VICTOR A. LASKIN, 2015

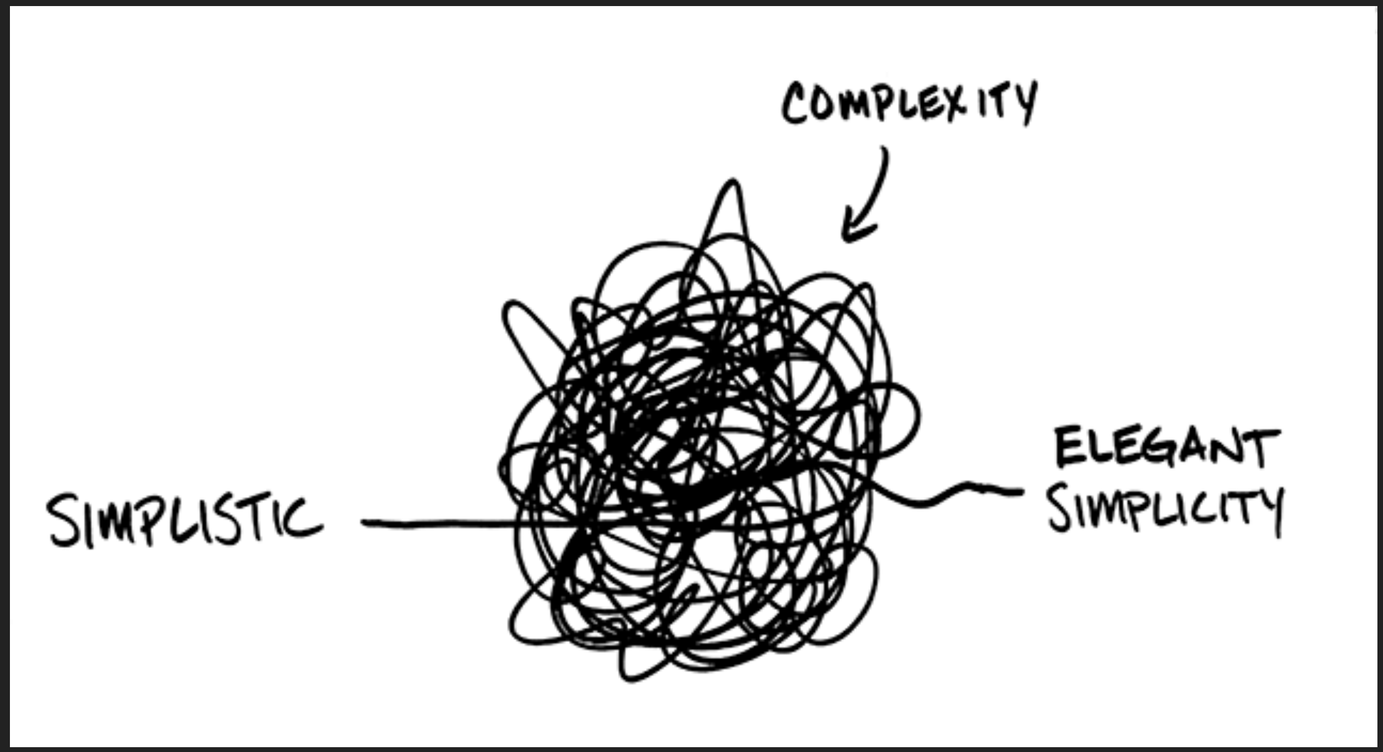
THE WAYS TO AVOID COMPLEXITY IN MODERN C++

PERFECTION IS ACHIEVED NOT WHEN THERE IS NOTHING MORE TO ADD, BUT WHEN THERE IS NOTHING MORE TO TAKE AWAY.

–Antoine de Saint Exupery

WHAT THE PROBLEM?

