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)

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich
CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre
software carpentry

Paradigms: OOP principles

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

Classes and instances

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <iostream>
4 #include <vector>
5
6 namespace view {
7     class Square {
8     public:
9         Square(int m, int rgb = 0)
10        : points(m*m), dim(m), color(rgb)
11        {
12        }
13        void draw(){
14        }
15        void erase(){
16        }
17        int get_dim() const {
18            return dim;
19        }
20        void set_dim(int m){
21            this->dim = m;
22            this->erase();
23            this->draw();
24        }
25       ~Square(){
26           this->erase();
27        }
28    private:
29        int dim;
30        int color;
31        std::vector<std::pair<int,int> > points;
32    };
33 }
34
35 int main(){
36     view::Square A(10,255);
37     A.draw();
38
39     view::Square* B = new view::Square(10);
40     B->draw();
41
42     delete B;
43     return 0;
44 }
```

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich
CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre
software carpentry

Classes and inheritance

```
19  class Circle : public Shape {
20  public:
21      Circle(int r, int rgb = 0) : Shape(rgb), radius(r) {
22          this->reserve(radius);
23      }
24      void reserve(int r){
25          Shape::reserve(M_PI*r*r);
26      }
27      void set_radius(int r){
28          this->erase();
29          this->radius = r;
30          this->reserve(r);
31          this->draw();
32      }
33      int get_radius() const {
34          return radius;
35      }
36  private:
37      int radius;
38  };
```

```
39  class Square : public Shape {
40  public:
41      Square(int m, int rgb = 0) : Shape(rgb), dim(m) {
42          this->reserve(m);
43      }
44      void reserve(int n){
45          Shape::reserve(n*n);
46      }
47      void set_dim(int m){
48          this->erase();
49          this->dim = m;
50          this->reserve(m);
51          this->draw();
52      }
53      int get_dim() const {
54          return dim;
55      }
56  private:
57      int dim;
58  };
```

```
1   class Shape {
2   public:
3       Shape(int rgb) : color(rgb) { }
4       void draw(){
5       }
6       void erase(){
7       }
8       void reserve(int n){
9           this->points.resize(n);
10      }
11     ~Shape(){
12         this->erase();
13      }
14  protected:
15      std::vector<std::pair<int,int> > points;
16  private:
17      int color;
18  };
```

```
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

```
20  class Circle : public Shape {
21  public:
22      Circle(int r, int rgb = 0) : Shape(rgb), radius(r) {
23          this->reserve(radius);
24      }
25      void reserve(int r){
26          Shape::reserve(M_PI*r*r);
27      }
28      virtual void set_dim(int r){
29          this->erase();
30          this->radius = r;
31          this->reserve(r);
32          this->draw();
33      }
34      int get_radius() const {
35          return radius;
36      }
37  private:
38      int radius;
39  };
```

```
1   class Shape {
2   public:
3       Shape(int rgb) : color(rgb) { }
4       void draw(){
5       }
6       void erase(){
7       }
8       void reserve(int n){
9           this->points.resize(n);
10      }
11      virtual void set_dim(int dim) = 0;
12     ~Shape(){
13         this->erase();
14      }
15  protected:
16      std::vector<std::pair<int,int> > points;
17  private:
18      int color;
19  };
```

```
40  class Square : public Shape {
41  public:
42      Square(int m, int rgb = 0) : Shape(rgb), dim(m) {
43          this->reserve(m);
44      }
45      void reserve(int n){
46          Shape::reserve(n*n);
47      }
48      virtual void set_dim(int m){
49          this->erase();
50          this->dim = m;
51          this->reserve(m);
52          this->draw();
53      }
54      int get_dim() const {
55          return dim;
56      }
57  private:
58      int dim;
59  };
```

```
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;
}
```

ETH Eidgenössische Technische Hochschule Zürich Swiss Federal Institute of Technology Zurich
CSCS Centro Svizzero di Calcolo Scientifico Swiss National Supercomputing Centre
software carpentry

