Italian C++ Community

www.italiancpp.org

Cat’s anatomy

...overview of a functional library for C++14

Nicola Bonelli <nicola@pfq.io>

Overview

What is cat?

- Cat is a C++14 functional library inspired by Haskell
- It consists of 97 headers roughly divided into 3 major groups
  - Utilities for the functional programmer
  - Common type-classes
  - Instances of type classes (instances of standard C++ types)
- Is the library mature?
  - No, it is still a work in progress
  - Documentation is still missing… :(

To whom is it useful?

- C++ coders that need some utility functions
  - Cat fills the gap of C++ libraries with respect to functional programming, perfect forwarding, tuple manipulation, type traits, etc...
- C++ coders that want to move their first steps toward functional programming
  - if you master function thinking the pain to switch to Haskell will be relieved
  - porting projects from Haskell to C++ become much more simple
- Functional thinking improves the quality of C++ code!

Library tree

```
├── cat
│   ├── applicative/...
│   ├── applicative.hpp
│   ├── bifunctor/...
│   ├── bifunctor.hpp
│   ├── bits/...
│   ├── __config.hpp
│   ├── container.hpp
│   ├── foldable/...
│   ├── foldable.hpp
│   ├── functional.hpp
│   ├── functor/...
│   ├── functor.hpp
│   ├── infix.hpp
│   ├── iterator.hpp
│   ├── match.hpp
│   ├── meta.hpp
│   ├── monad/...
│   ├── monad.hpp
│   ├── monoid/...
│   ├── monoid.hpp
│   ├── optional.hpp
│   ├── placeholders.hpp
│   ├── read/...
│   ├── read.hpp
│   ├── section.hpp
│   ├── show/...
│   ├── show.hpp
│   ├── string_view.hpp
│   ├── tuple.hpp
│   ├── type_traits.hpp
│   └── utility.hpp
├── example/...
│   └── parser/...
└── test/...
    ├── CMakeLists.txt
    ├── inc/...
    ├── test/...
```

Utility functions

Category	Functions
functional utilities	`curry, curry_as, compose (^), flip, on...`
container	`fold operations, etc.`
infix, sections, match	`utility for infix operators, sections (binary operators), pattern matching...`
type traits	`function_type, function_arity, return_type, arg_type_at...`
perfect forwarding	`const_forward, forward_as, forward_iterator...`
tuple	`make_type, tuple_assign, tuple_foreach, tuple_map, tuple_apply...`

Type classes and instances

Type-class	Instances
functor	`vector, deque, list, forward_list, string, map, multimap, (unordered_*), optional, pair, shared_ptr, unique_ptr, future...`
bifunctor	`pair`
applicative, alternative	`vector, deque, list, forward_list, optional, pair, shared_ptr, unique_ptr, future...`
foldable	`vector, deque, list, forward_list, string, map, multimap, set, multiset, (unordered_*), optional, pair, shared_ptr, unique_ptr...`

Type classes and instances

Type-class	Instances
monoid	vector, deque, list, forward_list, string, map, multimap, set, multiset, (unordered_*), optional, pair, shared_ptr, unique_ptr, future...
monad, monad plus	vector, deque, list, forward_list, string, optional, shared_ptr, unique_ptr, future...
read, show	containers, chrono, fundamentals, optional, string types, tuples

A functional approach...

In functional programming (FP)...

- Function is a first-class citizen
  - it can be passed to or returned from functions
- Functions are composable and with partial app.
  - improve code reusability
- Pure functions
  - have no observable side effects
  - referential transparency
- Data immutability
  - concurrent programming

Functions and arguments in C++?

- PF encourages any C++ callable type that
  - is pure, that does not evaluate on the basis of a global states
  - does not change the arguments passed

- A pure function can takes arguments by:
  - value
  - const L-value reference
  - R-value reference
  - universal reference (T &&)

- What's the recommended method?

