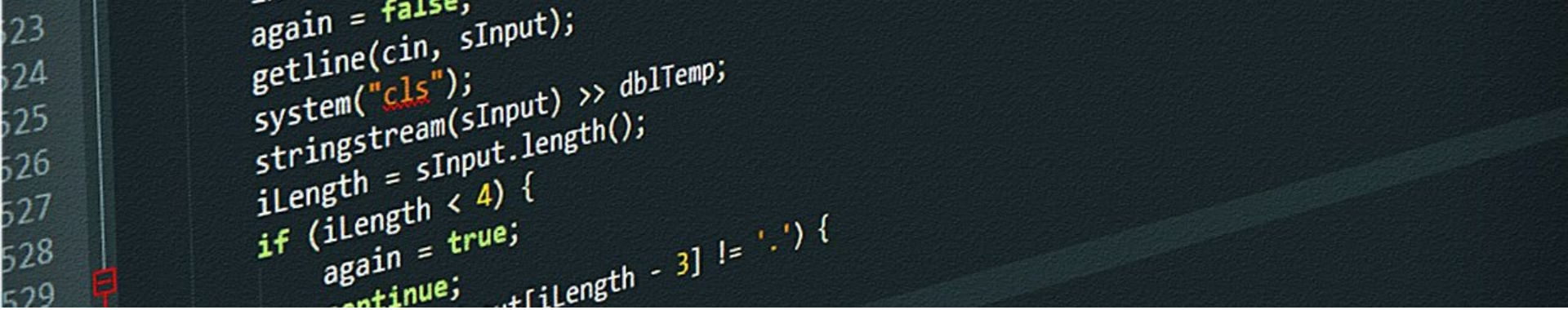


Thomas

C23-07 Operators, conversions

Advanced algorithms and programming

v5

Hochschule für Technik und Wirtschaft Berlin
University of Applied Sciences

Slides created using material from R. Peters and P. Puschmann / HTW Berlin

Operator overloading

- In C++, operators can be overloaded in the same way as functions
- This allows classes to behave like built-in types

- Overloading rules:
  - No new operators can be defined
  - The relative order of operators is kept as well as the number of operands
  - The operators should keep their meaning

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

Operator overloading

Example – class `Vector`

- Useful operations
  - Add (Vector + Vector)
  - Subtract (Vector - Vector)
  - Multiply (Vector * Vector) → scalar product
  - Multiply (Vector * number) → multiply length
  - Divide (Vector / umber) → divide length
  - Input and output operations
  - ...

Operator overloading

Operators as member functions

- The left operand is always the class object (instance)
  - Not suitable for some operations (e.g. << operator for std::cout)
- Some operators can be overloaded only as member function:
  ->, [], (), conversion operators

Syntax

```
// first operand is the instance itself!
RETURN_TYPE operator OP (); // with one operand
RETURN_TYPE operator OP (TYPE second_operand); // with two operands
```

Operator overloading

Operators as member functions – example

Vector2.h

```
#pragma once
#include <string>

class Vector2
{
public:
    Vector2() : m_x{ 0 }, m_y{ 0 } {}
    Vector2(double x, double y) : m_x{ x }, m_y{ y } { }

    Vector2 operator+(const Vector2& rVec) const; // Add vectors
    Vector2 operator-(const Vector2& rVec) const; // Subtract vectors
    Vector2 operator-() const; // Minus prefix (one operand)

    int operator*(const Vector2& rVec) const; // Scalar product
    Vector2 operator*(double num) const; // Multiply length
    Vector2 operator/(double num) const; // Divide length

    double& operator [](unsigned int index); // Access to x or y;
    Vector2& operator=(const Vector2& rVec);
    Vector2& operator+=(const Vector2& rVec);

    Vector2& operator++(); // Pre-increment (vector length += 1)
    Vector2 operator++(int); // Post-increment (vector length += 1)

    double getLength() const
    { return sqrt(m_x * m_x + m_y * m_y); };

    Vector2 getNormalized() const;
    double getX() const { return m_x; }
    double getY() const { return m_y; }
    std::string toString() const;

private:
    double m_x, m_y;
};
```

