12       fraction operator-(void) const;   // negative of fraction
13       fraction operator*(void) const;   // reciprocal of fraction
14       fraction& operator+=(const fraction&);
15       fraction& operator-=(const fraction&);
16       fraction& operator*=(const fraction&);
17       fraction& operator/=(const fraction&);
18       fraction operator+(int) const;
19       fraction operator-(int) const;
20       fraction operator*(int) const;
21       fraction operator/(int) const;
22       bool operator>(const fraction&) const;
23       bool operator<(const fraction&) const;
24       bool operator>=(const fraction&) const;
25       bool operator<=(const fraction&) const;
26       bool operator==(const fraction&) const;
27       bool operator!=(const fraction&) const;
28       fraction operator+(const fraction&) const;
29       fraction operator-(const fraction&) const;
30       fraction operator*(const fraction&) const;
31       fraction operator/(const fraction&) const;
32       fraction& operator++();  // prefix op returns by ref
33       fraction operator++(int);      // postix op returns by value
34 };
35
36 // member function definitions
37 fraction::fraction(int n, int d) {
38   assert(d != 0);
39   numer = n;
40   denom = d;
41 }
42
43 // print the fraction
44 void fraction::operator!(void) const {
45   cout << numer << '/' << denom << endl;
46 }
```

```
47
48  // reduce the fraction
49  fraction& fraction::operator~(void) {
50  int min;
51  // find the minimum of the denom and numer
52    min = denom < numer ? denom : numer;
53    for (int i = 2; i <= min; i++) {
54        while ((numer % i == 0) && (denom % i == 0)) {
55            numer /= i;
56            denom /= i;
57        }
58    }
59    return *this;
60  }
61
62  // negate the fraction
63  fraction fraction::operator-(void) const {
64    return fraction(-numer,denom);
65  }
66
67  // fraction reciprocal
68  fraction fraction::operator*(void) const {
69    return fraction(denom,numer);
70  }
71
72  fraction& fraction::operator+=(const fraction& f) {
73    numer = numer*f.denom+denom*f.numer;
74    denom = denom*f.denom;
75    return *this;
76  }
77
78  fraction& fraction::operator-=(const fraction& f) {
79    *this += (-f);
80    return *this;
81  }
82
83  fraction& fraction::operator*=(const fraction& f) {
84    numer = numer*f.numer;
85    denom = denom*f.denom;
86    return *this;
87  }
88
89  fraction& fraction::operator/=(const fraction& f) {
90    *this *= (*f);
91    return *this;
92  }
93
94  bool fraction::operator>(const fraction& f) const {
95    return (float) numer/denom > (float) f.numer/f.denom;
96  }
97
98  bool fraction::operator<(const fraction& f) const {
```

```
99    return f>*this;
100 }
101
102 bool fraction::operator==(const fraction& f) const {
103   return numer*f.denom == denom*f.numer;
104 }
105
106 bool fraction::operator!=(const fraction& f) const {
107   return !(*this == f);
108 }
109
110 bool fraction::operator<=(const fraction& f) const {
111   return !(*this > f);
112 }
113
114 bool fraction::operator>=(const fraction& f) const {
115   return !(*this<f);
116 }
117
118 fraction fraction::operator+(const fraction& f) const {
119   return fraction(numer*f.denom+denom*f.numer,denom*f.denom);
120 }
121
122 fraction fraction::operator-(const fraction& f) const {
123   return fraction(numer*f.denom-denom*f.numer,denom*f.denom);
124 }
125
126 fraction fraction::operator*(const fraction& f) const {
127   return fraction(numer*f.numer,denom*f.denom);
128 }
129
130 fraction fraction::operator/(const fraction& f) const {
131   return (*this) * (*f);
132 }
133
134 fraction fraction::operator+(int i) const {
135   return fraction(numer+i*denom,denom);
136 }
137
138 fraction fraction::operator-(int i) const {
139   return (*this) + -i;
140 }
141
142 fraction fraction::operator*(int i) const {
143   return fraction(numer*i,denom);
144 }
145
146 fraction fraction::operator/(int i) const {
147   return fraction(numer,i*denom);
148 }
149
150 // prefix increment operator
```