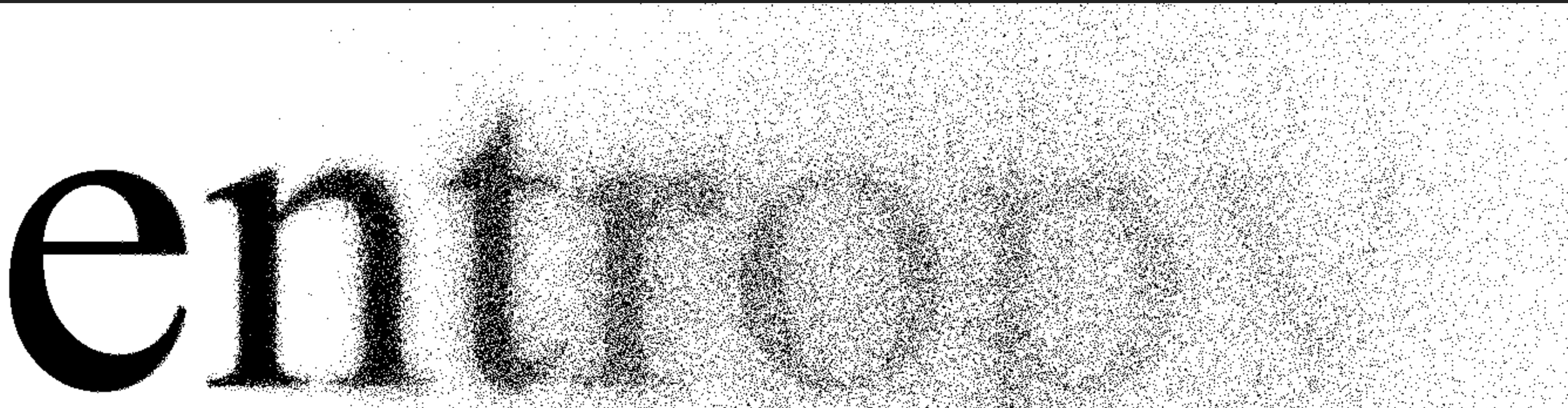
- Programmers tend to write complicated solutions as they see it as part of **creativity**.
- C++ is so flexible…
- It’s actually easy to produce ‘complex’ spaghetti architecture.



WHAT THE PROBLEM – PART 2

- During developer's life we get used to **apply same 'not perfect' solutions over and over** so they start to seem simple to us.
- We use **libs**. As libs often want to cover wide range of cases - their interfaces become not so simple as they could be.
- **Copy-paste** does not simplify things.

OVER TIME ENTROPY OF PROJECT IS GROWING…



OBVIOUS WAY – TRY TO MAKE IT “SIMPLE”

- “Fighting complexity” as true way of experienced developer
- “make simple things simple” Bjarn
- So this presentation is about the ways to do it!

IDEA

SO THE IDEA IS TO GATHER

C++11/14'S POWER

TO CREATE NEW WAYS / FUNCTIONAL INSTRUMENTS / TOOLS, ETC TO PRODUCE MORE PERFECT CODE

PRINCIPLE WAYS TO ACHIVE SIMPLICITY

- Organize (classifications, patterns, GOF, etc)
- Hide/encapsulate (OOP - Hide complexity)
- Divide (factorization)
- Reduce/Eliminate (Occam’s razor)
- Speed up
- Combine, reuse (create tools). Extract functionality.

THIS LINE IS ENOUGH REASON TO SWITCH TO C++11

```
for(auto item : list) ...
```

- Avoid non-ranged cycles - use them only at low algorithmic layer

DONT USE NEW/DELETE -> ONLY RAII & SMART POINTERS

- Use RAII where it’s possible
- Use smart-pointers where you absolutely can’t use RAII
- DON’T USE new/delete
- Use make_shared to allocate shared_ptr<>

USE CONST WHERE ITS POSSIBLE

- Less chance to make logic mistake, complex workflow or produce corrupted state
- Not only data: mark methods as const
- Immutable data paradigm from functional programming will be discussed further