Argument passing comparisons

semantic	***retain ownership***	**pros**	**cons**
value	• copyable? yes • non-copyable ? **no** (need move)	• clean design • good for sink fun. • **1 function to maintain**	• **non copyable object needs to be moved** • non-cheap copyable object is inefficient
const L-value ref.	• **yes**	• ideal interface • no extra copy/move	• **2^n overloads**
R-value ref.	• no, but desidered	• good perf.	• **2^n overloads**
universal ref (T &&) note: (std::vector<T> &&) is not a forwarding reference...	• yes	• no extra copy/move • **1 function to maintain**	• enforce the const L-value ref. • the argument must be a template • wrong types passed generate template error hell.

Argument passing with perfect forwarding

- Allows to pass both L-value and R-value arguments
- No extra copy or extra move is required
- Allows to retain the ownership of passed objects
    - non-copyable objects does not have to be moved (as req. in pass-by-value)
- If R-value expression is passed then the the function can take advantage of it
    - moving elements out of an expiring container

Cat general design

- Cat takes advantage of
  - Extensibility
    - instances of a given type are implemented as partial specializations of certain class
  - Static polymorphism
    - exploits the C++ inheritance only to ensure that interfaces are complete and correct
  - Non being OOP
    - free functions (or constexpr callables types) are the user APIs
  - modern C++
    - **`constexpr`** constructors for building objects at compile time
    - **`override/final`** help compiler devirtualize C++ methods
    - **`auto`** for trailing return type deduction

Utility functions

Perfect forwarding

- Universal reference allows to perfect forward expressions to/from generic functions
  - T && arg
    - T && seems a R-value ref. (but it is not)
    - arg is an L-value expression.
  - std::forward is used to restore the original type:
    - std::forward<T>(arg)
      - T can either be T& or T
- Reduce the number of functions to maintain:
  - for each argument one should maintain two versions
    - Object &&arg, and Object const &arg (2^n with the number of arguments)
  - T && can also be used in variadic functions: Ts && ...args

Perfect forwarding

`template <typename F, typename T>`
`auto map(F fun, std::vector<T> const & xs)`
`{       …`
`        for(auto & x : xs)`
`                … = fun(x);`
	`template <typename F, typename T>`
	`auto map(F fun, std::vector<T> && xs)`
	`{       …`
	`        for(auto & x : xs)`
	`                … = fun(std::move(x));`

Perfect forwarding

```
template <typename F, typename T>
auto map(F fun, std::vector<T> const & xs)
{       …
        for(auto & x : xs)
                … = fun(x);
```

```
template <typename F, typename T>
auto map(F fun, std::vector<T> && xs)
{       …
        for(auto & x : xs)
                … = fun(std::move(x));
```

```
template <typename F, typename Vector>
auto map(F fun, Vector && xs)
{       …
        for(auto & x : xs)
                … = fun(???(x));
```

how to forward this?

Perfect forwarding

```
template <typename F, typename T>
auto map(F fun, std::vector<T> const & xs)
{       …
        for(auto & x : xs)
                … = fun(x);
```

```
template <typename F, typename T>
auto map(F fun, std::vector<T> && xs)
{       …
        for(auto & x : xs)
                … = fun(std::move(x));
```

```
template <typename F, typename Vector>
auto map(F fun, Vector && xs)
{       …
        for(auto & x : xs)
                … = fun(cat::forward_as<Vector>(x));
```

forward_as

- Forward_as implementation:

```
template<typename T, typename V>
    decltype(auto)
    forward_as(V && value)
    {
        return static_cast<
            std::conditional_t<
                std::is_lvalue_reference<T>::value,
                    std::remove_reference_t<V> &,
                    std::remove_reference_t<V> &&>>(value);
    }
```

iterator or move_iterator?

- Homework
  - “...write a function that takes two vectors and returns a new one, result of concatenation.”

- Be aware that...
  - elements contained could be:
    - cheap to move but expensive to copy

- Suggestion:
  - take into account the L/R value-ness of the vectors passed...

iterator or move_iterator?

