Lecture 8: Operator overloading

Jonas Kusch and Martina Prugger

University of Innsbruck

April 18, 2023

Last goals: You are able to

- ☑ use classes (constructor, destructor, copy constructor)
- ☑ understand the programming project

Last goals: You are able to

- ☑ use classes (constructor, destructor, copy constructor)
- ☑ understand the programming project

Today's learning goals: Today we will learn about

- ☐ operator overloading
- ☐ work with multi arrays
- ☐ inheritance

Last goals: You are able to

- ☑ use classes (constructor, destructor, copy constructor)
- ☑ understand the programming project

Today's learning goals: Today we will learn about

- ☐ operator overloading
- ☐ work with multi arrays
- ☐ inheritance

Ask questions any time!

Classes - recap

```
class Data{
    double* _d;
public:
    Data(double d): _d(new double) {*_d = d;}
    Data(const Data& a){
        _d = new double;
        *_d = *a._d;
    }
    ~Data() {delete _d;}
};
void foo(Data a){} \\ calls copy constructor

int main(){
    Data a(1.234);
    if(true) Data b(a); \\ calls copy constructor
    foo(a);
    std::cout<< *a._d <<std::endl;
}
```

Classes - operators

- You can also define operations for classes!

```
Data a(1.234);
Data b(2.345);
a = a + b;
```

- We need to tell the compiler what it is supposed to do when adding ($+$) or assigning ($=$) objects of type data!
- The operator `Data& operator=(const Data&)` is defined per default, but might not do what we want.
  - $\rightarrow$ default behaviour copies all class data (dynamic memory not reallocated)
  - $\rightarrow$ `Data&` returns a reference.
  - $\rightarrow$ `Data c = a + b` uses copy constructor!

Classes - operators (any issues?)

```
class Data{
public:
    double* _d;
    Data(double d): _d(new double) {*_d = d;}
    ~Data(){delete _d;}
};

Data a(0.1); Data b(0.2);
a = b;
std::cout<<a._d<<" "<<b._d<<std::endl;
```

- The standard $=$ operator is

```
Data& operator=(const Data& a){
    _d = a._d;
    return *this;
}
```

Classes - operators (any issues?)

```
class Data{
public:
    double* _d;
    Data(double d): _d(new double) {*_d = d;}
    ~Data(){delete _d;}
    Data& operator=(const Data& a){
        delete _d; _d = new double; *_d = *a._d;
        return *this;
    }
};

int main(){
    Data a(0.1); Data b(0.2);
    b = a;
    return 0;
}
```

Classes - operators (any issues?)

```
class Data{
public:
    double* _d;
    Data(double d): _d(new double) {*_d = d;}
    ~Data(){delete _d;}
    Data& operator=(const Data& a){
        delete _d; _d = new double; *_d = *a._d;
        return *this;
    }
};

int main(){
    Data a(0.1); Data b(0.2);
    a = a;
    return 0;
}
```

Classes - operators (any issues?)

```
class Data{
public:
    double* _d;
    Data(double d): _d(new double) {*_d = d;}
    ~Data(){delete _d;}
    Data& operator=(const Data& a){
        if( this != &a ){
            delete _d; _d = new double; *_d = *a._d;
        }
        return *this;
    }
};

int main(){
    Data a(0.1); Data b(0.2);
    a = a;
    return 0;
}
```

Classes - operators

- You can also define operations for classes!

```
Data a(1.234);
Data b(2.345);
a = a + b;
```

- We need to tell the compiler what it is supposed to do when adding ($+$) or assigning ($=$) objects of type data!
- The operator `Data& operator=(const Data&)` is defined per default, but might not do what we want.
- Operators `Data& operator+(const Data&)` not defined per default.

Addition operator (example)

```
class Data{
    double* _d;
public:
    Data(double d): _d(new double) {*_d = d;}
    ~Data(){delete _d;}
    Data& operator=(const Data& a){...}
    Data& operator+(const Data& a){
        *_d = *_d + *(a._d);
        return *this;
    }
    void Print()const{std::cout<<*_d<<std::endl;}
};

Data a(0.1); Data b(0.2);
a = a + b;
a.Print();
```

Addition operator - What do you expect?

```
class Data{
    double* _d;
public:
    Data(double d): _d(new double) {*_d = d;}
    ~Data(){delete _d;}
    Data& operator=(const Data& a){...}
    Data& operator+(const Data& a){
        *_d = *_d + *(a._d);
        return *this;
    }
    void Print()const{std::cout<<*_d<<std::endl;}
};

Data a(0.1); Data b(0.2);
Data c = a + b;
a.Print(); b.Print(); c.Print();
```