```
151 fraction& fraction::operator++() {
152  numer += denom;
153  return *this;
154 }
155
156 // postfix increment operator
157 fraction fraction::operator++(int) {   // Note dummy int argument
158   fraction temp = *this;
159   ++(*this);                        // call the prefix operator
160   return temp;
161 }
162
163
164 int main(void)
165 {
166   fraction f(3,4);           // initialize fraction f & g
167   fraction g(1,2);
168   cout << "!f ";  !f;
169   cout << "!g ";  !g;
170   cout << endl;
171   cout << "-g ";  !-g;
172   cout << "*g ";  !*g;
173   fraction h = g + f;
174   cout << endl;
175   cout << "h=g+f " << " !h ";  !h;
176   cout << "!~h ";  !~h;
177   cout << endl;
178   cout << "f+g ";  ! (f + g);
179   cout << "f-g ";  ! (f - g);
180   cout << "f*g ";  ! (f * g);
181   cout << "f/g ";  ! (f / g);
182   cout << endl;
183   cout << "f+=g "; !~(f+=g);
184   cout << "f-=g "; !~(f-=g);
185   cout << "f*=g "; !~(f*=g);
186   cout << "f/=g "; !~(f/=g);
187   cout << endl;
188   cout << "f<g " << (f<g) << endl;
189   cout << "f>g " << (f>g) << endl;
190   cout << "f==g " << (f==g) << endl;
191   cout << "f!=g " << (f!=g) << endl;
192   cout << "f<=g " << (f<=g) << endl;
193   cout << "f>=g " << (f>=g) << endl;
194   cout << endl;
195   cout << "f+5 ";  !(f+5);
196   cout << "f-5 ";  !(f-5);
197   cout << "f*5 ";  !(f*5);
198   cout << "f/5 ";  !(f/5);
199   cout << endl;
200   cout << "f+=5 "; f+=5; cout << "!~f "; !~f;     // What's this?
201   cout << "++f "; !++f; cout << "f="; !f;
202   cout << "f++ "; !f++; cout << "f="; !f;
```

```
203
204     return 0;
205   }
```

```
******  Output  ******

!f 3/4
!g 1/2

-g -1/2
*g 2/1

h=g+f  !h 10/8
!~h 5/4

f+g 10/8
f-g 2/8
f*g 3/8
f/g 6/4

f+=g 5/4
f-=g 3/4
f*=g 3/8
f/=g 3/4

f<g 0
f>g 1
f==g 0
f!=g 1
f<=g 0
f>=g 1

f+5 23/4
f-5 -17/4
f*5 15/4
f/5 3/20

f+=5 !~f 23/4
++f 27/4
f=27/4
f++ 27/4
f=31/4
```

Should any of these member functions be specified as friend functions?

Why do operator~ and unary operator- have different return types?

How do the increment operators work?

Example 6-6 - "More power"

```
1  // File: ex6-6.cpp
2
3  #include <iostream>
4  #include <cstdlib>
5  #include <cmath>
6  using namespace std;
7
8  class Integer
9  {
10   private:
11    long x;
12   public:
13    Integer(long i) { x = i;}
14    long operator^(int);
15 };
16
17 long Integer::operator^(int power)
18 {
19   if (power == 0) return 1;
20   long temp = x;
21   for (int i = 1; i < power; i++) temp *= x;
22   return temp;
23 }
24
25
26 class Real
27 {
28    double d;
29   public:
30    Real(double arg) { d = arg;}
31    double operator^(double);
32 };
33
34 double Real::operator^(double power)
35 {
36   if (d == 0 && power == 0)
37   {
38     cout << "0 ^ 0 is undefined\n";
39     exit (1);
40   }
41
42   if (d < 0 && power != floor(power))
43   {
44     cout <<
45         "You may only take integer powers of negative numbers\n";
46     exit (1);
47   }
48
49  return pow(d,power);
50 }
```