Operator overloading

Operators as member functions – example

Vector2.cpp (partial)

```
#include <stdexcept>
#include <sstream>
#include "Vector2.h"

Vector2& Vector2::operator=(const Vector2& rVec) {
    m_x = rVec.m_x;  m_y = rVec.m_y;
    return *this;
}

Vector2 Vector2::operator+(const Vector2& rVec) const {
    return Vector2(m_x + rVec.m_x, m_y + rVec.m_y);
}

Vector2& Vector2::operator+=(const Vector2& rVec) {
    m_x += rVec.m_x;
    m_y += rVec.m_y;
    return *this;
}

double& Vector2::operator[](unsigned int index) {
    if (index > 1)
        throw std::out_of_range("Index of Vector2 may be only 0
or 1!");
    return index == 0 ? m_x : m_y;
}
```

```
Vector2& Vector2::operator++() {
    *this += getNormalized();
    return *this;
}

Vector2 Vector2::operator++(int) {
    Vector2 tmp = *this;
    *this += getNormalized();
    return tmp;
}

Vector2& Vector2::operator+=(const Vector2& rVec) {
    m_x += rVec.m_x;
    m_y += rVec.m_y;
    return *this;
}

Vector2 Vector2::getNormalized() const {
    double length = getLength();
    if (length > 0) {
        return *this / length;
    }
    return Vector2(0, 0);
}
```

Operator overloading

Operators as member functions – example

Main.cpp

```cpp
#include <iostream>
#include "Vector2.h"

int main()
{
    Vector2 vec1{4, 3}, vec2{5, 10}, vec3, vec4, vec5, vec6, vec7;

    vec3 = vec1 + vec2;
    vec4 = vec1 * 2;
    // vec4 = 2 * vec1; // Error. Object must be on left side!
    vec4[1] = 20;
    vec5 = vec1;
    vec6 = vec5++;
    vec7 = ++vec5;

    std::cout << "Vector1: " << vec1.toString() << std::endl;
    std::cout << "Vector2: " << vec2.toString() << std::endl;
    std::cout << "Vector3: " << vec3.toString() << std::endl;
    std::cout << "Vector4: " << vec4.toString() << std::endl;
    std::cout << "Vector5: " << vec5.toString() << std::endl;
    std::cout << "Vector6: " << vec6.toString() << std::endl;
    std::cout << "Vector7: " << vec7.toString() << std::endl;
}
```

```
Vector1: [4,3] : 5
Vector2: [5,10] : 11.1803
Vector3: [9,13] : 15.8114
Vector4: [8,20] : 21.5407
Vector5: [5.6,4.2] : 7
Vector6: [4,3] : 5
Vector7: [5.6,4.2] : 7
```

Operator overloading

Operators outside a class

- Motivation:
  During implementation, free choice of placement of class object (left or right operand)
- Implementation as friend function outside of class

Syntax

1. in the class as `friend`, if access to `private area` is required:

```
friend RETURN_TYPE operator OP (TYPE operand); // 1 operand
friend RETURN_TYPE operator OP (TYPE_1 operand1, TYPE_2 operand2); // 2 operands
```

2. Implement somewhere outside the class

```
RETURN_TYPE operator OP (TYPE operand) { ... } // 1 operand
RETURN_TYPE operator OP (TYPE_1 operand1, TYPE_2 operand2) { ... } // 2 operands
```

Operator overloading

Operators outside a class – example

Example

Vector2.h

```
#pragma once
#include <iostream>

class Vector2
{
public:
    Vector2() : m_x{ 0 }, m_y{ 0 } {}
    Vector2(double x, double y) : m_x{ x }, m_y{ y } { }

    double getLength() const { return sqrt(m_x * m_x + m_y * m_y); };

    // ... to be continued as in previous example

private:
    double m_x, m_y;

    friend Vector2 operator*(double num, const Vector2& rVec); // Multiply length
    friend Vector2 operator*(const Vector2& rVec, double num);
    // Input and output operators cannot be declared as member functions
    // because object must be on the right
    friend std::ostream& operator<<(std::ostream& os, const Vector2& rVec);
    friend std::istream& operator>>(std::istream& is, Vector2& rVec);
};
```

Operator overloading

Operators outside a class – example

Vector2.cpp

```
#include "Vector2.h"

Vector2 operator*(const Vector2& rVec, double num) {
    return Vector2(rVec.m_x * num, rVec.m_y * num);
}

Vector2 operator*(double num, const Vector2& rVec) { return rVec * num; }

std::ostream& operator<<(std::ostream& os, const Vector2& rVec) {
    os << "[" << rVec.m_x << "," << rVec.m_y << "] : " << rVec.getLength();
    return os;
}

std::istream& operator>>(std::istream& is, Vector2& rVec) {
    double x, y;
    if ((is >> x) && (is >> y)) {
        // Reading was successful ...
        rVec.m_x = x;
        rVec.m_y = y;
    }
    return is;
}
```

Operator overloading

Operators outside a class – example

main.cpp (continued)

```
#include "Vector2.h"

int main()
{
    Vector2 vec1, vec2, vec3;

    std::cout << "Please enter two values for a vector [X Y]:" << std::endl;
    // direct input operation via stream
    std::cin >> vec1;

    // Now the object can also be placed left or right of operator
    vec2 = 2 * vec1;
    vec3 = vec1 * 2;

    // direct output operation via stream
    std::cout << "Vector1: " << vec1 << std::endl;
    std::cout << "Vector2: " << vec2 << std::endl;
    std::cout << "Vector3: " << vec3 << std::endl;
}
```

```
Please enter two values for a vector [X Y]:
1 -3
Vector1: [1,-3] : 3.16228
Vector2: [2,-6] : 6.32456
Vector3: [2,-6] : 6.32456
```

Operator overloading

Return value and parameters

- Always pay attention to parameter passing method and return value passing method
  - const reference, reference, or value? → influences result of operation!

