Object Oriented Programming and Meta-Programming (C++)

Presented by Alex Kosenkov

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

software carpentry

Agenda

- Programming paradigms
- Classes: derivation and templates
- Design patterns
- Meta-programming

Paradigms (imperative)

- Procedural (C, Fortran etc)
- Object-oriented (C++, Java, Python etc)
- Aspect-oriented (AspectJ)
- Dataflow (Verilog, VHDL, Linda etc)

Paradigms: OOP principles

- Abstraction
- Encapsulation (hiding implementation)
- Inheritance
- Polymorphism

Classes and instances

```cpp
#include <stdio.h>
#include <stdlib.h>
#include <iostream>
#include <vector>

namespace view {
    class Square {
    public:
        Square(int m, int rgb = 0)
        : points(m*m), dim(m), color(rgb)
        {
        }
        void draw(){
        }
        void erase(){
        }
        int get_dim() const {
            return dim;
        }
        void set_dim(int m){
            this->dim = m;
            this->erase();
            this->draw();
        }
       ~Square(){
           this->erase();
        }
    private:
        int dim;
        int color;
        std::vector<std::pair<int,int> > points;
    };
}

int main(){
    view::Square A(10,255);
    A.draw();

    view::Square* B = new view::Square(10);
    B->draw();

    delete B;
    return 0;
}
```

Classes and inheritance

```cpp
class Circle : public Shape {
public:
    Circle(int r, int rgb = 0) : Shape(rgb), radius(r) {
        this->reserve(radius);
    }
    void reserve(int r){
        Shape::reserve(M_PI*r*r);
    }
    void set_radius(int r){
        this->erase();
        this->radius = r;
        this->reserve(r);
        this->draw();
    }
    int get_radius() const {
        return radius;
    }
private:
    int radius;
};
```

```cpp
class Shape {
public:
    Shape(int rgb) : color(rgb) { }
    void draw(){
    }
    void erase(){
    }
    void reserve(int n){
        this->points.resize(n);
    }
   ~Shape(){
       this->erase();
    }
protected:
    std::vector<std::pair<int,int> > points;
private:
    int color;
};
```

```cpp
class Square : public Shape {
public:
    Square(int m, int rgb = 0) : Shape(rgb), dim(m) {
        this->reserve(m);
    }
    void reserve(int n){
        Shape::reserve(n*n);
    }
    void set_dim(int m){
        this->erase();
        this->dim = m;
        this->reserve(m);
        this->draw();
    }
    int get_dim() const {
        return dim;
    }
private:
    int dim;
};
```

```cpp
void resize(Circle* s, int size){
    s->set_radius(size);
}
void resize(Square* s, int size){
    s->set_dim(size);
}

int main(){
    Square A(10,255);
    Circle B(10,255);

    resize(&A, 100);
    resize(&B, 100);
    return 0;
}
```

Classes and run-time polymorphism

```cpp
class Circle : public Shape {
public:
    Circle(int r, int rgb = 0) : Shape(rgb), radius(r) {
        this->reserve(radius);
    }
    void reserve(int r){
        Shape::reserve(M_PI*r*r);
    }
    virtual void set_dim(int r){
        this->erase();
        this->radius = r;
        this->reserve(r);
        this->draw();
    }
    int get_radius() const {
        return radius;
    }
private:
    int radius;
};
```

```cpp
class Shape {
public:
    Shape(int rgb) : color(rgb) { }
    void draw(){
    }
    void erase(){
    }
    void reserve(int n){
        this->points.resize(n);
    }
    virtual void set_dim(int dim) = 0;
   ~Shape(){
       this->erase();
    }
protected:
    std::vector<std::pair<int,int> > points;
private:
    int color;
};
```

```cpp
class Square : public Shape {
public:
    Square(int m, int rgb = 0) : Shape(rgb), dim(m) {
        this->reserve(m);
    }
    void reserve(int n){
        Shape::reserve(n*n);
    }
    virtual void set_dim(int m){
        this->erase();
        this->dim = m;
        this->reserve(m);
        this->draw();
    }
    int get_dim() const {
        return dim;
    }
private:
    int dim;
};
```

```cpp
void resize(Shape* s, int size){
    s->set_dim(size);
}

int main(){
    Shape* A = new Square(10,255);
    Shape* B = new Circle(10,255);

    resize(A, 100);
    resize(B, 100);
    return 0;
}
```