Addition operator - What do you expect?

```
class Data{
    double* _d;
public:
    Data(double d): _d(new double) {*_d = d;}
    ~Data(){delete _d;}
    Data& operator=(const Data& a){...}
    Data operator+(const Data& a){
        Data c(*this);
        *_d += *(a._d);
        return c;
    }
    void Print()const{std::cout<<*_d<<std::endl;}
};

Data a(0.1); Data b(0.2);
Data c = a + b;
a.Print(); b.Print(); c.Print();
```

Addition operator - non-member function

```
class Data{
    double* _d;
public:
    Data(double d): _d(new double) {*_d = d;}
    ~Data(){delete _d;}
    Data& operator=(const Data& a){...}
    friend Data& operator+(const Data& a, const Data& b);
    void Print()const{std::cout<<*_d<<std::endl;}
};

Data& operator+(const Data& a, const Data& b){
    Data c(a);
    *(c._d) = *(a._d) + *(b._d);
    return c;
}

Data a(0.1); Data b(0.2);
a = a + b;
```

Your turn!

```
class Data{
    double* _d;
public:
    Data(double d): _d(new double) {*_d = d;}
    ~Data(){delete _d;}
};
```

Task

Implement the operator ++ such that `Data b = a++` fulfills `b._d == a._d + 1`. What additional operator(s) need to be implemented for this to work?

Solution

```
class Data{
    double* _d;
public:
    Data(double d): _d(new double) {*_d = d;}
    ~Data(){delete _d;}
    Data(const Data& a){
        _d = new double;
        *_d = *(a._d);
    }
    Data operator++();
};

Data Data::operator++(){
    Data b(*this);
    *(b._d) += 1;
    return b;
}
```

Bracket operator

```
class Vector{
    unsigned _dim;
    double* _v;
public:
    Vector(unsigned dim): _v(new double [dim]), _dim(dim) {}
    ~Vector(){...}
    Vector& operator=(const Vector& a){...}
    double& operator[](const unsigned i){
        return _v[i];
    }
};

Vector a(10); a[0] = 1;
```

- You can overload the bracket operator to directly access elements.
- What does the `double& ...` do? → code example `vector.cpp`

Your turn!

What does the `Print` function do? Does this work?

```
class Vector{
    unsigned _dim;
    double* _v;
public:
    Vector(unsigned dim): _v(new double [dim]), _dim(dim) {}
    double& operator[](const unsigned i){
        return _v[i];
    }
    void Print(unsigned int i)const{
        std::cout<<(*this)[i]<<std::endl;
    }
};

Vector a(10); a[0] = 1;
Print(0);
```

Compiles with...

```
class Vector{
    unsigned _dim;
    double* _v;
public:
    Vector(unsigned dim): _v(new double [dim]), _dim(dim) {}
    double& operator[](const unsigned i){
        return _v[i];
    }
    double operator[](const unsigned i)const{
        return _v[i];
    }
    void Print(unsigned int i)const{
        std::cout<<(*this)[i]<<std::endl;
    }
};
```

Boost multiarrays

```
#include <iostream>
#include <boost/multi_array.hpp>

int main(){
    // Create a 3D array of dimensions 3x4x2
    boost::multi_array<int, 3> myArray(boost::extents[3][4][2]);

    for (int i = 0; i < 3; i++) // Fill with some values
        for (int j = 0; j < 4; j++)
            for (int k = 0; k < 2; k++)
                myArray[i][j][k] = i + j + k;

    std::cout<<myArray.shape()[0]<<std::endl;
    return 0;
}
```

Your turn

Exercise (Homework)

Write your own vector class which provides addition, substraction of two vectors and multiplication of a vector and a scalar. Use the boost library to store the vector entries. Provide all needed operators such as copy constructor, destructor,...

Exercise

Write outputs into constructors, destructors etc. Check when and where these are called.

