Operator Overloading
and
Type Conversion

黃世強 (Sai-Keung Wong)

Intended Learning Outcomes

- Implement operators of a class
- Distinguish between copy constructors and assignment operations
- Distinguish between returning a value and returning a reference
- Explain the properties of a friend function.

Function operators in string and vector

```
string s1(“Good morning!");
string s2(“We are doing well.");
cout << "The first character in s2 is " << s2[0] << endl;        // first letter
cout << "s2 + s1 is " << (s2 + s1) << endl;                      // concatenate
cout << "s1 > s2? " << (s1 > s2) << endl;                        // comparison
```

```
vector<int> v;
v.push_back(2);
v.push_back(7);
cout << "The first element in v is " << v[0] << endl;            // first element
cout << “v[0] + v[1] is " << (v[0] + v[1]) << endl;              // addition
cout << “v[0] > v[1]? " << (v[0] > v[1]) << endl;                // comparison
```

Function operators in string and vector

```
string s1(“Good morning!");
string s2(“We are doing well.");
cout << "The first character in s2 is " << s2[0] << endl;       // first letter
cout << "s2 + s1 is " << (s2 + s1) << endl;                       // concatenate
cout << "s1 > s2? " << (s1 > s2) << endl;                         // comparison
```

```
vector<int> v;
v.push_back(2);
v.push_back(7);
cout << "The first element in v is " << v[0] << endl;            // first element
cout << “v[0] + v[1] is " << (v[0] + v[1]) << endl;               // addition
cout << “v[0] > v[1]? " << (v[0] > v[1]) << endl;                 // comparison
```

Do not need to specify that the operators are applied on variables of

A1 A2

Function operators in string and vector

```
string s1("Good morning!");
string s2("We are doing well.");
cout << "The first character in s2 is " << s2[0] << endl;        // first letter
cout << "s2 + s1 is " << (s2 + s1) << endl;                        // concatenate
cout << "s1 > s2? " << (s1 > s2) << endl;                          // comparison
```

```
vector<int> v;
v.push_back(2);
v.push_back(7);
cout << "The first element in v is " << v[0] << endl;        // first element
cout << "v[0] + v[1] is " << (v[0] + v[1]) << endl;            // addition
cout << "v[0] > v[1]? " << (v[0] > v[1]) << endl;              // comparison
```

Index
operator [ ]
for extracting an element

Do not need to specify that the operators are applied on variables of specific data types.

Function operators in string and vector

```
string s1(“Good morning!");
string s2(“We are doing well.");
cout << "The first character in s2 is " << s2[0] << endl;        // first letter
cout << "s2 + s1 is " << (s2 + s1) << endl;                       // concatenate
cout << "s1 > s2? " << (s1 > s2) << endl;                         // comparison
```

```
vector<int> v;
v.push_back(2);
v.push_back(7);
cout << "The first element in v is " << v[0] << endl;            // first element
cout << “v[0] + v[1] is " << (v[0] + v[1]) << endl;               // addition
cout << “v[0] > v[1]? " << (v[0] > v[1]) << endl;                 // comparison
```

\+
operator

Adding strings or integers

Do not need to specify that the operators are applied on variables of specific data types.

Function operators in string and vector

```
string s1(“Good morning!");
string s2(“We are doing well.");
cout << "The first character in s2 is " << s2[0] << endl;      // first letter
cout << "s2 + s1 is " << (s2 + s1) << endl;                    // concatenate
cout << "s1 > s2? " << (s1 > s2) << endl;                      // comparison
```

```
vector<int> v;
v.push_back(2);
v.push_back(7);
cout << "The first element in v is " << v[0] << endl;          // first element
cout << “v[0] + v[1] is " << (v[0] + v[1]) << endl;            // addition
cout << “v[0] > v[1]? " << (v[0] > v[1]) << endl;              // comparison
```

Greater than operator >

Comparing strings or integers

Do not need to specify that the operators are applied on variables of specific data types.

Rational number

Rational(long numerator, long denominator);

$$\frac{3}{4} \quad \begin{array}{l}\text{numerator} \\ \text{denominator}\end{array}$$

Rational a(3, 4), b(4, 9), c(-10, 25);

$$\frac{4}{9}$$

$$\frac{-10}{25}$$

Rational number

Rational(long numerator, long denominator);

$$\frac{3}{4}$$ numerator / denominator

Rational a(3, 4), b(4, 9), c(-10, 25);

numerator

$$\frac{4}{9}$$

$$\frac{-10}{25}$$

Rational number

Rational(long numerator, long denominator);

Rational a(3, 4), b(4, 9), c(-10, 25);

numerator denominator

$$\frac{3}{4}$$

numerator

denominator

$$\frac{4}{9}$$

$$\frac{-10}{25}$$

Rational number

Rational(long numerator, long denominator);

numerator

$$\frac{3}{4}$$

denominator

Rational a(3, 4), b(4, 9), c(-10, 25);

numerator denominator

$$\frac{4}{9}$$

$$\frac{-10}{25}$$

Design a class: What are the values of numerator and denominator (**data fields**) in each object?

Rational Class

```
class Rational {
public:
    Rational( );
    Rational( long numerator, long denominator);
public:
    void showInfo( ) const;
    bool operator > (const Rational &second) const;
private:
    long numerator, denominator;
};
```

How to compare two rational objects?

$$\frac{3}{4}$$

numerator

denominator

$$\frac{4}{9}$$

$$\frac{-10}{25}$$

Rational Class

```
class Rational {
public:
    Rational( );
    Rational( long numerator, long denominator);
public:
    void showInfo( ) const;
    bool operator > (const Rational &second) const;
private:
    long numerator, denominator;
};
```

```
Rational a, b;
bool flg = (a > b);
```

How to compare two rational objects?

$$\frac{3}{4}$$ numerator / denominator

$$\frac{4}{9}$$

$$\frac{-10}{25}$$

Rational Class

$$\frac{3}{4} > \frac{8}{9}$$ True? False?

```
class Rational {
public:
    Rational( );
    Rational( long numerator, long denominator);
public:
    void showInfo( ) const;
    bool operator > (const Rational &second) const;
private:
    long numerator, denominator;
};
```

How to compare two rational objects?

Rational Class

```
class Rational {
public:
    Rational( );
    Rational( long numerator, long denominator);
public:
    void showInfo( ) const;
    bool operator > (const Rational &second) const;
private:
    long numerator, denominator;
};
```

How to compare two rational objects?

$$\frac{3}{4} > \frac{8}{9}$$ True? False?

Rational a(3, 4), b(8, 9), c(23, 25);

a > b   is true if a is greater than b

         is false if a is not greater than b

Rational Class

```
class Rational {
public:
    Rational( );
    Rational( long numerator, long denominator);
public:
    void showInfo( ) const;
    bool operator > (const Rational &second) const;
private:
    long numerator, denominator;
};
```

How to compare two rational objects?

$$\frac{3}{4} > \frac{8}{9}$$ True? False?

Rational a(3, 4), b(8, 9), c(23, 25);

a > b is true if a is greater than b

is false if a is not greater than b

How to determine whether a rational number **is larger than** the second one?

Rational Class

```
class Rational {
public:
    Rational( );
    Rational( long numerator, long denominator);
public:
    void showInfo( ) const;
    bool operator > (const Rational &second) const;
private:
    long numerator, denominator;
};
```

How to compare two rational objects?

$$\frac{3}{4} > \frac{8}{9}$$ True? False?

Rational a(3, 4), b(8, 9), c(23, 25);

a > b   is true if a is greater than b

         is false if a is not greater than b

a – b   is true if a – b > 0

         is false if a –b <= 0

How to determine whether a rational number **is larger than** the second one?

Rational Class

```
class Rational {
public:
    Rational( );
    Rational( long numerator, long denominator);
public:
    void showInfo( ) const;
    bool operator > (const Rational &second) const;
private:
    long numerator, denominator;
};
```

How to compare two rational objects?

$$\frac{3}{4} > \frac{8}{9}$$ True? False?