Classes and run-time polymorphism (alt)

```cpp
class Circle : public Shape {
public:
    Circle(int r, int rgb = 0) : Shape(r, rgb) {
        this->reserve(r);
    }
    virtual void reserve(int r){
        Shape::reserve(M_PI*r*r);
    }
    int get_radius() const {
        return this->dim;
    }
};
```

```cpp
class Square : public Shape {
public:
    Square(int m, int rgb = 0) : Shape(m, rgb) {
        this->reserve(m);
    }
    virtual void reserve(int n){
        Shape::reserve(n*n);
    }
    int get_dim() const {
        return this->dim;
    }
};
```

```cpp
class Shape {
public:
    Shape(int dim, int rgb) : dim(dim), color(rgb) { }
    void draw(){
    }
    void erase(){
    }
    virtual void reserve(int n){
        this->points.resize(n);
    }
    void set_dim(int dim){
        this->erase();
        this->dim = dim;
        this->reserve(dim);
        this->draw();
    }
   ~Shape(){
       this->erase();
    }
protected:
    std::vector<std::pair<int,int> > points;
    int dim;
private:
    int color;
};
```

```cpp
void resize(Shape* s, int size){
    s->set_dim(size);
}

int main(){
    Shape* A = new Square(10,255);
    Shape* B = new Circle(10,255);

    resize(A, 100);
    resize(B, 100);
    return 0;
}
```

Classes and compile-time polymorphism

Function templates:

```cpp
template <typename T>
T max(T a, T b) {
    return a > b ? a : b;
}

int main(){
    std::cout << max(13, 14)   << "\n"; // same as max<int>(13, 14)
    std::cout << max(4.5, 4.6) << "\n"; // same as max<float>(4.5, 4.6)
    std::cout << max<int>(4.5, 4.6) << "\n"; // explicit template / implicit casting
    return 0;
}
```

Function overloading:

```cpp
inline double distance(const std::complex<double>& a, const std::complex<double>& b){
    return fabs(std::norm(a) - std::norm(b));
}

inline double distance(double a, double b){
    return fabs(fabs(a) - fabs(b));
}
```

Classes and compile-time polymorphism

Class template:

```cpp
template <typename T>
class vector {
public:
    typedef T value_type;
    explicit vector(size_t n, T init = T()) : length(n){
        elements = (T*)std::malloc(sizeof(T)*length);
        for(size_t i = 0; i < n; i++) elements[i] = init;
    }
    T operator[](size_t k) const {
        return elements[k];
    }
    T& operator[](size_t k){
        return elements[k];
    }
    template<typename S>
    vector<T>& operator += (const vector<S>& other){
        for(size_t i = 0; i < length; i++) (*this)[i] += other[i];
        return *this;
    }
   ~vector(){
        std::free(elements);
    }
private:
    value_type* elements;
    size_t length;
};
```

```cpp
int main(){
    vector<int> a(29, 7);
    vector<float> b(29, 4.5);

    a += b;
    return 0;
}
```

Classes and compile-time polymorphism

Template Class Specialisation:

```cpp
template <>
class vector<bool> {
public:
    typedef bool value_type;
    explicit vector(size_t n, bool init = false) : length(n){
        int size = std::ceil((float)length/8);
        elements = (char*)std::malloc(size);
        memset(elements, (init ? 255 : 0), size);
    }
    bool operator[](size_t k) const {
        return (elements[k/8] & (1 << (k % 8))) >> k % 8;
    }
    void set(size_t k, bool value){
        if(value) elements[k/8] |= 1 << k % 8;
        else elements[k/8] &= ~(1 << k % 8);
    }
   ~vector(){
        std::free(elements);
    }
private:
    char* elements;
    size_t length;
};
```

Design patterns: principles

- Inversion of Control
- S.O.L.I.D.

Design patterns: S.O.L.I.D.

