Lecture 9: Templates and smart pointers

Jonas Kusch and Martina Prugger
University of Innsbruck

April 25, 2023

Last goals: You are able to

- ☑ define your own operators
- ☑ use inheritance
- ☑ use multi arrays

Last goals: You are able to

- ☑ define your own operators
- ☑ use inheritance
- ☑ use multi arrays

Today's learning goals: Today we will learn about

- ☐ templates
- ☐ smart pointers
- ☐ inheritance

Last goals: You are able to

- ☑ define your own operators
- ☑ use inheritance
- ☑ use multi arrays

Today's learning goals: Today we will learn about

- ☐ templates
- ☐ smart pointers
- ☐ inheritance

Ask questions any time!

Operators - recap

- 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!

Operators - recap

- 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.

Operators - recap

```
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;
}
```

Operators - recap

```
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();
```

Operators - recap

```
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 - recap

```cpp
#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;
}
```

Inheritance - recap

- 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`

Dominance of functions/data - recap

- 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 - recap

```
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 - recap

```
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 - recap

```
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

Polymorphism - recap

- 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 - recap

- 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
```

Virtual destructors - recap

```
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, operators etc. Create a vector of type DoubleVector with polymorphism.

Exercise (if you are done already)

Extend this to a class IntMatrix and IntVector. Provide an operator for matrix vector multiplication.

Templates

There is an easier solution to this task:

```
template <typename T>
T foo( T a, T b ){
    ...
}

double a, b;
T out = foo<double>(a, b)
```

Templates are

- 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;
```

- 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 - 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;
```

- No delete in main needed!
- Concept of smart pointers will be discussed later.

Templates - Any issues?

```
template<typename T>
class Pointer{
    T* _p;
public:
    Pointer(double* p) : _p(p) {}
    ~Pointer(){delete _p;}
    T& operator*() { return *_p; }
};

Pointer<double> ptr(new double);
*ptr = 0.1;
std::cout<<*ptr<<std::endl;
delete ptr;
```

Templates - Any issues?

```
template<typename T>
class Pointer{
    T* _p;
public:
    Pointer(double* p) : _p(p) {}
    ~Pointer(){delete _p;}
    T& operator*() { return *_p; }
};

Pointer<double> ptr(new double);
*ptr = 0.1;
std::cout<<*ptr<<std::endl;
delete ptr;
```

- No delete in main needed!
- Concept of smart pointers will be discussed later.

Templates - Any issues? What happens?

```
template<typename T>
class Pointer{
    T* _p;
public:
    Pointer(double* p) : _p(p) {}
    ~Pointer(){delete _p;}
    T& operator*();
};

template<typename T>
T& Pointer<T>::operator*(){
    return *_p;
}

Pointer<double> ptr(new double);
*ptr = 0.1;
std::cout<<*ptr<<std::endl;
```

Templates - Any issues? What happens?

```
template<typename T>
class Pointer{
    T* _p;
public:
    Pointer(double* p) : _p(p) {}
    ~Pointer(){delete _p;}
    T& operator*() { return *_p; }
    template<typename S> S* foo(){return new S;}
};

Pointer<double> ptr(new double);
*ptr = 0.1;
std::cout<<ptr.foo<int>()<<std::endl;
```

Templates - Any issues? What happens?

```cpp
template<typename T>
class Pointer{
    T* _p;
public:
    Pointer(double* p) : _p(p) {}
    ~Pointer(){delete _p;}
    T& operator*() { return *_p; }
    template<typename S> S* foo();
};

template<typename T> template<typename S> S* Pointer<T>::foo(){
    return new S;
}

Pointer<double> ptr(new double);
*ptr = 0.1;
std::cout<<ptr.foo<int>()<<std::endl;
```

Templates - Not like this

```
template<typename T>
class Pointer{
    T* _p;
public:
    Pointer(double* p) : _p(p) {}
    ~Pointer(){delete _p;}
    T& operator*() { return *_p; }
    template<typename S> S* foo();
};

template<typename T, typename S> S* Pointer<T>::foo(){
    return new S;
}

Pointer<double> ptr(new double);
*ptr = 0.1;
std::cout<<ptr.foo<int>()<<std::endl;
```

Templates - Performance

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

double* a = zero<3>(1000);
```

→ https://godbolt.org/

Templates - Performance

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

double* a = zero<3>(1000);
```

→ https://godbolt.org/

Smart pointers

```
template<typename T>
class Pointer{
    T* _p;
public:
    Pointer(double* p) : _p(p) {}
    ~Pointer(){delete _p;}
    T& operator*() { return *_p; }
};

Pointer<double> ptr(new double);
*ptr = 0.1;
```

- smart pointers can be used to handle memory deletion for you
- “Garbage Collection of C++”

Smart pointers - unique

```
#include <iostream>
#include <memory>

int main(){
    std::unique_ptr<double> p1(new double);
    *(p1+10) = 0.123;
    std::unique_ptr<double> p2 = std::move(p1);
    std::cout<<*p2<<" "<<*p1<<std::endl;
    return 0;
}
```

- unique_ptr stores unique reference
- pointer to reference can only exist once in memory
- p1 no longer stores address after giving it to p2

Smart pointers - shared

```
#include <iostream>
#include <memory>

int main(){
    std::shared_ptr<double> p1(new double);
    std::shared_ptr<double> p2 = p1;

    std::cout<<*p2<<" "<<*p1<<std::endl;
    std::cout << p1.use_count() << std::endl; // #pointers to address
    return 0;
}
```

- shared_ptr stores shared reference
- different pointers can point to same address
- p1 still stores address (assignment operator allowed)

Your turn - What goes wrong? (smart2.cpp)

```
template <typename T>
class Container{
    T _data;
public:
    Container(T data) : _data(data){}
    T Get()const{return _data;}
    T& Get(){return _data;}
};

std::unique_ptr<double> p(new double);
Container<std::unique_ptr<double>> c(p);
*(c.Get()) = 0.1;
std::cout<<"out = "<<*(c.Get())<<std::endl;
```

Your turn - What goes wrong? (smart3.cpp)

```
template <typename T>
class Container{
    T _data;
public:
    Container(T data){_data = move(data);}
    T Get()const{return _data;}
    T& Get(){return _data;}
};

std::unique_ptr<double> p(new double);
Container<std::unique_ptr<double>> c(p);
*(c.Get()) = 0.1;
std::cout<<"out = "<<*(c.Get())<<std::endl;
```

Your turn - Does this work? (smart4.cpp)

```
template <typename T>
class Container{
    T _data;
public:
    Container(T& data){_data = move(data);}
    T Get()const{return _data;}
    T& Get(){return _data;}
};

std::unique_ptr<double> p(new double);
Container<std::unique_ptr<double>> c(p);
*(c.Get()) = 0.1;
std::cout<<"out = "<<*(c.Get())<<std::endl;
```

Your turn - Does this work? (smart.cpp)

```cpp
template <typename T>
class Container{
    T _data;
public:
    Container(T data) : _data(data){}
    T Get()const{return _data;}
    T& Get(){return _data;}
};

std::shared_ptr<double> p(new double);
Container<std::shared_ptr<double>> c(p);
*(c.Get()) = 0.1;
std::cout<<"out = "<<*(c.Get())<<std::endl;
```

Homework

Exercise

Write your own shared smart pointer class which can store dynamical arrays. Use templates to make your code more flexible.