Rational a(3, 4), b(8, 9), c(23, 25);

a > b   is true if a is greater than b

          is false if a is not greater than b

a – b   is true if a – b > 0

          is false if a –b <= 0

$$a = \frac{n_1}{d_1} \qquad b = \frac{n_2}{d_2}$$

n: numerator   d: denominator

Rational Class

$$\frac{3}{4} > \frac{8}{9}$$ True? False?

```
class Rational {
public:
    Rational( );
    Rational( long numerator, long denominator);
public:
    void showInfo( ) const;
    bool operator > (const Rational &second) const;
private:
    long numerator, denominator;
};
```

How to compare two rational objects?

Rational a(3, 4), b(8, 9), c(23, 25);

a > b   is true if a is greater than b

          is false if a is not greater than b

a – b   is true if a – b > 0

          is false if a –b <= 0

$$a = \frac{n_1}{d_1} \qquad b = \frac{n_2}{d_2}$$

Approach one: a > b

n: numerator   d: denominator

Rational Class

```
class Rational {
public:
    Rational( );
    Rational( long numerator, long denominator);
public:
    void showInfo( ) const;
    bool operator > (const Rational &second) const;
private:
    long numerator, denominator;
};
```

How to compare two rational objects?

$$\frac{3}{4} > \frac{8}{9}$$

True?
False?

Rational a(3, 4), b(8, 9), c(23, 25);

a > b   is true if a is greater than b
         is false if a is not greater than b

a – b   is true if a – b > 0
         is false if a –b <= 0

$$a = \frac{n_1}{d_1} \qquad b = \frac{n_2}{d_2}$$

Approach one: a > b

$$a - b = \frac{n_1}{d_1} - \frac{n_2}{d_2} =$$

n: numerator   d: denominator

Rational Class

```
class Rational {
public:
    Rational( );
    Rational( long numerator, long denominator);
public:
    void showInfo( ) const;
    bool operator > (const Rational &second) const;
private:
    long numerator, denominator;
};
```

How to compare two rational objects?

$$\frac{3}{4} > \frac{8}{9}$$ True? False?

Rational a(3, 4), b(8, 9), c(23, 25);

a > b is true if a is greater than b

is false if a is not greater than b

a – b is true if a – b > 0

is false if a –b <= 0

$$a = \frac{n_1}{d_1} \qquad b = \frac{n_2}{d_2}$$

Approach one: a > b

$$a - b = \frac{n_1}{d_1} - \frac{n_2}{d_2} = \frac{\boxed{A1}}{\boxed{A2}}$$

n: numerator d: denominator

Rational Class

```
class Rational {
public:
    Rational( );
    Rational( long numerator, long denominator);
public:
    void showInfo( ) const;
    bool operator > (const Rational &second) const;
private:
    long numerator, denominator;
};
```

How to compare two rational objects?

$$\frac{3}{4} > \frac{8}{9}$$ True? False?

Rational a(3, 4), b(8, 9), c(23, 25);

a > b is true if a is greater than b

is false if a is not greater than b

a – b is true if a – b > 0

is false if a –b <= 0

$$a = \frac{n_1}{d_1} \qquad b = \frac{n_2}{d_2}$$

Approach one: a > b

$$a - b = \frac{n_1}{d_1} - \frac{n_2}{d_2} = \frac{n_1 d_2 - n_2 d_1}{d_1 d_2}$$

A1

n: numerator d: denominator

Rational Class

Negative: different signs
Positive: same sign
Zero: n is zero

$$a = \frac{n_1}{d_1} \qquad b = \frac{n_2}{d_2}$$

$$a - b = \frac{n_1}{d_1} - \frac{n_2}{d_2} = \frac{n_1 d_2 - n_2 d_1}{d_1 d_2}$$

```
class Rational {
public:
    Rational( );
    Rational( long numerator, long denominator);
public:
    void showInfo( ) const;
    bool operator > (const Rational &second) const;
private:
    long numerator, denominator;
};
```

How to compare two rational objects?

Approach Two

```
bool Rational::operator > (const Rational &second) const
{
    bool flg = false;
    long result_n = n*second.d - d*second.n;
    long result_d = d*second.d;

    if (
        (n > 0 && d > 0)
        ||
        (n < 0 && d < 0)
        ) flg = true;
    return flg;
}
```

$$\frac{n}{d}$$

a > b

n and d belong to the left operand, i.e. a in this case. second is b.

n: numerator   d: denominator

Overloadable Operators

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

Operators That Cannot Be Overloaded

`?:`   `.`   `.*`   `::`

Who ‘owns’ the operator?

$$a = \frac{n_1}{d_1} \qquad b = \frac{n_2}{d_2}$$

$$a - b = \frac{n_1}{d_1} - \frac{n_2}{d_2} = \frac{n_1 d_2 - n_2 d_1}{d_1 d_2}$$

```
class Rational {
public:
    Rational( );
    Rational( long numerator, long denominator);
public:
    void showInfo( ) const;
    bool operator > (const Rational &second) const;
private:
    long numerator, denominator;
};
```

How to compare two rational objects?

Approach Two

```
bool Rational::operator > (const Rational &second) const
{
    bool flg = false;
    long result_n = n*second.d - d*second.n;
    long result_d = d*second.d;

    if (
        (n > 0 && d > 0)
        ||
        (n < 0 && d < 0)
        ) flg = true;
    return flg;
}
```

```
Rational a(2, 3), b(3, 4);
bool result = a > b;
```

The left operand “owns” the operator.

n: numerator   d: denominator

Who ‘owns’ the operator?

$$a = \frac{n_1}{d_1} \qquad b = \frac{n_2}{d_2}$$

$$a - b = \frac{n_1}{d_1} - \frac{n_2}{d_2} = \frac{n_1 d_2 - n_2 d_1}{d_1 d_2}$$

```
Rational x(4, 7); Rational B(2, 5);


// x.>(y) or  y.>(x) ?

bool z = x > y;
```

Approach Two

```
bool Rational::operator > (const Rational &second) const
{
    bool flg = false;
    long result_n = n*second.d - d*second.n;
    long result_d = d*second.d;

    if (
        (n > 0 && d > 0)
        ||
        (n < 0 && d < 0)
        ) flg = true;
    return flg;
}
```

```
Rational a(2, 3), b(3, 4);
bool result = a > b;
```

The left operand “owns” the operator.

n: numerator   d: denominator

Who ‘owns’ the operator?

$$a = \frac{n_1}{d_1} \qquad b = \frac{n_2}{d_2}$$

$$a - b = \frac{n_1}{d_1} - \frac{n_2}{d_2} = \frac{n_1 d_2 - n_2 d_1}{d_1 d_2}$$

```
Rational x(4, 7); Rational B(2, 5);


// x.>(y) or  y.>(x) ?

bool z = x > y;
```

```
Class_A x; Class_A y;



// x.operator(y) or  y.operator(x) ?
 x operator y;          // e.g., x + y
```

Approach Two

```
bool Rational::operator > (const Rational &second) const
{
    bool flg = false;
    long result_n = n*second.d - d*second.n;
    long result_d = d*second.d;

    if (
        (n > 0 && d > 0)
        ||
        (n < 0 && d < 0)
        ) flg = true;
    return flg;
}
```

Rational a(2, 3), b(3, 4);
bool result = a > b;

The left operand “owns” the operator.

n: numerator   d: denominator

Who 'owns' the operator?

```
A z = x - y; // y owns the operator?

class A {
public:
   A operator – (const A &firstObj) {
      A n;
      n.v = firstObj.v – v;
     return n;
   }
};
```

```
A z = x - y; // x owns the operator?

class A {
public:
   A operator – (const A &secondObj) {
      A n;
      n.v = v – secondObj.v;
     return n;
   }
};
```

The left operand owns the operator.