AUTO

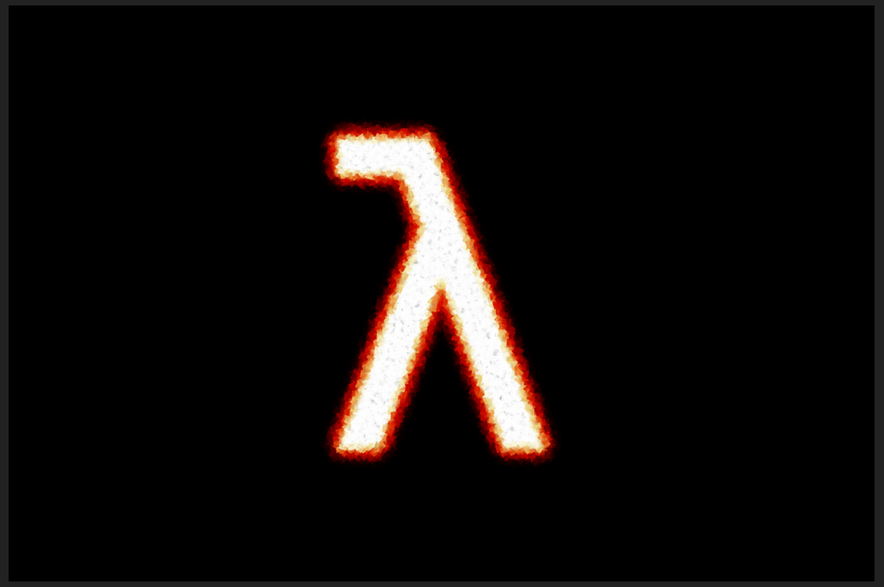
- *Auto* gives a way to save A LOT OF SPACE
- Places where a lot of *auto*s decreases readability are rare!
- Avoid type casting mistakes
- Return type detection (C++14) - saves a lot of space
- Polymorfic lambdas

USE LAMBDAS / STD::FUNCTION

- Every non trivial system has need for event handlers, etc - C++11 has not so bad syntax to live with
- Don’t be afraid of unreadable lambda hell (JS)
- Use structured approach to work with *lamdas*
- Everywhere when you want to create new class: think - may be it’s just simple function?
- Messaging architectures, Distributed systems - place code where it should be located, not where you forced to write it

WHY FUNCTIONAL PROGRAMMING?

- Where is simplicity here?
- Pure functions ARE SIMPLE
- Immutable state is simple
- Math-like data processing is easy write and read
- Easy to build concurrent/distributed systems
- C++11/14 is functional enough



IMMUTABLE DATA

- Immutable data concept is SIMPLE
- Stateless -> no errors
- No problem of inconsistent data (requires full construction)
- Perfect for data processing, especially inside pure functions: y = f(x)
- Perfect for concurrency, distributed systems
- Share same data over *std::shared_ptr*

POSSIBLE IMPLEMENTATION

- True CONSTs
- JSON serialization

```
class EventData : public IImmutable {
public:
    const int id;
    const string title;
    const double rating;

    SERIALIZE_JSON(EventData, id, title, rating);
};

using Event = EventData::Ptr;
```

MORE REALISTIC DATA – NESTED JSON SERIALIZATION

```
class ScheduleItemData : public IImmutable {
public:
    const time_t start;
    const time_t finish;
    SERIALIZE_JSON(ScheduleItemData, start, finish);
};

using ScheduleItem = ScheduleItemData::Ptr;


class EventData : public IImmutable {
public:
    const int id;
    const string title;
    const double rating;
    const vector<ScheduleItem> schedule;
    const vector<int> tags;

    SERIALIZE_JSON(EventData, id, title, rating, schedule, tags);
};

using Event = EventData::Ptr;
```

