



UNIVERSITY OF WATERLOO
FACULTY OF ENGINEERING
Department of Electrical & Computer Engineering

ECE 150 *Fundamentals of Programming*

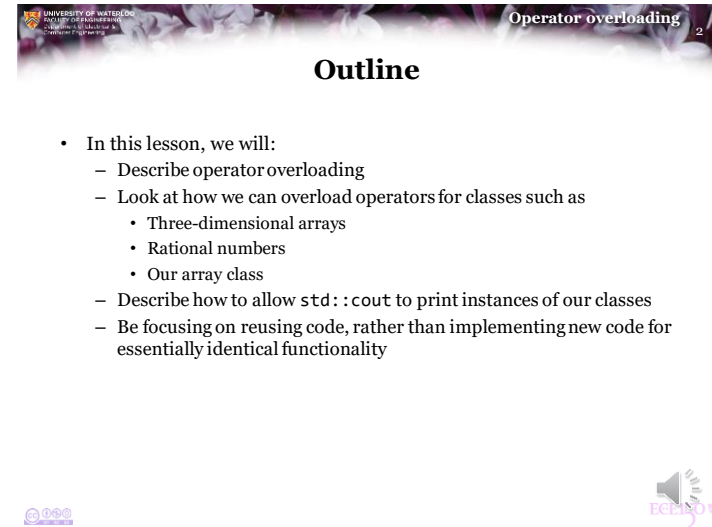
Operator overloading

CC BY NC SA

Douglas Wilhelm Harder, M.Math. LEL
Prof. Hiren Patel, Ph.D., P.Eng.
Prof. Werner Dietl, Ph.D.

© 2018 by Douglas Wilhelm Harder and Hiren Patel. Some rights reserved.

1



UNIVERSITY OF WATERLOO
FACULTY OF ENGINEERING
Department of Electrical & Computer Engineering

Operator overloading 2

Outline

- In this lesson, we will:
 - Describe operator overloading
 - Look at how we can overload operators for classes such as
 - Three-dimensional arrays
 - Rational numbers
 - Our array class
 - Describe how to allow `std::cout` to print instances of our classes
 - Be focusing on reusing code, rather than implementing new code for essentially identical functionality

CC BY NC SA

2



UNIVERSITY OF WATERLOO
FACULTY OF ENGINEERING
Department of Electrical & Computer Engineering

Operator overloading 3

Functions on vectors

- In the previous topic, we described classes for:
 - 3-dimensional vectors
 - Rational numbers
 - Arrays
 - Pair

3



UNIVERSITY OF WATERLOO
FACULTY OF ENGINEERING
Department of Electrical & Computer Engineering

Operator overloading 4

Using vectors

- Contrast the use of integers and rational numbers:

```
int main() {
    int m{ 15 };
    int n{ 17 };
    Rational p{ 15, 1 };
    Rational q{ 17, 1 };

    std::cout << "m + n = " << (m + n) << std::endl;
    std::cout << "p + q = "
        << to_string( add(p, q) ) << std::endl;
    std::cout << "-m = " << (-m) << std::endl;
    std::cout << "-p = "
        << to_string( negate( p ) ) << std::endl;

    return 0;
}
```

4



Using vectors

- Wouldn't this be nice?

```
int main() {
    int m{ 15 };
    int n{ 17 };
    Rational p{ 15, 1 };
    Rational q{ 17, 1 };

    std::cout << "m + n = " << (m + n) << std::endl;
    std::cout << "p + q = " << (p + q) << std::endl;

    std::cout << "-m = " << (-m) << std::endl;
    std::cout << "-p = " << (-p) << std::endl;

    return 0;
}
```

5



Functions on vectors

- What we need is a way to tell the compiler:
 - If you have *rational + rational*, then call an appropriate function
 - If you have *-rational*, then call a different appropriate function
- This is done through *operator overloading*

```
Rational operator+( Rational const &p, Rational const &q );
```
- If the left and right operands of a binary "+" are instances of the class *Rational*, then call this function


```
Rational operator+( Rational const &p, Rational const &q ) {
    return Rational{ p.numer_*q.denom_ + q.numer_*p.denom_,
                    p.denom_*q.denom_ };
}
```