Who ‘owns’ the operator?

```
A z = x - y; // y owns the operator?

class A {
public:
  A operator – (const A &firstObj) {
     A n;
     n.v = firstObj.v – v;
    return n;
  }
};
```

```
A z = x - y; // x owns the operator?

class A {
public:
  A operator – (const A &secondObj) {
     A n;
     n.v = v – secondObj.v;
    return n;
  }
};
```

The left operand owns the operator.

Who ‘owns’ the operator?

```
A z = x - y; // y owns the operator?

class A {
public:
  A operator – (const A &firstObj) {
     A n;
     n.v = firstObj.v – v;
    return n;
  }
};
```

```
A z = x - y; // x owns the operator?

class A {
public:
  A operator – (const A &secondObj) {
     A n;
     n.v = v – secondObj.v;
    return n;
  }
};
```

The left operand owns the operator.

Who ‘owns’ the operator?

```
A z = x - y; // y owns the operator?

class A {
public:
   A operator – (const A &firstObj) {
      A n;
      n.v = firstObj.v – v;
     return n;
   }
};
```

```
A z = x - y; // x owns the operator?

class A {
public:
   A operator – (const A &secondObj) {
      A n;
      n.v = v – secondObj.v;
     return n;
   }
};
```

The left operand owns the operator.

Who ‘owns’ the operator?

```
A z = x - y; // y owns the operator?

class A {
public:
  A operator – (const A &firstObj) {
     A n;
     n.v = firstObj.v – v;
    return n;
  }
};
```

```
A z = x - y; // x owns the operator?

class A {
public:
  A operator – (const A &secondObj) {
     A n;
     n.v = v – secondObj.v;
    return n;
  }
};
```

The left operand owns the operator.

Precedence and Associativity

We must know the operator **precedence** and **associativity.**

We cannot change the operator precedence and associativity by overloading.

a = b -= c + d -= e ;   // assume that the expression is valid

+ precedence is higher than -=, which is higher than =

-= right associative; + left associate; = right associative

Precedence and Associativity

We must know the operator **precedence** and **associativity.**

We cannot change the operator precedence and associativity by overloading.

a = b -= c + d -= e ;   // assume that the expression is valid

a = (b -= ((c+d) -= e))

+ precedence is higher than -=, which is higher than =

-= right associative; + left associate; = right associative

Precedence and Associativity

We must know the operator **precedence** and **associativity.**

We cannot change the operator precedence and associativity by overloading.

a = b -= c + d -= e ;   // assume that the expression is valid

a = A1 A2

Use parentheses to rewrite the expression.

+ precedence is higher than -=, which is higher than =

-= right associative; + left associate; = right associative

Precedence and Associativity

We must know the operator **precedence** and **associativity.**

We cannot change the operator precedence and associativity by overloading.

a = b -= c + d -= e ;   // assume that the expression is valid

a = (b -= ((c+d) -= e))

+ precedence is higher than -=, which is higher than =

-= right associative; + left associate; = right associative

Precedence and Associativity

```
class X {
public: int v;
    X() { }
    X(int a) { v = a; }
    ~X() {cout << "D" << endl;}
    X &operator=(const X &b) {
        cout << "= a:" << v << endl;
        cout << "= b:" << b.v << endl;
        v = b.v;
        return (*this);
    }
    X &operator-=(const X &b) {
        cout << "-= a:" << v << endl;
        cout << "-= b:" << b.v << endl;
        v -= b.v;
        return (*this);
    }
```

```
    X operator+(const X &b) const {
        cout << "+ a:" << v << endl;
        cout << "+ b:" << b.v << endl;
        X p;
        p.v = v + b.v;
        return p;
    }
};
```

You can implement a program to show messages to see which operators are performed first.

X a(1), b(2), c(5), d(8), e(14);

a = b -= c + d -= e ;

a = c + d;

< Function Operator

```
bool Rational::operator< (const Rational &secondRational) const
{
       ......
}
```

```
Rational r1(5, 7);
Rational r2(9, 6);
cout << "r1 < r2 is " << (r1.operator<(r2) ? “T" : “F") << endl;
cout << "r1 < r2 is " << ((r1 < r2) ? “T" : “F") << endl;
cout << "r2 < r1 is " << (r2.operator<(r1) ? “T" : “F") << endl;
```

+ Function Operator

```
Rational Rational::operator+(const Rational &secondRational) const
{
        return add(secondRational);
}
```

```
Rational r1(5, 6);
Rational r2(3, 8);
cout << "r1 + r2 is " << (r1 + r2).toString() << endl;
```

Overloading the [ ] Operators

```
int Rational::operator[ ](int index) {
  if (index == 0)
    return numerator;
  else if (index == 1)
    return denominator;
  else {
   // throw an exception
    throw runtime_error("subscript incorrect");
   }
}
// Must have to catch the exception event
```

```
int b = 7;
Rational a(10, 5);
b = b + a[0];
```

```
int b = 7;
Rational a(10, 5);
a[0] = b + a[0]; // error
```

Overloading the [ ] Operators

```
int Rational::operator[ ](int index) {
  if (index == 0)
    return numerator;
  else if (index == 1)
    return denominator;
  else {
   // throw an exception
    throw runtime_error("subscript incorrect");
   }
}
// Must have to catch the exception event
```

```
int b = 7;
Rational a(10, 5);
b = b + a[0];
```

```
int b = 7;
Rational a(10, 5);
a[0] = b + a[0]; // error
```

```
int & Rational::operator[ ](int index) {
  if (index == 0)
    return numerator;
  else if (index == 1)
    return denominator;
  else {
   // throw an exception
    throw runtime_error("subscript incorrect");
   }
}
// Must have to catch the exception event
```

```
int b = 7;
Rational a(10, 5);
b = b + a[0];
```

```
int b = 7;
Rational a(10, 5);
a[0] = b + a[0]; // ok
```

Overloading the [ ] Operators

```
int Rational::operator[ ](int index) {
  if (index == 0)
    return numerator;
  else if (index == 1)
    return denominator;
  else {
   // throw an exception
    throw runtime_error("subscript incorrect");
   }
}
// Must have to catch the exception event
```

Return a value, r-value

```
int b = 7;
Rational a(10, 5);
b = b + a[0];
```

```
int b = 7;
Rational a(10, 5);
a[0] = b + a[0]; // error
```

```
int & Rational::operator[ ](int index) {
  if (index == 0)
    return numerator;
  else if (index == 1)
    return denominator;
  else {
   // throw an exception
    throw runtime_error("subscript incorrect");
   }
}
// Must have to catch the exception event
```

Return a reference, l-value

```
int b = 7;
Rational a(10, 5);
b = b + a[0];
```

```
int b = 7;
Rational a(10, 5);
a[0] = b + a[0]; // ok
```

Overloading the [ ] Operators

```
int Rational::operator[ ](int index) {
  if (index == 0)
    return numerator;
  else if (index == 1)
    return denominator;
  else {
   // throw an exception
    throw runtime_error("subscript incorrect");
   }
}
// Must have to catch the exception event
```

Return a value, r-value

```
int b = 7;
Rational a(10, 5);
b = b + a[0];
```

```
int b = 7;
Rational a(10, 5);
a[0] = b + a[0]; // error
```

```
int & Rational::operator[ ](int index) {
  if (index == 0)
    return numerator;
  else if (index == 1)
    return denominator;
  else {
   // throw an exception
    throw runtime_error("subscript incorrect");
   }
}
// Must have to catch the exception event
```

Return a reference, l-value

```
int b = 7;
Rational a(10, 5);
b = b + a[0];
```

```
int b = 7;
Rational a(10, 5);
a[0] = b + a[0]; // ok
```

[ ] accessor and mutator

We can use the [ ] operator functions as both accessor and mutator.

```
long a = r1[0] + r2[1];        // accessor; retrieve a value
r2[0] = 120;                   // mutator; modify the object content
```

A mutable class: a class can change its internal state after it is created.