Return value as reference

```
// Vector2::operator=()
Vector2& operator=(const Vector2& rVec)
{
   m_x = rVec. m_x;
   m_y = rVec. m_y;
   return *this;
}

// main:
Vector2 vec2(4,8), vec3, vec4;
vec4 = ++(vec3 = vec2);
```

```
Vector2: [4,8] : 8.94427
Vector3: [4.44721,8.89443] : 9.94427
Vector4: [4.44721,8.89443] : 9.94427
```

Return value as value

```
// Vector2::operator=()
Vector2 operator=(const Vector2& rVec)
{
   m_x = rVec. m_x;
   m_y = rVec. m_y;
   return *this;
}

// main:
Vector2 vec2(4,8), vec3, vec4;
vec4 = ++(vec3 = vec2);
```

```
Vector2: [4,8] : 8.94427
Vector3: [4,8] : 8.94427
Vector4: [4.44721,8.89443] : 9.94427
```

Operator overloading

Return value and parameters

Return value as reference

```
// Vector2::operator=()
Vector2& operator=(const Vector2& rVec)
{
    m_x = rVec. m_x;
    m_y = rVec. m_y;
    return *this;
}

// main:
Vector2 vec2(4,8), vec3, vec4;
vec4 = ++(vec3 = vec2);
```

```
Vector2: [4,8] : 8.94427
Vector3: [4.44721,8.89443] : 9.94427
Vector4: [4.44721,8.89443] : 9.94427
```

1. &vec3 <= (vec3 = vec2)
2. ++(&vec3)
3. vec4 = vec3

Return value as value

```
// Vector2::operator=()
Vector2 operator=(const Vector2& rVec)
{
    m_x = rVec. m_x;
    m_y = rVec. m_y;
    return *this;
}

// main:
Vector2 vec2(4,8), vec3, vec4;
vec4 = ++(vec3 = vec2);
```

```
Vector2: [4,8] : 8.94427
Vector3: [4,8] : 8.94427
Vector4: [4.44721,8.89443] : 9.94427
```

1. tmp <= (vec3 = vec2)
2. ++tmp
3. vec4 = tmp

Conversion of classes

Implicit conversion

- With implicit conversion the compiler automatically converts a data type into another data type.
- Internally the compiler has a conversion table, e.g:

Data type	Possible conversion to
`int`	`float, double, char ...`
`double`	`int, float, char ...`
`char[]`	`std::string`

Conversion of classes

Implicit conversion example

```cpp
#include <string>

int add(int a, int b) { return a + b; }

int main()
{
    int a = 65;
    float b = a; // int -> float
    char c = a; // int -> char (65 corresponds to ASCII 'A')

    char buf[] = "Hello World!";

    std::string s = buf; // char[] -> std::string

    int d = add(b /* float -> int */, c /* char -> int */);
}
```

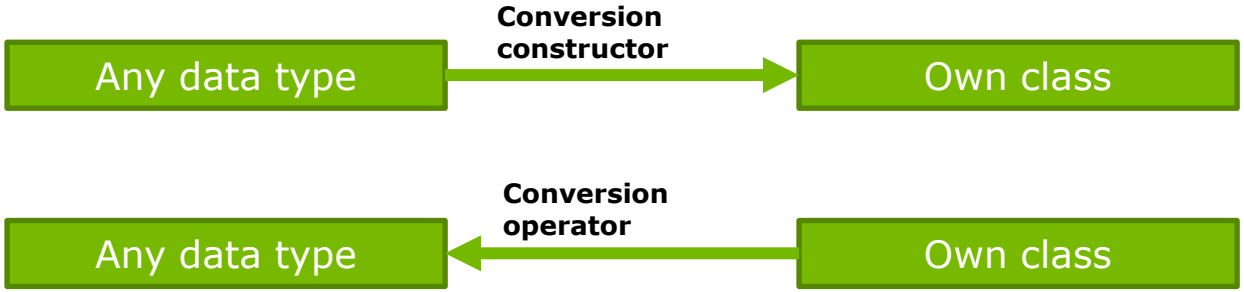
Conversion of classes

Conversion of own classes

Motivation

- Create compatibility between own classes and built-in types or to types from libraries
- Reuse functions that were implemented for other (compatible) data types