6



Operator overloading

- For almost every operator in C++, that operator can be overloaded for new operands
 - This is true for all arithmetic, comparison, logical, bitwise, bit-shifting, assignment and auto-assignment operators
 - The few that cannot be overloaded such as `::` or `.`

7



Operators on vectors

- Thus, all the previous functions we implemented for vectors could implemented using operator overloading:


```
Vector_3d operator+( Vector_3d const &u, Vector_3d const &v );
Vector_3d operator*( Vector_3d const &u, double const s );
Vector_3d operator*( double const s, Vector_3d const &u );
bool operator==( Vector_3d const &u, Vector_3d const &v );
```
- Important: The compiler knows NOTHING about arithmetic properties such as commutativity or symmetry
 - You must implement all variations such as **su** or **us**

8



Operators on vectors

- To be fair, we should also implement any other operator that may a user may use
- ```
// Unary + and -
Vector_3d operator+(Vector_3d const &u);
Vector_3d operator-(Vector_3d const &u);

// Binary - and /
Vector_3d operator-(Vector_3d const &u, Vector_3d const &v);
Vector_3d operator/(Vector_3d const &u, double const s);

// Binary !=
bool operator!=(Vector_3d const &u, Vector_3d const &v);
```



9



## Operators on vectors

- To be fair, we should also implement any other operator that may a user may use
- ```
// Unary + and -
Vector_3d operator+( Vector_3d const &u );
Vector_3d operator-( Vector_3d const &u );

// Binary - and /
Vector_3d operator-( Vector_3d const &u, Vector_3d const &v );
Vector_3d operator/( Vector_3d const &u, double const s );

// Binary !=
bool operator!=( Vector_3d const &u, Vector_3d const &v );
```



10



Operators on vectors

- One rule we will focus on is the idea of reuse:
 - If you can implement something using a function already written, do so
- ```
Vector_3d operator-(Vector_3d const &u) {
 return Vector_3d{ -u.x_, -u.y_, -u.z_ };
}

Vector_3d operator-(Vector_3d const &u, Vector_3d const &v) {
 return u + (-v);
}

bool operator==(Vector_3d const &u, Vector_3d const &v) {
 return (u.x_ == v.x_) && (u.y_ == v.y_) && (u.z_ == v.z_);
}

bool operator!=(Vector_3d const &u, Vector_3d const &v) {
 return !(u == v);
}
```



11



## Operators on vectors

- What not to overload?
  - Can you divide a scalar by a vector?
 

```
Vector_3d operator/(double const s, Vector_3d const &u);
```

    - What should this do?
 

```
double operator*(Vector_3d const &u, Vector_3d const &v);
Vector_3d operator*(Vector_3d const &u, Vector_3d const &v);
```

      - Is it the inner product, or is it the cross product?
      - For clarity, this author would leave both as named functions:
 

```
inner(...) and cross(...)
```
  - Should `u + 2.5` add 2.5 to each entry of the vector `u`?
 

```
Vector_3d operator+(Vector_3d const &u, double const s);
Vector_3d operator+(double const s, Vector_3d const &u);
```



12

UNIVERSITY OF WATERLOO  
Faculty of Engineering  
Department of Electrical and Computer Engineering

Operator overloading 13

## Operators on vectors

- You can also overload all the automatic assignment operators:
 

```
Vector_3d &operator+=(Vector_3d &u, Vector_3d const &v);
Vector_3d &operator-=(Vector_3d &u, Vector_3d const &v);
Vector_3d &operator*=(Vector_3d &u, double const s);
Vector_3d &operator/=(Vector_3d &u, double const s);
```



13



UNIVERSITY OF WATERLOO  
Faculty of Engineering  
Department of Electrical and Computer Engineering

Operator overloading 14

## Operators on vectors

- In each of these, we could reuse code we have already authored:
 

```
Vector_3d &operator+=(Vector_3d &u, Vector_3d const &v) {
 u = u + v;
 return u;
}
```
- One critical point: the automatic assignment operators must return the left-hand operand by reference



14



UNIVERSITY OF WATERLOO  
Faculty of Engineering  
Department of Electrical and Computer Engineering

Operator overloading 15

## Operators on vectors

- What is most desirable, perhaps, however, is to get the vector class to work with `std::cout`

```
std::cout << u;
- The class of std::cout is std::ostream
std::ostream &operator<<(ostream &out, Vector_3d const &u) {
 out << "(" << u.x_ << ", " << u.y_ << ", " << u.z_ << ")";

 return out;
}
- Inside, treat the parameter out as if you would std::cout
- The argument std::cout is passed by reference,
 and returned by reference
```



