FUNCTIONAL PROGRAMMING IN C++

Nicola Gigante
Meeting C++ 2015
What will we talk about

Wasn’t C++ object oriented last time I checked?

  What’s wrong with OOP?

  But what is functional programming after all?

Elements of Functional Style

  Let’s Get High Tonight

  Enemy of the State

  Being lazy at work

  Types, types everywhere...

  Lawful abstractions

Back to the future
Wasn’t C++ object oriented last time I checked?
C++ supports object oriented programming very well:

- Abstract Classes
- Multiple Inheritance
- Polymorphism via virtual functions

So this is the main programming paradigm of the language, right?

Nope.
Let’s take a look at the C++ library: the Standard Library.

- A lot of classes
- Very few classes meant to be derived
- Very few virtual functions and pure virtual functions
- Mostly in I/O streams

So it seems the committee has forgotten how to object-orient the standard library.
So is (modern) C++ an object oriented language?

- Yes, if you mean that it supports object-oriented programming
- No, if you mean that it is the main paradigm of the language
But wait, what’s wrong with OOP?

Object Oriented Programming is simply not enough.

Why? Because it does not answer a lot of important questions

- How to better write and compose algorithms?
  OOP is nothing more than dear-old imperative programming in this regard

So we need something more ...

C++ programmers often already know it.
Object Oriented Programming is simply not enough.

Why? Because it does not answer a lot of important questions

- How to write high-level *and fast* code?
  OOP data structures are often slow because of too much indirection and type erasure

So we need something more ...

C++ programmers often already know it.
Object Oriented Programming is simply not enough. Why? Because it does not answer a lot of important questions:

- How to manage complexity in concurrent software? OOP does not simply say anything here

So we need something more ...

C++ programmers often already know it.
But what is functional programming after all?

Functional Programming has a few core principles:

- Functions are first-class citizens
- Immutable state
- Composable, generic, reusable, pure functions
- Make heavy use of the type system

Your mileage may vary.
We cannot say that C++ is a functional programming language.

So why bending C++ to program functionally?

• Some applications need to be very fast, yet handling a lot of complexity

• C++ is fast, and functional programming helps at handling complexity
We cannot say that C++ is a functional programming language. So why bending C++ to program functionally?

- Functional Programming is not very much about language features, as it is a way of thinking about programming
- C++ programmers can often benefit a lot from thinking (and coding) functionally
- Modern C++ has introduced a number of interesting functional features, by the way
Elements of Functional Style
One of the main abstraction tools in Functional Programming is the concept of *higher order functions*.

- Functions that take other functions as arguments and return other functions
- Functions in header `<algorithm>` are a good old example
- The introduction of lambdas in C++11 allows to officially say that this pattern is encouraged in C++
Lambdas are anonymous function objects.

```cpp
    // Sort by absolute value
    std::sort(begin(v), end(v), [](int i, int j){
      return std::abs(i) < std::abs(j);
    });
```
This was (roughly) equivalent to:

```cpp
class abs_cmp
{
public:
    bool operator()(int i, int j) const {
        return std::abs(i) < std::abs(j);
    }
};

std::sort(begin(v), end(v), abs_cmp{});

Only sane.
```
How to take a lambda as an argument?
Use a template parameter.

```cpp
template<
typename Iterator,
typename F>
void for_each(Iterator b, Iterator e, F f) {
  for(; b != e; ++b) {
    f(*b);  
  }
}
```
What’s great about C++ lambdas is that they do not introduce any runtime overhead:

- Each lambda is an instance of its own separate anonymous type:
  - The function body is put into the type’s `operator()` member function
  - Captured variables are put into data members
- Plays nicely with how templates work and how compilers do inlining of function calls
C++11 lambdas were great but they had a few limitations:

- They couldn’t have generic arguments (i.e. templated `operator()`)
- They couldn’t capture variables by move
- The anonymous type cannot be named, so lambdas could not be returned from a function!
C++14 changed the game regarding lambdas:

- Generic arguments
- Automatic return type deduction in regular functions
- Generalized capture lists
Each lambda object has its unique type.

So how to pass a lambda to a non-template function? How to store such a lambda for later use?

Enter `std::function`:

```cpp
std::function<int(int)> f = [](int x) {
    return x * 2;
};
```
Understanding `std::function`:

- It is a *polymorphic* wrapper around any *Callable* object
- Implemented through *type-erasure* techniques
- It adds overhead because of the indirect call and the lack of inlining
- The overhead is the same on every other language supporting lambdas, anyway
Higher-Order Combinators

Functional-style higher order combinators are possible in C++

• The key ingredients are here: lambdas, automatic type deduction, etc...