<pre>template <typename T>
auto concat(std::vector<T> const & xs, std::vector<T> const & ys)
{
    auto ret = xs;
    ret.insert(std::end(ret),
               std::begin(ys)),
               std::end(ys)));
    return ret;
}</pre>
	<pre>template <typename T>
	auto concat(std::vector<T> const & xs, std::vector<T> && ys)
	{
	    auto ret = xs;
	    ret.insert(std::end(ret),
	               std::make_move_iterator(std::begin(ys)),
	               std::make_move_iterator(std::end(ys)));
	    return ret;
	}</pre>

<pre>template <typename T>
auto concat(std::vector<T> && xs, std::vector<T> const & ys)
{
    auto ret = std::move(xs);
    ret.insert(std::end(ret),
               std::begin(ys)),
               std::end(ys)));
    return ret;
}</pre>
	<pre>template <typename T>
	auto concat(std::vector<T> && xs, std::vector<T> && ys)
	{
	    auto ret = std::move(xs);
	    ret.insert(std::end(ret),
	               std::make_move_iterator(std::begin(ys)),
	               std::make_move_iterator(std::end(ys)));
	    return ret;
	}</pre>

forward_iterator

- Concat example with a single function?

```
template <typename Vec1, typename Vec2>
auto append(Vec1 && xs, Vec2 && ys)
{
    auto ret = std::forward<Vec1>(xs);
    ret.insert(std::end(ret),
               ???(std::begin(ys)),
               ???(std::end(ys)));
    return ret;
}
```

how to forward these?

forward_iterator

```
template <typename Vec1, typename Vec2>
auto append(Vec1 && xs, Vec2 && ys)
{
    auto ret = std::forward<Vec1>(xs);
    ret.insert(std::end(ret),
               cat::forward_iterator<Vec2>(std::begin(ys)),
               cat::forward_iterator<Vec2>(std::end(ys)));
    return ret;
}
```

forward_iterator

```
template <typename Iter>
 auto forward_iterator_impl(Iter it, std::true_type)
 {  return it; }

 template <typename Iter>
 auto forward_iterator_impl(Iter it, std::false_type)
 {  return std::make_move_iterator(it); }

 template <typename Ref, typename Iter>
 auto forward_iterator(Iter it)
 {
     return forward_iterator_impl(std::move(it), std::is_lvalue_reference<Ref>{});
 }
```

Additional functional utilities...

- Cat provides a callable type generated by curry function which supports:
  - composition and lazy evaluation

```
auto f = [](int a, int b) { return a+b; };
auto g = [](int a) { return a+1; };

auto f_ = cat::curry(f);
auto g_ = cat::curry(g);
cout << f_(1)(2);
int c = f_(1,2);
auto l = f_.apply(1,2); // lazy evaluation
c += l();
```

Additional functional utilities...

- Functional composition (math style)

```
auto h = f_ ^ g_;
cout << h(1,2); // > 4
```

- Argument flipping

```
auto g = cat::flip(f);
```

- Infix on operator

```
vector<pair<int, string>> vec = { {2, “world”}, {2, “hello”} };
sort(begin(xs), end(xs), less<int>{} |on| first);
```

cat callable and curry

- FP does not encourage passing arguments by L-value ref.
    - however Cat library is able to handle them.
- The callable type generated by `curry` is a closure?
    - If the target argument is a L-value ref. then an L-value ref. is stored into the callable.
    - A copy of the decayed argument is stored otherwise.
- What are the implications?
    - targets may take arguments by copy, L-value or R-value reference.
    - because the callable may hold an L-value reference, an undefined behavior is expected if evaluated with expired referred arguments

cat callables vs. std bind

cat::curry

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

int n = 0;
auto f_ = cat::curry(f)(1);

std::cout << f_(n) << std::endl;
std::cout << n << std::endl;
```

std::bind

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

int n = 0;
auto f_ = std::bind(f, 1, _1);

std::cout << f_(n) << std::endl;
std::cout << n << std::endl;
```

cat callables vs. std bind

cat::curry

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

int n = 0;
auto f_ = cat::curry(f)(1);

std::cout << f_(n) << std::endl;
std::cout << n << std::endl;

> 2
> 1
```

OK

std::bind

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

int n = 0;
auto f_ = std::bind(f, 1, _1);

std::cout << f_(n) << std::endl;
std::cout << n << std::endl;

> 2
> 1
```

