Lambdas – *uses and abuses*

Dawid Zalewski

github.com/zaldawid
zaldawid@gmail.com
@zaldawid

15-Nov-20

```cpp
auto forty_two = 42;

std::vector functions = {
                          [forty_two](){ return forty_two; },
                          [](auto denominator){ return 1.0 / denominator; },
                          [](auto a, auto b){ return a + b; }
                         };
```

```
error: class template argument deduction failed
```

```cpp
auto forty_two = 42;

std::vector functions = {
                           [](int a, int b){ return a + b; },
                           [](int a, int b){ return a - b; },
                           [](int a, int b){ return a * b; },
                         };
```

```
error: class template argument deduction failed
```

```cpp
auto forty_two = 42;

container_t functions = {
                            [forty_two](){ return forty_two; },
                            [](auto denominator){ return 1.0 / denominator; },
                            [](auto a, auto b){ return a + b; }
                          };

auto answer      = functions.call();
auto reciprocal  = functions.call(42);
auto sum         = functions.call(4, 2);
```

Lambdas' anatomy

lambda introducer (capture list) — *lambda declarator (params & specifiers)* — *compound statement (lambda body)*

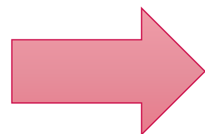
```
[cnt] <typename T> (T a, T b) mutable { while (cnt--) a+=b; return a; }
```

template params (c++20 only) — *lambda params* — *specifiers*

Closures

Lambda expression

```
auto lmb = [](int x, int y) {
  return x + y;
};
```



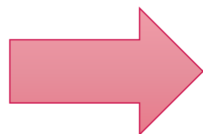
Closure type

```
class lmb_t{
public:

  inline constexpr auto
  operator()(int x, int y) const {
    return x + y;
  }

  lmb_t() = default;
};

auto lmb = lmb_t();
```

Closures with function templates

Lambda expression

```
auto lmb = [](auto x, auto y) {
  return x + y;
};
```



Closure type

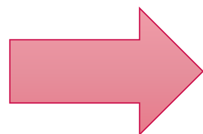
'Invented' types

```
class lmb_t{
public:
  template <typename T1, typename T2>
  inline constexpr auto
  operator()(T1 x, T2 y) const {
    return x + y;
  }

  lmb_t() = default;
};

auto lmb = lmb_t();
```

Closures with concepts

Lambda expression

```
auto lmb =
[] <std::integral T> (T x, T y) {
  return x + y;
};
```



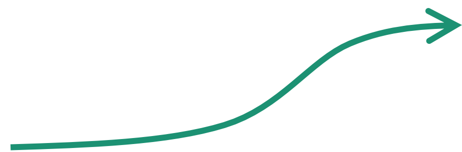
Closure type

```
class lmb_t{
public:
  template <std::integral T>
  inline constexpr auto
  operator()(T x, T y) const {
    return x + y;
  }

  lmb_t() = default;
};

auto lmb = lmb_t();
```

Closures: member variables

Lambda expression

```
void func(){
  auto k = 42;

  auto lmb1 = [=] () {
    return k += 11;
  };
  return lmb1();
}
```



Closure type

```
class lmb1_t{
public:
  int operator()() const {
    return k += 11;
  }
private:
  int k;
}

auto lmb1 = lmb1_t(k);
```

```
error: assignment of read-only variable 'k'
```

Closures: member variables

Lambda expression

```
void func(){
  auto k = 42;

  auto lmb2 = [=] () mutable {
    return k += 11;
  };
  return lmb2();
}
```

Closure type

```
class lmb2_t{
public:
  int operator()() {
    return k += 11;
  }
private:
  int k;
};

auto lmb2 = lmb2(k);
```

Lambdas before C++20

- Limited generic types (no `template <typename…>`) [p0428]
- Lambdas are not default-constructible [p0624]
- Lambdas cannot appear in unevaluated context [p0315]
- No pack expansion in init capture [p0780]
- No capturing of structured bindings [p1091]
- Weirdness around captures in member functions [p0806, p0409]
- No self-referencing (recursive) lambdas [p0839]