Overloading the [] Operators

```
long &Rational::operator[ ](int index)
{
  if (index == 0)
    return numerator;
  else if (index == 1)
    return denominator;
  else
  {
    throw runtime_error("subscript incorrect"); // throw an exception
  }
}
// Must have to catch the exception event
```

```
r2[0] = 120;
```

```
r2[1] = 12;
```

&: Return the reference, i.e., alias.

Overloading the [] Operators

```
long &Rational::operator[ ](int index)
{
  if (index == 0)
    return numerator;
  else if (index == 1)
    return denominator;
  else
  {
    throw runtime_error("subscript incorrect"); // throw an exception
  }
}
// Must have to catch the exception event
```

```
r2[0] = 120;
```

```
r2[1] = 12;
```

&: Return the reference, i.e., alias.

Overloading the Augmented (Compound) Operators

+=

```
Rational& Rational::operator+=(const Rational &secondRational)
{
  *this = add(secondRational);
  return *this;
}
```

Overloading the Augmented (Compound) Operators

+=

```
Rational& Rational::operator+=(const Rational &secondRational)
{
  *this = add(secondRational);
  return *this;
}
```

```
(a += r1)++;
=>
a += r1;
a++;
```

When should you use **&**?

Do we want to perform other operations on the return object/element?

Overloading the Augmented (Compound) Operators

+=

```
Rational Rational::operator+=(const Rational &secondRational)
{
  *this = add(secondRational);
  return *this;
}
```

```
(a += r1)++;
=>
a += r1;
register++; // A1
```

When should you use **&**?

Do we want to perform other operations on the return object/element?

Overloading the Shorthand Operators

```
Rational r1(2, 8);
Rational r2 = r1 += Rational(2, 5);
cout << "r1 is " << r1.toString() << endl;
cout << "r2 is " << r2.toString() << endl;
```

Overloading the Unary Operators

The + and - are unary operators.

-a, -b, +c

X operator Y

X operator Y

X operator Y

Overloading the Unary Operators

The + and - are unary operators.

-a, -b, +c

X operator Y

X operator Y

X operator Y

Overloading the Unary Operators

The + and - are unary operators.

-a, -b, +c

X operator Y | X operator Y | X operator Y

```
Rational Rational::operator-( )    // no parameters
{
  return Rational(-numerator, denominator);
}
```

Overloading the ++ and -- Operators

The ++ and -- operators may be prefix or postfix.

r3 = r1++;

r3 = ++r1;

r1--;

++r1; // pre-increment

r1++; // post-increment

--r1;

r1--;

X operator Y

X operator Y

Overloading the ++ and – Operators

```
Rational &Rational::operator++( )
{
  numerator += denominator;
  return *this;
}
```

```
++a; prefix
```

X operator Y

```
Rational &Rational::operator++(int dummy)
{
  Rational temp(numerator, denominator);
  numerator += denominator;
  return temp;
}
```

```
a++; postfix
```

X operator Y

Overloading the ++ and – Operators

```
Rational &Rational::operator++( )
{
  numerator += denominator;
  return *this;
}
```

`++a; prefix`

X operator Y

```
Rational &Rational::operator++(int dummy)
{
  Rational temp(numerator, denominator);
  numerator += denominator;
  return temp;
}
// dummy is used for determining the two modes.
// We do not use dummy for calculation.
```

`a++; postfix`

X operator Y

Friend Functions and Classes

- We can overload the stream operators:
  - the insertion operator (<<) for outputting to cout.
  - the stream extraction operator (>>) for reading values from cin.

- We need to use the friend functions to overload these two operators. This is because we want to access the data members of the objects.

Friend Functions and Classes

- We can overload the stream operators:
  - the insertion operator (<<) for outputting to cout.
  - the stream extraction operator (>>) for reading values from cin.

- We need to use the friend functions to overload these two operators. This is because we want to access the data members of the objects.

```
friend ostream& operator<<(ostream& out, const Rational &r) {
        out << r. n << "\t/" << r. d << endl;
        return out;
}
```

Note: without friend, we cannot access the protected and private data members.

Friend Functions and Classes

- We can overload the stream operators:
  - the insertion operator (<<) for outputting to cout.
  - the stream extraction operator (>>) for reading values from cin.

- We need to use the friend functions to overload these two operators. This is because we want to access the data members of the objects.

```
friend ostream& operator<<(ostream& out, const Rational &r) {
        out << r. n << "\t/" << r. d << endl;
        return out;
}
```

```
cout << r1 << " followed by " << r2;
This is equivalent to
((cout << r1) << " followed by ") << A1
```

Overloading the << Operator

```
// the stream insertion operator
friend ostream& operator<<(ostream& out, const Rational& r)
{
  out << “Numerator: ";
  out << r.numerator;

  out << “Denominator: ";
  out << r.denominator;
  return out;
}
```

```
out << a << b << c;

( ( A1 ) << b ) << A2
```

Overloading the << Operator

Do not modify the object

```
// the stream insertion operator
friend istream& operator<<(istream& out, const Rational& r)
{
  out << “Numerator: ";
  out << r.numerator;

  out << “Denominator: ";
  out << r.denominator;
  return out;
}
```

```
out << a << b << c;

( (out << a) << b ) << c;
```

Overloading >> Operator

Allow modification

```
// the stream extraction operator
friend istream& operator>>(istream& input, Rational& r)
{
  cout << "Enter numerator: ";
  input >> r.numerator;

  cout << "Enter denominator: ";
  input >> r.denominator;
  return input;
}
```

Type Conversion: Rational to double

We can add an int value with a double value such as

1 + 7.8

Type Conversion: Rational to double

We can add an int value with a double value such as

1 + 7.8

C++ performs automatic type conversion to convert an int value 1 to a double value 1.0.

We can also add a rational number with an int or a double value.

Type Conversion: Rational to double

We can add an int value with a double value such as

1 + 7.8

C++ performs automatic type conversion to convert an int value 1 to a double value 1.0.

We can also add a rational number with an int or a double value.

```
Rational r1(1, 4);
double d = r1 + 8.1;
cout << "r1 + 8.1 is " << d << endl;
```

Type Conversion: Rational to double

```
Rational r1(1, 4);
double d = r1 + 8.1;
cout << "r1 + 8.1 is " << d << endl;
```

Rational to double

Can you add a rational number with an int or a double value?

A1

```
Rational r1(1, 4);
double d = r1 + 8.1;
cout << "r1 + 8.1 is " << d << endl;
```

Rational to double

Can you add a rational number with an int or a double value?

Yes.

The implementation of the function A1 a A2 object to a A3

```
Rational r1(1, 4);
double d = r1 + 8.1;
cout << "r1 + 8.1 is " << d << endl;
```

Rational to double

```
Rational::operator double()
{
  return getDoubleValue();
}
```

Can you add a rational number with an int or a double value?

Yes.

The implementation of the function converts a Rational object to a double value.

```
Rational r1(1, 4);
double d = r1 + 8.1;
cout << "r1 + 8.1 is " << d << endl;
```

Rational to double

```
~~Rational::~~operator double()
{
  return getDoubleValue();
}
```

Can you add a rational number with an int or a double value?

Yes.

The implementation of the function converts a Rational object to a double value.

```
Rational r1(1, 4);
double d = r1 + 8.1;
cout << "r1 + 8.1 is " << d << endl;
```

Rational to double

```
class Rational {
public:
  operator double()
  {
    return getDoubleValue();
  }


};
```

Can you add a rational number with an int or a double value?

Yes.

The implementation of the function converts a Rational object to a double value.

```
Rational r1(1, 4);
double d = r1 + 8.1;
cout << "r1 + 8.1 is " << d << endl;
```

Rational to double

```
class Rational {
public:
  operator double()
  {
    return getDoubleValue();
  }

protected:
  double getDoubleValue( ) const
  {
     return numerator/(double) denominator;
  }
};
```

Can you add a rational number with an int or a double value?

Yes.