Inheritance

- Assume you have defined a new class and realize that it is very similar to a previously defined class.

```
class Triangle{
    std::vector<int> _points;
private:
    double Area()const;
};

class RightTriangle{
    std::vector<int> _points;
private:
    double Area()const;
    int Hypotenuse()const;
};
```

Inheritance

- Assume you have defined a new class and realize that it is very similar to a previously defined class. Let's say you have cells of type

```
class Triangle{
    std::vector<int> _points;
public:
    double Area()const;
};

class RightTriangle : public Triangle{
public:
    int Hypotenuse()const;
};
```

- `Triangle` is parent, `RightTriangle` is child class
- `RightTriangle` inherits all functions and variables from `RightTriangle`
- means that a `RightTriangle` is a `Triangle`, but not every `Triangle` is a `RightTriangle`

Inheritance - Example

```
class Triangle{
    std::vector<int> _points;
public:
    Triangle(std::vector<int> points) : _points(points) {}
    double Area()const{ return 1.0;}
};

class RightTriangle : public Triangle{
public:
    RightTriangle(std::vector<int> points) : Triangle(points) {}
    int Hypotenuse()const{ return 1;}
};

std::vector<int> points;
points.push_back(1); points.push_back(2); points.push_back(3);
RightTriangle A(points);
std::cout<<A.Area()<<std::endl;
```

Inheritance - Example

```
class Triangle{
    std::vector<int> _points;
public:
    Triangle(std::vector<int> points) : _points(points) {}
    double Area()const{ return 1.0;}
};

class RightTriangle : public Triangle{
public:
    RightTriangle(std::vector<int> points) : Triangle(points) {}
    double Area()const{ return 2.0;}
};

std::vector<int> points;
points.push_back(1); points.push_back(2); points.push_back(3);
RightTriangle A(points);
std::cout<<A.Area()<<std::endl;
```

Inheritance - Example

```
class Triangle{
    std::vector<int> _points;
public:
    Triangle(std::vector<int> points) : _points(points) {}
    double Area()const{ return 1.0;}
};

class RightTriangle : public Triangle{
public:
    RightTriangle(std::vector<int> points) : Triangle(points) {}
    double Area()const{ return 2.0;}
};

std::vector<int> points;
points.push_back(1); points.push_back(2); points.push_back(3);
RightTriangle A(points);
std::cout<<A.Triangle::Area()<<std::endl;
```

Dominance of functions/data

- The functions/data in the child class dominates the parent class.
- Functions/data in parent class are however still available via `object.Parent::Data`.
- Constructor of child first calls the parent constructor
- destructor vice-versa

Constructors

```
class Cell{
public:
    Cell(){std::cout<<"create Cell"<<std::endl;}
};

class Triangle : public Cell{
public:
    Triangle(){std::cout<<"create Triangle"<<std::endl;}
};

class RightTriangle : public Triangle{
public:
    RightTriangle(){std::cout<<"create RightTriangle"<<std::endl;}
};
```

- create Cell, then Triangle, then RightTriangle

Destructors

```
class Cell{
public:
    ~Cell(){std::cout<<"free Cell"<<std::endl;}
};

class Triangle : public Cell{
public:
    ~Triangle(){std::cout<<"free Triangle"<<std::endl;}
};

class RightTriangle : public Triangle{
public:
    ~RightTriangle(){std::cout<<"free RightTriangle"<<std::endl;}
};
```

- free `RightTriangle`, then `Triangle`, then `Cell`

Access rights

```
class Cell{
    double _area;
protected:
    std::vector<int> _points;
public:
    double Area()const;
};

class Triangle : public Cell{};
\\ all protected variables/fcts will be potected, all public public

class RightTriangle : protected Triangle{};
\\ all protected/public variables/fcts will be potected

class Quadrangle : private Cell{};
\\ all protected/public variables/fcts will be private
```

- private is not inherited, protected not accessible in main

Your turn - Does this compile?

```
class Triangle{
protected:
    std::vector<int> _points;
public:
    Triangle(std::vector<int> points) : _points(points) {}
    double Area()const{ return 1.0;}
};

class RightTriangle : private Triangle{
public:
    RightTriangle(std::vector<int> points) : Triangle(points) {}
    double Area()const{ return 1.2 * _points[0];}
    std::cout<<A.Area()<<std::endl;
};

RightTriangle A(points);
std::cout<<A.Area()<<std::endl;
```

Your turn - Does this compile?

```
class Triangle{
protected:
    std::vector<int> _points;
public:
    Triangle(std::vector<int> points) : _points(points) {}
    double Area()const{ return 1.0;}
};

class RightTriangle : private Triangle{
public:
    RightTriangle(std::vector<int> points) : Triangle(points) {}
    double Area()const{ return 1.2 * _points[0];}
    std::cout<<A.Area()<<std::endl;
};

RightTriangle A(points);
std::cout<<A.Triangle::Area()<<std::endl;
```

