

A workshop on monads with C++14



Meetup C/C++ de Madrid
Joaquín M^a López Muñoz <joaquin@tid.es>
Madrid, January 2015

Telefónica

Workshop styles



Workshop styles



Workshop styles



Potential outcome



Potential outcome



Prerequisites



Prerequisites

- You are expected to be reasonably fluent with some post-2003 C++ stuff
 - `auto`
 - `std::function`, lambda functions, generic lambda functions
 - function return type deduction, trailing return type declaration
 - Template template parameters (this is C++03, anyway)
 - `decltype`, `std::declval`
- You are encouraged to pre-read about functional programming and monads
 - For instance, go to github.com/joaquintides/usingstdcpp2014
- You are required to bring a computer with
 - Internet access (hopefully provided by host)
 - A C++14 compiler such as GCC 4.9 (`-std=c++1y`) with a recent Boost distro
 - Alternatively, you can use an online environment such as Coliru (coliru.stacked-crooked.com)

Maybe Monad




```
template<typename T>
struct optional
{
    optional(const T& x);
    optional(none_t);

    const T& get()const;
    T&      get();

    operator bool()const; // sort of
};

optional<double> inv(double x){
    if(x==0.0)return none;
    else      return 1.0/x;
}

optional<double> sqr(double x){
    if(x<0.0)return none;
    else      return std::sqrt(x);
}

optional<double> arcsin(double x){
    if(x<-1.0||x>1.0)return none;
    else              return std::asin(x);
}
```


- Implement the following composition of functions

```
optional<double> ias(double x)
{
    // return inv(arcsin(sqrt(x))) where it makes sense
}
```


- Implement this helper function

```
template<typename F>
optional<double> call(const optional<double>& x, F f)
{
    // call f if x is valid, none otherwise
}
```

- Implement **ias** using **call**

- Now **inv** is modified so that it truncates its result to **int**:

```
optional<int> inv(double x)
{
    if(x==0.0) return none;
    else      return int(1.0/x);
}
```

- Modify **call** and **ias** to cope with type mismatches


```
template<template<typename> class M,typename T>
M<T> mreturn(const T& x)
{
    return x;
}
```

// must be overloaded

```
template<template<typename> class M,typename T,typename F>
auto operator>>=(const M<T>& m, F f)->decltype(f(std::declval<const T>()));
```

■ Signatures

$\text{return} : T \rightarrow M<T>$

$>>= : (M<T>, T \rightarrow M<T'>) \rightarrow M<T'>$

List Monad



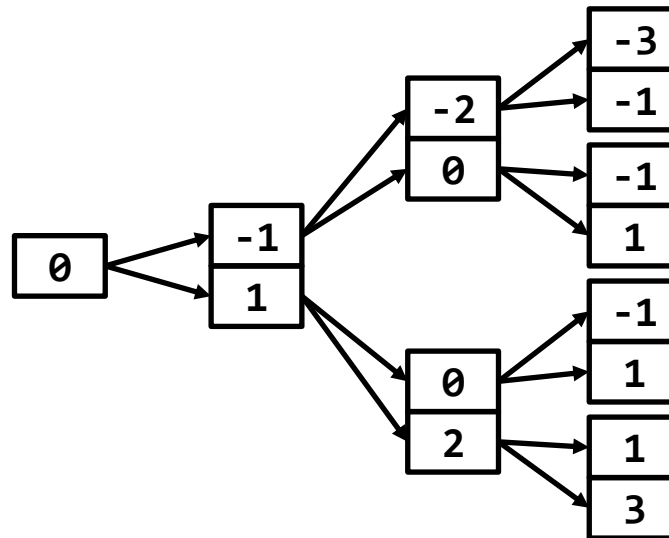
list (function composition) as a monad

list1.cpp

```
template<typename T> struct list::std::list<T>
{
    using base=std::list<T>;
    using base::base;
    list(const T& x):base(1,x){} // compatibility with mreturn
};
```

```
list<int> decinc(int x){return {x-1,x+1};}
```

- How do we compose **decinc** with itself?