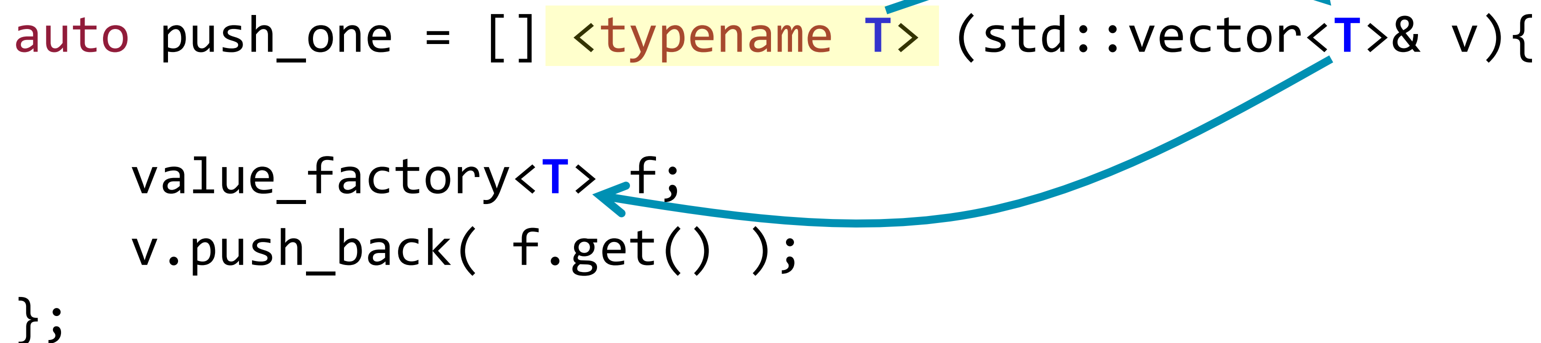
Lambdas in C++20: templates

Generic lambdas with implicit invented types are no fun:

```
std::vector<double> v;
push_one(v);

auto push_one = [](auto& v){
    using T = typename std::remove_reference_t<decltype(v)>::value_type;
    value_factory<T> f;
    v.push_back( f.get() );
};
```

Lambdas in C++20: templates

Explicit templates remove the boilerplate code:



Lambdas in C++20: templates++

```
#include <concepts>

auto sum = [] <std::integral T> (T a, T b) { return a + b; };

auto sum = [] (T a, T b) requires std::integral<T> { return a + b; };

sum(21.0, 21.0);
```

```
error: (…) candidate template ignored: constraints not satisfied
           with T = double] because 'double' does not satisfy 'Integral'
```

Lambdas in C++20: unevaluated & def-constructible

```
struct Process { int priority; };

using Container = std::vector<Process>;

using ProcessOrdering =

  decltype( [](auto&& lhs, auto&& rhs){ return lhs.priority > rhs.priority; } );
```

Only for capture-less lambdas!

```
std::priority_queue< Process, Container, ProcessOrdering > queue;
…
auto ordering = ProcessOrdering();
```

Lambdas in C++20: init capture pack expansion

```
template<class F, class... Args>
auto make_task(F&& f, Args&&... args) {
  return [f = std::forward<F>(f), args...]() mutable {
    return std::forward<F>(f)(std::forward<Args>(args)...);
  };
}

auto f = [](const auto& ... s) {((std::cout << s) ,...);};

auto task = make_task(f, std::string("bob"));

task();
```

unnecessary copy

task closure

Lambdas in C++20: init capture pack expansion

```
template<class F, class... Args>
auto make_task(F&& f, Args&&... args) {
  return [f = std::forward<F>(f), ...args=std::forward<Args>(args)]() mutable {
    return std::forward<F>(f)(std::forward<Args>(args)...);
  };
}

auto f = [](const auto&& ... s) {((std::cout << s) ,...);};

auto task = make_task(f, std::string("bob"));
```

task closure

Init-captures with pack expansions help avoiding copies.

Lambdas in C++20: recursive lambdas

Let's build a recursive lambda:

```
auto sum = [](int n) { return n == 0? 0 : n + sum(n-1); };

>> error: use of 'sum' before deduction of 'auto'


auto sum = [](int n) { return n == 0? 0 : n + operator()(n-1); };

>> error: use of undeclared 'operator()'
```

Lambdas in C++20: recursive lambdas

Function pointers to the rescue?

```
int (*sum)(int) = [](int n) { return n == 0? 0 : n + sum(n-1); };

>> error: 'sum' is not captured

int (*sum)(int) = [&](int n) { return n == 0? 0 : n + sum(n-1); };

>> error: cannot convert '<lambda>' to 'int (*)(int)' in initialization


std::function<int(int)> sum = [&](int n) { return n == 0? 0 : n + sum(n-1); };

>> but std::function, really?
```