15



UNIVERSITY OF WATERLOO  
Faculty of Engineering  
Department of Electrical and Computer Engineering

Operator overloading 16

## Rational number class

- Next, let's look at overriding operators for our rational number class
 

```
class Rational {
public:
 int numer_;
 int denom_;
};
```
- There are many more operations that can be performed on rational numbers, including all arithmetic operations but also comparison operators



16





## Rational number class

- Again, reuse is very powerful here, for if one or two functions are authored correct, so are the rest:

```
bool operator==(Rational const &p, Rational const &q) {
 return p.numer_*q.denom_ == q.numer_*p.denom_;
}
```

```
bool operator!=(Rational const &p, Rational const &q) {
 return !(p == q);
}
```



17



## Rational number class

- Here is comparing if  $p < q$ :

```
bool operator<(Rational const &p, Rational const &q) {
 if (p.denom_*q.denom_ > 0) {
 return p.numer_*q.denom_ < q.numer_*p.denom_;
 } else {
 return p.numer_*q.denom_ > q.numer_*p.denom_;
 }
}
```

```
bool operator<=(Rational const &p, Rational const &q) {
 return (p < q) || (p == q);
}
```

```
bool operator>(Rational const &p, Rational const &q) {
 return !(p <= q);
}
```

```
bool operator>=(Rational const &p, Rational const &q) {
 return !(p < q);
}
```



18



## Rational number class

- As for `std::cout`,

```
- We always want to see printed 3/5 or -2/5, and not -3/-5 or 2/-5
std::ostream operator<<(std::ostream &out, Rational const &p) {
 if (p.denom_ < 0) {
 out << -p.numer_ << "/" << -p.denom_;
 } else {
 out << p.numer_ << "/" << p.denom_;
 }
}
```

- Also, should we print integer rational numbers as integers without a denominator?

- For example, 1 instead of 3/3,  
-5 instead of -5/1



19



## Array class

- For the array class, we can get it to print using `std::cout`:

```
std::ostream &operator<<(std::ostream &out, Array const &array) {
 if ((array.array_ == nullptr) || (array.capacity_ == 0)) {
 out << "{}";
 } else {
 out << "{" << array.array_[0];

 for (std::size_t k{1}; k < array.capacity_; ++k) {
 out << ", " << array.array_[k];
 }

 out << "}";
 }

 return out;
}
```



20



## Pair class

- For the Pair class, we may consider overloading == and != if this was considered appropriate

```
// Class declarations
class Pair;

// Class definitions
class Pair {
public:
 std::size_t first_;
 std::size_t second_;
};
```



21



## Complex numbers

- Now, in your linear algebra and calculus courses, you have likely already been exposed to complex numbers
  - How could you implement a complex number class?
    - What operators could you implement?
    - What additional functions would be required?



22



## Summary

- Following this lesson, you now
  - Have seen numerous examples of operator overloading
  - Understand that operator overloading can make programming much more intuitive
  - Specifically understand how to extend the printing capabilities of `std::cout` to print objects from classes you have authored



23



## References

- [1] [https://en.wikipedia.org/wiki/C++\\_classes](https://en.wikipedia.org/wiki/C++_classes)
- [2] [https://en.wikipedia.org/wiki/Operator\\_overloading](https://en.wikipedia.org/wiki/Operator_overloading)



24



## Colophon

These slides were prepared using the Georgia typeface. Mathematical equations use Times New Roman, and source code is presented using Consolas.

The photographs of lilacs in bloom appearing on the title slide and accenting the top of each other slide were taken at the Royal Botanical Gardens on May 27, 2018 by Douglas Wilhelm Harder. Please see

<https://www.rbg.ca/>

for more information.



25



## Disclaimer

These slides are provided for the ECE 150 *Fundamentals of Programming* course taught at the University of Waterloo. The material in it reflects the authors' best judgment in light of the information available to them at the time of preparation. Any reliance on these course slides by any party for any other purpose are the responsibility of such parties. The authors accept no responsibility for damages, if any, suffered by any party as a result of decisions made or actions based on these course slides for any other purpose than that for which it was intended.



26