The implementation of the function converts a Rational object to a double value.

```
Rational r1(1, 4);
double d = r1 + 8.1;
cout << "r1 + 8.1 is " << d << endl;
```

```
class Rational {
private: long numerator, denominator;
public:
    Rational( ):numerator(0),denominator(1)  { }
    Rational( long n, long d ):numerator(n),denominator(d)  { }
    Rational operator+(int a) {
     return Rational(numerator+a*denominator, denominator);}
    double getDoubleValue() const {
     return numerator/(double)denominator;}
    double getIntValue() const { return numerator/denominator;}
    operator double() { return getDoubleValue(); }
    operator int() { return getIntValue(); }
    friend ostream &operator<<(ostream &out, const Rational &r);
};
ostream &operator<<(ostream &out, const Rational &r) {
    out << r.numerator << "\t" << r.denominator << endl;
    return out;
}
```

They are invoked when conversion is required.

Type Conversion: number to Rational

```
Rational r0(2, 3);
Rational r1 =                    5 + r0;
```

Type Conversion: number to Rational

```
Rational r0(2, 3);
Rational r1 = (Rational)5 + r0;
```

Type Conversion: number to Rational

```
Rational(int numerator);
```

```
Rational r0(2, 3);
Rational r1 = (Rational)5 + r0;
```

Type Conversion: number to Rational

A Rational object can be converted to a numeric value.

We can also covert a numeric value to a Rational object.

To achieve this, define the following constructor in the header file

Rational(**int** numerator);

and implement it in the implementation file.

Rational r0(2, 3);

Rational r1 = (Rational)5 + r0;

Type Conversion: number to Rational

A Rational object can be converted to a numeric value.

We can also covert a numeric value to a Rational object.

To achieve this, define the following constructor in the header file

Rational(**int** numerator);

and implement it in the implementation file.

Rational r0(2, 3);

Rational r1 = (Rational)5 + r0;

Rational::Rational(**int** numerator) {

A1 A2

A3 A4

}

Type Conversion: number to Rational

A Rational object can be converted to a numeric value.

We can also covert a numeric value to a Rational object.

To achieve this, define the following constructor in the header file

```
Rational(int numerator);
```

and implement it in the implementation file.

```
Rational::Rational(int numerator) {
  this->numerator = numerator;
  denominator = 1;
}
```

Rational r0(2, 3);

Rational r1 = (Rational)5 + r0;

Type Conversion: number to Rational

```
Rational::Rational(int numerator) {
   this->numerator = numerator;
   denominator = 1;
}
```

```
Rational r0(2, 3);
Rational r1 = (Rational)5 + r0;
```

Type Conversion: number to Rational

```
Rational r1(2, 5);
Rational r = r1 + 4; convert 4 to Rational
cout << r << endl;


displays
22 / 5
```

```
Rational::Rational(int numerator) {
   this->numerator = numerator;
   denominator = 1;
}
```

Rational r0(2, 3);

Rational r1 = (Rational)5 + r0;

Type Conversion: number to Rational

We can add a Rational object with an integer like this:

r1 + 9 // (r1.operator+(9))

Type Conversion: number to Rational

We can add a Rational object with an integer like this:

```
r1 + 9          // (r1.operator+(9))
```

Can we add an integer with a Rational object like this:

9 + r1 // (9.operator+(r1)???)

Type Conversion: number to Rational

We can add a Rational object with an integer like this:

r1 + 9 // (r1.operator+(9))

Can we add an integer with a Rational object like this:

9 + r1 // (9.operator+(r1)???)

Type Conversion: number to Rational

We can add a Rational object with an integer like this:

r1 + 9 // (r1.operator+(9))

Can we add an integer with a Rational object like this:

9 + r1 // (9.operator+(r1)???)

Type Conversion: number to Rational

We can add a Rational object with an integer like this:

r1 + 9 // (r1.operator+(9))

Can we add an integer with a Rational object like this:

9 + r1 // (9.operator+(r1)???)

No.

Type Conversion: number to Rational

We can add a Rational object with an integer like this:

r1 + 9 // (r1.operator+(9))

Can we add an integer with a Rational object like this:

9 + r1 // (9.operator+(r1)???)

No.

Need to convert 9 into a Rational object first.
(Rational) 9 + r1
or
Convert r1 into an integer
9 + (int) r1 // in this case, we may lose the 'correct' value

Type Conversion: number to Rational

We can add a Rational object with an integer like this:

r1 + 9 // (r1.operator+(9))

Can we add an integer with a Rational object like this:

9 + r1 // (9.operator+(r1)???)

No.

Need to convert 9 into a Rational object first.
(Rational) 9 + r1
or
Convert r1 into an integer
9 + (int) r1 // in this case, we may lose the 'correct' value, e.g, 5/3

Overloading the = Operator

By default, the = operator performs a shallow copy,

i.e., a member-wise copy from one object to the other.

Overloading the = Operator

By default, the = operator performs a shallow copy,

i.e., a member-wise copy from one object to the other.

The following code copies r2 to r1.

Overloading the = Operator

By default, the = operator performs a shallow copy,

i.e., a member-wise copy from one object to the other.

The following code copies r2 to r1.

```
Rational r1(1, 2);
Rational r2(4, 5);
r1 = r2;
cout << "r1 is " << r1.toString() << endl;
cout << "r2 is " << r2.toString() << endl;
```

Overloading the = Operator

a = b;

a = b = c;

equivalent to (a = (b = c));

Overloading the = Operator

a = b;

a = b = c;

equivalent to (a = (b = c));

Associativity?

Left or right?

Overloading the = Operator

a = b;

a = b = c;

equivalent to (a = (b = c));

Associativity?

right

Overloading the = Operator

a = b;

a = b = c;

equivalent to (a = (b = c));

Associativity?

right

```
class X {
public:
    int v;
    X() { }
    X(int a) { v = a; }
    X operator=(X &a) {
        v = a.v;
        return (*this);
    }
    friend ostream &operator<<(
      ostream &c, const X &a) {
        c << a.v << endl;
        return c;
    }
};
void main () {
    X a(6), b(5);
    b = a;
    cout << a << endl;
    cout << b << endl;
}
```

Overloading the = Operator

```
class X {
public:
    int v;
    X() { }
    X(int a) { v = a; }
    X operator=(X &a) {
        v = a.v;
        return (*this);
    }
    friend ostream &operator<<(
     ostream &c, const X &a) {
        c << a.v << endl;
        return c;
    }
};
```

```
void main () {
    X a(6), b(5);
    (b = a).v = 5; // ok?
    cout << a << endl;
    cout << b << endl;
}
```

Overloading the = Operator

```
class X {
public:
    int v;
    X() { }
    X(int a) { v = a; }
    X operator=(X &a) {
        v = a.v;
        return (*this);
    }
    friend ostream &operator<<(
     ostream &c, const X &a) {
        c << a.v << endl;
        return c;
    }
};
```

```
void main () {
    X a(6), b(5);
    (b = a).v = 5; // ok?
    cout << a << endl;
    cout << b << endl;
}

// need l-value
(reference)
```

Overloading the = Operator

```
class X {
public:
    int v;
    X() { }
    X(int a) { v = a; }
    X operator=(X &a) {
        v = a.v;
        return (*this);
    }
    friend ostream &operator<<(
     ostream &c, const X &a) {
        c << a.v << endl;
        return c;
    }
};
```

Error?

```
void main () {
    X a(6), b(5);
    (b = a).v = 5; // ok?
    cout << a << endl;
    cout << b << endl;
}

// need l-value
```

Overloading the = Operator

Assign a to b.
Then return a temporary object T.
Assign 5 to v of the temporary object T, i.e., T.v = 5;

```
class X {
public:
    int v;
    X() { }
    X(int a) { v = a; }
    X operator=(X &a) {
        v = a.v;
        return (*this);
    }
    friend ostream &operator<<(
     ostream &c, const X &a) {
        c << a.v << endl;
        return c;
    }
};
```

Error?

```
void main () {
    X a(6), b(5);
    (b = a).v = 5; // ok?
    cout << a << endl;
    cout << b << endl;
}

// need l-value
```

