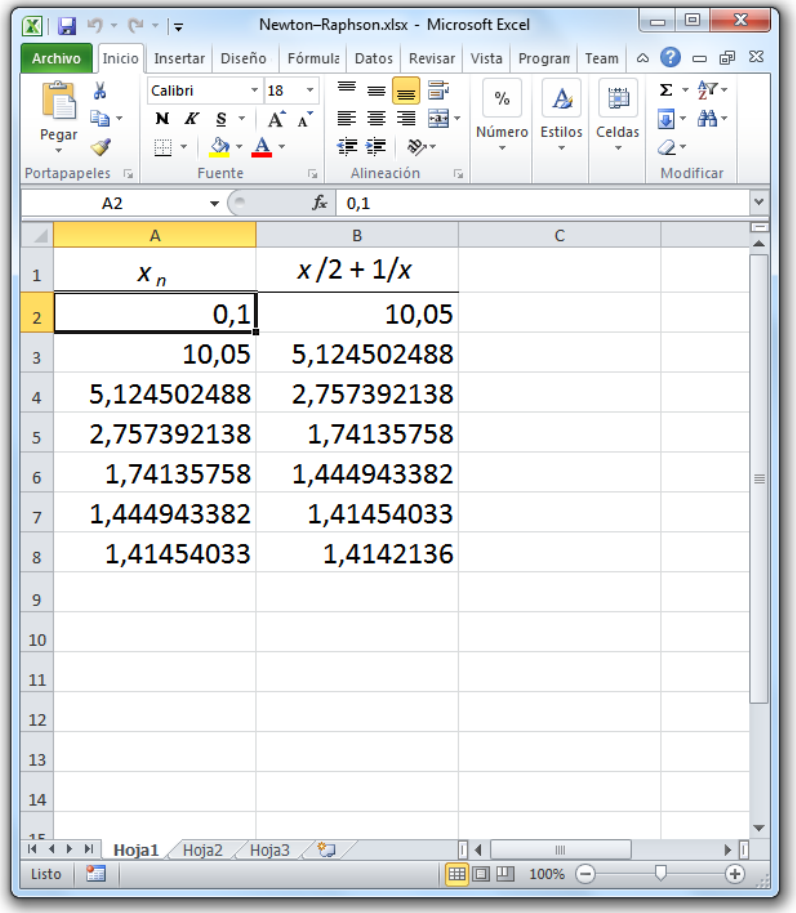
Some fun with Reactive Programming in C++17

`using std::cpp 2019`

Joaquín M López Muñoz <joaquin.lopezmunoz@gmail.com>

Madrid, March 2019

What are these systems doing (most of the time)?



	A	B	C
1	x_n	$x/2 + 1/x$	
2	0,1	10,05	
3	10,05	5,124502488	
4	5,124502488	2,757392138	
5	2,757392138	1,74135758	
6	1,74135758	1,444943382	
7	1,444943382	1,41454033	
8	1,41454033	1,4142136	
9			
10			
11			
12			
13			
14			

usingstdcpp2019::urp product pitch

function_basic.cpp

```
#include <iostream>
#include "urp.hpp"

int main()
{
  using namespace usingstdcpp2019::urp;

  value x=0,y=0;
  auto  z=(x*x)+y+1;

  x=6;
  y=5;
  std::cout<<"z="<<z.get()<<"\n";
}
```

- Guess output

usingstdcpp2019::urp product pitch

function_basic.cpp

```cpp
#include <iostream>
#include "urp.hpp"

int main()
{
  using namespace usingstdcpp2019::urp;

  value x=0,y=0;
  auto  z=(x*x)+y+1;

  x=6;
  y=5;
  std::cout<<"z="<<z.get()<<"\n";
}
```

```
joaquin@machine:~$ ./function_basic

z=42

joaquin@machine:~$
```

usingstdcpp2019::urp product pitch

event_basic.cpp

```cpp
#include <iostream>
#include <string>
#include "urp.hpp"

int main()
{
  using namespace usingstdcpp2019::urp;

  trigger<std::string> s;
  auto n=s|map([](const std::string& s){return s.size();});
  auto e=combine(s,n)
        |filter([](const auto& p){return std::get<1>(p)>=4;})
        |map([](const auto& p){return std::get<0>(p);})
        |accumulate(std::string{},std::plus<>{});
  e.connect([](const auto&,const std::string& str){std::cout<<str<<" ";});

  for(const auto& str:{"welcome","to","using","std","cpp","2019"})s=str;
}
```