- Hint: returning lists is like having multiple potential results
- Hint: look at the signature of `>>=`

■ Define >>= for **list**

```
template<typename T,typename F>  
auto operator>>=(const list<T>& l, F f)  
{  
    // ...  
}
```

■ Calculate

```
((decinc(0)>>=decinc)>>=decinc)
```

■ Calculate

```
list<int>{0,1,4,5}>>=decinc
```


- Define **transform** in terms of **>>=** (generically!)

```
template<template<typename> class M, typename T, typename F>
auto transform(const M<T>& m, F f)
{
    // ...
}
```

- Hint: $F: T \rightarrow R$, but **>>=** expects $F': T \rightarrow M<R>$

- Calculate

```
transform(list<int>{0,1,2}, [](int x){return x+1;})
```

- What will these be?

```
transform(optional<int>{1}, [](int x){return x+1;})
transform(optional<int>{none}, [](int x){return x+1;})
```

Exercise (extra bonus points!)

list4.cpp

■ List comprehension:

```
{x+1 | x <- [0 1 2]}
```

implemented as

```
list<int>{0,1,2}>>=[](int x){return mreturn<list>(x+1);}
```

■ Implement

```
{x+y | x <- [0 1 2], y <- [1 2 3]} // [1 2 3 2 3 4 3 4 5]
```


do notation

```
{x+y | x <- [0 1 2], y <- [1 2 3]}
```

- Equivalent in **do** notation (Haskell, beware **return** is a monadic function, not the C++ keyword)

```
do x <- [0 1 2]
   y <- [1 2 3]
   return (x + y)
```

equivalent to (Haskell)

```
[0 1 2] >>= \ x ->
  [1 2 3] >>= \ y ->
    return (x + y)
```

translated to (C++)

```
list<int>{0,1,2}>>=[&](int x){
  return list<int>{1,2,3}>>=[&](int y){
    return mreturn<list>(x+y);
  };
}
```

■ Do you hate macros?

```
#define DO(var,monad,body) \
((monad)>>=[=](const auto& var){ \
    return body; \
})
```

■ Just for recreational purposes

```
auto res=
    DO(x,list<int>({0,1,2}),
    DO(y,list<int>({1,2,3}),
        mreturn<list>(x+y)
    ));
```


■ Implement

```
{x+y | x <- [0 1 2], y <- [1 2 3], even (x+y)}  
{x+y | x <- [0 1 2], even x, y <- [1 2 3]}  
{x+y | x <- [0 1 2], y <- [1 2 3], even y}  
{(x,y) | x <- [0 1 2], y <- [1 2 3]}  
{(x,y) | x <- [0 1 2], y <- [1 2 3], (even x) || (even y)}
```

■ Hint: use a **filter** helper defined like this

```
template<typename Pred>  
auto filter(Pred pred)  
{  
    // return function object that accepts some x and returns  
    // mreturn<list>(x) if pred(x) or an empty list otherwise  
}
```

Recap so far



Recap so far

- Monads are type constructors (aka class templates) along with operations **return** and $>>=$ for $T \rightarrow M<T'>$ function composition
- Monads are a **design pattern** for composition of functions returning **extended types**
 - An extended type $M<T>$ is a “container” with 0 or more T values
- Monad contents can **only** be read via $>>=$
 - This allows for controlled access and execution semantics
- What other value-holding types can be seen as monads?
- **do** notation emulates imperative style in a functional setting
- What about the **monadic laws**, Cap'n?
 - Trust your heart (just kidding)

Continuation Monad (simplified)



- In CPS functions are equipped with a continuation or callback object to feed the result to

```
template<typename C>
auto add(int x,int y,C c){return c(x+y);}

template<typename C>
auto square(int x,C c){return c(x*x);}

template<typename C>
auto pyth(int x,int y,C c) // x*x + y*y
{
    return square(x,[=](int xx){
        return square(y,[=](int yy){
            return add(xx,yy,c);
        });
    });
}

int main()
{
    pyth(3,4,[](int r){std::cout<<r<<"\n";});
}
```

(example adapted from en.wikibooks.org/wiki/Haskell/Continuation_passing_style)

A twist of syntax

- Say we return some suitable continuation object rather than accept one

```
template<typename T>
class cont
{
public:
    cont(const T& x):run_([](){return x;}){}

    T run()const{return run_();}

private:
    std::function<T()> run_;
};

auto add(int x,int y){return cont<int>(x+y);}

auto square(int x){return cont<int>(x*x);}

auto pyth(int x,int y){ /* ?? */ }
```

