Functional Programming in C++

Krzysztof Paprocki

C++ Meetup Zurich

22.02.2017

Properties of a good Software "Product"

1. Correct

2. Cheap

Short view at the history of programming

- “On Computable Numbers, with an application to the Entscheidungsproblem” - 1936, A. Turing
- Fortran (**For**mula **Tran**slation) – 1953 (> 60 years ago), procedural, imperative
- ALGOL 58 – 1958 (~60 years ago), procedural, imperative, structured
- LISP (**LIS**t **P**rocessor) – 1958 (~60 years ago), **functional**
- Simula – 1965 (> 50 years ago), **OO**
- Smalltalk – 1972 (~45 years ago), OO, **dynamic**
- C – 1972 (~45 years ago), imperative (robust, fast, easy)
- C++ - 1983 (~35 years ago), OO, ...

What's changed?

Why FP now?

What's changed?

Why FP now?

1. Ignorance (is power)

2. HW

HW is the game changer!

- HW has been getting cheaper and cheaper
- In the same time faster and faster
- With more and more memory…
- More complex problems could be tackled with computers
- Mainstream industry and research needed efficient, abstract languages

Is FP just a next Hype?

Is FP just a next Hype?

- To some extent – yes, but same was/is with OO

Is FP just a next Hype?

- To some extent – yes, but same was/is with OO
- What is happening in SW development:

  - The complexity of large scale SW systems sky rocketed

  - Moore's Law is not coupled with speed anymore

  - But we get always more cores

  - Need for concurrent programs – OO doesn't cut it, no problem for **FP**

Is FP just a next Hype?

- To some extent – yes, but same was/is with OO
- What is happening in SW development:

  - The complexity of large scale SW systems sky rocketed

  - Moore's Law is not coupled with speed anymore

  - But we get always more cores

  - Need for concurrent programs – OO doesn't cut it, no problem for **FP**
- Why should I learn FP?

Is FP just a next Hype?

- To some extent – yes, but same was/is with OO
- What is happening in SW development:

  - The complexity of large scale SW systems sky rocketed

  - Moore's Law is not coupled with speed anymore

  - But we get always more cores

  - Need for concurrent programs – OO doesn't cut it, no problem for **FP**
- Why should I learn FP?

  - May be your tool for utilizing many cores, for algorithms

  - New paradigm makes you a better programmer

  - Its mentally challenging and rewarding

Programming paradigms

- Imperative (OO)
  - “how”
  - mutability
- Declarative (functional, regex)
  - “what”
  - no side-effects
  - focus on results over steps
- Actor-based
- Generic
- ...

C++ is a multi paradigm language

- Procedural (“C style”)
- OO
- Generic (templates)
- **Functional** (well, sort of since C++11)

Functional Programming

Functional Programming

- Is all about Math, mainly functions: $f: X \to Y$, Category Theory in general

Functional Programming

- Is all about Math, mainly functions: $f\colon X \to Y$, Category Theory in general
- FP building blocks:

  - Pure functions (no side-effects), referential transparency

  - First-class functions (a.k.a. Higher-order functions)

Functional Programming

- Is all about Math, mainly functions: $f: X \rightarrow Y$, Category Theory in general
- FP building blocks:

  - Pure functions (no side-effects), referential transparency

  - First-class functions (a.k.a. Higher-order functions)

  - Immutability, Persistent Data Structures, Pattern Matching

  - Expressions instead of statements

Functional Programming

- Is all about Math, mainly functions: $f: X \rightarrow Y$, Category Theory in general
- FP building blocks:

  - Pure functions (no side-effects), referential transparency

  - First-class functions (a.k.a. Higher-order functions)

  - Immutability, Persistent Data Structures, Pattern Matching

  - Expressions instead of statements

  - ADT (Algebraic Data Types), e.g. Optional

Functional Programming

- Is all about Math, mainly functions: $f: X \rightarrow Y$, Category Theory in general
- FP building blocks:

  - Pure functions (no side-effects), referential transparency

  - First-class functions (a.k.a. Higher-order functions)

  - Immutability, Persistent Data Structures, Pattern Matching

  - Expressions instead of statements

  - ADT (Algebraic Data Types), e.g. Optional

  - Recursion… Good programmers don't use recursion, right?