SIMPLE USAGE

```
Event event = EventData(136, "Nice event", 4.88, {ScheduleItemData(1111,2222),
ScheduleItemData(3333,4444)}, {45,323,55});

// serialisation
string json = event->toJSON();

// deserialisation
Event eventCopy = EventData::fromJSON(json);


// How to "set" fields
Event anotherEvent = event->set_rating(3.56);
```

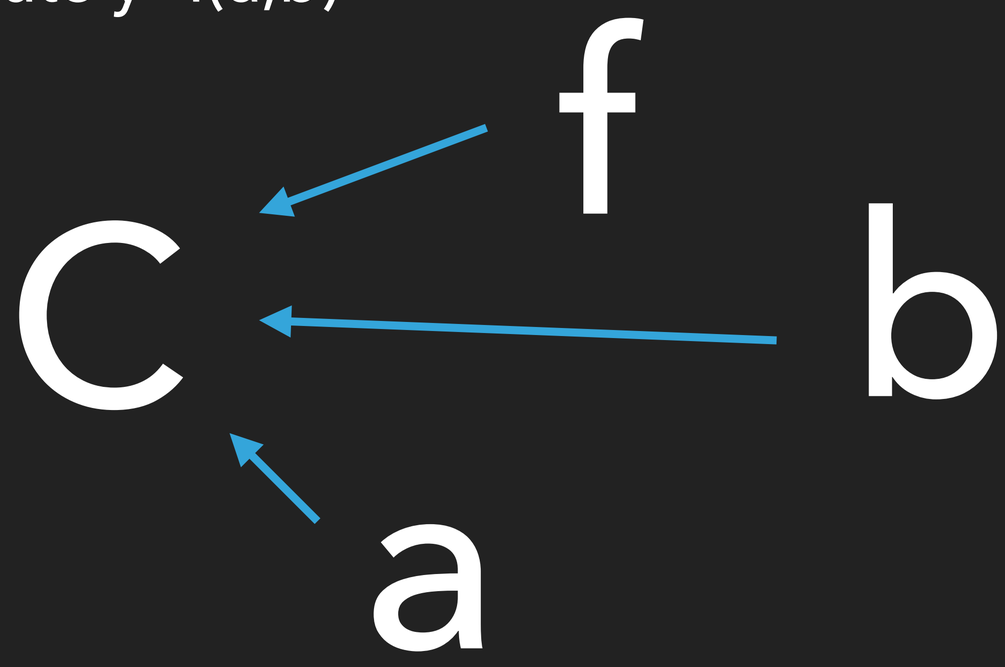
Implementation details - http://vitiy.info/immutable-data-and-serialisation-in-cpp11/

DATA PROCESSING

- Filter, map, reduce - simple blocks to build functional data processing
- Factorisation! Gives ability to extract iterations into **utility functors**
- Using C++11 it’s easy to implement such functional utils
- More compact code
- Easy to write
- Easy to read

CHANGE THE WAY OF THINKING – PART 1

- Let’s say we want to compute y=f(a,b)



- C - accumulator of parts - f,a,b, which can come in any order

CHANGE THE WAY OF THINKING – PART 2

- Step towards simplicity - Segregation:

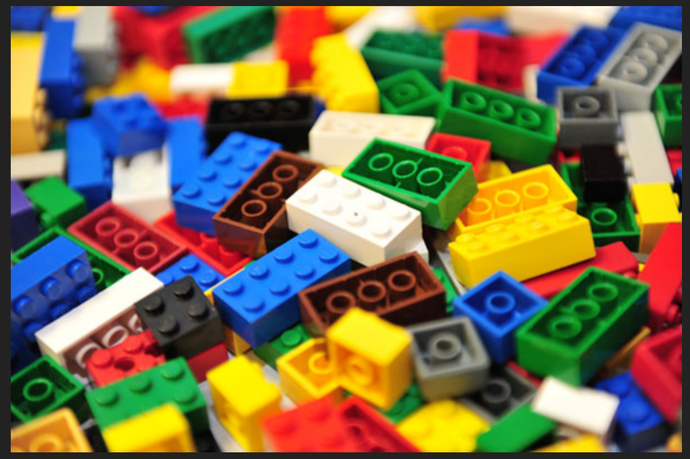
DATA

ITERATION

ACTION

CHANGE THE WAY OF THINKING – PART 3

- Any combination of parts can be stored as variable, passed to function and reused
- Functional composition - You can build long data processing chains for your business data
- AIM: Build constructor of functional blocks as way towards simplicity