- How do we compose continuation objects?
- Let's look at the **compositional** interface of **cont**

Exercise

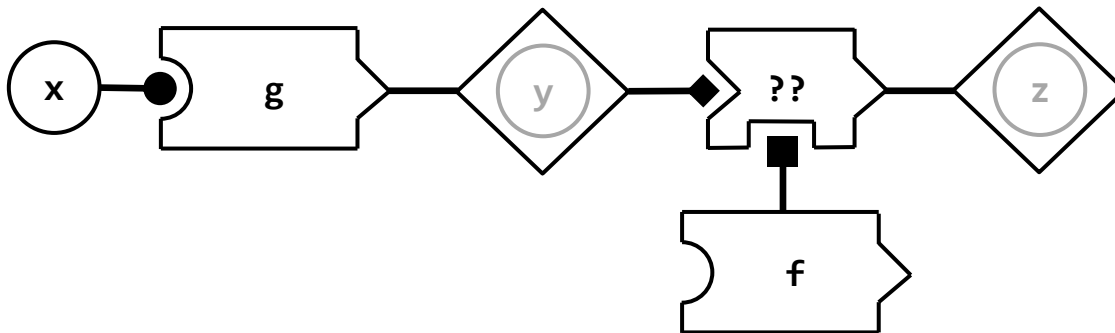
- Figure out the implementation of **cont** composition ctor

```
template<typename T>
class cont
{
public:
    // ...

    template<typename F>
    cont(const cont& c, F f):run_(/* figure this out */){}

    //...

private:
    std::function<T()> run_;
};
```



- I guess you saw it coming 😊

```
template<typename T,typename F>
auto operator>>=(const cont<T>& c, F f)
{
    return cont<T>{c,f};
}
```

```
auto pyth(int x,int y)
{
    return
        DO(xx,square(x),
           DO(yy,square(y),
              add(xx,yy)
            ));
}
```

- Not the real Continuation Monad: we've made a simplifying assumption
- Say which

- Recursively construct a continuation object that computes factorial

```
cont<int> fac(int n);
```

```
int main()
{
    auto f=fac(5);
    std::cout<<"running f\n";
    auto r=f.run();
    std::cout<<"fac(5)="<<r<<"\n";
}
```

- Bonus points: do it lazily (i.e. continuation chain is constructed at `f.run()` time)
- Extra bonus points: do it tail-recursively

Homework (hard)

■ Implement the general Continuation Monad

```
template<typename T,typename R>
class cont;

template<typename T,typename R,typename F>
auto operator>>=(const cont<T>& c, F f);
// f: T -> cont<T',R> with T' not necessarily the same as T
```

