

# Operator Overloading

Michael Hanke

School of Engineering Sciences

Program construction in C++ for Scientific Computing



# Outline

- 1 Introduction
- 2 Copy-Assignment
- 3 Operator Overloading
- 4 Good Practices
- 5 Overloading I/O Operators
- 6 Summary

# Introduction

- Our Point class is a model for the vector space  $\mathbb{R}^2$ . In this space, operations like vector addition and multiplication by a scalar are defined.
- What we can do right now: Define a function

```
const Point add(const Point &P, const Point &Q);  
Point W = add(P,Q);
```

Note: The symbol = indicates an initialization of W. This is the copy constructor.

- Wish: How can we make sense to statements like

```
Point P, Q, W;  
W = P+Q;
```

Note: Here, the = symbol stands for the assignment operator. We will need another type of constructor here (copy-assignment).

# Operators

## Operator Overloading

In C++, operators are considered to be special kind of function (with predefined structure of the argument list, compatible with the C++ syntax). So they can be redefined for new data types. This is called overloading.

# Lvalues and Rvalues

- In C, **lvalues** are expressions which can appear at the left-hand side of the assignment operator; **rvalues** appear on the right-hand side of it:

$$lvalue = rvalue;$$

- The semantics is that rvalue will be evaluated and assigned to the lvalue.
- **The assignment operator is right-associative** meaning that  $a = b = c$ ; is equivalent to  $a = (b = c)$ ;
- Consequence: *An assignment  $a = b$  has a value!*
- Question: What is the result of  $(a = b) = c$ ? (Demo!)

## Lvalues and Rvalues (cont)

- Operators differ as to whether they require an lvalue or an rvalue as operands or whether they return lvalues or rvalues.
- C++ has a rather complex behavior here.
- Heuristically:
  - When we use an object as rvalue, we use the object's value (its contents).
  - When we use an object as lvalue, we use the object's identity (its address).
- Most often, an lvalue can be used when an rvalue is required.
- Example: An expression like  $a + b$  can never be an lvalue.

*Why?*

# Operators in C++

- Arithmetic operators (+, -, \*, /, %) - standard precedence and associativity rules
- Logical and relational operators - standard precedence and associativity rules
- Assignment operator (=) - right associative, lowest precedence
- Increment and decrement operators (i++, ++i, i--, --i)
- Member access operator (left associative), function call, subscript (., (), [])
- many more

All these operators can be overloaded (exception here: member access operator)!

A Final Bit To Know: *this*

- When calling a member function there is always a concrete object involved:

```
Point P, Q; P.X();
```

The latter will return the contents of `x` of the instance `P`.

- How can the runtime system decide between `x` belonging to `P` and the one belonging to `Q`?
- The distinction is provide by an implicit variable defined by the compiler for every class:

```
class *this;
```

- In our example, the statement `return x;` is interpreted as

```
return (*this).x;
```

equivalently, `return this -> x;`

- All member functions have the implicit first argument `this`.



## Aim

How to define a version of the assignment operator such that an assignment of the kind

$$P = Q = W;$$

makes sense?

- Since the interpretation is rather close to the copy constructor, we should expect similar properties.
- However, as an operator, the assignment must have a value.
- After assignment both objects should be the same (but not identical!).
- The value returned should be (a reference to) the rightmost expression.
- For consistency with the built-in types, it should be a reference to the left-hand operator.

# The Copy-Assignment Constructor

- The copy-assignment constructor **must** be a public member function.
- It **must** have the form

```
class& operator=(const class&);
```
- The `this` pointer points to the left-hand side operand of the assignment.
- The argument should be a reference.

## Extending The Point Class

```
class Point {
private:          // Can be omitted here
    double x;
    double y;
public:
    Point(double xx = 0.0, double yy = 0.0) :
        x(xx), y(yy) { }
    Point(const Point& Q): x(Q.x), y(Q.y) { }
    ~Point() { }
    double X() const { return x; }
    double Y() const { return y; }
    void zero() { x = y = 0.0; }
};
```

## The Point Class (cont)

```
Point& Point::operator=(const Point& P) {  
    if (this != &P) {  
        x = P.x; // equivalent: (*this).x = P.x;  
        y = P.y;  
    }  
    return *this; // dereferencing!  
}
```

- This copy-assignment constructor corresponds to the automatically generated one.
- As a rule of thumb, an individual version is necessary if you need an individual copy constructor.