Functional Programming

- Is all about Math, mainly functions: $f\colon X \to Y$, Category Theory in general
- FP building blocks:

  - Pure functions (no side-effects), referential transparency

  - First-class functions (a.k.a. Higher-order functions)

  - Immutability, Persistent Data Structures, Pattern Matching

  - Expressions instead of statements

  - ADT (Algebraic Data Types), e.g. Optional

  - Recursion… Good programmers don't use recursion, right?

  - Laziness

Functional Programming

- Is all about Math, mainly functions: $f: X \rightarrow Y$, Category Theory in general
- FP building blocks:

  - Pure functions (no side-effects), referential transparency

  - First-class functions (a.k.a. Higher-order functions)

  - Immutability, Persistent Data Structures, Pattern Matching

  - Expressions instead of statements

  - ADT (Algebraic Data Types), e.g. Optional

  - Recursion… Good programmers don't use recursion, right?

  - Laziness
- Languages examples:

  - Pure functional languages: Haskell

  - Languages with support for FP: Scala, Java 8, Python, C++, Closure, JavaScript, etc.

Higher-order functions

- Take other functions as arguments (<algorithm> is a kind of example)
- Return other functions

Higher-order functions

- Take other functions as arguments (<algorithm> is a kind of example)
- Return other functions
- Lambdas introduced in C++11 made it (almost) possible, better with C++14
- Lambda calculus – anonymous function

Lambda expressions - Anonymous functions

- C++11: `[](int x, int y) { return x + y; }`
- Java 8: `(int x, int y) -> x + y`
- Scala: `(x: Int, y: Int) => x + y`
- Python: `lambda x, y: x + y`
- Ruby: `l = ->(x, y) { x + y }`
- Haskell: `\x y -> x + y`

Lambda expressions - Anonymous functions

- C++11: `[](int x, int y) { return x + y; }`
- Java 8: `(int x, int y) -> x + y`
- Scala: `(x: Int, y: Int) => x + y`
- Python: `lambda x, y: x + y`
- Ruby: `l = ->(x, y) { x + y }`
- Haskell: `\x y -> x + y`

- In C++ lambdas are equivalent to function objects, so classes!
- Each lambda has a unique type

Passing lambda as an argument

```
// sort
std::sort(begin(v), end(v), [](int i, int j){
  return i < j;
});

// defining own function which takes a lambda
template<typename F>
auto h(F f) {
  // ...
  int i = 11;
  return f(i);
}
```

Returning lambda from a function

- Possible since C++14 thanks to automatic return type deduction

```
auto k() {
  return [] (int i, int j) { return i + j; };
}



auto l = k();
```

Closures

“A closure is a function that carries an implicit binding to all the variables referenced within it. In other words, it encloses a context around the things it references.”

```
[&](int x, int y) { return x + y; }
```

`[]`	no capture
`[=]`	all captured by value
`[&]`	all captured by ref
`[=outer1, &outer2]`	x by value, y by ref

Currying

- Converts a function of N arguments into a function of one argument that returns another function as its result
- No support from core language yet
- May be realized with boost::hana

```
#include <iostream>
#include "boost/hana/functional/curry.hpp"

auto f = boost::hana::curry<3>([](int x, int y, int z) {
  return x + y + z;
});

int main() {
  std::cout << f(1)(2)(3) << std::endl;
  auto g = f(1);
  std::cout << g(2)(3) << std::endl;
}
```

Partial application

- Returns a new function while providing some arguments to a original function and letting the rest of the arguments be provided later.
- boost::hana and to some extent with std::bind

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <boost/hana/functional/placeholder.hpp>

using boost::hana::_;

int main() {
  std::vector<int> v = { 4, 6, 8, 10 };
  std::transform(begin(v), end(v), begin(v), _ - 4);
}
```

Expressions vs Statements

- Expressions evaluate to a value

```
z = x + y;
```

- While Statements don't:

```
z = if (x) { /*...*/ } else { /*...*/ } // error: expected expression
```

- In Scala it is legal though

Expressions