Your turn - Does this compile?

```
class Triangle{
protected:
    std::vector<int> _points;
public:
    Triangle(std::vector<int> points) : _points(points) {}
    double Area()const{ return 1.0;}
};

class RightTriangle : protected Triangle{
public:
    RightTriangle(std::vector<int> points) : Triangle(points) {}
    double Area()const{ return 1.2 * _points[0];}
    std::cout<<A.Area()<<std::endl;
};

RightTriangle A(points);
std::cout<<A.Triangle::Area()<<std::endl;
```

Your turn - Does this compile?

```
class Cell{
public:
    double Area()const;
};

class Triangle : public Cell{};

class RightTriangle : public Triangle{};

RightTriangle A;
std::cout<<A.Area()<<std::endl;
```

Your turn - Does this compile?

```
class Cell{
public:
    double Area()const;
};

class Triangle : protected Cell{};

class RightTriangle : public Triangle{};

RightTriangle A;
std::cout<<A.Area()<<std::endl;
```

Your turn - Does this compile?

```
class Cell{
public:
    double Area()const;
};

class Triangle : protected Cell{};

class RightTriangle : public Triangle{
    void PrintArea()const {std::cout<<Area()<<std::endl;}
};
```

Polymorphism

- Often you define if you want to use a Triangle, RightTriangle, Quadrangle mesh during runtime when reading the mesh.

```
if(meshType == "Triangle"){
    Triangle* c = new Triangle;
    //... do all the computations using triangles
}else if(meshType == "RightTriangle"){
    RightTriangle* c = new RightTriangle;
    //... do all the computations using right triangle
}...
```

- Way out: A pointer of the parent class can point on the child class

```
Cell *c;
if(meshType == "Triangle"){
    c = new Triangle;
}else if(meshType == "RightTriangle"){
    c = new RightTriangle;
}
//... do all the computations using cell
```

Polymorphism

- Often you define if you want to use a Triangle, RightTriangle, Quadrangle mesh during runtime when reading the mesh.

```
if(meshType == "Triangle"){
    Triangle* c = new Triangle;
    //... do all the computations using triangles
}else if(meshType == "RightTriangle"){
    RightTriangle* c = new RightTriangle;
    //... do all the computations using right triangle
}...
```

- Way out: A pointer of the parent class can point on the child class

```
Cell *c;
if(meshType == "Triangle"){
    c = new Triangle;
}else if(meshType == "RightTriangle"){
    c = new RightTriangle;
}
//... do all the computations using cell
```

Polymorphism

```
class Cell{
    double _area;
public:
    void Area()const { std::cout<<"Cell"<<std::endl; }
};

class Triangle : public Cell{
    void Area()const { std::cout<<"Triangle"<<std::endl; }
};

class RightTriangle : public Triangle{
    void Area()const { std::cout<<"RightTriangle"<<std::endl; }
};

Cell* c = new RightTriangle;
c->Area();
```

Polymorphism

```
class Cell{
    double _area;
public:
    virtual void Area()const { std::cout<<"Cell"<<std::endl; }
};

class Triangle : public Cell{
    virtual void Area()const { std::cout<<"Triangle"<<std::endl; }
};

class RightTriangle : public Triangle{
    virtual void Area()const { std::cout<<"RightTriangle"<<std::endl; }
};

Cell* c = new RightTriangle;
c->Area();
```

What happens?

```
class Cell{
public:
    virtual void AllocateMem() { std::cout<<"Cell"<<std::endl; }
};

class Triangle : public Cell{
    double* _data;
    virtual void AllocateMem() { _data = new double; }
    ~Triangle(){
        std::cout<<"delete"<<std::endl;
        delete _data;
    }
};

Cell* c = new Triangle;
c->AllocateMem();
delete c;
```

Virtual destructors

```
class Cell{
public:
    virtual void AllocateMem() { std::cout<<"Cell"<<std::endl; }
    virtual ~Cell(){
};

class Triangle : public Cell{
    double* _data;
    virtual void AllocateMem() { _data = new double; }
    virtual ~Triangle(){delete _data;}
};

Cell* c = new Triangle;
c->AllocateMem();
delete c;
```

- Always use virtual destructors to ensure correct deallocation.

Abstract Classes

```
class Cell{
public:
    virtual double Area() = 0;
};

class Triangle : public Cell{
    virtual double Area() { return 1.0; }
};
```

- It does not make sense to create an object `Cell c`, since a cell must always be a triangle, quadrangle, ...
- Every child of cell needs to have an area.
- You can impose this with a pure virtual function `virtual double Area() = 0`.

Your turn

Exercise

Write an abstract class Vector, a class IntVector and a class DoubleVector. Connect these classes with inheritance and provide all needed constructors, destructors etc. Create a vector of type DoubleVector with polymorphism.