OPERATOR OVERLOADING

Using standard library functions and providing an intuitive interface

---

Florian Warg, Max Staff

May 11, 2017

How do you sort your own objects?

- do not use your own containers
- do not build your own sorting function
- use the standard library

How do you sort your own objects?

- do not use your own containers
- do not build your own sorting function
- use the standard library
- solution: build own comparison operators

How do you sort your own objects?

- do not use your own containers
- do not build your own sorting function
- use the standard library
- solution: build own comparison operators
- also helps for making your object ”printable”...

DECLARING AN OPERATOR I

```
class Gear {
    float gearRatio;
public:
    float getRatio() { return this->gearRatio; }
    bool operator<(const Gear& other);
    bool operator==(const Gear& other);

    bool operator<=(const Gear& other);
    bool operator>=(const Gear& other);
    bool operator>(const Gear& other);
    bool operator!=(const Gear& other);
}
```

DECLARING AN OPERATOR II

```
class Matrix {
    int content[width * height];
public:
    int* operator[](std::size_t index);
    Matrix operator+(const Matrix& other) const;
    Matrix& operator+=(const Matrix& other);
}

Matrix a, b;
a += b;
Matrix c = a + b;
```

DECLARING AN OPERATOR III

```
class Matrix {
    int content[width * height];
public:
    Matrix& operator++();
    Matrix operator++(int);
    Matrix& operator--();
    Matrix operator--(int);
}

Matrix a;
++a;
a++;
--a;
a--;
```

OVERLOADING AN INCREMENT OPERATOR

```
Matrix& Matrix::operator++() {
    for (auto& i : this->content) {
        ++i; // increment
    }
    return *this; // return incremented state
}

Matrix Matrix::operator++(int) {
    Matrix tmp(*this); // copy
    this->operator++(); // increment
    return tmp; // return previous state
}

Matrix a;
++a;
a++;
```

OVERLOADING A FUNCTION OPERATOR

```
class Matrix {
    int content[width * height];
public:
    Matrix& operator()(std::size_t x, std::size_t y);
}

int Matrix::operator()(std::size_t x, std::size_t y) {
    return this->content[y * width + x];
}

Matrix a;
int b = a(5, 3);
```

FUNCTORS

- Functor = object with overloaded `( )`-operator
- better optimizable than function-pointers
- useful with templates (coming soon)

WHICH OPERATORS ARE OVERLOADABLE?

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

WHICH OPERATORS ARE NOT OVERLOADABLE?

- `::` scope resolution
- `.` member access
- `.*` member access through pointer to member
- `?:` ternary conditional

WHICH OPERATORS ARE NOT OVERLOADABLE?

- `::` scope resolution
- `.` member access
- `.*` member access through pointer to member
- `?:` ternary conditional
- new or nonexistent operators (`**`, `<>`, `&|`)

WHICH OPERATORS ARE NOT OVERLOADABLE?

- `::` scope resolution
- `.` member access
- `.*` member access through pointer to member
- `?:` ternary conditional
- new or nonexistent operators (`**`, `<>`, `&|`)
- `&&`, `||` lose short-circuit (aka ”lazy”) evaluation

WHICH OPERATORS ARE NOT OVERLOADABLE?

- `::` scope resolution
- `.` member access
- `.*` member access through pointer to member
- `?:` ternary conditional
- new or nonexistent operators (`**`, `<>`, `&|`)
- `&&`, `||` lose short-circuit (aka ”lazy”) evaluation
- precedence, grouping or amount of operands cannot be changed

OVERLOADING OUTSIDE OF THE CLASS

expression	member function	non-member
`@a`	`(a).operator@()`	`operator@(a)`
`a@b`	`(a).operator@(b)`	`operator@(a, b)`
`a=b`	`(a).operator=(b)`	not possible
`a(b...)`	`(a).operator()(b...)`	not possible
`a[b]`	`(a).operator[](b)`	not possible
`a->`	`a.operator->()`	not possible
`a@`	`a.operator@(0)`	`operator@(a, 0)`

EXAMPLE

```
class Matrix {
    int content[width * height];
    // ...
public:
    void print(std::ostream& stream) const;
}

std::ostream& operator<<(std::ostream& stream, const Mat
    matrix.print(stream);
    return stream;
}

Matrix a, b;
std::cout << a << b << "\n";
```

WHEN TO OVERLOAD

Three Basic Rules of Operator Overloading in C++

- Only overload if the meaning of the operator is obviously clear and undisputed
- Always stick to the operator's well-known semantics
- Always provide all out of a set of related operations

Source: http://stackoverflow.com/q/4421706/3729508

HOW TO OVERLOAD

The Decision between Member and Non-member

- Some operators are required to be members
- unary operators should be members
- if one operand is treated specially (e.g. is modified) should be members
- if both operands are treated equally (e.g. addition) should be non-member
- if access to private attributes is required should be non-member
- try to avoid declaring friend functions

Source: http://stackoverflow.com/q/4421706/3729508

WHICH COMPARISON OPERATORS TO USE

- standard library algorithms (e.g. `std::sort`) will always expect `operator<`
- users expect other operators if `operator<` exists

```
bool operator==(const X& l, const X& r){ /* actually compare */ }
bool operator!=(const X& l, const X& r){return !operator==(l,r);}
bool operator< (const X& l, const X& r){ /* actually compare */ }
bool operator> (const X& l, const X& r){return  operator< (r,l);}
bool operator<=(const X& l, const X& r){return !operator> (l,r);}
bool operator>=(const X& l, const X& r){return !operator< (l,r);}
```

Source: http://stackoverflow.com/q/4421706/3729508

CONVERSION OPERATORS

- conversion to other types (e.g. bool, string) can be overloaded
- difference between implicit and explicit conversion

```
class MyString {
public:
    operator const char*() const; {return data_;} // implicit
    explicit operator const char*() const {return data_;}
private:
    const char* data_;
}
```

Source: http://stackoverflow.com/q/4421706/3729508