Functional Programming In C++

Vitaly Fanaskov

Oslo, 2023

About me

- Vitaly Fanaskov
- Senior software engineer
- 10+ years of C++ experience
- GIS, VFX, frameworks, and libraries
- Ph.D (CS)

- reMarkable – digital notebook designed for tasks that demand focus
- https://careers.remarkable.com/jobs

Agenda

- Functional programming
- Composition and decomposition
- Abstractions available in C++
- Conclusions

Functional Programming

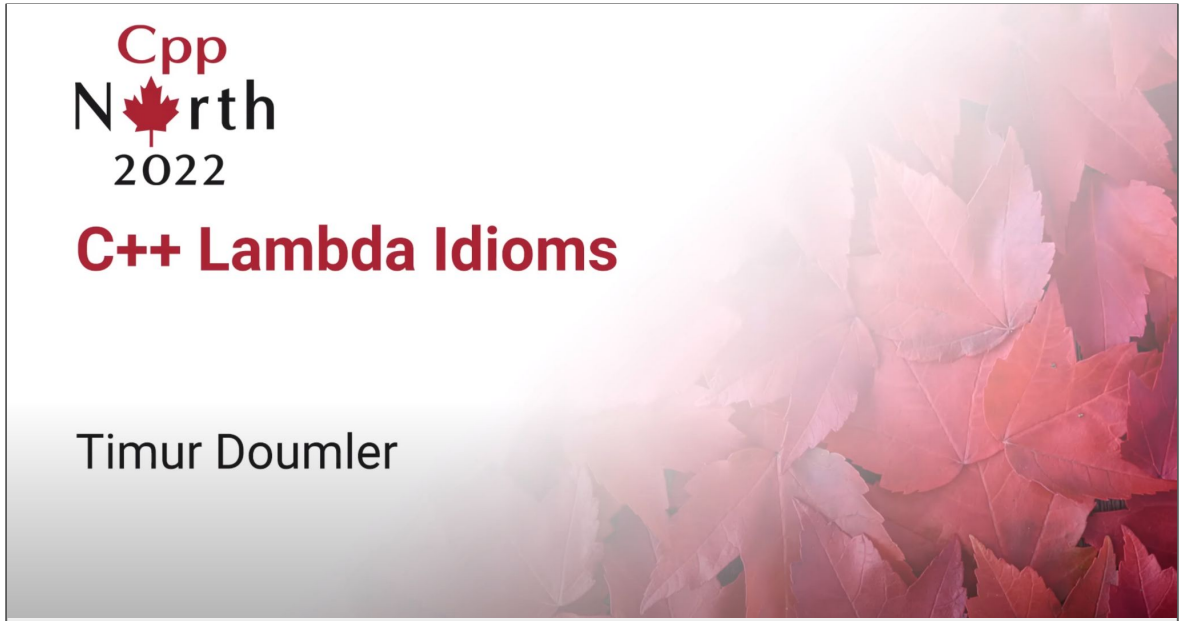
Main characteristics

- Applying and composing functions
- Functions as first-class citizens
- High-order functions, pure functions, and recursion
- Declarative and composable style
- High level of modularity

Abstractions in C++: functions

Lambda functions

```
const auto add = [] (int a, int b) { return a + b; };
```



https://www.youtube.com/watch?v=iWKewYYKPHk

std::function

```
const auto add_l = [] (int a, int b) { return a + b; };
int add_func (int a, int b) { return a + b; }

// ...

std::function<int(int, int)> add;
add = add_l;
add = add_func;
```

Function composition

f: string → int
g: int → bool

h = g ◦ f

```
auto f = [] (std::string_view) → int {};
auto g = [] (int) → bool {};

auto compose = [] (auto g, auto f) {
  return [=] (std::string_view v) → bool {
    return std::invoke(g, std::invoke(f, v));
  };
};

auto h = compose(g, f);
```

Abstractions in C++: “monadic” operations

std::optional<T>

- and_then
- transform
- or_else

```
std::optional<Json> createUser() { /* ... */ }

// ...

const auto userID = createUser()
  .transform(extractID)
  .or_else(defaultID)
  .value();
```

std::expected<T, E>

- and_then
- transform
- or_else
- transfrom_error (*returns the expected itself if it contains an expected value; otherwise, returns an expected containing the transformed unexpected value*)

```
std::expected<User, Error> createUser() { /* ... */ }

// ...

const auto userID = createUser()
  .transform(extractID)
  .or_else(defaultID)
  .value();
```

Abstractions in C++: utilities

C++ provides a lot of helpers

- std::invoke
- std::reference_wrapper
- Type traits (e.g., std::is_invocable)
- Concepts (e.g., std::invocable)
- Fold expressions

3rd-party libraries

There are quite a few useful libraries

- Processing ranges
- Reactive programming
- Async programming
- Strong types
- Monads

Should I give it a try?

Pros of using functional-programming

- Modularity
- Testability
- Lazy calculation
- Chain computation
- Value semantic

Thank you!