Approach



- A conversion constructor converts any other data type to a class
- A conversion operator converts a class into any other data type

- The possible conversions are entered into the conversion table of the compiler

Conversion of classes

Conversion of own classes

Conversion constructor

- Constructor with a single parameter of the type to be converted (either *Call By Reference* or *Call by Value*)

```
// TYPE -> Class
Class_name(TYPE parameter); // for basic data types
Class_name(const TYPE& parameter); // for classes, structs, etc...
```

Conversion Operator

- Operator without return value and parameter, with `const`
- TYPE is the datatype to which the class should be converted

```
// Class -> TYPE
operator TYPE() const;
```

Conversion of classes

Conversion of own classes – example

Fraction.h

```
#pragma once
#include <string>

class Fraction // math. fraction
{
public:

    Fraction(int numerator, int denominator)
        : m_numerator(numerator), m_denominator(denominator) {}

    Fraction(double d); // double -> Fraction
    Fraction(int i) : m_numerator(i), m_denominator(1) {} // int -> Fraction

    Fraction operator+(const Fraction& rOther) const;
    Fraction operator+(int num) const;
    Fraction operator*(const Fraction& rOther) const;
    Fraction operator/(const Fraction& rOther) const;
    std::string toString() const;

    // Fraction -> double
    operator double() const { return (double)m_numerator / (double)m_denominator; }
    operator std::string() const; // Fraction -> string

private:
    void cancel();
    int m_numerator, m_denominator;
};
```

Conversion of classes

Conversion of own classes – example

Fraction.cpp (partial)

```
#include <sstream>
#include <iostream>
#include "Fraction.h“

// conversions

Fraction::Fraction(double d)
{
    m_denominator = 1;
    const double tolerance = 0.0000000001;
    while ((double)d - (int)d > tolerance) {
        d *= 10;
        m_denominator *= 10;
    }
    m_numerator = d;
    cancel();
}

Fraction::operator std::string() const
{
    std::stringstream s;
    s << "( " << m_numerator << " / " << m_denominator << " )";
    return s.str();
}
```

Conversion of classes

Conversion of own classes – example

Main.cpp

```
#include <iostream>
#include <string>
#include "Fraction.h"

double add(double v1, double v2) { return v1 + v2; }

// does not have to be a 'friend' of Fraction! (here only public access):
std::ostream& operator<<(std::ostream& os, const Fraction& rVec) {
    os << (std::string)rVec;
    return os;
}

int main() {
    Fraction frac1(5, 4);
    Fraction frac2 = 0.75;
    //frac1 = frac1 * 2; Error, why?
    Fraction frac3 = frac1 * frac2;
    Fraction frac4 = frac1 / frac2;
    Fraction frac5 = add(frac1, frac2); // Fraction => double => Fraction
    std::cout << "Fraction 1 = " << frac1 << "\nFraction 2 = " << frac2
        << "\nFraction 1*2 = " << frac3 << "\nFraction 1/2 = " << frac4
        << "\nFraction 1+2 = " << frac5 << std::endl;
}
```

```
Fraction 1 = ( 5 / 4 )
Fraction 2 = ( 3 / 4 )
Fraction 1*2 = ( 15 / 16 )
Fraction 1/2 = ( 5 / 3 )
Fraction 1+2 = ( 2 / 1 )
```

Conversion of classes

Multiple conversion options

Error with multiple conversion options

- The compiler has two options for the expression:

```
frac1 = frac1 * 2;
```

1. convert `frac1` to `double`, then use the built-in operator:

```
double operator*(double, double);
```

2. Convert literal `2` to `Fraction`, then use the operator of the `Fraction class`

```
Fraction operator*(const Fraction&, const Fraction&);
```

- The compiler cannot decide which option is the correct one

Error: Conversion must always be unique!

Conversion of classes

Multiple conversion options

Error with multiple conversion options

- Such errors occur if a class is convertible in both directions
- Use keyword `explicit` to prevent that the compiler includes the conversion constructor in the conversion table

```
explicit Fraction(double d);
explicit Fraction(int i) : m_numerator(i), m_denominator(1) {}
```

- Now `double` or `int` can no longer be converted implicitly to `fraction`
  - As consequence, there is only one unambiguous possibility left:

```
double operator*(double, double);
```

Conversion of classes

Conversion of own classes - best practice

- Conversion is only useful, if the other datatype can truly represent the class
  Avoid data loss during conversions!

- Make the own class implicitly convertible only in one direction
  Use the keyword `explicitly` for this purpose

Hochschule für Technik
und Wirtschaft Berlin

University of Applied Sciences

www.htw-berlin.de