# Lecture 6: Classes and Templates

Johannes Gerstmayr and Markus Walzthöni

Material: Jonas Kusch and Martina Prugger
University of Innsbruck

November 24, 2023

# Classes: contents of lecture 6

**Today's plan:**

- More constructors
- Typical Operators
- Default constructor, copy operator, move mechanism, etc.
- Overloading, Inheritance
- Templates

# Move constructor

```
class A {
public:
    std::string s;
    int k;
    A() : s("test"), k(-1) {}
    A(const A& o) : s(o.s), k(o.k) { std::cout << "move failed!\n"; }
    A(A&& o) noexcept :               // important for optimizer / std lib
        s(std::move(o.s)),            // explicit move of a member of class type
        k(std::exchange(o.k, 0)) {} // explicit move of a member of non-class type
};
A f(A a) { return a; } //function potentially copies

int main() {
    std::cout << "Trying to move A\n";
    A a1 = f(A()); // return by value move-constructs the target from the function
    A a2 = std::move(a1); // move-constructs from xvalue
    std::cout << "now:   a1.k = " << a1.k << '\n';
}
```

# More constructors

```cpp
#include <initializer_list>
#include <vector>

class D {
public:
    std::vector<int> values;
    //old school:
    //D(x): values({x}) {}
    //D(x, y): values({x,y}) {}
    D(std::initializer_list<int> list) : values(list) {}
};

int main() {
    D d = {1, 2, 3, 4, 5}; // Initializer list constructor
    for (int v : d.values) { std::cout << v << " "; }
}
```

## Operator overloading (1)

For classes A and B, we can tell the compiler how to handle, e.g., `auto C = A+B;`

```
class SimpleString {
public:
    std::string data;
    SimpleString(const std::string& str = "") : data(str) {}
    // Overload the assignment operator
    SimpleString& operator=(const SimpleString& other) {
        if (this != &other) { data = other.data; }
        return *this; }
    // Overload the + operator for string concatenation
    SimpleString operator+(const SimpleString& other) const
        { return SimpleString(data + other.data); }
};
int main() {
    SimpleString s1("Hello "), s2("World!!!");
    SimpleString s3 = s1 + s2;
    std::cout << "s3: " << s3.data << std::endl;
}
```

## Operator overloading (2)

For simple printing, add `ostream operator` !

```
class SimpleString {
    ...
    // Overload the ostream << operator
    friend std::ostream& operator<<(std::ostream& os, const SimpleString& ss) {
        os << ss.data;
        return os;
    }
};
int main() {
    SimpleString s1("Hello "), s2("World!!!");
    SimpleString s3 = s1 + s2;
    std::cout << s1 << "+" << s2 << " = " << s3 << std::endl;
}
```

# Operator overloading (3)

```
class SimpleString {
    ...
    // Overload the equality (==) operator
    bool operator==(const SimpleString& other) const
        { return data == other.data; }
    // Overload the * operator for string length
    int operator*(const SimpleString& other) const
        { return Size() + other.Size(); }
    // Overload the bracket [] operator (read)
    char operator[](int i) const { return data[i]; }
};
int main() {
    ...
    std::cout << "s1 and s2 are equal" << (s1==s2) ? "true" : "false" << std::endl;
    std::cout << "length of s1 + s2: " << s1*s2 << std::endl;
    std::cout << "s1[1] = " << s1[1] << std::endl;
}
```

Note: there is also a move operator!

# Inheritance

# 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> pointsInit) : points(pointsInit) {}
    double Area()const{ return 1.0;}
};

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

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

# Inheritance - Example

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

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

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

## Inheritance - Example

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

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

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

# Dominance of functions/data

## Some rules ...

- 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

```cpp
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

```cpp
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: //not accessible from outside, but in derived class
    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> pointsInit) : points(pointsInit) {}
    double Area() const{ return 1.0;}
};
class RightTriangle : private Triangle{
public:
    RightTriangle(std::vector<int> pointsInit) : Triangle(pointsInit) {}
    double Area() const{ return 1.2 * points[0];}
};

std::vector<int> points({1,2});
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> pointsInit) : points(pointsInit) {}
    double Area() const{ return 1.0;}
};
class RightTriangle : private Triangle{
public:
    RightTriangle(std::vector<int> pointsInit) : Triangle(pointsInit) {}
    double Area() const{ return 1.2 * points[0];}
};

std::vector<int> points({1,2});
RightTriangle A(points);
std::cout << A.Triangle::Area() << std::endl; //only works if "public Triangle" is
```

## Your turn - Does this compile?

```
class Triangle{
protected:
    std::vector<int> points;
public:
    Triangle(std::vector<int> pointsInit) : points(pointsInit) {}
    double Area() const{ return 1.0;}
};
class RightTriangle : public Triangle{
public:
    RightTriangle(std::vector<int> pointsInit) : Triangle(pointsInit) {}
    double Area() const{ return 1.2 * points[0];}
};

std::vector<int> points({1,2});
RightTriangle A(points);
std::cout << A.Triangle::Area() << std::endl; //only works if "public Triangle" is
```

## Your turn - Does this compile?