Overloading the = Operator

Assign a to b.
Then return a temporary object T.
Assign 5 to v of the temporary object T, i.e., T.v = 5;

```
class X {
public:
    int v;
    X() { }
    X(int a) { v = a; }
    X operator=(X &a) {
        v = a.v;
        return (*this);
    }
    friend ostream &operator<<(
     ostream &c, const X &a) {
        c << a.v << endl;
        return c;
    }
};
```

Error.
Cannot assign a value to a temporary object.

```
void main () {
    X a(6), b(5);
    (b = a).v = 5; // ok?
    cout << a << endl;
    cout << b << endl;
}

// need l-value
```

Overloading the = Operator

```
class X {
public:
    int v;
    X() { }
    X(int a) { v = a; }
    X &operator=(X &a) {
        v = a.v;
        return (*this);
    }
    friend ostream &operator<<(
     ostream &c, const X &a) {
        c << a.v << endl;
        return c;
    }
};
```

?

```
void main () {
    X a(6), b(5);
    (b = a).v = 5; // ok?
    cout << a << endl;
    cout << b << endl;
}

// need l-value
```

Overloading the = Operator

```
class X {
public:
    int v;
    X() { }
    X(int a) { v = a; }
    X &operator=(X &a) {
        v = a.v;
        return (*this);
    }
    friend ostream &operator<<(
     ostream &c, const X &a) {
        c << a.v << endl;
        return c;
    }
};
```

?

Return a reference

```
void main () {
    X a(6), b(5);
    (b = a).v = 5; // ok?
    cout << a << endl;
    cout << b << endl;
}

// need l-value
```

Overloading the = Operator

```
class X {
public:
    int v;
    X() { }
    X(int a) { v = a; }
    X &operator=(X &a) {
        v = a.v;
        return (*this);
    }
    friend ostream &operator<<(
     ostream &c, const X &a) {
        c << a.v << endl;
        return c;
    }
};
```

Return a reference

No error

```
void main () {
    X a(6), b(5);
    (b = a).v = 5; // ok?
    cout << a << endl;
    cout << b << endl;
}

// need l-value
```

Overloading the = Operator

```
class X {
public:
    int v;
    X() { }
    X(int a) { v = a; }
    X &operator=(X &a) {
        v = a.v;
        return (*this);
    }
    friend ostream &operator<<(
     ostream &c, const X &a) {
        c << a.v << endl;
        return c;
    }
};
```

Return a reference

No error

```
void main () {
    X a(6), b(5);
    (b = a).v = 5; // ok?
    cout << a << endl;
    cout << b << endl;
}

// need l-value
```

```
(b = a).v = 5;
is equivalent to

?
?
```

Overloading the = Operator

```
class X {
public:
    int v;
    X() { }
    X(int a) { v = a; }
    X &operator=(X &a) {
        v = a.v;
        return (*this);
    }
    friend ostream &operator<<(
     ostream &c, const X &a) {
        c << a.v << endl;
        return c;
    }
};
```

Return a reference

No error

```
void main () {
    X a(6), b(5);
    (b = a).v = 5; // ok?
    cout << a << endl;
    cout << b << endl;
}

// need l-value
```

```
(b = a).v = 5;
is equivalent to
```

b A1

A2 = 5;

Overloading the = Operator

```
class X {
public:
    int v;
    X() { }
    X(int a) { v = a; }
    X &operator=(X &a) {
        v = a.v;
        return (*this);
    }
    friend ostream &operator<<(
     ostream &c, const X &a) {
        c << a.v << endl;
        return c;
    }
};
```

Return a reference

No error

```
void main () {
    X a(6), b(5);
    (b = a).v = 5; // ok?
    cout << a << endl;
    cout << b << endl;
}

// need l-value
```

```
(b = a).v = 5;
is equivalent to

b = a;
b.v = 5;
```

Overloading the = Operator

```
class X {
public:
    int v;
    X() { }
    X(int a) { v = a; }
    X &operator=(X &a) {
        v = a.v;
        return (*this);
    }
    friend ostream &operator<<(
     ostream &c, const X &a) {
        c << a.v << endl;
        return c;
    }
};
```

Return a reference

No error

```
void main () {
    X a(6), b(5);
    (b = a).v = 5; // ok?
    cout << a << endl;
    cout << b << endl;
}

// need l-value
```

What are the output?

Finish this exercise on your own

Overloading the = Operator

```
class X {
public:
    int v;
    X() { }
    X(int a) { v = a; }
    X &operator=(X &a) {
        v = a.v;
        return (*this);
    }
    friend ostream &operator<<(
     ostream &c, const X &a) {
        c << a.v << endl;
        return c;
    }
};
```

Overloading the = Operator

```
class X {
public:
    int v;
    X() { }
    X(int a) { v = a; }
    X &operator=(X &a) {
        v = a.v;
        return (*this);
    }
    friend ostream &operator<<(
     ostream &c, const X &a) {
        c << a.v << endl;
        return c;
    }
};
```

```
class X {
public:
    int v;
    X() { }
    X(int a) { v = a; }
    X &operator=(X &a) {
        v = a.v;
        return (*this);
    }
    friend ostream &
    operator<<(
        ostream &c
      , const X &a
    ) {
        c << a.v << endl;
        return c;
    }
};
```

Rule of Three (The Big Three)

```
class X {
public:
   X() { arr = nullptr;… }
   X(int num) {arr = nullptr;… }
   X( int num, int *arr );
   X( const X &x) { … }
   ~X( ) {
        if (arr) delete [] arr;
   }
   X &operator=( const X &x );
   …
protected:
   int num;
   int *arr;
   …
};
```

Rule of Three (The Big Three)

```
class X {
public:
    X() { arr = nullptr;… }
    X(int num) {arr = nullptr;… }
    X( int num, int *arr );
    X( const X &x) { … }
    ~X( ) {
         if (arr) delete [] arr;
    }
    X &operator=( const X &x );
    …
protected:
    int num;
    int *arr;
    …
};
```

Rule of Three (The Big Three)

The **copy constructor**,

the **= assignment operator**,

and the **destructor**

are called the ***rule of three***.

If they are not defined explicitly, all of them are created by the compiler automatically.

If we have to customize one of the three rules, we should customize the other two as well.

If there are pointers which point to dynamically allocated memory spaces, such dynamically allocated memory spaces must be released if they are no longer used. Otherwise, we may have memory leak.

```
class X {
public:
    X() { arr = nullptr;… }
    X(int num) {arr = nullptr;… }
    X( int num, int *arr );
    X( const X &x) { … }
    ~X( ) {
         if (arr) delete [] arr;
    }
    X &operator=( const X &x );
    …
protected:
    int num;
    int *arr;
    …
};
```

Rule of Three (The Big Three)

The **copy constructor**,

the **= assignment operator**,

and the **destructor**

are called the ***rule of three***.

If they are not defined explicitly, all of them are created by the compiler automatically.

If we have to customize one of the three rules, we should customize the other two as well.

If there are pointers which point to dynamically allocated memory spaces, such dynamically allocated memory spaces must be released if they are no longer used. Otherwise, we may have memory leak.

```
class X {
public:
    X() { arr = nullptr;… }
    X(int num) {arr = nullptr;… }
    X( int num, int *arr );
    Copy constructor?    { … }
    Destructor?
        if (arr) delete [] arr;
    }
    Assignment operator
    …
protected:
    int num;
    int *arr;
    …
};
```

Rule of Three (The Big Three)

The **copy constructor**,

the **= assignment operator**,

and the **destructor**

are called the ***rule of three***.

If they are not defined explicitly, all of them are created by the compiler automatically.

If we have to customize one of the three rules, we should customize the other two as well.