Classes and run-time polymorphism (alt)

```
32  class Circle : public Shape {
33  public:
34      Circle(int r, int rgb = 0) : Shape(r, rgb) {
35          this->reserve(r);
36      }
37      virtual void reserve(int r){
38          Shape::reserve(M_PI*r*r);
39      }
40      int get_radius() const {
41          return this->dim;
42      }
43  };
```

```
44  class Square : public Shape {
45  public:
46      Square(int m, int rgb = 0) : Shape(m, rgb) {
47          this->reserve(m);
48      }
49      virtual void reserve(int n){
50          Shape::reserve(n*n);
51      }
52      int get_dim() const {
53          return this->dim;
54      }
55  };
```

```
 7  class Shape {
 8  public:
 9      Shape(int dim, int rgb) : dim(dim), color(rgb) { }
10      void draw(){
11      }
12      void erase(){
13      }
14      virtual void reserve(int n){
15          this->points.resize(n);
16      }
17      void set_dim(int dim){
18          this->erase();
19          this->dim = dim;
20          this->reserve(dim);
21          this->draw();
22      }
23     ~Shape(){
24         this->erase();
25      }
26  protected:
27      std::vector<std::pair<int,int> > points;
28      int dim;
29  private:
30      int color;
31  };
```

```
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:

```
1 template <typename T>
2 T max(T a, T b) {
3     return a > b ? a : b;
4 }
5
6 int main(){
7     std::cout << max(13, 14)   << "\n"; // same as max<int>(13, 14)
8     std::cout << max(4.5, 4.6) << "\n"; // same as max<float>(4.5, 4.6)
9     std::cout << max<int>(4.5, 4.6) << "\n"; // explicit template / implicit casting
10    return 0;
11 }
```

Function overloading:

```
12 inline double distance(const std::complex<double>& a, const std::complex<double>& b){
13     return fabs(std::norm(a) - std::norm(b));
14 }
15
16 inline double distance(double a, double b){
17     return fabs(fabs(a) - fabs(b));
18 }
```

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich
CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre
software carpentry

Classes and compile-time polymorphism

Class template:

```
1    template <typename T>
2    class vector {
3    public:
4        typedef T value_type;
5        explicit vector(size_t n, T init = T()) : length(n){
6            elements = (T*)std::malloc(sizeof(T)*length);
7            for(size_t i = 0; i < n; i++) elements[i] = init;
8        }
9        T operator[](size_t k) const {
10           return elements[k];
11       }
12       T& operator[](size_t k){
13           return elements[k];
14       }
15       template<typename S>
16       vector<T>& operator += (const vector<S>& other){
17           for(size_t i = 0; i < length; i++) (*this)[i] += other[i];
18           return *this;
19       }
20      ~vector(){
21           std::free(elements);
22       }
23   private:
24       value_type* elements;
25       size_t length;
26   };
```

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

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

ETH Eidgenössische Technische Hochschule Zürich Swiss Federal Institute of Technology Zurich

CSCS Centro Svizzero di Calcolo Scientifico Swiss National Supercomputing Centre

software carpentry

Template Class Specialisation:

```
28    template <>
29    class vector<bool> {
30    public:
31        typedef bool value_type;
32        explicit vector(size_t n, bool init = false) : length(n){
33            int size = std::ceil((float)length/8);
34            elements = (char*)std::malloc(size);
35            memset(elements, (init ? 255 : 0), size);
36        }
37        bool operator[](size_t k) const {
38            return (elements[k/8] & (1 << (k % 8))) >> k % 8;
39        }
40        void set(size_t k, bool value){
41            if(value) elements[k/8] |= 1 << k % 8;
42            else elements[k/8] &= ~(1 << k % 8);
43        }
44       ~vector(){
45            std::free(elements);
46        }
47    private:
48        char* elements;
49        size_t length;
50    };
```

Design patterns: principles

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

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich
CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre
software carpentry

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

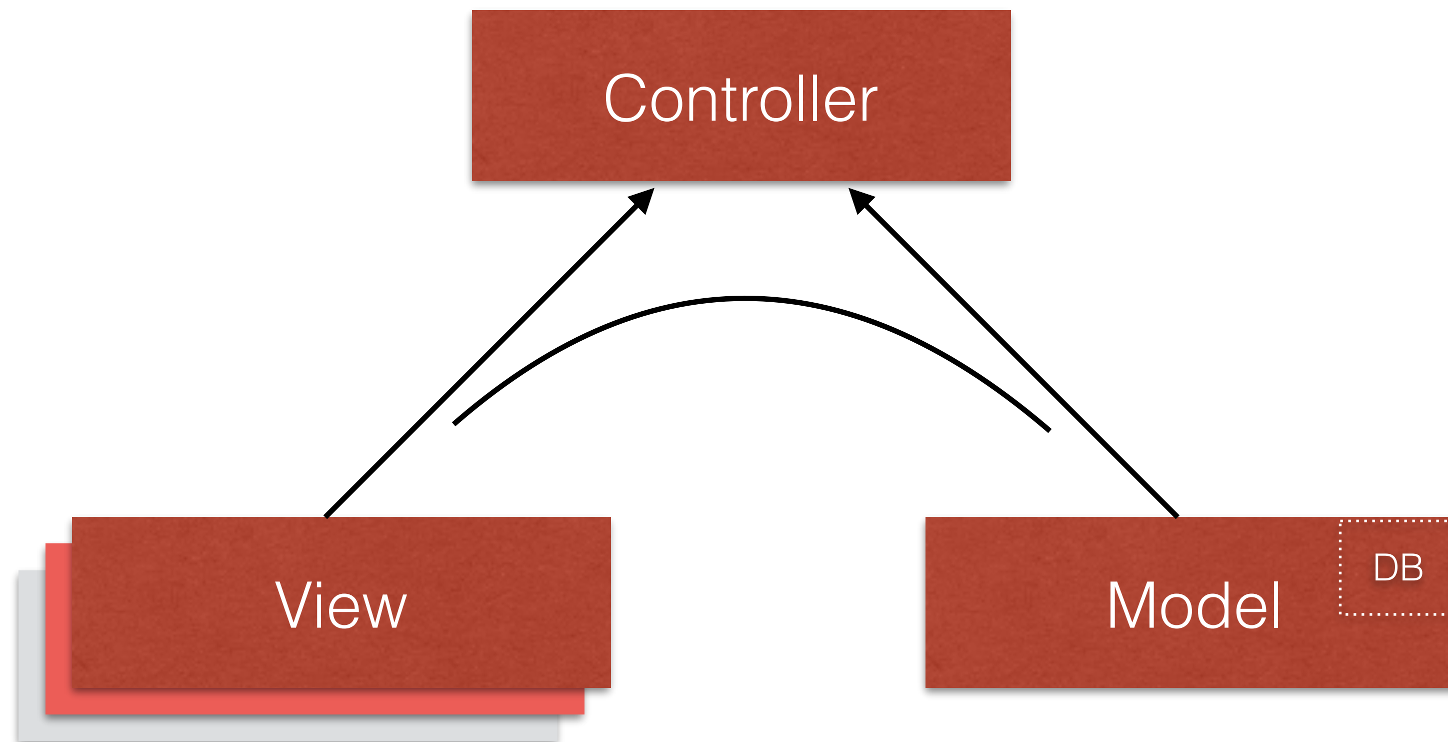
- Dependency Injection
- Template method pattern
- Model View Controller
- Adapter
- Singleton
- Factory

Design patterns: Template method pattern