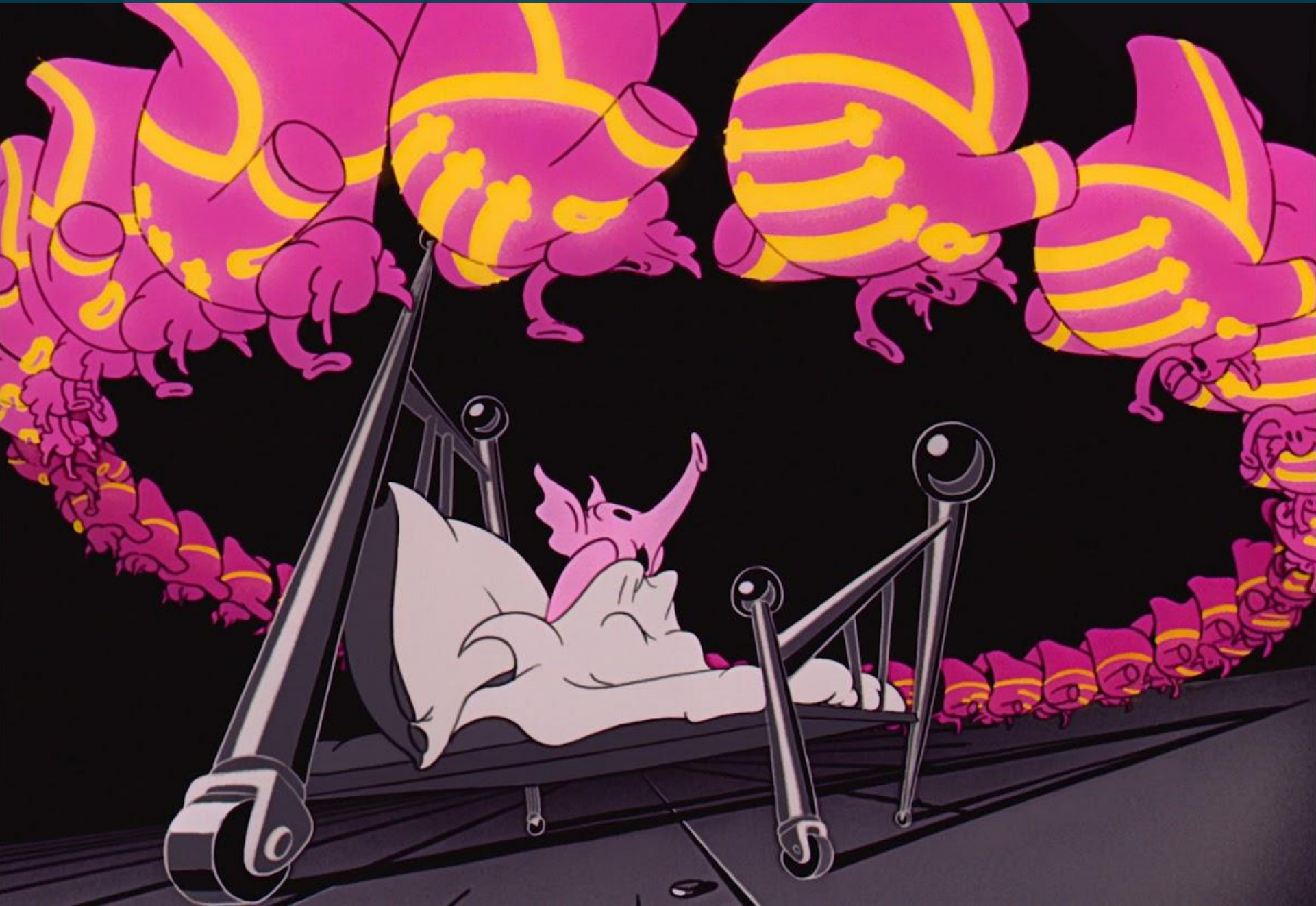
■ Implement a (restricted) continuation monad with stepwise computation

```
template<typename T>
class cont
{
public:
    // ...
    std::pair<T,boost::optional<cont>> step()const;
    // ...
};
```


Discussion

- Why is this monad interesting?
- The mother of all monads
- The Continuation Monad captures execution flow
- Imperative style in non-imperative settings
- Name application scenarios
- Which C++ entities resemble the Continuation Monad?

I see monads everywhere!



I see monads everywhere!

- Monads are a **design pattern** for composition of functions returning **extended types**
- In Haskell, monads + **do** emulate side-effects and imperative style
- In C++, monads are increasingly showing up
 - **std::expected** proposal
 - **then-like** extensions to **std::future**
- Have a look at “Resumable Functions (rev 3)” proposal (N4286)
 - **await** could potentially be extended to act as a monadic **>>=**
- My dream: C++ monadic GUI based on functional reactive programming

If you want to learn more about monads...



If you want to learn more about monads...

- “Monad tutorials online” compiles all relevant Internet stuff by time of publication! **haskell.org/haskellwiki/Monad_tutorials_timeline**
- “All about monads” **haskell.org/haskellwiki/All_About_Monads** is very comprehensive but assumes you know Haskell
 - Take a look at the monad catalog it provides
- Gabriel Gonzalez’s “Haskell for all” **haskellforall.com**
 - Very readable even if you don’t know the language, monads and more
- “Bartosz Milewski’s Programming Cafe” **bartoszmilewski.com** has some monadic stuff as applied to C++
 - “Rediscovering Monads in C++” has many interesting ideas **slideshare.net/sermp/rediscovering-monads**
 - Bartosz’s writing a book on Category Theory for programmers, due 2015
- My blog **bannalia.blogspot.com** has some (maybe not too friendly) articles on monads for C++ and C++ template metaprogramming

A workshop on monads with C++14

Thank you

github.com/joaquintides/cpp14monadworkshop

Meetup C/C++ de Madrid
Joaquín M^a López Muñoz <joaquin@tid.es>
Madrid, January 2015

Telefonica