usingstdcpp2019::urp product pitch

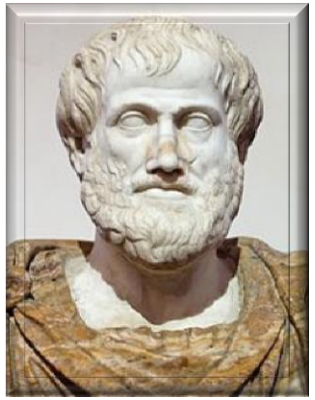
event_basic.cpp

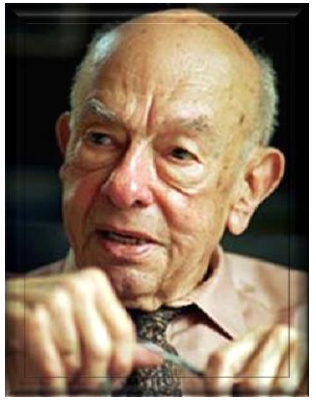
```cpp
#include <iostream>
#include <string>
#include "urp.hpp"

int main()
{
  using namespace usingstdcpp2019::urp;

  trigger<std::string> s;
  auto n=s|map([](const std::string& s){return s.size();});
  auto e=combine(s,n)
        |filter([](const auto& p){return std::get<1>(p)>=4;})
        |map([](const auto& p){return std::get<0>(p);})
        |accumulate(std::string{},std::plus<>{});
  e.connect([](const auto&,const std::string& str){std::cout<<str<<" ";});

  for(const auto& str:{"welcome","to","using","std","cpp","2019"})s=str;
}
```

```
joaquin@machine:~$ ./event_basic
welcome welcomeusing welcomeusing2019
joaquin@machine:~$
```

Reification is the thing

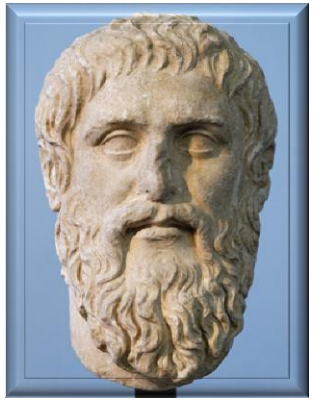
The pragmatic guys

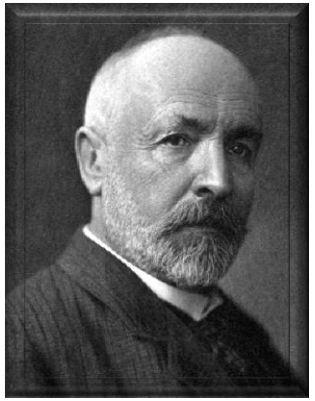
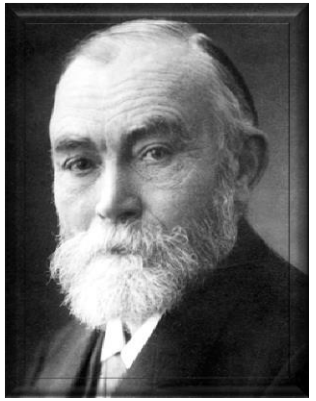
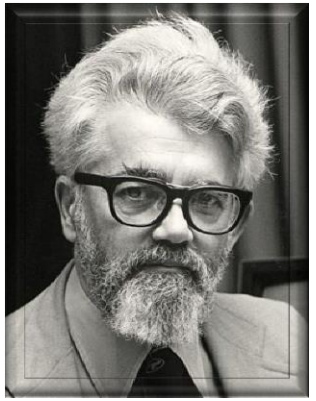