```
 1 class Algorithm {
 2 public:
 3     virtual void initialize() = 0;
 4     virtual void writeAnswer() = 0;
 5     virtual bool converged() = 0;
 6     virtual void iterate() = 0;
 7
 8     void compute() {
 9         initialize();
10         while(!converged()) iterate();
11         writeAnswer();
12     }
13 };
14
15 class IterativeAlgorithm : public Algorithm {
16 public:
17     virtual void initialize(){
18     }
19     virtual void writeAnswer(){
20     }
21     virtual bool converged(){
22     }
23     virtual void iterate(){
24     }
25 };
```

ETH Eidgenössische Technische Hochschule Zürich Swiss Federal Institute of Technology Zurich
CSCS Centro Svizzero di Calcolo Scientifico Swiss National Supercomputing Centre
software carpentry

Design patterns: Model View Controller

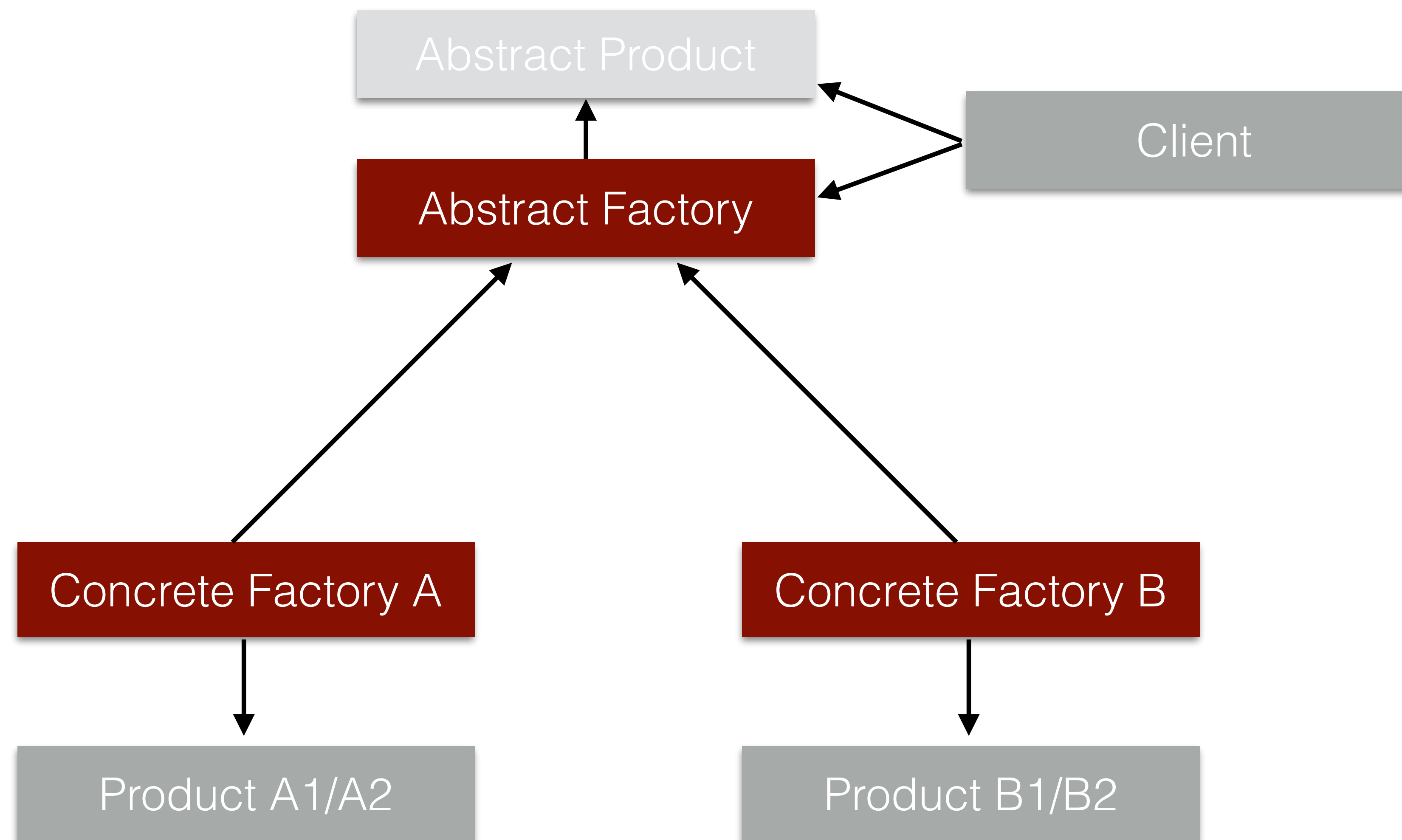


Design patterns: Singleton

```
1     class A {
2     protected:
3         A(){ }
4         A(A const&){ }
5     };
6
7     template <typename T>
8     class singleton : public T {
9     public:
10        static T& instance(){
11            static singleton s;
12            return s;
13        }
14    private:
15        inline singleton(){}
16        singleton(singleton const&);
17        singleton& operator=(singleton const&);
18    };
19
20    int main(){
21        A& a = singleton<A>::instance();
22        A b(a); // error
23        A c;    // error
24        return 0;
25    }
```

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich
CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre
software carpentry

Design patterns: Factory



Meta-programming

Example: prime number

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <iostream>
4
5 constexpr bool is_prime(unsigned long n, unsigned long div){
6     return n % div == 0 ?
7            false : (div * div > n ? true : is_prime(n, div + 2));
8 }
9 constexpr bool is_prime(unsigned long n){
10     return n == 2 || n == 3 || (n % 2 && is_prime(n, 3));
11 }
12
13 template<bool Prime>
14 class Number {
15 public:
16     Number(){ printf("Default number constructor\n"); }
17 };
18
19 template<>
20 class Number<true> {
21 public:
22     Number(){ printf("Prime number constructor\n"); }
23 };
24
25 int main(){
26     Number<is_prime(997)> a;
27     Number<is_prime(169)> b;
28     return 0;
29 }
```