OK

cat callables vs. std bind

cat::curry

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

int n = 0;
auto f_ = cat::curry(f)(n);

std::cout << f_(1) << std::endl;
std::cout << n << std::endl;
```

std::bind

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

int n = 0;
auto f_ = std::bind(f, n, _1);

std::cout << f_(1) << std::endl;
std::cout << n << std::endl;
```

cat callables vs. std bind

cat::curry

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

int n = 0;
auto f_ = cat::curry(f)(n);

std::cout << f_(1) << std::endl;
std::cout << n << std::endl;

> 2
> 1
```

unspecified

unspecified

OK

std::bind

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

int n = 0;
auto f_ = std::bind(f, n, _1);

std::cout << f_(1) << std::endl;
std::cout << n << std::endl;

> 2
> 0
```

store with decay

perfect forwarding

what's going on here?!?

Type classes

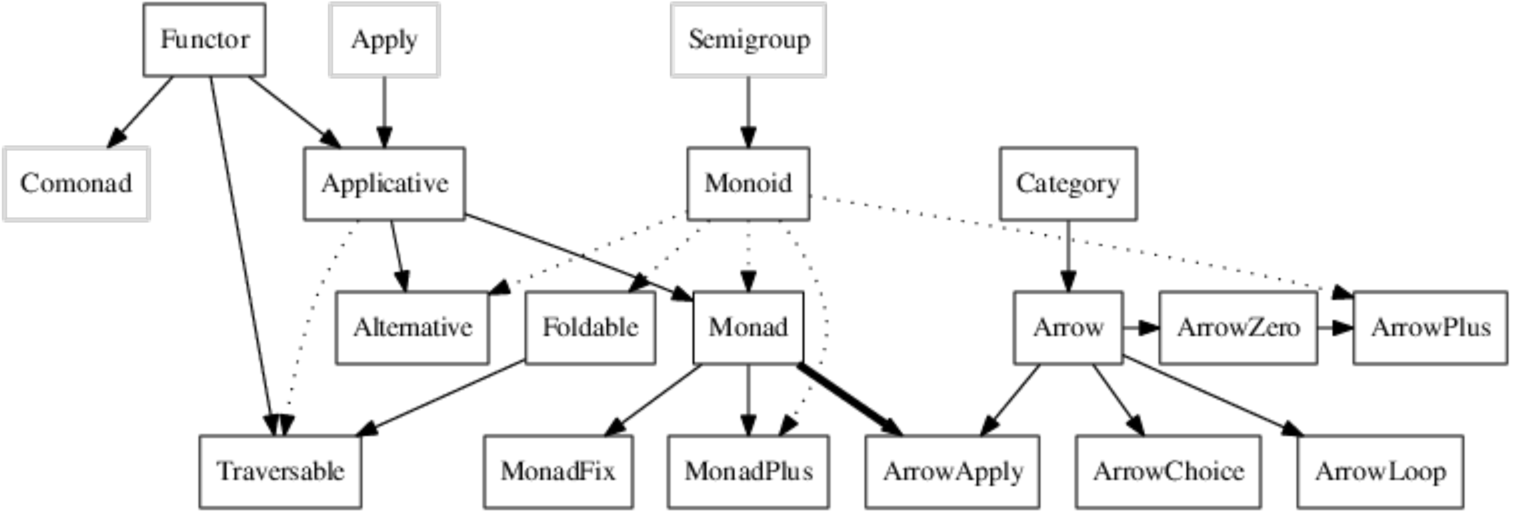
Type classes

- Type class is a type system construct that supports *ad hoc polymorphism*
  - In other words a type class is a collection of types with a common property…