- The Single Responsibility Principle: one reason to change
- The Open Closed Principle: read-only, open for extension
- The Liskov Substitution Principle: substitution-robust behaviour
- The Interface Segregation Principle: dependance - means usage
- The Dependency Inversion Principle: low-level depends on high-level

Design patterns: examples

- Dependency Injection
- Template method pattern
- Model View Controller
- Adapter
- Singleton
- Factory

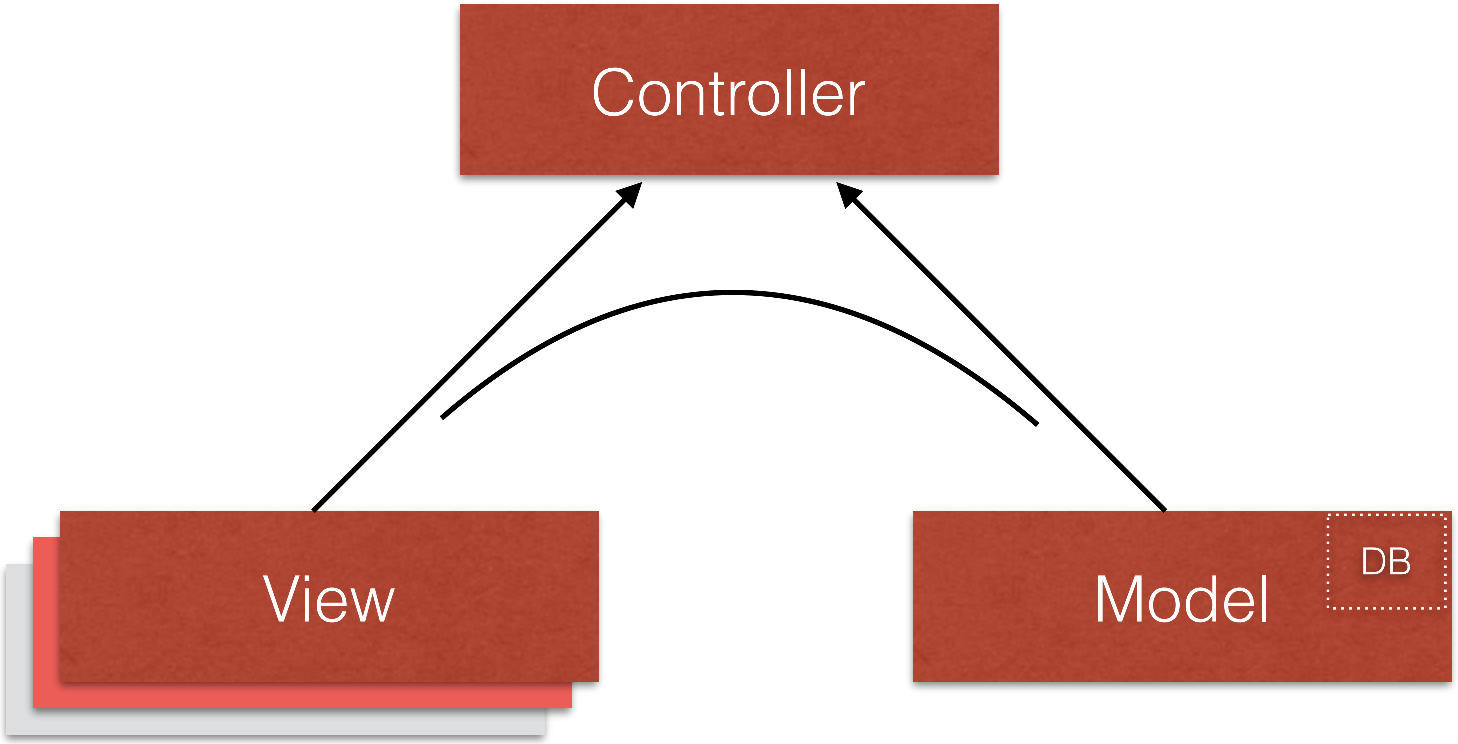
Design patterns: Template method pattern

```cpp
class Algorithm {
public:
    virtual void initialize() = 0;
    virtual void writeAnswer() = 0;
    virtual bool converged() = 0;
    virtual void iterate() = 0;

    void compute() {
        initialize();
        while(!converged()) iterate();
        writeAnswer();
    }
};

class IterativeAlgorithm : public Algorithm {
public:
    virtual void initialize(){
    }
    virtual void writeAnswer(){
    }
    virtual bool converged(){
    }
    virtual void iterate(){
    }
};
```