```
51  int main (void)
52  {
53    Integer z(2), y(3);
54    cout << (z^5) << endl;
55    cout << (y^0) << endl;
56    Real r1(3.14), r2(6.02e23), r3(1.2345);
57    cout<< (r1^2) << endl;
58    cout<< (r2^3) << endl;
59    cout<< (r3^0) << endl;
60    cout<< (r1^3.14) << endl;
61    Real r4(-1.4);
62    cout << (r4^3) << endl;
63    cout << (r4^1.3) << endl;
64
65    return 0;
66  }
```

```
******  Output  ******

32
1
9.8596
2.18167e+71
1
36.3378
-2.744
```

You may only take integer powers of negative numbers

What if you want to evaluate an expression like πr^2 ? Is this correct?

```
3.141592654*r^2
```

Unary vs Binary, Member vs.Non-Member

Only two types of overloaded operator functions may exist, unary or binary, and they may be defined as member or non-member functions. So, there are only four ways to define these functions. The following table summarizes these possibilities. Assume @ represents an overloaded operator. There is, of course, no such operator available in C++.

		Unary Operator	**Binary Operator**
Member function	prototype	**? operator@();**	**? operator@(Arg);**
	Functional notation call	Arg1.operator@() [1]	Arg1.operator@(Arg2)[1]
	Infix notation call	@Arg1[1]	Arg1 @ Arg2[1]
Non-member function	prototype	**? operator@(Arg1);** [1]	**? operator@(Arg1,Arg2);**[2]
	Functional notation call	operator@(Arg1) [1]	operator@(Arg1,Arg2) [2]
	Infix notation call	@Arg1 [1]	Arg1 @ Arg2 [2]

[1] Arg1 would have to be a class object
[2] Either Arg1 or Arg2 or both would have to be a class object

Example 6-7 - Matrix Arithmetic

The following example is an implementation of Matrix addition. It is meant to demonstrate the overloaded + and = operators. This example also uses the this operator in member functions, so that objects can be followed in the program using the program output.

```
1  // File: ex6-7.cpp
2
3  #include <iostream>
4  #include <cstdlib>
5  using namespace std;
6
7  class Matrix
8  {
9  private:
10      int** element;
11      int rows;
12      int cols;
13      void alloc(void);
14      void release(void);
15  public:
16      Matrix(int = 0, int = 0);  // also default constructor
17      Matrix(const Matrix&); // copy constructor
18      ~Matrix();
19      void print(void) const;
20      Matrix operator+(const Matrix&) const;
21      Matrix& operator=(const Matrix&);
22  };
23
24  Matrix::Matrix(int r, int c) : rows(r), cols(c)
25  {
26      cout << "Constructor called for object " << this <<endl;
27      alloc();
28
29      // initialize Matrix elements with random numbers 0-9
30      for (int i = 0; i < rows; i++)
31          for (int j = 0; j < cols; j++)
32              element[i][j] = rand()%10;
33  }
34
35  Matrix::Matrix(const Matrix& arg) : rows(arg.rows), cols(arg.cols)
36  {
37      cout << "\nIn copy constructor for object " << this;
38      cout << ", argument: " << &arg << endl;
39
40      alloc();
41      for (int i = 0; i < rows; i++)
42          for (int j = 0; j < cols; j++)
43              element[i][j] = arg.element[i][j];
44  }
45
```