STEP 1: FEED TUPLE TO FUNCTION

- We can push function arguments into tuple and pass it to function as a bunch

```
auto f = [](int x, int y, int z) { return x + y - z; };
auto params = make_tuple(1,2,3);
auto res = fn_tuple_apply(f, params);
print(res);
// output: 0
```

PIPING

- We can introduce some OPTIONAL sugar
- Simple: simular to unix pipeline
- You could find another way

```
vector<int> numbers{4,8,15,16,23,42};
auto result = numbers | where([](int x){ return (x > 10); }) | map([](int x){ return x + 5; }) | log();
// List: 20 21 28 47
```

Details - http://vitiy.info/functional-pipeline-in-c11/

CURRYING

- One more term from functional programming
- f(a,b,c) = f(a)(b)(c)
- *std::bind* - not so ‘simple’
- I want it more simple way -> right and left curry
- Actually i want special operator for it… (optional)

Implementation details - http://vitiy.info/templates-as-first-class-citizens-in-cpp11/

EXAMPLE OF CURRYING: >> & <<

- [SYNTAX IS OPTIONAL !] Exact operator choice is up to You

```
// currying....
auto f = [](int x, int y, int z) { return x + y - z; };
auto uf = fn_to_universal(f);
auto uf1 = uf << 1;
auto uf2 = uf1 << 2 << 5;
uf2() | print;
// result: -2

// Piping:

1 | (uf << 4 << 6) | print; // 4+6-1 = 9

3 | (uf >> 6 >> 7) | print; // 3+6-7 = 2
```

MAP/REDUCE – SIMPLE!

```
// MAP
template <typename T, typename... TArgs, template <typename...>class C, typename F>
auto fn_map(const C<T,TArgs...>& container, const F& f)        {
    using resultType = decltype(f(std::declval<T>()));
    C<resultType> result;
    for (const auto& item : container)
        result.push_back(f(item));
    return result;
}

// REDUCE (FOLD)
template <typename TResult, typename T, typename... TArgs,
          template <typename...>class C, typename F>
TResult fn_reduce(const C<T,TArgs...>& container, const TResult& startValue, const F& f)
{
    TResult result = startValue;
    for (const auto& item : container)
        result = f(result, item);
    return result;
}
```

FILTER

- Also simple to write:

```
// FILTER
template <typename T, typename... TArgs, template <typename...>class C, typename F>
C<T,TArgs...> fn_filter(const C<T,TArgs...>& container, const F& f)
{
    C<T,TArgs...> result;
    for (const auto& item : container)
        if (f(item))
            result.push_back(item);
    return result;
}
```

EXAMPLE

```
vector<string> slist = {"one", "two", "three"};

// all strings as one
slist | (reduce >> string("") >> sum) | (print << "All: ");
// All: onetwothree

// sum of elements of array
vector<int>{1,2,3} | (reduce >> 0 >> sum) | (print << "Sum: ");
// Sum: 6

// count sum length of all strings in the list
slist | (fmap >> fcount) | (reduce >> 0 >> sum) | (print << "Total: " >> " chars");
// Total: 11 chars
```

TEMPLATED FUNCTIONS AS FIRST CLASS CITIZENS

- Feel the power of variadic templates here

```
template <typename... Args>
void fprint(Args... args)
{
    (void)(int[]){((cout << args), 0)...}; cout << endl;
}

fn_make_universal(print, fprint);

template <typename T, typename... Args>
T sum_impl(T arg, Args... args)
{
    T result = arg;
    [&result](...){}((result += args, 0)...);
    return result;
}

fn_make_universal(sum, sum_impl);
```