Design patterns: Model View Controller



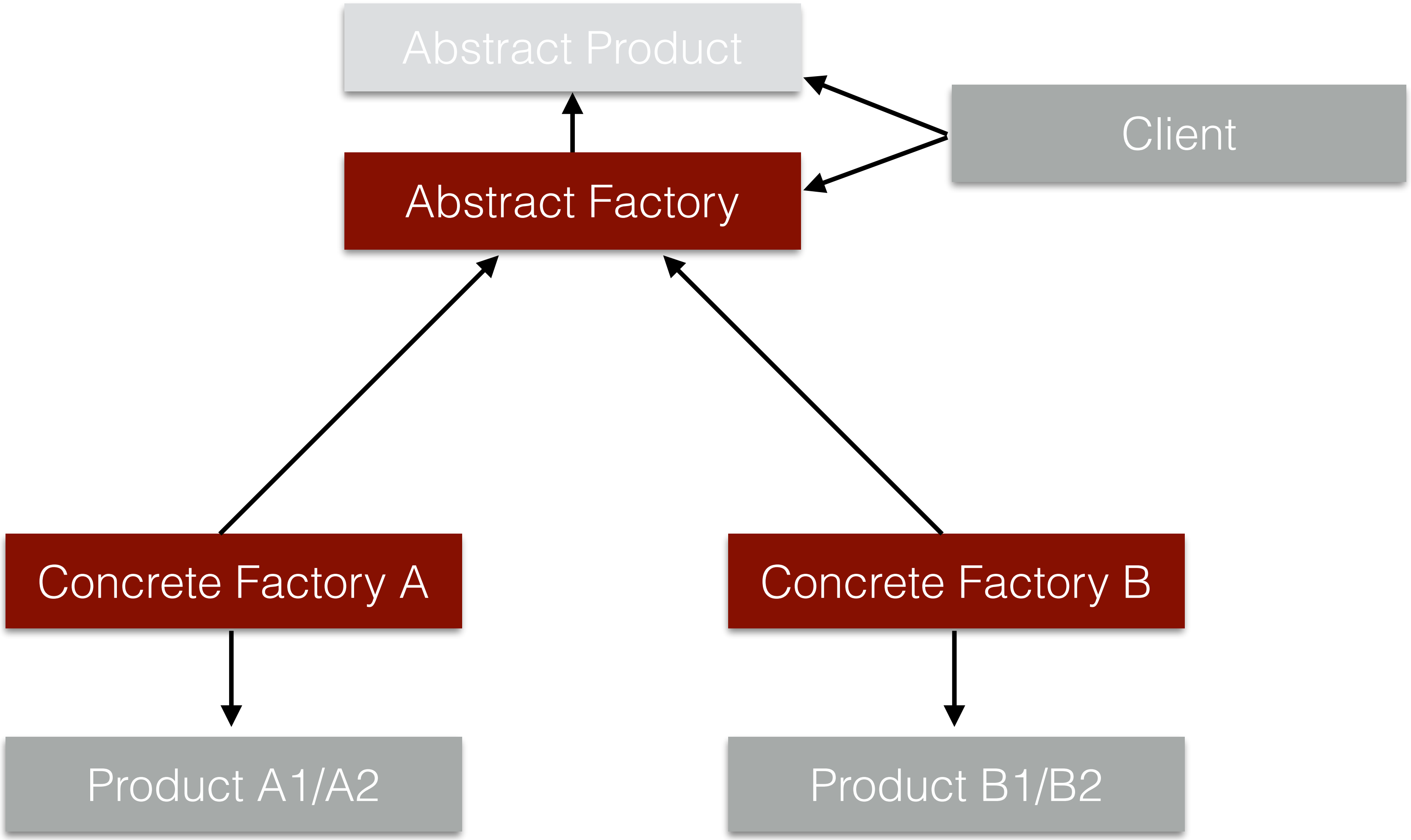
Design patterns: Singleton

```
class A {
protected:
    A(){ }
    A(A const&){ }
};

template <typename T>
class singleton : public T {
public:
    static T& instance(){
        static singleton s;
        return s;
    }
private:
    inline singleton(){}
    singleton(singleton const&);
    singleton& operator=(singleton const&);
};

int main(){
    A& a = singleton<A>::instance();
    A b(a); // error
    A c;    // error
    return 0;
}
```