• C++ makes life more challenging: handling perfect forwarding, reduce copy/moves operations, etc...

• A lot of libraries allow us to have fun without too much headaches
  • Boost.Fusion, Boost.Hana, Fit, ...
As an example, we want to implement the classic function composition:

```cpp
auto f(std::vector<float>) -> float;
auto g(std::string) -> std::vector<float>;
auto h(std::istream&) -> std::string;

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

int x = fgh(std::cin); // x = f(g(h(std::cin)))
```
C++14 made it easy:

```cpp
template<typename F>
auto compose(F f) {
    return [=](auto x) { return f(x); };}
}

template<typename F, typename ...Fs>
auto compose(F f, Fs ...fs) {
    return [=](auto x) { return f(compose(fs...)(x)); };}
}
```
*Boost.Hana* is a cutting edge library for C++ metaprogramming:

- Wonderful tools to make C++ metaprogramming a funny activity again
- Blends compile-time and run-time computations
- Functional in nature (you cannot mutate types)
- Also provides a number of nice combinators
Currying is the best friend of higher-order programmers:

- A “curried” function can be invoked with only one argument even if it expects more
- The result is another function that still expects the missing arguments
Currying is provided by `hana::curry()`:

```cpp
auto f = curry([](int x, int y, int z) {
    return x * y + z;
});

auto s = f(4);
auto t = s(5);
auto w = t(6); // w = 26
```
Overloaded function objects:

```cpp
auto f = hana::overload(
    [](std::string s) { return s + s; },
    [](auto x) { return x * 2; }
);
std::vector<T> v = ...;
std::transform(begin(v), end(v), begin(v), f);
```

Useful in generic code to specialize actions based on type while staying generic.
Partial application of binary operators:

```cpp
using hana::_;

std::vector<int> v = {1,2,3};

std::transform(begin(v), end(v), begin(v), _ * 2);
```
The `on()` function and infix function application:

```cpp
std::sort(begin(v), end(v), [] (int i, int j) {
    return std::abs(i) < std::abs(j);
});
```
The `on()` function and infix function application:

```cpp
std::sort(begin(v), end(v), (_ < _) ^on^ &std::abs<T>);
```
The `on()` function and infix function application:

```cpp
class auto abs = [](auto x) { return std::abs(x); };
std::sort(begin(v), end(v), (_ < _) ^on^ abs);
```
Fix point combinator for recursive lambdas:

```cpp
auto fact =
    hana::fix([](auto fact, auto n) -> int {
        if (n == 0)
            return 1;
        else
            return n * fact(n - 1);
    });
```
Functional Programming is the realization that (explicitly) manipulating mutable (and shared) state is a bad idea.

Why?

- Reliability
- Maintainability
- Concurrency hell

How to avoid it?
Functional Programming is all about pure functions:

- The result of a pure function does not depend in any way on anything other than its input values
- Do not access nor share external state
- Do not *mutate* anything (even internally)

In C++ we can relax a little on the last point, but still...
With purity you avoid “spooky action at a distance”:  

- You separate computation from side effects (e.g. I/O) 
- Pure components are easier to compose 
- Pure components are easier to test 
- Pure components are inherently thread safe
What does it mean to be “composable”?

It risks to sound like a buzzword if not properly framed.
Composability means:

You have two “good” components.

Is their composition (very) likely to be “good”, too?

Substitute “good” for “fast”, “correct”, “thread-safe”, and so on...
Every functional language has a map function.

The C++ Standard Library equivalent is \texttt{std::transform}:

\begin{verbatim}
std::vector<int> v = {1, 2, 3, 4};
std::transform(begin(v), end(v), begin(v),
  [](int x) {
    return x * 2;
  });
\end{verbatim}

Simple, fast...

But what if I want to change the type of the elements?

What if you want to chain transformations?
Example: Standard Algorithms

```cpp
std::vector<int> v = {1, 2, 3, 4};
std::transform(begin(v), end(v), begin(v),
    &to_string<int>); // Nope...
```
std::vector<int> v = {1, 3, 4};
std::vector<std::string> outs;

std::transform(begin(v), end(v),
               std::back_inserter(outs),
               &to_string<int>);

What’s the problem?
Manual management of the resulting container is required:
  • Repetition of the return type of `to_string`
  • Explicit management of its size (`std::back_inserter`)
Can I really compose those transforms?

```cpp
std::vector<T1> input;
std::vector<T2> step_one;
std::vector<T3> step_two;
std::vector<T4> output;

std::transform(begin(input), end(input),
                std::back_inserter(step_one), &f);
std::transform(begin(step_one), end(step_one),
                std::back_inserter(step_two), &g);
std::transform(begin(step_two), end(step_two),
                std::back_inserter(output), &h);
```

No, it’s a mess. And slow!
So why should I prefer all this standard algorithm stuff over a simple tight loop?

```cpp
std::vector<T1> input;
std::vector<T4> output;

for (auto &x : input) {
    output.push_back(h(g(f(x))));
}
```

Maybe it’s the wrong use case? No, it’s the wrong function. The problem is that `std::transform` *mutates* its inputs.
Ranges are the solution to the STL standard algorithms composability problems.