SIMPLE CUSTOM BLOCKS

- we could build a lot of custom functional blocks - *limit, count, contains, find, sort, each2, reverse,* etc…
- avoid a lot of code duplication
- isolation of iteration code from actual action
- prevents mistakes at iteration’s part
- it’s possible to build blocks which are specific for your business

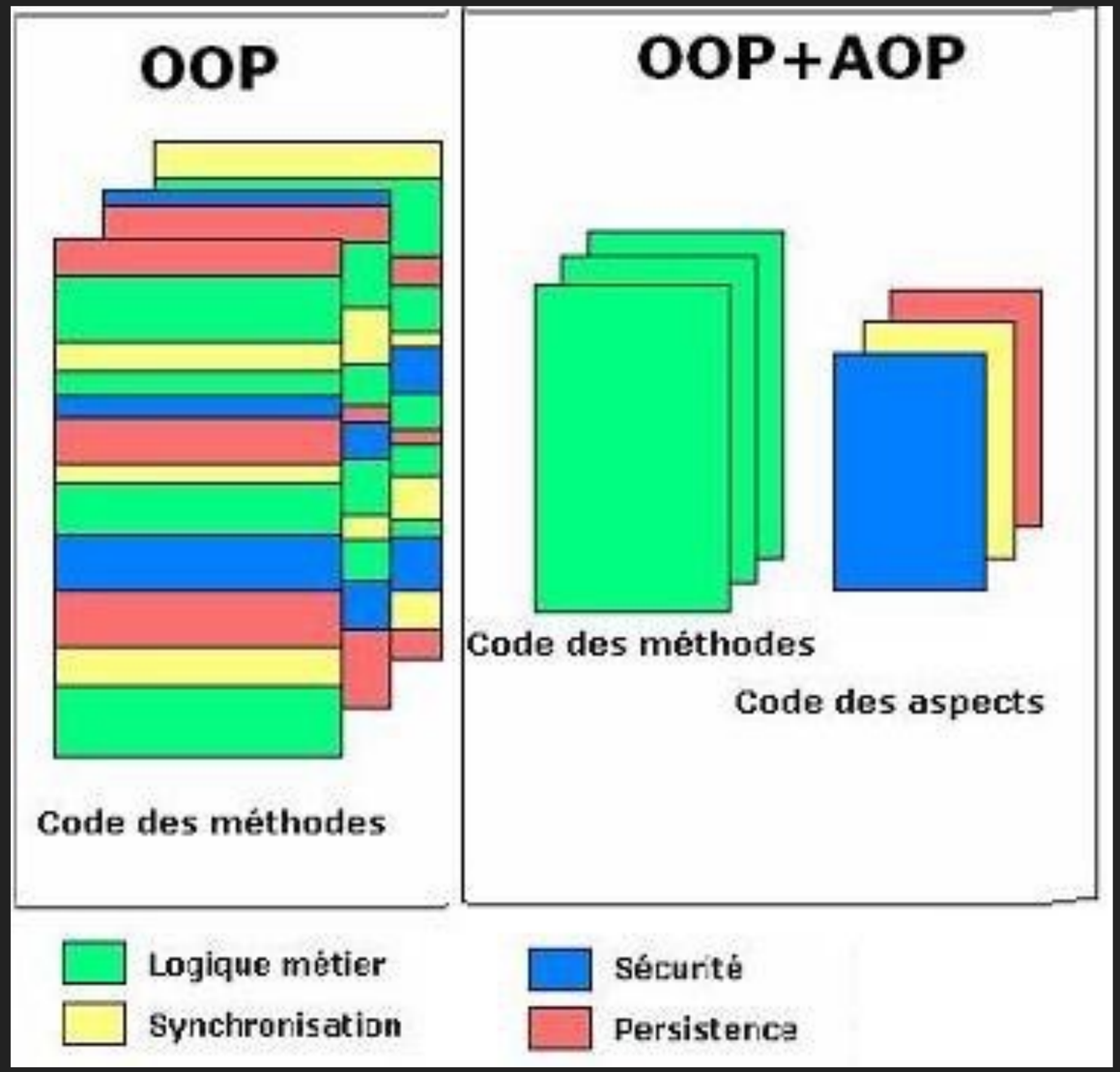
HIGH-ORDER FUNCTIONS IN APP STRUCTURE

$$ComplexF() = f1(f2(f3(f4)))()$$

- Functional factorization
- Each step is more transparent and independent
- Reuse of steps -> Avoid a lot of boilerplate code

ASPECT ORIENTED PROGRAMMING

- Separate your logical parts inside different functions
- Extract Security, Synchronisation, Persistence, Logging, etc
- Main logic core becomes way more minimalistic



AOP: LOGGING

- Some trivial example - logging:

```
auto logAspect = [](auto f){
    return [=](auto&&... args){
        LOG << "start" << NL;
        f(std::forward<decltype(args)>(args)...);
        LOG << "finish" << NL;
    };
};

auto plus = [](int a, int b) { LOG << a + b << NL; };
