Applying functional programming in code design

Michał Dominiak
Wrocław University of Technology
Nokia Networks
griwes@griwes.info

Outline

- Introduction; ideas vs techniques
- Purity of functions
- Functional data structures
- Avoiding shared state
- Composability and laziness
- Designing for "success"
- Writing generic code
- Noticing patterns
- Open Content session: functors and monads

Introduction

- Functional programming - an increasingly important declarative paradigm.
- As every paradigm, it comes with some techniques and ideas.
- Techniques are less important than ideas.
- Techniques are just implementations of the ideas.
- The ideas are applicable in most programming languages - especially in one as multi-paradigm as C++.
- We will focus on *ideas.*
- Code design - all the steps taken from the point of creating a top-level overview of the architecure up to implementation of specific interfaces.

What is this talk actually about?

- A set of "guidelines" for following some of the ideas of functional programming.
- An explanation of some possible choices when designing your systems.
- **NOT:** A clear set of "do"s and "do not"s.
- **NOT:** "Functional programming is the only true paradigm!"

Purity of functions

- Pure function - a function that does not cause side effects and that always returns the same value whenever called with the same set of (explicit) arguments.
- *It's the value that matters, not state (and identity only matters if it's a part of the value).*
- It's easier to reason about pure functions.
- Compiler can optimize them better, for example by replacing multiple identical calls with a single one.
- Pure functions are implicitly thread-safe.

Purity of functions, pt. 2

- N3744 from 2013 by Walter Brown.
- The proposed [[pure]] disallows exceptions to be thrown.
- Feature status in C++: [[pure]] - kind of, constexpr - kind of. GNU extension - __attribute__((pure)).

- Beware of floating point.

Pure functions and data structures

- Hard to write a pure function doing something useful (and mutating) with a data structure.
- Immutability helps.
- In C++: structs and classes with all members const - nothing short of that really helps (and this is still fragile to const_cast, even when it's undefined).
- const member functions aren't really there (mutable members).
- Now... how is such a type useful?
- Can't do some specific things with, for example, a vector of those, like call .erase().

Functional data structures

- In other words, immutable data structures.
- Make sense mostly in cases of node-based data structures *(I am not going to talk about performance in this talk).*
- Nodes can be shared between multiple instances.
- "Mutating" functions return new instances of the structure instead.
- By consequence, these functions become pure.
- Due to all of that, we gain the profits of being thread-safe implicitly again.

Avoiding shared (mutable) data

- Almost a summary of the previous two points.
- The idea: avoiding shared (mutable is implied on this slide) data to be thread-safe without the need to use other synchronization primitives.
- Global variables should be avoided for known reasons.
- Design your functions to be pure. Let your functions take inputs and generate outputs, instead of operating on some implicit inputs (for the purpose of this topic, the this pointer is considered an implicit input).
- Try to use functional data structures for the parts of data that needs to be shared.
- *Do not force this.* Sometimes sharing the data (*with proper synchronization)* will be a better solution for a particular problem you are facing.

Composability

- Functions that follow value semantics are composable.
- Pure functions follow value semantics.
- A lot of mutating functions follow reference semantics, which makes them less composable.
- It’s even worse when a function needs two references into the same object instead of just one.
- That’s iterators. Standard algorithms are not composable!

Composability pt. 2

- That was noticed – and that's why Boost.Range and the proposals for range libraries appeared.
- Some of currently proposed algorithms are still not pure.
- Some of them do not follow value semantics and mutate the passed value instead.
- Which feels better?

```
auto rng = get_range();
auto sorted = sort(rng, std::less<>{});
or
auto rng = get_range();
auto sorted = rng;
sort(rng, std::less<>{});
```

Composability and laziness

- A mutating algorithm is alright... when it’s used to implement one that follows value semantics, not reference semantics.
- Breaking rules locally is fine.
- Laziness – so only computing the result when it’s actually wanted – can make the code do less work, but requires composability.
- Mutating algorithms are eager.
- Sort then filter vs filter then sort.