There are a couple of ranges libraries available:

- *Boost.Ranges*, stable and very good
- *Ranges v3* by Eric Niebler, the cutting edge research project on ranges, base of the forthcoming Ranges Technical Specification.
How the same piece of code looks like with *Ranges v3*:

```cpp
type::vector<T1> input;

type::vector<T4> output =
    type::transform(
        type::transform(
            type::transform(input, f), g), h);
```
How the same piece of code looks like with Ranges v3:

```cpp
std::vector<T1> input;

std::vector<T4> output = input
    | view::transform(f)
    | view::transform(g)
    | view::transform(h);
```
How does `view::transform()` avoid the performance problems of `std::transform()`?

- Range views are *lazy*
- Each element of the result is computed only when needed
- Intermediary ranges are never completely built
- This results into the same tight loop of the imperative code
Lazyness is a flagship feature of functional programming languages:

- Pure code does not depend on the order of evaluation
- Most non-pure functional languages support lazyness, too (It is still better to limit side effects in lazy computations)
- C++ can implement ad-hoc lazyness in libraries quite well
Lazyness allows the creation of infinite sequences:

```cpp
auto random_numbers(int seed)
{
    std::uniform_int_distribution<> dist;
    std::mt19937 mt{seed};

    return view::generate([=](){ mutable {
        return dist(mt);
    }});
}
```
Lazyness allows the creation of infinite sequences:

```cpp
extern int interesting_work(int);

std::vector<int>
    results = random_numbers(42)
    | view::transform(interesting_work)
    | view::take(100);
```

Are these functions composable?

std::vector<int> veryEfficientSource(int);
int veryEfficientSink(std::vector<int>);

int notSoEfficientComposition(int x) {
  return veryEfficientSink(veryEfficientSource(x));
}

Composing the functions in the naïve way leads to suboptimal performance
Lazyness and Composability

SomeRangeTy veryEfficientSource(int);

template<typename Range>
int veryEfficientSink(Range);

int veryEfficientComposition(int x) {
    return veryEfficientSink(veryEfficientSource(x));
}

Lazyness allows the naïve composition to be fast!
Note that:

- Single functions can still be coded in isolation
- Single functions are still implemented in the most effective way for their single task

So, I have two fast components. Is their composition fast, too? Yes? Composability!
Purity and lazyness help not only the design of algorithms, but data structures as well

- Immutable data structures are often called *persistent*
- Update the data structure by creating a new one which shares unchanged parts of the old one
- Concurrency and lock-free programming are easier!
- They would merit a whole talk by themselves, we are only teasing them...
Functional programming gives its best with a strongly-typed programming language.

Advantages of a strong type system:

- Types help the code to self-document itself
- The compiler can help catching bugs before testing
- Refactoring is easier

Modern C++ programmers know these advantages very well.
Algebraic Data Types are the natural way of representing types in a functional setting.

Why “algebraic”? Because they are represented recursively by applying three operations over simpler basic types:

- $T_1 + T_2$: The type of values made of a $T_1$ or a $T_2$
- $T_1 \times T_2$: The type of values composed of a $T_1$ and a $T_2$
- $T_2^{T_1}$: The type of functions that accepts a $T_1$ and return a $T_2$
C++ does not have an ad-hoc syntax to declare ADTs, but many common types are used as ADTs in principle:

- `std::tuple` is a product type
- `boost::variant`, `std::optional<T>`, `expected<T,E>` are sum types
Inheritance hierarchies may seem to cover the same use cases of sum types.

They’re actually one the opposite of each other:

- Class hierarchies support a *fixed* set of operations over an *open* range of possibilities
- Sum types support an *open* set of operations over a *fixed* range of possibilities

*Hidden secret*: Each time you need to apply the Visitor pattern, you probably need a sum type
Languages supporting Abstract Data Types usually supports doing *Pattern Matching* on them.