Design patterns: Factory



Meta-programming

Example: prime number

```
#include <stdio.h>
#include <stdlib.h>
#include <iostream>

constexpr bool is_prime(unsigned long n, unsigned long div){
    return n % div == 0 ?
           false : (div * div > n ? true : is_prime(n, div + 2));
}
constexpr bool is_prime(unsigned long n){
    return n == 2 || n == 3 || (n % 2 && is_prime(n, 3));
}

template<bool Prime>
class Number {
public:
    Number(){ printf("Default number constructor\n"); }
};

template<>
class Number<true> {
public:
    Number(){ printf("Prime number constructor\n"); }
};

int main(){
    Number<is_prime(997)> a;
    Number<is_prime(169)> b;
    return 0;
}
```

Meta-programming

Example: looping

```cpp
#include <stdio.h>
#include <stdlib.h>
#include <iostream>

constexpr int max(int a, int b){
    return a > b ? a : b;
}

template<int N, template<int I> class Iteration>
constexpr bool repeat(){
    return N != 0 ? Iteration<max(1,N)>::iterate() && repeat<max(0,N-1),Iteration>() : false;
}

constexpr int factorial(int N){
    return N != 1 ? N*factorial(N-1) : 1;
}

template<int N>
struct compile_time {
    static void check(){ std::cout << N << "\n"; }
};

template<int I>
class loop_iteration{
public:
    static bool iterate(){
        std::cout << "Loop iteration: " << I << "; ";
        std::cout << "Value: ";
        compile_time<factorial(I)>::check();
        return true;
    }
};

int main(){
    repeat<10,loop_iteration>();
    return 0;
}
```

Meta-programming

Example: lambda deduction

```cpp
#include <stdio.h>
#include <stdlib.h>
#include <iostream>
#include <functional>

template <typename Function>
struct function_traits : public function_traits<decltype(&Function::operator())> {};

template <typename ClassType, typename ReturnType, typename... Args>
struct function_traits<ReturnType(ClassType::*)(Args...) const> {
    typedef ReturnType (*pointer)(Args...);
    typedef const std::function<ReturnType(Args...)> function;
};

template <typename Function>
typename function_traits<Function>::function to_function (Function& lambda) {
    return static_cast<typename function_traits<Function>::function>(lambda);
}

template <class L>
struct overload_lambda : L {
    overload_lambda(L l) : L(l) {}
    template <typename... T>
    void operator()(T&& ... values){
        to_function(*(L*)this)(std::forward<T>(values)...);
    }
};

template <class L>
overload_lambda<L> lambda(L l){
    return overload_lambda<L>(l);
}

int main(){
    auto call = lambda([](int a){ std::cout << a << "\n"; });
    call(10);
    return 0;
}
```

Meta-programming

Example: Substitution failure is not an error (SFINAE) lookup

```
#include <stdio.h>
#include <stdlib.h>
#include <iostream>
#include <functional>

namespace allocators {
    template<typename T> struct default_allocator {
        static void info(){ std::cout << "Default allocator!\n"; }
    };
    template<typename T> struct my_allocator {
        static void info(){ std::cout << "My allocator!\n"; }
    };
}
template <typename T>
struct has_allocator {
    template <typename T1> static typename T1::allocator_type test(int);
    template <typename>    static void test(...);
    enum { value = !std::is_void<decltype(test<T>(0))>::value };
};
template <bool HAS, typename T> struct checked_get_allocator {};
template <typename T> struct checked_get_allocator<true, T> { typedef typename T::allocator_type type; };
template <typename T> struct checked_get_allocator<false, T> { typedef typename allocators::default_allocator<T> type; };
template <typename T> struct get_allocator { typedef typename checked_get_allocator<has_allocator<T>::value, T>::type type; };

template <class T> struct smart_type : public T {
    typedef typename get_allocator<T>::type allocator_type;
};
int main(){
    struct user_A { };
    struct user_B { typedef allocators::my_allocator<double> allocator_type; };
    smart_type<user_A>::allocator_type::info();
    smart_type<user_B>::allocator_type::info();
    return 0;
}
```

Questions

End of the second part

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

cscs
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

software carpentry