- We can write simply  $P = Q = W$ ; now if  $P$ ,  $Q$ , and  $W$  are instances of class `Point`.
- We can even write  $P = 1.0$ ; since we have available a type conversion `double` to `Point`.
- The latter is slightly inefficient because first, a temporary object of class `Point` is created and only then the assignment takes place.
- For efficiency reasons, it might be better to have an explicit definition:

```
Point& Point::operator=(const double& xx) {  
    x = xx;  
    y = 0.0;  
    return *this;  
}
```

# A First Operator

- Previously, we defined the negative of a Point (**nonmember function**):

```
const Point negative(const Point& P) {  
    return Point(-P.X(), -P.Y());  
}
```

- This can easily be transformed into a unary minus operator (**member function**):

```
const Point Point::operator-() const {  
    return Point(-x, -y);  
}
```

- Note the first implicit parameter `this`!
  - The old object will not change, therefore `const`.
  - As previously, the result cannot be a reference.
- Now we can write

```
P = -Q;
```

# Another Operator: +=

It is as simple as this:

```
const Point& Point::operator+=(const Point& Q) {  
    x += Q.x;  
    y += Q.y;  
    return *this;  
}
```

- We can write now  $W = P += Q$ ; but not  $(P += Q) = W$ . *Why?*  
*Design error?*
- We can also write  $P += 1.0$ ; Might be better to define it explicitly.

## And Finally: +

```
const Point Point::operator+(const Point& Q) const {  
    return Point(x+Q.x,y+Q.y);  
}
```

Note: Creation of a temporary object!



# Compare

```
Point& operator= (const Point&);  
const Point operator- () const;  
const Point& operator+=(const Point&);  
const Point operator+ (const Point&) const;
```

# A Final Subtlety

- The following code is valid:

```
Point P, Q(1.0,2.0);  
P = Q+3;
```

Note: The `int 3` is converted to a `double` is converted to a `Point`.

- Addition should be commutative. However, the expression `P = 3.0+Q;` leads to a compile time error. *Why?*
- *Operators defined as member functions are “unsymmetric”!*

We will need operators that are not member functions.

## Friend Functions

- For implementing our operation, we would need a nonmember function of the kind

```
const Point operator+(double x, const Point& Q);
```

- Even if this function belongs to the interface, it is not part of the class. Consequently, it does not have access to the private members.
- Access can be granted by providing the `friend` attribute in the class declaration:

```
class Point {
    friend const Point operator+(const double x,
                                const Point& Q);
}
```

- In the implementation, the actual definition takes place:

```
const Point operator+(double x, const Point& Q) {
    return Point(x+Q.x,Q.y);
}
```

- In the previous case, a non-friend, nonmember implementation could have been provided:

```
const Point operator+(double x, const Point& Q) {  
    return Q+x;  
}
```

- It is slightly more expensive because of the additional function call.
- It is slightly better maintainable because it does not use the internals of `Point` directly.

`typetst.cpp`

# Good Practices

- Define overloaded operators consistently with the user's expectations. For example,
  - $P + Q$  and  $P += Q$  should deliver identical values.
  - The operator  $+$  should not be overloaded with a subtraction-like operation.
- If the class does **I/O**, define the shift operator consistently with those of the built-in types.
- If a class has `operator==`, it should also provide `operator!=`.
- Be careful when overloading **logical operators**. Evaluation rules of the built-in functions do not survive (short-circuit evaluation).
- Assignment and compound assignment should return a reference to the left-hand operand.
- If a (commutative) binary operator accepts operands of different types, both orders should be available.

# Conventions

- Similarly to arithmetic or assignment operators, both << and >> should return a reference to its left-hand argument.
- << and >> are left associative. So the left-hand argument is always a stream.
- So the overloaded operators can neither be a member of the stream class (we cannot add members to library classes) nor a member of our own class.
- The declaration looks something like this:

```
ostream& operator<<(ostream& os, const class& item)  
istream& operator>>(istream& is,      class& item)
```

- Note: Input operators must deal with the possibility that the input might fail; output operators usually don't bother.

## Example: &lt;&lt; for Point

```
ostream& operator<<(ostream& os, const Point& P) {  
    os << "(" << P.X() << ", " << P.Y() << " )";  
    return os;  
}
```

Note: This operator could be made slightly more efficient by defining it as a friend.

*How?*



## Example: &gt;&gt; for Point

```
istream& operator>>(istream& is, Point& P) {  
    double x, y;  
    is >> x >> y;  
    if (is) // Success?  
        P = Point(x,y);  
    else  
        P = Point();  
    return is;  
}
```

# Summary

- How to overload operators
  - Assignment operators
  - this
  - friend
- 
- What comes next:
    - Some more details about I/O