auto loggedPlus = logAspect(plus);
loggedPlus(2,2);
```

AOP: EXAMPLE

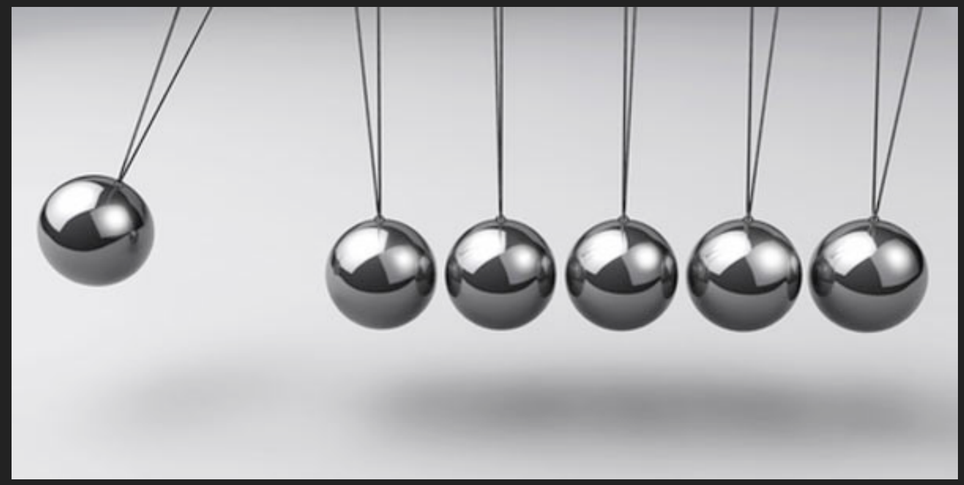
- Possible usage

```
auto findUserFinal = secured(session, notEmpty( cached(userCache,
triesTwice( logged("findUser",findUser)))));

auto user = findUserFinal(2);
LOG << (user.hasError() ? user.getError()->message : user()->name) << NL;

// output:
// 2015-02-02 18:11:52.025 [83151:10571630] findUser start
// 2015-02-02 18:11:52.025 [83151:10571630] Elapsed: 0us
// 2015-02-02 18:11:52.025 [83151:10571630] Bob
```

C++11 implementation - http://vitiy.info/c11-functional-decomposition-easy-way-to-do-aop/

FUNCTIONAL CHAINS & PIPING

- Different functional composition: Combine functions into chains



```
// Build functional chain:
auto f1 = [](int x){ return x+3; };
auto f2 = [](int x){ return x*2; };
auto f3 = [](int x) { return (double)x / 2.0; };
auto f4 = [](double x) { return SS::toString(x); };
auto f5 = [](string s) { return "Result: " + s; };
auto testChain = fn_chain<>() | f1 | f2 | f3 | f4 | f5;

// execution:
testChain(3) | print;
```