```
46  Matrix::~Matrix(void)
47  {
48      cout << "\n~~ Destructor called for object: " << this << endl;
49
50      release();
51  }
52
53  void Matrix::alloc(void)       // allocate heap memory for elements
54  {
55      cout << "Allocate heap memory for Matrix " << this << "
    elements\n";
56
57      element = new int*[rows];
58      for (int i = 0; i < rows; i++)
59          element[i] = new int[cols];
60  }
61
62  void Matrix::release(void)
63  {
64      cout << "I got rid of Matrix " << this << "'s elements\n";
65
66      for (int i = 0; i < rows; i++)
67          delete [] element[i];
68      delete [] element;
69  }
70
71  void Matrix::print(void) const
72  {
73      cout << "\nMatrix values for object: "<< this << endl;
74
75      for (int i = 0; i < rows; i++)
76      {
77          for (int j = 0; j < cols; j++)
78              cout << element[i][j] << '\t';
79          cout << endl;
80      }
81  }
82
83  Matrix Matrix::operator+(const Matrix& arg) const
84  {
85      cout << "\nExecuting operator+ for object: " << this;
86      cout << ", argument: " << &arg << endl;
87
88      if (rows != arg.rows || cols != arg.cols)
89      {
90          cerr << "Invalid Matrix addition\n";
91          return (*this);
92      }
93
94      Matrix temp(rows,cols);
95
96      for (int i = 0; i < rows; i++)
```

```
97          for (int j = 0; j < cols; j++)
98              temp.element[i][j] = element[i][j] + arg.element[i][j];
99
100     temp.print();
101     return temp;
102 }
103
104 Matrix& Matrix::operator=(const Matrix& arg)
105 {
106     cout << "\nExecuting operator= for object: " << this;
107     cout << ", argument: " << &arg << endl;
108
109     // Make sure rows and cols match the argument
110     if (rows != arg.rows || cols != arg.cols)
111     {
112         release();
113         rows = arg.rows;
114         cols = arg.cols;
115         alloc();
116     }
117
118     for (int i = 0; i < arg.rows; i++)
119         for (int j = 0; j < arg.cols; j++)
120             element[i][j] = arg.element[i][j];
121
122     return *this;
123 }
124
125 int main(void)
126 {
127     Matrix A(3,4), B(3,4), C;
128     A.print();
129     B.print();
130     C.print();
131     C = A + B;
132     C.print();
133 }
```

****** OUTPUT ******

```
Constructor called for object 0x28fee8
Allocate heap memory for Matrix 0x28fee8 elements
Constructor called for object 0x28fedc
Allocate heap memory for Matrix 0x28fedc elements
Constructor called for object 0x28fed0
Allocate heap memory for Matrix 0x28fed0 elements

Matrix values for object: 0x28fee8
1       7       4       0
9       4       8       8
2       4       5       5
```

```
Matrix values for object: 0x28fedc
1       7       1       1
5       2       7       6
1       4       2       3

Matrix values for object: 0x28fed0

Executing operator+ for object: 0x28fee8, argument: 0x28fedc
Constructor called for object 0x28fe3c
Allocate heap memory for Matrix 0x28fe3c elements

Matrix values for object: 0x28fe3c
2       14      5       1
14      6       15      14
3       8       7       8

In copy constructor for object 0x28fef4, argument: 0x28fe3c
Allocate heap memory for Matrix 0x28fef4 elements

~~ Destructor called for object: 0x28fe3c
I got rid of Matrix 0x28fe3c's elements

Executing operator= for object: 0x28fed0, argument: 0x28fef4
I got rid of Matrix 0x28fed0's elements
Allocate heap memory for Matrix 0x28fed0 elements

~~ Destructor called for object: 0x28fef4
I got rid of Matrix 0x28fef4's elements

Matrix values for object: 0x28fed0
2       14      5       1
14      6       15      14
3       8       7       8

~~ Destructor called for object: 0x28fed0
I got rid of Matrix 0x28fed0's elements

~~ Destructor called for object: 0x28fedc
I got rid of Matrix 0x28fedc's elements