```
Wire disarmBomb(WeekDay weekDay) {
  switch (weekDay) {
    case WeekDay::Monday:
      return Wire::BLUE;
    case WeekDay::Friday:
      return Wire::RED;
    default:
      return Wire::GREEN;
  }
}
```

Expressions

```
Wire
  swi
    c

    case WeekDay::Friday:
      return Wire::RED;
    default:
      return Wire::GREEN;
```

Rule 6–6–5	**(Required)**	**A function shall have a single point of exit at the end of the function.**

Recursion: Tail-call

- Non-tail-call:

```
int f(int i) {
  if (i == 0) {
    return 0;
  }
  return i + f(i - 1);
}
```

Recursion: Tail-call

- Non-tail-call:

```
int f(int i) {
  if (i == 0) {
    return 0;
  }
  return i + f(i - 1);
}
```

- Tail-call occurs when recursive call is the last evaluated expression

```
int f(int i, int j = 0) {
  if (i == 0) {
    return j;
  }
  return f(i - 1, i + j);
}
```

Recursion: Tail-call optimization

- Tail-call occurs when recursive call is the last evaluated expression

```
int f(int i, int j = 0) {
  if (i == 0) {
    return j;
  }
  return f(i - 1, i + j);
}
```

- Above compiled with optimization **should** generate `jmp` instead of `call`
- No stack frame created, fast and safe
- Equivalent to `while()` loop

Recursion: Tail-call optimization support in C++

- Tail-call optimization is not specified, depends on compiler implementation
- If used, needs to be verified with assembler if really optimized
- Check must be done in “release” build, means with optimization switched on
- Quite big effort, so practical direct usage for C++ it is rather limited...

Recursion: Tail-call optimization support in JVM (Scala)

- JVM is even worse: does not support tail-call optimization at all
- But scalac does it on its own
- Can be easily checked with annotations:

```
import scala.annotation.tailrec

class TailCall {
  @tailrec final def f(i: Int, j: Int = 0): Int = {
    if (i == 0) j
    else f(i - 1, i + j);
  }
}
```

Recursion: Tail-call optimization support in JVM (Scala)

- Non-Tail-call with annotation:

```
import scala.annotation.tailrec

class TailCall {
  @tailrec final def f(i: Int, j: Int = 0): Int = {
    if (i == 0) j
    else 1 + f(i - 1, i + j);
  }
}
```

- Output:

```
[error] .../tailCall.scala:6: could not optimize @tailrec annotated
method f: it contains a recursive call not in tail position
[error]     else 1 + f(i - 1, i + j);
[error]            ^
[error] one error found
[error] (compile:compileIncremental) Compilation failed
```

Laziness

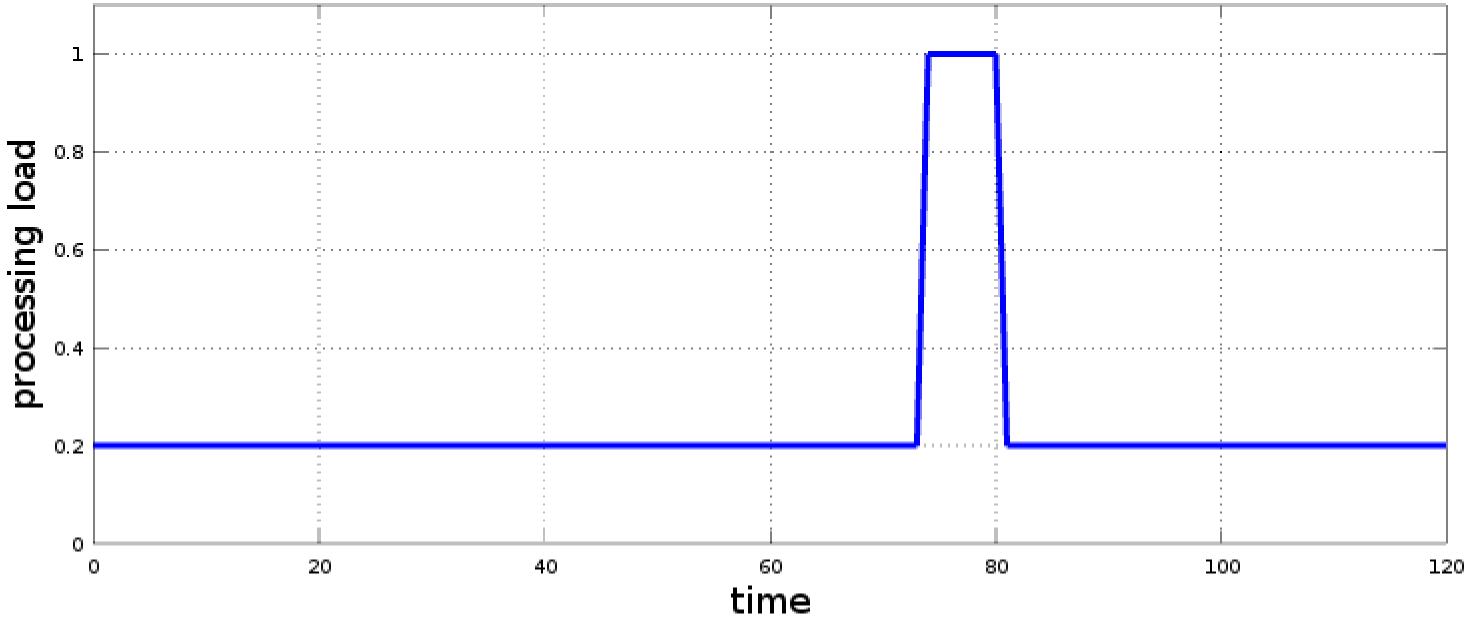
- Evaluation of the expression is postponed until the result is needed
- Pass lambda for expensive computation which might be not always evaluated:

```
template<typename F>
auto h(bool b, int x, F f) {
  if (b) {
    return f(); // expensive computation
  } else {
    return x;
  }
}
```

- Ranges

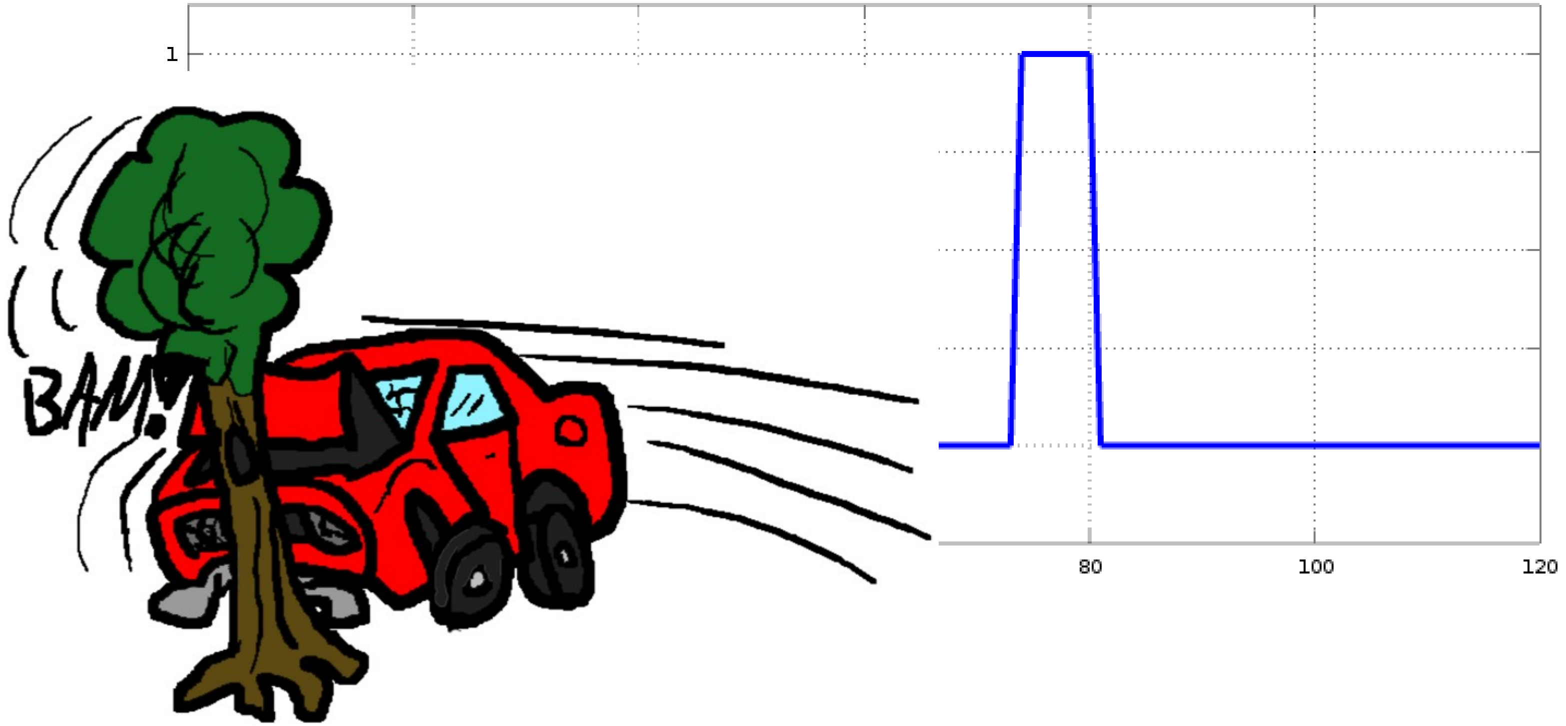
Laziness

- Evaluation of the expression postponed until result needed



Laziness

- Evaluation of the expression postponed until result needed



Common FP functions

- Map – applies function on each element of the collection, is a best example of functor in mathematical sense
- Filter – applies filtering function on each element of the collection and decides which elements stays in the collection (goes to the new collection)
- Fold (reduce) – takes data in one format and gives back in another

Common FP functions

- Map => std::transform
- Filter => std::copy_if
- Fold (reduce) => std::accumulate

Fold

1. Takes/Uses a collection, range, etc.

2. Takes init value

3. Takes a function. This function takes two arguments: the accumulated value and the current item in the collection

4. At the beginning of execution the init value is used as the first argument for the above function. Second argument is the first element in the collection

5. The result of the function is then passed as first argument to the function in the next iteration. Second argument is next element of the collection

6. Repeats step 5 until the end of the collection

Fold