Lambdas applied: recursive lambdas (1)

Add another lever of indirection:

```
auto sum = [](auto n){

    auto sum_impl = [](auto&& self, auto n){
        if (n == 0) return 0;
        return n + self(self, n - 1);
    };

    return sum_impl(sum_impl, n);

};

sum(42); //903
```

Lambdas applied: recursive lambdas (2)

Use the magic:

```
auto sum_ = [](auto&& sum, auto n) -> int { return n == 0? 0 : sum(n-1) + n; };

auto sum = magic_something{std::move(sum_)};

auto value = sum(42);
```

Lambdas applied: recursive lambdas (3)

Use the magic: a higher-order function (Y-combinator):

```
template <typename F>
struct recurse {
  F func;
  template <typename... Args>
  decltype(auto) operator()(Args&&... args) const {
    return func(*this, std::forward<Args>(args)...);
  }
};

auto sum_ = [](auto&& sum, auto n) -> int { return n == 0? 0 : sum(n-1) + n; };

auto sum = recurse{std::move(sum_)};
```

Calls recurse::operator()

Calls the lambda

Aggregate initialization

Lambdas applied: recursive lambdas (3)

Use the magic: a higher-order function (Y-combinator):

```
template <typename F>
struct recurse {
  F func;
  template <typename... Args>
  decltype(auto) operator()(Args&&... args) const {
    return func(*this, std::forward<Args>(args)...);
  }
};

template <typename F>
recurse(F) -> recurse<F>;
```

Recursive lambdas: better composition

Composing through inheritance

```
template <typename F>
struct recurse : F {
  F func;
  template <typename... Args>
  decltype(auto) operator()(Args&&... args) const {
    return F::operator()(*this, std::forward<Args>(args)...);
  }
};

template <typename F>
recurse(F) -> recurse<F>;
auto sum = recurse{std::move(sum_)};
```

Inheriting from a lambda

Aggregate initialization still works!*

*Each direct public base (F) is copy-initialized from the corresponding lambda in the list.

Lambdas applied: initializers

Default function arguments

```
void print_number(int number =
            [](auto n){ auto sum = n; while(n--) sum += n; return sum;}(42) ){
  std::cout << number;
}
```

Lambdas applied: initializers

Complex initialization using a function

```
auto x = 42;

auto x = log_init("init x with: ", 42);

template <typename Arg>
decltype(auto) log_init(std::string_view msg, Arg&& arg){
  std::cout << msg << arg;
  return std::forward<Arg>(arg);
}
```

Lambdas applied: initializers

Re-usable & composable initialization: inheritance?

```
std::ofstream flog{"log.out"};

log_init_t log_init{
  [fout=std::move(flog)](auto msg, auto&& value) mutable { fout << msg << value; },
  [](auto msg, auto&& value){ std::cout << msg << value; }
};

auto x = 42;

auto x = log_init("init x with: ", 42);
```

Lambdas applied: initializers

Composing through multiple inheritance

```
template <typename...Fs>
struct log_init_t : Fs...{
  template <typename Arg>
  decltype(auto) operator()(std::string_view msg, Arg&& arg){
    (Fs::operator()(msg, arg),...);
    return std::forward<Arg>(arg);
  }
};

Fs... -> {Fs1, Fs2, Fs3}
Fs1::operator()(msg, arg), Fs2::operator()(msg, arg), Fs3::operator()(msg, arg);
```

