# Unary Operators

## Lecture 26
## Section xx.xx

Robb T. Koether

Hampden-Sydney College

Mon, Nov 4, 2019

# Outline

# Outline

# Operators that Must be Member Functions

- The following operators must be implemented as member functions.
  - The assignment operator `=` (Lab 9).
  - The subscript operator `[]` (CS II).

# Outline

# Unary Operators

- Unary operators should be implemented as member functions.
- The operator is invoked by a single operand.
- The expression `-a` is interpreted as `a.operator-()`
- There is no issue of left operand vs. right operand.

# Outline

# The Negation Operator

## The Negation Operator

```
type type::operator-() const
{
//  Compute the negative
        :
    return negated-value;
}
```

- The negation operator (unary $-$) returns the negative of the value of the invoking object.

# The Negation Operator

## The `Rational` Negation Operator

```
Rational Rational::operator-() const
{
//  Return the negative
    return Rational(-nNumerator, mDenominator);
}
```

- The negation operator (unary $-$) returns the negative of the value of the invoking object.

# Outline

# The Assignment Operator =

## The Assignment Operator =

```
Rational& Rational::operator=(const Rational& r)
{
    mNumerator = r.mNumerator;
    mDenominator = r.mDenominator;
    return *this;
}
```

- Recall the definition of the assignment operator from Lab 9.

# The `this` Pointer

- Every member function (except **`static`** functions) includes a reference to the invoking object.

# The `this` Pointer

- Every member function (except **`static`** functions) includes a reference to the invoking object.
- The variable built-in "variable" **`this`** holds the address of the invoking object.

# The `this` Pointer

- Every member function (except **`static`** functions) includes a reference to the invoking object.
- The variable built-in "variable" **`this`** holds the address of the invoking object.
- The invoking object itself if obtained by applying the dereferencing operator `*`:

```
*this
```

# Outline

# The Pre-Increment Operator ++

## The Pre-Increment Operator ++

```
type& type::operator++()
{
    //  Increment the object
        ⋮
    return *this;
}
```

- The pre-increment operator should return the object *by reference*.
- That is, the return value is the very object that invoked ++, not a copy.
- The expression uses the returned value ("increment before use").
- What will `++(++a)` do?

# The `Rational ++` Operator

## The `Rational ++` Operator

```
Rational& Rational::operator++()
{
    mNumerator += mDenominator;
    return *this;
}
```

- We define $(a/b)$++ to be $(a/b) + 1$, which is the rational

$$(a + b)/b.$$

- Thus, we simply add the denominator to the numerator.
- (The denominator remains the same.)

# The Pre-Decrement Operator --

## The Pre-Decrement Operator --

```
type& type::operator--()
{
    //  Decrement the object
        :
    return *this;
}
```

- The pre-decrement operator – follows exactly the same pattern.

# The `Rational --` Operator

## The `Rational --` Operator

```
Rational& Rational::operator--()
{
    mNumerator -= mDenominator;
    return *this;
}
```

- We define $(a/b)--$ to be $(a/b) - 1$, which is the rational

$$(a - b)/b.$$

- Thus, we simply subtract the denominator from the numerator.
- (The denominator remains the same.)

# The Post-Increment Operator

- The post-increment operator is somewhat different from the pre-increment operator.
- It must increment (change) the object, but it must also return the value of the object before it was changed.
- The only way to accomplish this is to
  - Save a copy of the invoking object before incrementing it.
  - Increment the invoking object.
  - Return the copy of the unchanged object.

# The Post-Increment Operator

- The post-increment operator should return the object *by value*.
- That is, the value returned is a *copy* of the invoking object, before it was incremented.
- A consequence is that expressions such as `(a++)++` are not permitted.

# The Post-Increment Operator

- Another problem is that, so far, the pre-increment and post-increment operators have identical prototypes, except for the precise return type (by value vs. by reference):

```
type type::operator++();
```

- To remedy this, we include one unused and unnamed **`int`** parameter to distinguish post-increment from pre-increment.
- (This is a completely artificial mechanism.)

# The Post-Increment Operator

## The Post-Increment Operator

```
type type::operator++(int) // int is unnamed, unused
{
    type original = *this;
    //  Increment the object, using *this
        ⋮
    return original;
}
```

# The Post-Increment Operator

## The Post-Increment Operator

```
Rational Rational::operator++(int)
{
    Rational original = *this;
    mNumerator += mDenominator;
    return original;
}
```

# Example

## Example (Example)

- Add negation, pre- and post-increment, and pre- and post-decrement to the `Rational` class.

# Outline

# Converting Dates to Integers

- To facilitate calculations with dates, we will write functions that will convert dates to integers and integers to dates.
- Our scheme is to assign 0 to Jan 1, 1601; 1 to Jan 2, 1602; and so on.
- If we create the appropriate functions, then we can "cast" a `Date` object as an integer and cast an integer as a `Date` object.

# The `Date` class `add()` Function

## The `Date` class `add()` Function

```
Date Date::add(int n) const
{
    return Date(int(*this) + n);
}
```

- To add `n` days to a date, we will
  - Convert the date to an integer.
  - Add `n` to the integer.
  - Convert the integer to a date.

- For the `Date` class, how would we implement
  - The pre-increment operator `++`?
  - The post-increment operator `++`?