# Type Conversions
## Lecture 8

Robb T. Koether

Hampden-Sydney College

Wed, Jan 31, 2018

1 Conversion to Non-primitive Types

2 Conversion to Primitive Types

# Outline

1. Conversion to Non-primitive Types

2. Conversion to Primitive Types

# Conversion of Types

- Frequently in a program an object must be converted from one type to another.
- For the primitive types, this is done automatically whenever it is sensible and unambiguous.
  - Convert **`float`** to **`int`**.
  - Convert **`int`** to **`float`**.
- How can it be done with non-primitive types?

# Converting to a Non-primitive Type

## Converting to a Non-primitive Type

```
Type::Type(Other-type);
```

- A class uses its constructors to define rules for converting an object of another type to an object of that type.

# Example

## Example (Convert **int** to Rational)

```
//  Rational constructor
    Rational::Rational(int n)
    {
        num = n;
        den = 1;
        return;
    }

//  Usages
    Rational r(100);
    Rational r = 100;
    r = (Rational)100;
    r = Rational(100);
```

# Example

- How would you convert a `Point` with components

```
double m_x;
double m_y;
```

  to a `Vectr` with components

```
int m_size;
double* m_element;
```

# Outline


1 Conversion to Non-primitive Types

2 Conversion to Primitive Types

# Converting to a Primitive Type

- Sometimes we want to convert an object of a non-primitive type to an object of a primitive type.
- For example, we might want to convert
  - A `Rational` to a **`double`**.
  - A `Date` to an **`int`**.
- For this we need a conversion operator.

# Conversion Operators

## Conversion Operator Prototype

```
Type::operator primitive-type() const;
```

## Conversion Operator Usage

```
(primitive-type)Object;   // Old style (casting)
primitive-type(Object);   // New style (function call)
```

- The operator converts the non-primitive-type object to the primitive type and returns the object of the primitive type.

# Conversion Operators

**Example (Conversion Operators)**

```
Rational::operator double() const;
Date::operator int() const;
```

# Example

**Example (Convert `Rational` to `double`)**

```
Rational::operator double() const
{
    return (double)num/ (double)den;
}
```

# Example

## Example (Convert Date to int)

```
enum Month (Jan, Feb, Mar, ..., Dec);
int days_in_month[] = {31, 28, 31, ..., 31};

Date::operator int() const
{
    int years = m_year - 1601;          // Since 1601
    int day_number = 365 * years;       // 365 days/year
    day_number += (years / 4);          // For leap years
    day_number -= (years / 100);        // For century years
    day_number += (years / 400);        // For cntry leap years

    for (Month m = Jan; m < m_month; m = (Month) (m + 1))
        day_number += days_in_month(m, m_year);  // Past months

    day_number += m_day - 1;            // This month
    return day_number;
}
```

# Example

**Example (Convert `Date` to `int`)**

```
Date start("Jan", 31, 2018);
Date stop("Dec", 25, 2018);
int elapsed = stop - start;
```

- What exactly happens when the above code is executed?
- What would happen if we also had a function that would convert a `Date` object to a floating-point number of days?