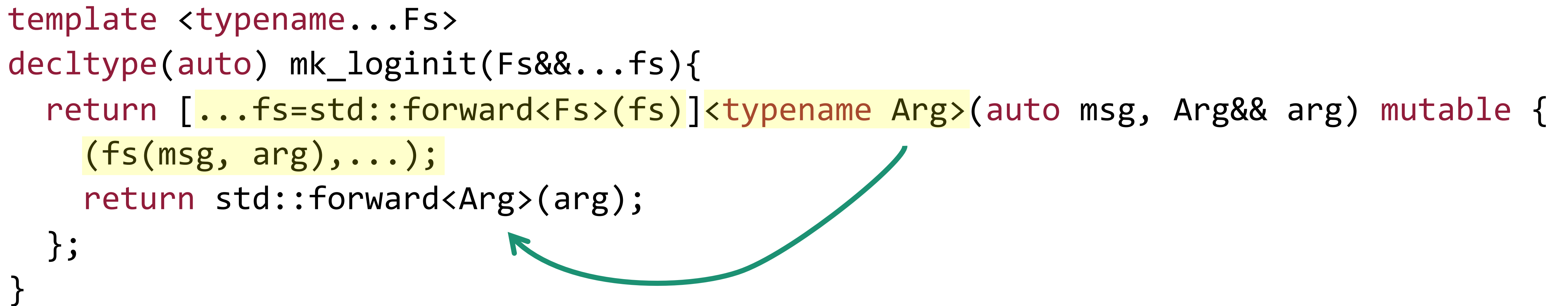
Lambdas applied: initializers

Composing through multiple inheritance

```
template <typename...Fs>
struct log_init_t : Fs...{
  template <typename Arg>
  decltype(auto) operator()(std::string_view msg, Arg&& arg){
    (Fs::operator()(msg, arg),...);
    return std::forward<Arg>(arg);
  }
};

template <typename...Fs>
log_init_t(Fs...) -> log_init_t<Fs...>;
```

Lambdas applied: initializers

C++20 lambdas can do the same:




```
auto log_init =  mk_loginit(
  [fout=std::move(flog)](auto msg, auto&& value) mutable { fout << msg << value; },
  [](auto msg, auto&& value){ std::cout << msg << value; }
);
```

Lambdas applied: wrappers

When just lambdas aren't enough:

```
auto logger = [](const auto& msg) { std::cout << msg; };

logger << "alice" << 42;


logger{[](const auto& msg) { std::cout << msg; }} << "alice" << 42;
```

Lambdas applied: wrappers

Lambdas used as building blocks for a composition / adapter

```
std::ofstream fout{"out.txt"};

logger{
  [](const auto& msg){ std::cout << msg; }
  [fout=std::move(fout)](const auto& msg) mutable { fout << msg; },
} << "alice" << 42;
```

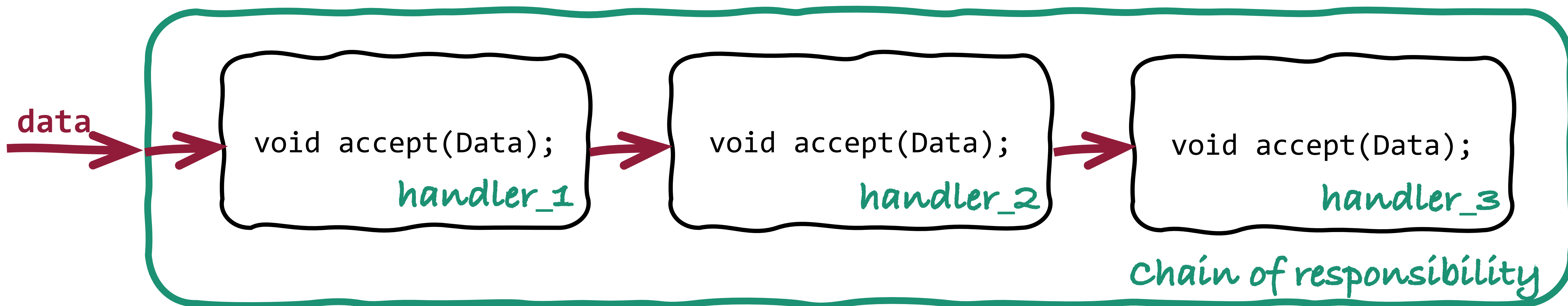
Lamdas applied: multiple inheritance

Lambdas used as building blocks for an adapter

```
logger{/*lambdas*/} << "alice" << 42;

template <typename...Fs>
struct logger: Fs...{

  template <typename T>
  logger& operator<<(const T& arg){
    (Fs::operator()(arg),...);
    return *this;
  }
};

template <typename...Fs> logger(Fs...) -> logger<Fs...>;
```

Chain of responsibility

But this look like the *chain of responsibility* design pattern!