Writing generic code (consciously)

- In C++ there's a weird trend to seemingly forcefully avoid genericity: shared_ptr's .get() vs future's .get() vs optional's .value(), shared_ptr's and optional's operator* vs lack of such for future.
- Possible cause for this: trying to avoid some "pitfalls" of the parts of STL that were accepted into the standard (looking at you, allocators).
- Possible cause for this: not understanding - or being afraid of - templates.
- Unified interfaces between similar types are beneficial.
- Having to additionally write generic wrappers over slightly different "low level" interfaces is not.

Digression: portability

- There seem to exist two understandings of the word "portability", or the term "portable code".
- One of them means "code that can be compiled on any platform that is compliant with some standards (against which the code was written)".
- The other one means "code that works on many platforms, but must be made aware of every single one of them to work there".
- In the first case, you just... build the software.
- In the second one, you need tons of configuration; detection whether a function (assumed to work as the standard says by the first understanding) has a bug A, a bug B, a bug C...

Writing generic code

- Templates that "just assume" that a function is available are like the first approach to portability.
- Having to write additional helpers that go around slight differences between close-to-type interfaces is like the second one.
- **Guideline:** write identical interfaces for similar types (*or:* try to think what typeclasses - or concepts - can a type you are implementing fulfill and follow that).
- It’s better to define a function near a type than to define it near the use of a functionality.

Not forcing generic code

- Sometimes a piece of code is not generic. *That is perfectly fine.*
- AbstractSingletonFactory is probably "genericity" gone too far.
- Write code and provide interfaces in a generic way when it makes sense; *do not force it.*
- Come back to types you've implemented after learning that they in fact almost fulfill a concept, and add the missing interfaces to it, so it can be used efficiently without having to add boilerplate on the user side in the future.

Designing for "success"

- Write interfaces primarily to handle successful situations.
- Handle erroneous situations, but don't design control flow around them.
- std::future's .then() (N3857):
  - *One option which was considered was to follow JavaScript's approach and take two functions, one for success and one for error handling. However this option is not viable in C++ as there is no single base type for exceptions as there is in JavaScript. The lambda function takes a future as its input which carries the exception through. This makes propagating exceptions straightforward. This approach also simplifies the chaining of continuations.*
  - Alternatives: pass "any", use dispatch similar to chained catch blocks (most of the mechanism is already in place!).

Designing for "success" pt. 2

- With the current design, we get either:
  - verbosity in the form of

    auto && value = future.get();
  - verbosity *together* with explicit, too early exception handling and the constant overhead of rethrowing an exception (across multiple continuations) until one that catches that specific exception is found
- This design doesn't solve the case of *weird* exceptions (not derived from std::exception), which just get rethrown outside of any continuation.
- The continuation that handles all exceptions with catch (…) has no direct ways of getting the exception without rethrowing it, which is not actually different from having the erroneous case by passing it an exception_ptr.
- All this doesn't matter when the code in question doesn't throw (which *should* be the more common case).

Noticing patterns and turning them into useful abstractions

- Some types have common operations. Same example as before: smart pointers, optional, future – all wrap values.
- It’s useful to be able to do similar things in the same way for similar types – genericity.
- People tend to design abstractions and force them upon types – this happens with most Gang of Four patterns.
- Actually useful patterns and abstractions are not *designed* or *invented*; they are *noticed*, or *discovered*.

Noticing patterns and turning them into useful abstractions pt. 2

- Many of the useful patterns are highly generic.
- To achieve this in a (sensible) way, you often need to pass callbacks – i.e. the implementation of the abstraction is a *higher-order functions.*
- The use of these got easier in C++11 with addition of lambdas.
- And easier still in C++14 with addition of generic lambdas.
- We knew this in C++ for a long time, since the invention of STL algorithms...
- ...and when doing this beautiful thing, we also overloaded a mathematical term: *functor*.

Summary

- Do not force guidelines mindlessly; do not follow them blindly.
- Breaking guidelines is fine, as long as the benefits outweight the potential costs.
- Immutable shared data is fine.
- Mutable local data is also fine.
- Explicit flow of data is easier to reason about (and to debug and optimize!) than one with implicit inputs.
- Abstractions are discovered iteratively, not artificially invented.

Questions?

- Please come to the standalone extension of this talk: *Functional programming – functors and monads*, at 8:30pm in McClintock (404) to learn something about the famous functional programming *techniques*.