TRANSDUCERS

- New term from closure world (by Rich Hickey)
- Video from CppCon2015 - https://www.youtube.com/watch?v=vohGJjGxtJQ
- What the idea? Simple! Just composition of reducing functions.
- Analogy - check bag, wrap bag, get weight, pass bag - and then next step

TRANSDUCER USAGE

- Usage example

```
vector<int> input{1,2,3,4,5,6,7};
auto comp = tr | tfilter([](int x){ return (x % 2 == 0); }) | tmap([](int x){ return x+1; });
auto result = into(vector<int>(), comp, output_rf, input);
// 3 5 7


auto result = input | into_vector << (tr | tfilter([](int x){ return (x % 2 == 0); }) |
tmap([](auto x){ return x+1; }));
// 3 5 7
```

MORE TRANDUCER USAGE

```
// multiple inputs
vector<int> input1{1,2,3,4};
vector<int> input2{6,7,8,9};
auto result = into_vector(tr | tmap([](int x, int y){ return x + y; }), input1, input2);
//  7 9 11 13

// any range as source
auto result = into_vector(tr | tfilter([](int x){ return (x % 5 == 0);}), ints(100));
// 0 5 10 15 20 25 30 35 40 45 50 55 60 65 70 75 80 85 90 95
```

- Early termination of main loop:

```
vector<int>{1,2,3,4,5,6,7} | into << nullptr << (tr | tfilter(is_even) | tenumerate() |
tlimit(2)) << tr_each([](int n, int x){ LOG << n << ":" << x << NL; }) ;
// 0:2
// 1:4
```

MORE TRANSDUCERS USAGE

```
LOG << "Count: " << into(0, tr | tfilter(is_even), tr_count, ints(2000)) << NL;
// Count: 1000
```

- Resumable process! Sequences (gui events, network events, etc)

```
auto enumerateStrings = (tr | tenumerate() | tmap([](int n, string s){ return s + " " +
SS::toString(n); }))(output_rf);
auto result = fn_tr_reduce(vector<string>(), enumerateStrings, vector<string>{"a","b","c"});
// a 0 b 1 c 2

fn_tr_reduce_more(result, enumerateStrings, vector<string>{"e","d"});
// a 0 b 1 c 2 e 3 d 4
```

TRANSDUCER IMPLEMENTATION

- Not so obvious inside

```
auto tr_map = [] (auto fn) {
    return [=] (auto step) {
        return [=] (auto& out, auto&& ...ins) {
            return step(out, fn(std::forward<decltype(ins)>(ins)...));
        };
    };
};
```

This even will not compile under VS2015 (for now I hope) - produces internal compiler error, so my final implementation is not so pretty

BUT: THIS SHOULD BE ON LOW UTILITY LAYER...

LAYERED ARCHITECTURE

- Separate layers
- Messaging, CQRS, etc
- Everything at it's place

REAL-LIFE APPLICATION EXAMPLE

https://iqoption.com/land/iqoption4-features/en/

C++14 CROSS-PLATFORM ENGINE

- All ideas are cross-platform as it should be
- OSX, Windows, IOS, Android, Web (Emscripten cross compilation into JS)
- One code base for all platforms
- GPU, Open GL ES 2.0, application logic is 100% C++ code

Minor details - http://vitiy.info/small-presentation-of-my-cross-platform-engine-for-mobile-and-desktop-applications/

ALWAYS ASK YOURSELF –
HOW TO DO IT MORE SIMPLE WAY?

VICTOR LASKIN

- You can find a lot of examples in my blog - http://vitiy.info
- New updates are published through twitter - https://twitter.com/VictorLaskin
- Linkedin - http://linkedin.com/in/victorlaskin

THANKS