Chain of responsibility: multiple loggers

A typical chain of responsibility

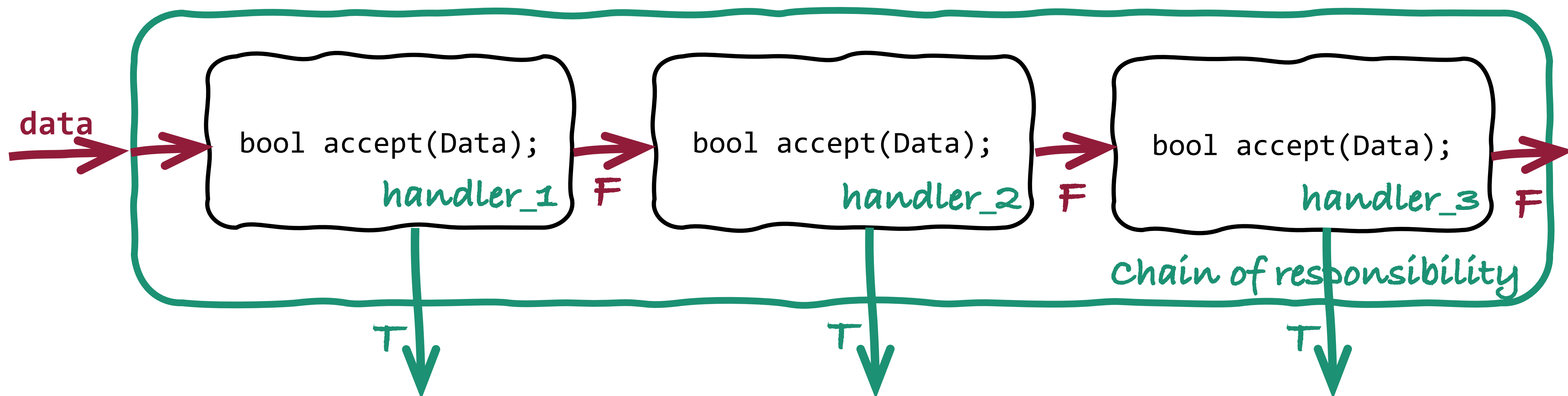
```
enum severity : int { info = 0, warn, error };

auto info_logger =
        [](severity s, auto msg){ if (s >= info) { /*handle logging*/ }};
auto warn_logger =
        [](severity s, auto msg){ if (s >= warn) { /*handle logging*/ }};

auto rc = resp_chain{info_logger, warn_logger};
…
rc(info, "System started");                // only info_logger should log
rc(warning, "Voltage too low (4.2)");  // only warn_logger should log
```

Chain of responsibility: multiple loggers

A typical chain of responsibility

```
template<typename...Fs>
struct resp_chain : Fs... {

  template<typename...Args>
  void operator()(const Args&...args) {
    ((Fs::operator()(args...)), ...);
  }

};
template<typename...Fs>
resp_chain(Fs...) -> resp_chain<Fs...>;
```

Chain of responsibility: the canonical version



Chain of responsibility: the canonical version

Forwarding only if not handled

```
auto info_logger = [](severity s, auto msg){
    if (s == info) { /*handle logging*/ return true; } return false; };

auto warn_logger = [](severity s, auto msg){
    if (s == warn) { /*handle logging*/ return true; } return false; };

auto rc = resp_chain{info_logger, warn_logger};
…
rc(info, "System started");
rc(warning, "Voltage too low (4.2)");
```

Chain of responsibility: the canonical version

Folding lambda parameter pack over binary or.

```
template<typename...Fs>
struct resp_chain : Fs... {

  template<typename...Args>
  void operator()(const Args&...args) {
    ((Fs::operator()(args...)) || ...);
  }

};
```

unary right fold of Fs... over ||

We could return the result here

Chain of responsibility: the canonical version

Folding lambda parameter pack over binary or.

```
template<typename...Fs>
struct resp_chain : Fs... {

  template<typename...Args>
  decltype(auto) operator()(const Args&...args) {
    return ((Fs::operator()(args...)) || ...);
  }

};
template<typename...Fs>
resp_chain(Fs...) -> resp_chain<Fs...>;
```

unary right fold of Fs... over ||

Chain of responsibility with optionals

Returning `std::optional` instead of `bool` complicates things...

```
struct log_entry{};

auto info_logger = [](severity s, auto msg) -> std::optional<log_entry> {
    if (s == info) { /*handle logging*/ return std::optional(log_entry{…}); }
                                        return std::nullopt; };
auto warn_logger = [](severity s, auto msg) -> std::optional<log_entry> {
    if (s == warn) { /*handle logging*/ return std::optional(log_entry{…}); }
                                           return std::nullopt; };

auto log_1 = rc(info, "System started");
auto log_2 = rc(error, "Low memory");
```