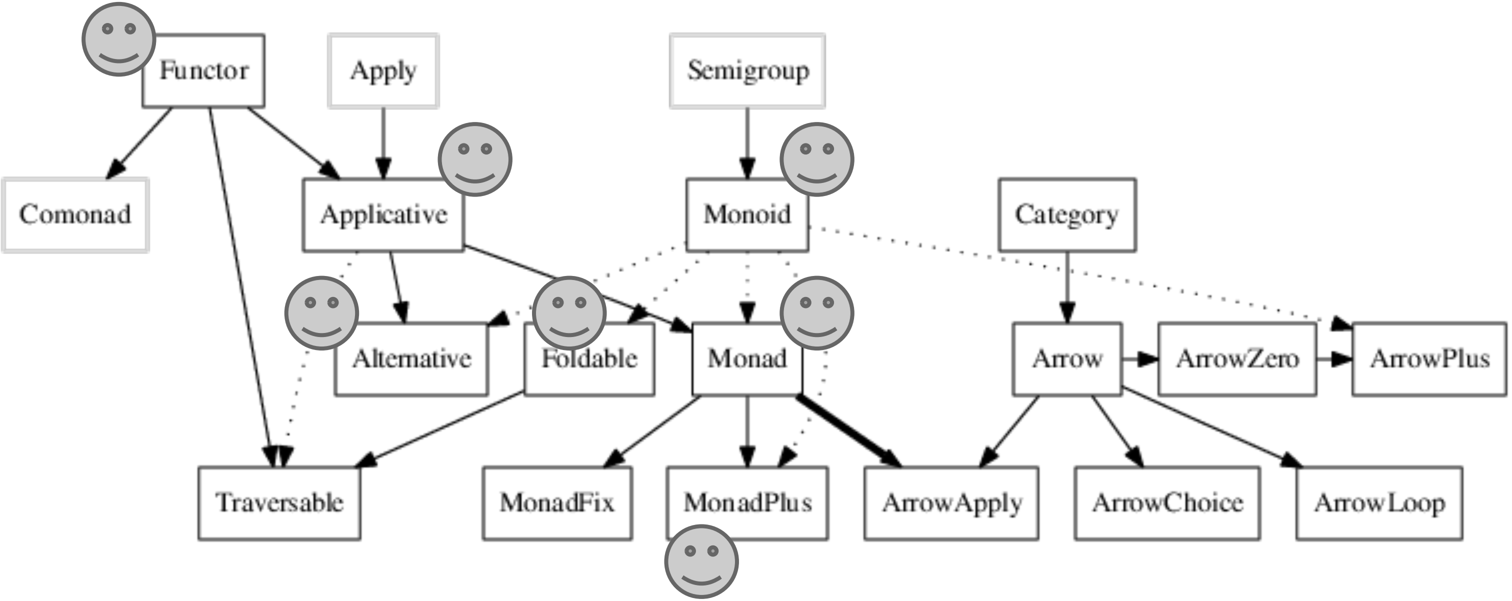
- Why type classes are useful in C++?
  - type classes can regulate overloading and generic programming
    - from the perspective of error handling
  - can be used effectively with *concepts*

- The Haskell language Eq typeclass
  - A very basic type-class is Eq with == and /=
  - A type is an instance of Eq typeclass if it does support ==, /= operators

Typeclassiopedia



Typeclassiopedia



Show type-class

- Show typeclass is useful for debugging, it provides the function show that converts a type into a string:

  ```
  std::string show(T const &);
  ```

- Each C++ standard type has an instance of show:
  - fundamental types, arrays, containers, pair, tuples, pointers, chrono types, optional and string_view.

- It’s possible to declare new instances of show:
  - a new or existing type becomes showable
  - is_showable<T> type traits is generated and can be used in TMP, concepts, static_asserts, etc…

Show type-class

```
template <typename T>
struct Show
{  virtual std::string show(T const &) = 0;
};
```

Type class

```
template <typename T> struct ShowInstance;
```

Base instance

```
template <typename T>
inline std::string show(T const &v)
{
    static_assert(is_showable<T>::value, "T is not showable!");
    return ShowInstance<T>{}.show(v);
}
```

free function

```
template <typename T>
struct is_showable : has_specialization<ShowInstance, T> { };
```

Useful type trait

Show instances...

```
template <typename T>
struct ShowInstance<int> final : Show<int>
{
    std::string
    show(const int &value) override
    { return std::to_string(value); }
};
template <typename T>
struct ShowInstance<bool> final : Show<bool>
{
    std::string
    show(const bool &value) override
    { return value ? “true” : “false”;}
};
```

int

bool

Show instances...

```
template <typename T>
struct ShowInstance<int> final : Show<int>
{
    std::string
    show(const int &value) override
    { return std::to_string(value); }
};
template <typename T>
struct ShowInstance<bool> final : Show<bool>
{
    std::string
    show(const bool &value) override
    { return value ? “true” : “false”;}
};
```

int

bool

```
template <typename T>
void print_(T const &elem)
{
        std::cout << show(elem) << std::endl;
};

 print_(42);
 print_(make_tuple(2.718281, "Hello World!", nullopt));
 print_(vector<int>{1,2,3});

 42
( 2.718281 "Hello World!" () )
[ 1 2 3 ]
```