```
struct Person {
  string name;
  int age;
};

vector<Person> v { Person{ "James", 24 }, Person{ "Alice", 28 } };

list<string> names;
accumulate(begin(v), end(v), &names, [](list<string>* l, Person e) {
                                          l->push_back(e.name);
                                          return l; });
```

Fold expression

- Since C++17, uses variadic templates
- Is one of the form:

```
( pack op ... )               // unary right fold
( ... op pack )               // unary left fold
( pack op ... op init )       // binary right fold
( init op ... op pack )       // binary left fold
```

allowed op binary operators:

```
+ - * / % ^ & | = < > << >> += -= *= /= %= ^= &= |= <<= >>= == != <=
>= && || , .* ->*
```

Fold expression

- Usage:

```
template<typename... Args>
auto reduceSum(Args&&... args) {
  return (... + args);
}
```

```
std::cout << reduceSum(2, 5, 8, 7) << std::endl;
std::cout << reduceSum(std::string("a"), std::string("b"),
                                              std::string("c")) << std::endl;
```

Algebraic Data Types (ADT)

- Is the FP feature used for creation of complex types from simpler ones
- Use simple type composition operators

ADT concepts

- Unit type – has exactly one possible value, denoted with *()*
- Product Operation

  - given types A and B, the type $A \otimes B$ is the product of A and B

  - a value of $A \otimes B$ contains an element of type A **and** an element of type B

  - can be generalized to n-tuple $A \otimes B \otimes C \otimes$ ...
- Sum Operation

  - given types A and B, the type $A \oplus B$ is the sum of A and B

  - a value of $A \oplus B$ contains an element of type A **or** an element of type B

ADT, Type Definitions

Shorthands for more complex types:

- $a := e$ means “is the same as”, where:

  a is new identifier

  e is the type expression

  - Example:

    $A := D \otimes E$

    from now on A is allowed to be used instead of $D \otimes E$

- $a ::= e$ creates new type based on type expression, “is implemented with”

  - Example:

    bool type may be created from *int*, when ignoring values other than 0 and 1:

    $B ::= int$

ADT, Type Functions

- Allow to create type templates in general form:

  $f\ a_1\ a_2\ \ldots\ a_n := e$ and $f\ a_1\ a_2\ \ldots\ a_n ::= e$ where:

  f is a new identifier

  $a_1\ a_2\ \ldots\ a_n$ are types, which may be used by expression e

- Example:

  $none ::= ()$

  $f\ a := none \oplus a$

  and concrete type:

  $fInt := f\ Int$

ADT in C++

- Product:

```
struct T {
  A a;
  B b;
};

std::pair<T, K>

boost::fusion::vector<T, K>

boost::fusion::map<>

…
```

ADT in C++

- Sum:

```
enum StreetLight {
  Red,
  Orange,
  Green
};

boost::variant<T, K>

std::optional<T>  // since C++17!
```

ADT in C++, std::optional

- Is very “functional”
- All FP languages have it (Haskell: Maybe, Scala: Option, etc.)
- This is an example of The Monad!
- Represents computations which might go wrong
- Provides clean error handling
- We already saw std::optional type function:

$$none ::= ()$$

$$f\ a := none \oplus a$$

ADT in C++, std::optional

- Sending some data...

```
std::experimental::optional<std::string> send(Data data) {

    /* wrap protocol... */

    int res = io_send(buf);

    if (res < Error::OK) {
        return {};
    } else {
        return std::to_string(res);
    }
}
```

ADT in C++, std::optional

- Sending some data...

```
std::experimental::optional<std::string> send(Data data) {

    /* wrap protocol... */

    int res = io_send(buf);

    if (res < Error::OK) {
        return {};
    } else {
        return std::to_string(res);
    }
}



std::cout << "sent: " << send(Data()).value_or("failed") << std::endl;
```

ADT in C++, std::optional

- Working with maps…
- In Scala one can write:

```
val m = Map[String, Int]("a" -> 1, "b" -> 2)
val s = m.getOrElse("d", 0)
```

- In C++ if one wants to get element in the std::map:

```
std::map<Type1, Type2> m;
/* inserts...*/
auto res = m.find(key);
if (res != m.end()) {
  /* do smth with res->second; */
} else {
  /* error? */
}
```

ADT in C++, std::optional

- More “optional” approach:

```
template<typename K, typename V>
class MyMap {
  public:
    std::experimental::optional<V> get(const K& key) {
      auto res = m.find(key);
      if (res != m.end()) {
        return res->second;
      } else {
        return {};
      }
    }

    V getOrElse(const K& key, const V& elseVal) {
      return get(key).value_or(elseVal);
    }

    std::map<K, V>& operator()() {
      return m;
    }
  private:
    std::map<K, V> m;
};
```

ADT in C++, std::optional

- More “optional” approach:

```
MyMap<std::string, int> myMap;
myMap().insert(std::pair<std::string,int>("a",3));
myMap().insert(std::pair<std::string,int>("b",6));
```

```
std::cout << myMap.get("b").value_or(0) << std::endl;

std::cout << myMap.getOrElse("5",0) << std::endl;
```

Pros & Cons of FP

- Pros:

  - Powerful tool for algorithms, data handling, clustering, some DSP

  - may make the SW “Product” more correct and way cheaper

  - A way to directly translate math into computer program

Pros & Cons of FP

- Pros:

  - Powerful tool for algorithms, data handling, clustering, some DSP

  - may make the SW “Product” more correct and way cheaper

  - A way to directly translate math into computer program

- Cons:

  - Poor implementation for non-mainstream compilers

  - It is difficult to convince an organization to update their compilers

  - Many concepts not suitable for embedded systems and mission critical applications (FP may be unpredictable)

  - May make the SW “Product” not correct and very expensive

  - Negative effect of social factor – It is hard to find devs which are open-minded and willing to learn outside office hours

Want more?

- Function composition
- Monoid, Monad, in general Category Theory
- Persistent Data Structures
- Pattern matching
- Trampolines
- Parallelism
- Other functional design patterns...

paprocki.krzysztof@gmail.com

www.linkedin.com/in/krzysztof-paprocki-a2695439