Chain of responsibility with optionals

Returning `std::optional` instead of `bool` complicates things...

```
template<typename F, typename ... Fs>
struct resp_chain : F, Fs... {

  template<typename...Args>
  decltype(auto) operator()(const Args&...args) {
    auto res{F::operator()(args...)};
    ((res) || (( res = std::move(Fs::operator()(args...))) || ...));
    return res;
  }

};
```

Fizz-buzz

All of the sudden fizz-buzz becomes a child's play

```
auto fizz_buzz = resp_chain{
  [](auto n) { return n % 15 == 0 ? std::optional("FizzBuzz") : std::nullopt; },
  [](auto n) { return n % 3  == 0 ? std::optional("Fizz") : std::nullopt; },
  [](auto n) { return n % 5  == 0 ? std::optional("Buzz") : std::nullopt; },
  [](auto n) { return std::optional(itoa(n)); }
 };

auto number = 0;
while (number++ != 100) {
  std::cout << number << ": " << *fizz_buzz(number) << "\n";
}
```

An ugly case…

Global state variable and side effects

```
auto _handled{false};

void set_handled();
bool is_handled();

auto info_logger = [](severity s, auto msg) {
      if (s == info) { /*handle logging*/ set_handled(); } };

auto warn_logger = [](severity s, auto msg){
     if (s == warn) { /*handle logging*/ set_handled(); } };
```

Global state

Side effects

An ugly case: inheriting from (deeper) self

Recursion + primary class template

```
template<typename F, typename ... Fs>
struct resp_chain : F, Fs... {

  template <typename SF, typename...Args>
  void operator()(SF& sf, const Args&...args){
    if (F::operator()(args...); !sf()){
      resp_chain<Fs...>::operator()(sf, args...);
    }
  }
};
```

```
bool is_handled();
```

Recursive call to the rest of the chain

An ugly case: inheriting from (deeper) self

Specialization for the final case

```
template<typename F>
struct resp_chain<F> : F {

  template <typename SF, typename...Args>
  void operator()(SF&, const Args&...args) {
    F::operator()(args...);
  }
};

template<typename... Fs>
resp_chain(Fs...) -> resp_chain<Fs...>;
```

All the inheritance on one slide

```
template<typename F>
struct lambdas : F {};
```

Simple cases

```
template<typename...Fs>
struct lambdas : Fs... {};
```

Multiple lambda's
Pack expansion
Fold expressions

```
template<typename F, typename...Fs>
struct lambdas : F, Fs... {};
```

For picking the result of the first lambda

```
template<typename F, typename...Fs>
struct lambdas : F, lambdas<Fs...> {};
```

Recursive calls

But there is more!

Adding new handlers to an existing chain

```
enum severity { info, warning, error };

auto info_logger = [](severity s, auto msg) { if (s == info)    ...};
auto warn_logger = [](severity s, auto msg) { if (s == warn) ...};

auto rc = resp_chain{info_logger, warn_logger};

rc(error, "Low memory");
```

But there is more!

Adding new handlers to an existing chain

```
auto rc = resp_chain{info_logger, warn_logger};
rc_with_error(error, "Low memory"); // won't be handled

auto err_logger = [](severity s, auto msg) -> std::optional<log_entry> {
     if (s == error){ /*handle logging*/ return std::optional(log_entry{…}); }
     return std::nullopt; };

auto rc_with_err = rc.set_next(std::move(err_logger);

rc_with_error(error, "Low memory");
```

Handled by err_logger

But there is more: programming with types

```
template<typename F, typename ... Fs>
struct resp_chain : F, Fs ... {

  template<typename...Args>
  decltype(auto) operator()(Args&& ...args) { ... }

  template<typename G>
  decltype(auto) set_next(G&& g) {
    using G_ = std::remove_cvref_t<G>;
    return resp_chain<F, Fs..., G_>{*this,
                                     static_cast<Fs&>(*this)...,
                                     std::forward<G>(g)};

  }
};
```

Copy-initialize base classes

set_next returns a new type!

Uses and abuses

- Folds + inheritance = composable lambdas
- Clean, reusable abstractions
- Strong types for safer code
- Compile-time checks

Lambdas,
uses & abuses

Dawid Zalewski
github.com/zaldawid
zaldawid@gmail.com
@zaldawid