Show instances...

```
template <typename T>
struct ShowInstance<optional<T>> final : Show<optional<T>>
{
    std::string
    show(const optional<T> &t) override
    {
        if (t) return std::string(1,’(‘) + cat::show(t.value()) + ')';
        return "()";
    }
};
template <>
struct ShowInstance<nullopt_t> final : Show<nullopt_t>
{
    std::string show(const nullopt_t &) override
    { return "()"; }
};
```

`optional<T>`

`nullopt_t`

Read type-class

```
template <typename T>
struct Read
{
    virtual optional<pair<T, string_view>> reads(string_view) = 0;
};
```

Type class

Read type-class

```
template <typename T>
struct Read
{
    virtual optional<pair<T, string_view>> reads(string_view) = 0;
};
```

Type class

```
const char * s = “13 42”;
if (auto x = reads<int>(s)) {
if (auto y = reads<int>(x->second)) {
    cout << show(x) << ' ' << show(y) << endl;
}}

((13 " 42")) ((42 ""))
```

Functor

- Functor is a class for types which can be mapped over (haskell-wikibooks)
    - It has a single method (high-order function) called fmap.

- The intuitive example about functor is that of a box
    - fmap takes a function from apples to eggs (Apple -> Egg) and a box of apples, and return a box of eggs

- A functor is any kind of type constructor that can contain a type
    - in C++ a container, an optional, a smart pointer etc. is a functor
        - functor properties for them are satisfied
    - with fmap that is able to apply a function over

Functor type-class

template template argument

```
template <template <typename ...> class F>
struct Functor
{
    template <typename A, typename Fun, typename Fa_>
    struct where
    {
        virtual auto fmap(Fun fun, Fa_ && fa) -> F<std::result_of_t<Fun(A)>> = 0;
    };
};
```

virtual template method

is this forwarding reference!?!?

return type deduction

```
template <typename Fun, typename Fa_>
 auto operator()(Fun f, Fa_ && xs) const
 {
     static_assert(..., "Type not a functor!");
     return FunctorInstance<std::decay_t<Fa_>, Fun, Fa_>{}.fmap(std::move(f), std::forward<Fa_>(xs));
 }
```

Functor instance

```
template <typename A, typename Fun, typename Fa_>
struct FunctorInstance<std::vector<A>, Fun, Fa_> final :
Functor<std::vector>::template where<A, Fun, Fa_>
{
    using B = std::result_of_t<Fun(A)>;

    std::vector<B>
    fmap(Fun f, Fa_ && xs) override
    {
        std::vector<B> out;
        out.reserve(xs.size());
        for(auto & x : xs)
            out.push_back(f(cat::forward_as<Fa_>(x)));
        return out;
    }
};
```

return type deduction

is this forwarding reference!?!?

Functor instance

```
template <typename A, typename Fun, typename Fa_>
struct FunctorInstance<std::vector<A>, Fun, Fa_> final :
Functor<std::vector>::template where<A, Fun, Fa_>
{
    using B = std::result_of_t<Fun(A)>;

    std::vector<B>
    fmap(Fun f, Fa_ && xs) override
    {
        std::vector<B> out;
        out.reserve(xs.size());
        for(auto & x : xs)
            out.push_back(f(cat::forward_as<Fa_>(x)));
        return out;
    }
};
```

return type deduction

is this forwarding reference!?!?

```
vector<string> v = {"hello", "world"};
auto s = fmap([](string const &s) { return s.size(); }, v);

cout << show (v) << endl;
cout << show (s) << endl;

[ "hello" "world" ]
[ 5 5 ]
```

Home page

https://http://cat.github.io/

Get the code!

git clone https://github.com/cat/cat.git

Volunteers?

nicola@pfq.io