- There is no universal way to simulate it in C++ yet, but...
- *boost::variant* offer something similar for that particular type
- Use *std::optional::value_or()* to simulate it on optional values
- The *Mach7* library from Bjarne Stroustrup goes very near to a generic and fast library-based pattern matching solution!
Best practices for error handling are an hotly debated topic. A good technique should provide:

- **Reliability**: the programmer should not be able to forget handling some error
- **Efficiency**: the code should not be slowed down by the error handling mechanism
- **Readability**: the error handling mechanism should not clutter the code too much
Exceptions are great, but they do have pros and cons:

- **Reliability**: there’s no way to prevent the programmer to forget to catch some exception type
- **Efficiency**: zero overhead on the non-exceptional path, but very slow on the throwing path
- **Readability**: exceptions are simple to throw, very intrusive catch, and *trivial to ignore*
So are exceptions bad? No!

They have to be used to signal *exceptional* events.

Sounds trivial enough. How about this interface:

```cpp
def parse(std::string); // throw on error```

Is a parsing error an *exceptional* or an *expected* result?

In the second case we need a different mechanism.
One of the simplest ADTs possible, `std::optional<T>`, solves the problem in this case:

```cpp
std::optional<int> parse(std::string);
```

**Reliability:**

- The programmer cannot implicitly ignore the handling of the error
- The type of the function clearly states that the return value may be missing
One of the simplest ADTs possible, `std::optional<T>`, solves the problem in this case:

```cpp
std::optional<int> parse(std::string);
```

**Efficiency:**

- There can be a small overhead on the non-error path when compared to zero-overhead exceptions
- The error path is faster
- More importantly, the two paths have *equal* performance
One of the simplest ADTs possible, `std::optional<T>`, solves the problem in this case:

```cpp
std::optional<int> parse(std::string);
```

**Readability:**

```cpp
auto result = parse("42");
if(!result) {
    // handle...
}

int x = *result;
```
A few libraries provide a similar sum type, `expected<T, E>`:

- Similar to `optional<T>`, but provides a clue on what caused the failure.
- Can be used in the same way.
- Heavily discussed proposals exist to include it in the next standard.
C++ exceptions and optional/expected cover similar use cases but are not direct competitors:

- Use exceptions where the failure is possible but rare or really unpredictable (example: I/O functions)
- Use optional values when the failure is a common case that has to be guaranteed to be dealt correctly
- Use optional values if you want a pure function: throwing an exception is very much a side effect
We may implement a `begin()`/`end()` pair of functions for `std::optional<T>` or `expected<T,E>`:

```cpp
auto opt = parse("42");
opt = view::transform(begin(opt), end(opt), _ * 2);
```

So optional values can be “transformed”/“mapped” like containers. Is this a coincidence?

Not at all.

What do containers and optional values have in common?
Containers and optional values are Functors.

From the documentation of Boost.Hana:

*Intuitively, a Functor is some kind of box that can hold generic data and map a function over this data to create a new, transformed box*

*Pedantic disclaimer:*

Every time a C++ programmer says *functor* instead of *function object*, a mathematician dies.
A **Functor** is a *lawful* abstraction

- The mere existence of a way to map the contained values is not enough
- The mapping function must also obey two *laws* for the type to be a real functor:

\[
\text{transform}(xs, \text{id}) == xs \\
\text{transform}(xs, \text{compose}(g, f)) == \\
\text{transform}(\text{transform}(xs, f), g)
\]
Abstractions like “Functor” help the programmer to reason about the code.

- We can be more confident of what we wrote
- We can recognize different and possibly faster ways to write the same thing
- We can abstract over things which are apparently very different at first

*Boost.Hana* and others (notably *Cat*), provide a nice concept-based infrastructure to use and reason about these abstractions.
Monads are just another lawful abstraction.

A Functor which can also chain computations:

```cpp
std::future<int> x = std::async(...);
std::future<int> y = x.then([](int x) {
  // When x is ready...
});

Unfortunately this is still pseudo-syntax...
```
Monads are just another lawful abstraction.

A Functor which can also chain computations:

```cpp
std::optional<int> x = parse(...);
std::optional<int> y = x.then([](int x) {
    // If x is not empty...
});
```

Unfortunately this is still pseudo-syntax...
Monads are everywhere:

- A lot of important C++ types are monads
  - Containers
  - `optional<T>, expected<T,E>`
  - Futures
  - `Coroutines`
- That means a lot of different tasks, done with different syntaxes every day, are actually the same thing
- If you are curious, follow “Awaiting for the Ranges” tomorrow
Back to the future
I’ve seen things you compilers wouldn’t parse...

C++17 is being shaped while we talk.

It will be *exciting* for functional programmers:

- Standardization of a ranges-based Standard Library
- Standardization of `optional<T>` and (hopefully) `expected<T,E>, variant<>`
- Coroutine `await` syntax will be (hopefully) general enough to support any monad
- Concepts-related changes in the language will make it easier to write generic higher-order functions
Questions?