```
class Cell{
public:
    double Area() const {return 3.14;}
};

class Triangle : protected Cell{};

class RightTriangle : public Triangle{};

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

## Your turn - Does this compile?

```cpp
class Cell{
public:
    double Area() const {return 3.14;}
};

class Triangle : protected Cell{};

class RightTriangle : public Triangle{
    void PrintArea() const {std::cout << Area() << std::endl;}
};
RightTriangle A;
std::cout << A.PrintArea() << 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(); //works for all 3 classes, but calls Cell::Area() !
```

# Polymorphism: virtual function specifier

```
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();
```

→ `virtual` functions are member functions **whose behavior can be overridden in derived classes**. Works for pointers and references (`Cell*` and `Cell&`). Note: there is also multiple inheritance (`class C: public A, public B`)

# 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 `Area()` implemented.
- You can impose this with a **pure virtual function** `virtual double Area() = 0`.
- Abstract classes have at least 1 pure virtual function.
- **Pure virtual functions** ensure that they are overridden in the derived class, the **override** specifier ensures that a function overrides a parent function, and the **final** specifier ensures that no derived class tries to override a function.

## 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 `Area()` implemented.
- You can impose this with a **pure virtual function** `virtual double Area() = 0`.
- Abstract classes have at least 1 pure virtual function.
- **Pure virtual functions** ensure that they are overridden in the derived class, the **override** specifier ensures that a function overrides a parent function, and the **final** specifier ensures that no derived class tries to override a function.

## 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 `Area()` implemented.
- You can impose this with a **pure virtual function** `virtual double Area() = 0`.
- Abstract classes have at least 1 pure virtual function.
- **Pure virtual functions** ensure that they are overridden in the derived class, the **override** specifier ensures that a function overrides a parent function, and the **final** specifier ensures that no derived class tries to override a function.

## 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 `Area()` implemented.
- You can impose this with a **pure virtual function** `virtual double Area() = 0`.
- Abstract classes have at least 1 pure virtual function.
- **Pure virtual functions** ensure that they are overridden in the derived class, the **override** specifier ensures that a function overrides a parent function, and the **final** specifier ensures that no derived class tries to override a function.

# Your turn

## Example

- check GitHub C++ codes for inheritance!
- see Exudyn `CNode.h`

## 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.

# Templates

## Templates

Consider you need a function which swaps the content of two variables:

```
void Swap(int& a, int& b) {int c=a; a=b; b=c;}
int a=3, b=5;
Swap(a,b);
```

Now we also would like to do this for other types:

```
void Swap(double& a, double& b) {double c=a; a=b; b=c;}
void Swap(bool& a, bool& b) {bool c=a; a=b; b=c;}
void Swap(long& a, long& b) {long c=a; a=b; b=c;}
void Swap(std::string& a, std::string& b) {std::string c=a; a=b; b=c;}
...
```

→ this is code duplication!!!

# Templates

Therefore, C++ has templates:

```
template <typename T>
T foo( T a, T b ){
    ...
}
double a, b;
double out = foo<double>(a, b)
```

Templates are very old C++98 features!

- construction plans for the compiler
- can be used to remove code redundancies, avoid repetition, performance

# Templates

Combine two typenames:

```
template <typename T, typename S>
S trafo(const T& a){
    S s(a);
    return s;
}

double a = 0.1;
long double c = trafo<double, long double>(a);
```

- typenames are seperated by a comma
- used when typenames are defined at once

## Templates

Use for classes

```
template <typename T>
class Container{
    T data;
    ...
}

Container<int> b;
typedef Container<int> ContainerInt; //just call it ContainerInt from now on
```

- Sometimes you will see the keyword `class` instead of `typename`.
- same meaning, `typename` can be used in all situations (with C++17 or newer)

**Exercise**

Rewrite your classes IntVector and DoubleVector into a single class using templates.

# Templates - Any issues?

```
template <typename T>
T max(const T& a, const T& b){
    if( a > b) { return a; }
    else { return b; }
}

double a = 0.1, b = 0.3;
double c = max<double>(a,b);
std::cout << c << std::endl;
```

# Templates - Performance

```
template<int n>
double* zero(int m){
    double* a = new double [n*m];
    for( int i = 0; i < m ; ++i )
        for( int j = 0; j < n ; ++j )
            {a[i*n+j] = 0.0;}
    return a;
}

double* a = zero<8>(100);
```

- → `https://godbolt.org/` (check with x86-64 gcc 13.2 and -O1, -O2, -O3)
- → Check some templated classes in Exudyn
- → Templates are extremely powerful, many extensions since C++98
- → Expression templates for creating special rules for operators (performance)

## Templates - Performance

```
template<int n>
double* zero(int m){
    double* a = new double [n*m];
    for( int i = 0; i < m ; ++i )
        for( int j = 0; j < n ; ++j )
            {a[i*n+j] = 0.0;}
    return a;
}

double* a = zero<8>(100);
```

- → `https://godbolt.org/` (check with x86-64 gcc 13.2 and -O1, -O2, -O3)
- → Check some templated classes in Exudyn
- → Templates are extremely powerful, many extensions since C++98
- → Expression templates for creating special rules for operators (performance)

## Templates - Performance

```
template<int n>
double* zero(int m){
    double* a = new double [n*m];
    for( int i = 0; i < m ; ++i )
        for( int j = 0; j < n ; ++j )
            {a[i*n+j] = 0.0;}
    return a;
}

double* a = zero<8>(100);
```

- → `https://godbolt.org/` (check with x86-64 gcc 13.2 and -O1, -O2, -O3)
- → Check some templated classes in Exudyn
- → Templates are extremely powerful, many extensions since C++98
- → Expression templates for creating special rules for operators (performance)

# Outlook

## This is it about classes, for now!

- Lecture 7 given by Markus

**Outlook lecture 8**:

- clean code, code duplication, refactoring
- coding rules / styles, Google standards (what is it?)
- exceptions, macros
- Numerical receipes in C++ (what is it?)
- Pybind11 (what is it)