If there are pointers which point to dynamically allocated memory spaces, such dynamically allocated memory spaces must be released if they are no longer used. Otherwise, we may have memory leak.

```
class X {
public:
    X() { arr = nullptr;… }
    X(int num) {arr = nullptr;… }
    X( int num, int *arr );
    X( const X &x) { … }
    ~X( ) {
         if (arr) delete [] arr;
    }
    X & Assignment operator
    …
protected:
    int num;
    int *arr;
    …
};
```

Rule of Three (The Big Three)

The **copy constructor**,
the **= assignment operator**,
and the **destructor**
are called the ***rule of three***.

If they are not defined explicitly, all of them are created by the compiler automatically.

If we have to customize one of the three rules, we should customize the other two as well.

If there are pointers which point to dynamically allocated memory spaces, such dynamically allocated memory spaces must be released if they are no longer used. Otherwise, we may have memory leak.

```
class X {
public:
    X() { arr = nullptr;… }
    X(int num) {arr = nullptr;… }
    X( int num, int *arr );
    X( const X &x) { … }
    ~X( ) {
          if (arr) delete [] arr;
    }
    X &operator=
    …
protected:
    int num;
    int *arr;
    …
};
```

Assignment operator

Rule of Three (The Big Three)

The **copy constructor**,

the **= assignment operator**,

and the **destructor**

are called the ***rule of three***.

If they are not defined explicitly, all of them are created by the compiler automatically.

If we have to customize one of the three rules, we should customize the other two as well.

If there are pointers which point to dynamically allocated memory spaces, such dynamically allocated memory spaces must be released if they are no longer used. Otherwise, we may have memory leak.

```
class X {
public:
    X() { arr = nullptr;… }
    X(int num) {arr = nullptr;… }
    X( int num, int *arr );
    X( const X &x) { … }
    ~X( ) {
          if (arr) delete [] arr;
    }
    X &operator=( const
    …
protected:
    int num;
    int *arr;
    …
};
```

Assignment operator

Rule of Three (The Big Three)

The **copy constructor**,

the **= assignment operator**,

and the **destructor**

are called the ***rule of three***.

If they are not defined explicitly, all of them are created by the compiler automatically.

If we have to customize one of the three rules, we should customize the other two as well.

If there are pointers which point to dynamically allocated memory spaces, such dynamically allocated memory spaces must be released if they are no longer used. Otherwise, we may have memory leak.

```
class X {
public:
    X() { arr = nullptr;… }
    X(int num) {arr = nullptr;… }
    X( int num, int *arr );
    X( const X &x) { … }
    ~X( ) {
          if (arr) delete [] arr;
    }
    X &operator=( const X &x );
    …
protected:
    int num;
    int *arr;
    …
};
```

Rule of Three (The Big Three)

The **copy constructor**,
the **= assignment operator**,
and the **destructor**
are called the ***rule of three***.

If they are not defined explicitly, all of them are created by the compiler automatically.

If we have to customize one of the three rules, we should customize the other two as well.

If there are pointers which point to dynamically allocated memory spaces, such dynamically allocated memory spaces must be released if they are no longer used. Otherwise, we may have memory leak.

```
class X {
public:
    X() { arr = nullptr;… }
    X(int num) {arr = nullptr;… }
    X( int num, int *arr );
    X( const X &x) { … }
    ~X( ) {
          if (arr) delete [] arr;
    }
    X &operator=( const X &x );
    …
protected:
    int num;
    int *arr;
    …
};
```

In a member function:
X a, b;
(a = b).num = 6;

Do deep copy….

Intended Learning Outcomes

- Implement operators of a class
- Distinguish between copy constructors and assignment operations
- Distinguish between returning a value and returning a reference
- Explain the properties of a friend function.

Supplemental Materials

Rational Class

Negative: different signs
Positive: same sign
Zero: n is zero

$$a = \frac{n_1}{d_1} \qquad b = \frac{n_2}{d_2}$$

$$a - b = \frac{n_1}{d_1} - \frac{n_2}{d_2} = \frac{n_1 d_2 - n_2 d_1}{d_1 d_2}$$

```
class Rational {
public:
    Rational( );
    Rational( long numerator, long denominator);
public:
    void showInfo( ) const;
    bool operator > (const Rational &second) const;
private:
    long numerator, denominator;
};
```

How to compare two rational objects?

Approach Two

```
bool Rational::operator > (const Rational &second) const
{
    bool flg = false;
    long result_n = n*second.d - d*second.n;
    long result_d = d*second.d;

    if (
        (n > 0 && d > 0)
        ||
        (n < 0 && d < 0)
        ) flg = true;
    return flg;
}
```

$$\frac{n}{d}$$

a > b

n and d belong to the left operand, i.e. a in this case. second is b.

n: numerator   d: denominator

Overloading the << Operator

Do not modify the object

```
// the stream insertion operator
friend ostream& operator<<(ostream& out, const Rational& r)
{
  out << “Numerator: ";
  out << r.numerator;

  out << “Denominator: ";
  out << r.denominator;
  return out;
}
```

```
out << a << b << c;

( (out << a) << b ) << c;
```

Overloading >> Operator

Allow modification

```
// the stream extraction operator
friend istream& operator>>(istream& input, Rational& r)
{
  cout << "Enter numerator: ";
  input >> r.numerator;

  cout << "Enter denominator: ";
  input >> r.denominator;
  return input;
}
```

Overloading the [] Operators

```
long &Rational::operator[ ](int index)
{
  if (index == 0)
    return numerator;
  else if (index == 1)
    return denominator;
  else
  {
    throw runtime_error("subscript incorrect"); // throw an exception
  }
}
// Must have to catch the exception event
```

```
r2[0] = 120;
```

```
r2[1] = 12;
```

&: Return the reference, i.e., alias.

Type Conversion: number to Rational

```
Rational r1(2, 5);
Rational r = r1 + 4; convert 4 to Rational
cout << r << endl;


displays
22 / 5
```

```
Rational::Rational(int numerator) {
   this->numerator = numerator;
   denominator = 1;
}
```

Rational r0(2, 3);

Rational r1 = (Rational)5 + r0;

Rational to double

```
class Rational {
public:
  operator double()
  {
    return getDoubleValue();
  }

protected:
  double getDoubleValue( ) const
  {
     return numerator/(double) denominator;
  }
};
```

Can you add a rational number with an int or a double value?

Yes.

The implementation of the function converts a Rational object to a double value.

```
Rational r1(1, 4);
double d = r1 + 8.1;
cout << "r1 + 8.1 is " << d << endl;
```

Overloading the = Operator

```
class X {
public:
    int v;
    X() { }
    X(int a) { v = a; }
    X &operator=(X &a) {
        v = a.v;
        return (*this);
    }
    friend ostream &operator<<(
     ostream &c, const X &a) {
        c << a.v << endl;
        return c;
    }
};
```

```
class X {
public:
    int v;
    X() { }
    X(int a) { v = a; }
    X &operator=(X &a) {
        v = a.v;
        return (*this);
    }
    friend ostream &
    operator<<(
        ostream &c
      , const X &a
    ) {
        c << a.v << endl;
        return c;
    }
};
```

Examples

Rational(long numerator, long denominator);

Rational b(3, 4), c(4, 9), d(-10, 25);

b > c?

$$\frac{3}{4} - \frac{4}{9}$$

c > d?