~~ Destructor called for object: 0x28fee8
I got rid of Matrix 0x28fee8's elements
```

✔ How would you change the operator=() function so that you could assign a matrix with a different number of rows or columns?

Is it a Copy Constructor or a Default Constructor and an Assignment Operator?

If you instantiate a class object x using an existing object y, such as,

Test x(y);

It is assumed that the copy constructor is called to perform the creation of the object. Further, if you use the syntax,

Test x = y;

Then, the same copy constructor is called. Is that correct, or is the default constructor called, then the assignment operator? Consider the following example.

Example 6-7a - Copy Constructor or a Default Constructor and an Assignment Operator?

```
134  // File: Ex6-7a.cpp –
135  // Copy Constructor or a Default Constructor and Assignment
   Operator
136
137  #include <iostream>
138  using namespace std;
139
140  class Test
141  {
142  public:
143   Test() { cout << "default ctor: " << this << endl; }
144   Test(const Test& arg) { cout << "copy ctor: " << this << "
   argument=" << &arg << endl; }
145   Test& operator=(const Test& arg) { cout << "operator=: " << this
   << " argument=" << &arg << endl; return *this; }
146   Test operator+(const Test& arg) { cout << "operator+: " << this
   << " argument=" << &arg << endl; return *this; }
147   Test operator-(const Test& arg) { cout << "operator-: " << this
   << " argument=" << &arg << endl; Test temp;  return temp; }
148  };
149
150  int main()
151  {
152   Test one;
153   Test two(one);
154   one = two;
155   Test three = one;
156   Test four = one + two;
157   Test five = one - two;
158   cout << "&five=" << &five << endl;
159
160   return 0;
161  }
```

******* Output *******

MS Visual C++ 2008

```
default ctor: 002BFABB                                          (19)
copy ctor: 002BFAAF argument=002BFABB                           (20)
operator=: 002BFABB argument=002BFAAF                           (21)
copy ctor: 002BFAA3 argument=002BFABB                           (22)
operator+: 002BFABB argument=002BFAAF                           (23)
copy ctor: 002BFA97 argument=002BFABB                           (23)
operator-: 002BFABB argument=002BFAAF                           (24/14)
default ctor: 002BF997                                          (14)
copy ctor: 002BFA8B argument=002BF997                           (24)
&five=002BFA8B                                                  (25)
```

g++ 4.4.0 on Linux

```
default ctor: 0x7fff43e46cff                                    (19)
copy ctor: 0x7fff43e46cfe argument=0x7fff43e46cff               (20)
operator=: 0x7fff43e46cff argument=0x7fff43e46cfe               (21)
copy ctor: 0x7fff43e46cfd argument=0x7fff43e46cff               (22)
operator+: 0x7fff43e46cff argument=0x7fff43e46cfe               (23)
copy ctor: 0x7fff43e46cfc argument=0x7fff43e46cff               (23)
operator-: 0x7fff43e46cff argument=0x7fff43e46cfe               (24/14)
default ctor: 0x7fff43e46cfb                                    (14)
&five=0x7fff43e46cfb                                            (25)
```

Comments

Line 19: default constructor call
Line 20: copy constructor call
Line 21: assignment operator call
Line 22: copy constructor call. Is this the answer to our question? Not yet, let's look further.
Line 23: operator+ is called, then the copy constructor (same result on both compilers)
Line 24: Now here's where it gets interesting. The call to operator-() invokes a default Test constructor call in line 14. Then the MS compiler copies the temporary object using the copy constructor, since the return is "by value". The g++ compiler appears to by "optimizing this onstructor call out". So, it appear that th g++ compiler does not use the copy compiler in this case to perform the "return by value" copy.
Line 25: This illustrates that the five object was created by the copy constructor for the MS compiler and by the default constructor for the g++ compiler.

Who ever said this was going to be easy? Haven't we got better things to do with our time?