Meta-programming

Example: looping

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <iostream>
4
5 constexpr int max(int a, int b){
6     return a > b ? a : b;
7 }
8
9 template<int N, template<int I> class Iteration>
10 constexpr bool repeat(){
11     return N != 0 ? Iteration<max(1,N)>::iterate() && repeat<max(0,N-1),Iteration>() : false;
12 }
13
14 constexpr int factorial(int N){
15     return N != 1 ? N*factorial(N-1) : 1;
16 }
17
18 template<int N>
19 struct compile_time {
20     static void check(){ std::cout << N << "\n"; }
21 };
22
23 template<int I>
24 class loop_iteration{
25 public:
26     static bool iterate(){
27         std::cout << "Loop iteration: " << I << "; ";
28         std::cout << "Value: ";
29         compile_time<factorial(I)>::check();
30         return true;
31     }
32 };
33
34 int main(){
35     repeat<10,loop_iteration>();
36     return 0;
37 }
```

ETH Eidgenössische Technische Hochschule Zürich Swiss Federal Institute of Technology Zurich
CSCS Centro Svizzero di Calcolo Scientifico Swiss National Supercomputing Centre
software carpentry

Meta-programming

Example: lambda deduction

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <iostream>
4 #include <functional>
5
6 template <typename Function>
7 struct function_traits : public function_traits<decltype(&Function::operator())> {};
8
9 template <typename ClassType, typename ReturnType, typename... Args>
10 struct function_traits<ReturnType(ClassType::*)(Args...) const> {
11     typedef ReturnType (*pointer)(Args...);
12     typedef const std::function<ReturnType(Args...)> function;
13 };
14
15 template <typename Function>
16 typename function_traits<Function>::function to_function (Function& lambda) {
17     return static_cast<typename function_traits<Function>::function>(lambda);
18 }
19
20 template <class L>
21 struct overload_lambda : L {
22     overload_lambda(L l) : L(l) {}
23     template <typename... T>
24     void operator()(T&& ... values){
25         to_function(*(L*)this)(std::forward<T>(values)...);
26     }
27 };
28
29 template <class L>
30 overload_lambda<L> lambda(L l){
31     return overload_lambda<L>(l);
32 }
33
34 int main(){
35     auto call = lambda([](int a){ std::cout << a << "\n"; });
36     call(10);
37     return 0;
38 }
```

Meta-programming

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

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <iostream>
4 #include <functional>
5
6 namespace allocators {
7     template<typename T> struct default_allocator {
8         static void info(){ std::cout << "Default allocator!\n"; }
9     };
10    template<typename T> struct my_allocator {
11        static void info(){ std::cout << "My allocator!\n"; }
12    };
13 }
14 template <typename T>
15 struct has_allocator {
16     template <typename T1> static typename T1::allocator_type test(int);
17     template <typename>    static void test(...);
18     enum { value = !std::is_void<decltype(test<T>(0))>::value };
19 };
20 template <bool HAS, typename T> struct checked_get_allocator {};
21 template <typename T> struct checked_get_allocator<true, T> { typedef typename T::allocator_type type; };
22 template <typename T> struct checked_get_allocator<false, T> { typedef typename allocators::default_allocator<T> type; };
23 template <typename T> struct get_allocator { typedef typename checked_get_allocator<has_allocator<T>::value, T>::type type; };
24
25 template <class T> struct smart_type : public T {
26     typedef typename get_allocator<T>::type allocator_type;
27 };
28 int main(){
29     struct user_A { };
30     struct user_B { typedef allocators::my_allocator<double> allocator_type; };
31     smart_type<user_A>::allocator_type::info();
32     smart_type<user_B>::allocator_type::info();
33     return 0;
34 }
```

ETH Eidgenössische Technische Hochschule Zürich Swiss Federal Institute of Technology Zurich
CSCS Centro Svizzero di Calcolo Scientifico Swiss National Supercomputing Centre
software carpentry

Questions

End of the second part