Implement the following operator >

```
bool Rational::operator>(const Rational &second) const
```

Operator Functions

```
bool Rational::operator>(const Rational &second) const
{
    bool flg = false;
    long n = numerator*second.denominator - denominator*second.numerator;
    long d = denominator*second.denominator;

    if (
        (n > 0 && d > 0)
        ||
        (n < 0 && d < 0)
        ) flg = true;
    return flg;}
```

$$\mathbf{a} \quad \mathbf{>} \quad \mathbf{b}$$

$$\frac{n1}{d1} - \frac{n2}{d2}$$

$$= \frac{n1*d2 - d1*n2}{d1*d2}$$

Who ‘owns’ the operator?

```
bool Rational::operator>(const Rational &second) const
{
    bool flg = false;
    long n = numerator*second.denominator - denominator*second.numerator;
    long d = denominator*second.denominator;

    if (
        (n > 0 && d > 0)
        ||
        (n < 0 && d < 0)
        ) flg = true;
    return flg;
}
```

```
Rational a(3,4), b(4,9)

bool result = a > b;
```

Precedence and Associativity

```
class X {
public: int v;
    X() { }
    X(int a) { v = a; }
    ~X() {cout << "D" << endl;}
    X &operator=(const X &b) {
        cout << "= a:" << v << endl;
        cout << "= b:" << b.v << endl;
        v = b.v;
        return (*this);
    }
    X &operator-=(const X &b) {
        cout << "-= a:" << v << endl;
        cout << "-= b:" << b.v << endl;
        v -= b.v;
        return (*this);
    }
```

```
    X operator+(const X &b) const {
        cout << "+ a:" << v << endl;
        cout << "+ b:" << b.v << endl;
        X p;
        p.v = v + b.v;
        return p;
    }
};
```

X a(1), b(2), c(5), d(8), e(14);

a = b -= c + d -= e ;

What are the output?

Operator Precedence

- In mathematics and computer programming, the **order of operations** (or **operator precedence**) is a collection of rules that define which procedures to perform first in order to evaluate a given mathematical expression.

1 * 4 - 5 / 6

2 * 4 + 5 - 6

- From wiki

Operator associativity

- In programming languages, the **associativity** (or **fixity**) of an operator is a property that determines how operators of the same precedence are grouped in the absence of parentheses.

1 – 4 - 5 – 6 // left associative

2 / 4 / 5 / 6 // left associative

- From wiki

Operator associativity

- In programming languages, the **associativity** (or **fixity**) of an operator is a property that determines how operators of the same precedence are grouped in the absence of parentheses.

1 – 4 - 5 – 6 // left associative

2 / 4 / 5 / 6 // left associative

a -= b -= d -= e ; // right associative

- From wiki

Operator associativity

- **Left-associative**

  The operations are grouped from the left

  2 / 3 / 4

- **Right-associative**

  The operations are grouped from the right

  a = b = c = d

Operator associativity

- **Left-associative**

  The operations are grouped from the left

  2 / 3 / 4

- **Right-associative**

  The operations are grouped from the right

  a = b = c = d

Overloading the = Operator

a = b;

a = b = c;

Associativity?   right-associative

Exercise
What are the output?

```
class X {
public:
    int v;
    X() { }
    X(int a) { v = a; }
    X operator=(X &a) {
        v = a.v;
        return (*this);
    }
    friend ostream &operator<<(
      ostream &c, const X &a) {
        c << a.v << endl;
        return c;
    }
};
void main () {
    X a(6), b(5);
    b = a;
    cout << a << endl;
    cout << b << endl;
}
```

6

6

```
class X {
public:
    int v;
    X() { }
    X(int a) { v = a; }
    X operator=(X &a) {
        v = a.v;
        return (*this);
    }
    friend ostream &operator<<(
      ostream &c, const X &a) {
        c << a.v << endl;
        return c;
    }
};
void main () {
    X a(6), b(5);
    b = a;
    cout << a << endl;
    cout << b << endl;
}
```

6

6

Exercise

What are the output?

```
class X {
public:
    int v;
    X() { }
    X(int a) { v = a; }
    X operator=(X &a) {
        v = a.v; ++a.v; return (*this);
    }
    friend ostream &operator<<(
      ostream &c, const X &a) {
        c << a.v << endl;
        return c;
    }
};
void main () {
    X a(6), b(5), c(4);
    a = b = c;
    cout << a << endl;
    cout << b << endl;
    cout << c << endl;
}
```

Exercise

What are the output?

```
class X {
public:
    int v;
    X() { }
    X(int a) { v = a; }
    X operator=(X &a) {
        v = a.v; ++a.v; return (*this);
    }
    friend ostream &operator<<(
      ostream &c, const X &a) {
        c << a.v << endl;
        return c;
    }
};
void main () {
    X a(6), b(5), c(4);
    a = b = c;
    cout << a << endl;
    cout << b << endl;
    cout << c << endl;
}
```

4
4
5

Exercise
What are the output?

```cpp
class X {
public:
    int v;
    X() { }
    X(int a) { v = a; }
    X operator=(X &a) {
        v = a.v; ++a.v; return (*this);
    }
    friend ostream &operator<<(
      ostream &c, const X &a) {
        c << a.v << endl;
        return c;
    }
};
void main () {
    X a(6), b(5), c(4);
    a = b = c;
    cout << a << endl;
    cout << b << endl;
    cout << c << endl;
}
```

4
4
5

Exercise
What are the output?

```
a = (b = c)     // b.v = 4
                // c.v = 5

b's = operation returns b1
a = b1          // a.v = 4
                // b1.v = 5
```

```
class X {
public:
    int v;
    X() { }
    X(int a) { v = a; }
    X &operator=(X &a) {
        v = a.v; ++a.v; return (*this);
    }
    friend ostream &operator<<(
      ostream &c, const X &a) {
        c << a.v << endl;
        return c;
    }
};
void main () {
    X a(6), b(5), c(4);
    a = b = c;
    cout << a << endl;
    cout << b << endl;
    cout << c << endl;
}
```

Exercise

What are the output?

```cpp
class X {
public:
    int v;
    X() { }
    X(int a) { v = a; }
    X &operator=(X &a) {
        v = a.v; ++a.v; return (*this);
    }
    friend ostream &operator<<(
      ostream &c, const X &a) {
        c << a.v << endl;
        return c;
    }
};
void main () {
    X a(6), b(5), c(4);
    a = b = c;
    cout << a << endl;
    cout << b << endl;
    cout << c << endl;
}
```

```
4
5
5
```

Exercise

What are the output?

Exercise
What are the output?

```
class X {
public:
    int v;
    X() { }
    X(int a) { v = a; }
    X &operator=(X &a) {
        v = a.v; ++a.v; return (*this);
    }
    friend ostream &operator<<(
      ostream &c, const X &a) {
        c << a.v << endl;
        return c;
    }
};
void main () {
    X a(6), b(5), c(4);
    a = b = c;
    cout << a << endl;
    cout << b << endl;
    cout << c << endl;
}
```

```
4
5
5
```

```
a = (b = c)      // b.v = 4
                 // c.v = 5

b's = operation returns b
a = b            // a.v = 4
                 // b.v = 5
```

Overloading the = Operator

```
class X {
public:
    int v;
    X() { }
    X(int a) { v = a; }
    ~X( ) { cout << "D" << endl;}
    X operator=(X &a) {
        v = a.v;
        return (*this);
    }
    friend ostream &operator<<(
     ostream &c, const X &a) {
        c << a.v << endl;
        return c;
    }
};
```

```
void main () {
    X a(6), b(5);
    b = a;
    cout << a << endl;
    cout << b << endl;
}
//what occurs after a=b?
```

What are the output?

Overloading the = Operator

```
class X {
public:
    int v;
    X() { }
    X(int a) { v = a; }
    ~X( ) { cout << "D" << endl;}
    X operator=(X &a) {
        v = a.v;
        return (*this);
    }
    friend ostream &operator<<(
     ostream &c, const X &a) {
        c << a.v << endl;
        return c;
    }
};
```

```
void main () {
    X a(6), b(5);
    b = a;
    cout << a << endl;
    cout << b << endl;
}
//what occurs after a=b?
```

What are the output?

D

6

6

Overloading the = Operator

```
class X {
public:
    int v;
    X() { }
    X(int a) { v = a; }
    ~X( ) { cout << "D" << endl;}
    X operator=(X &a) {
        v = a.v;
        return (*this);
    }
    friend ostream &operator<<(
     ostream &c, const X &a) {
        c << a.v << endl;
        return c;
    }
};
```

```
void main () {
    X a(6), b(5);
    b = a;
    cout << a << endl;
    cout << b << endl;
}
//what occurs after a=b?

(b=a).v = 7; // not good
```