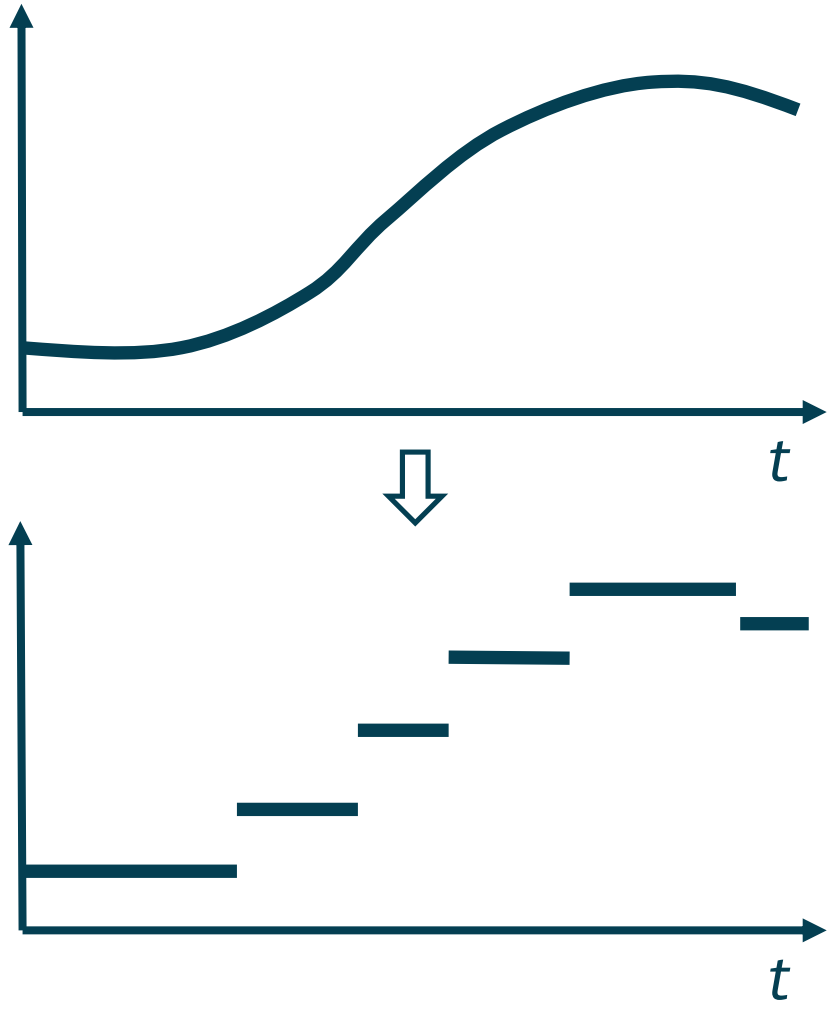

vs

The idealistic bunch

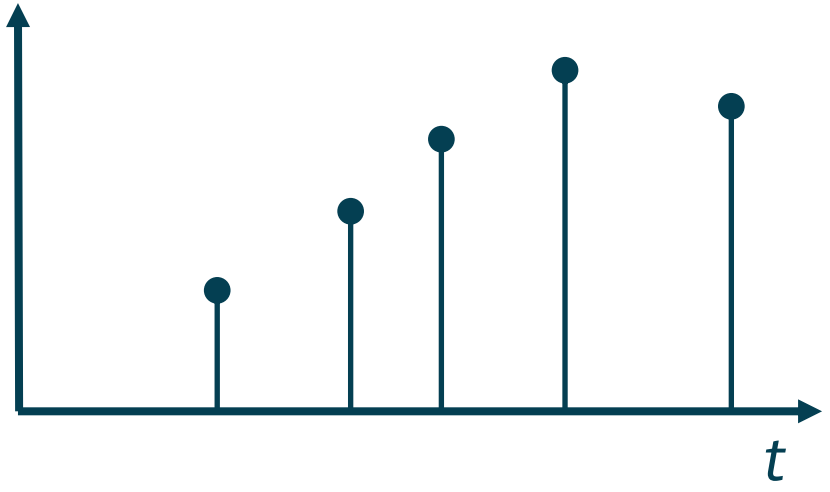


The two (non-competing) models of RP (Elliott and Hudak)



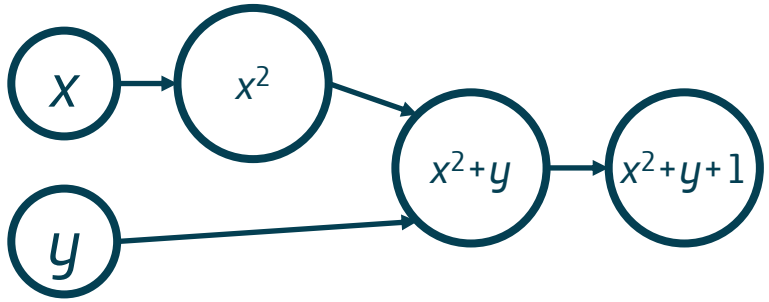
Function
(behavior, signal)



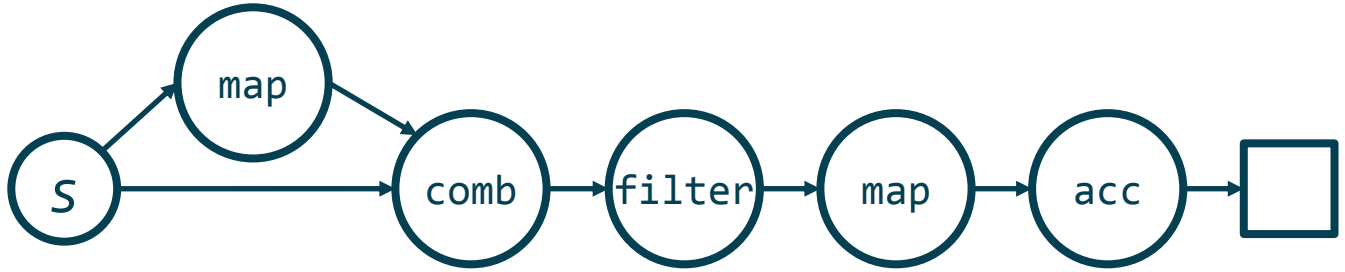
Event
(stream, observable)

Declarative reactivity (Elliott and Hudak)

```
auto z=(x*x)+y+1;
```




```
auto n=s|map(···);
auto e=combine(s,n)
      |filter(···)
      |map(···)
      |accumulate(···);
e.connect(···);
```



- Data dependencies form a Directed Acyclic Graph
- Declarativity: DAG is constructed **implicitly** as we create new nodes
- Is acyclicity guaranteed?

Enter `µrp` implementation

CARTON N.º 1.764 SERIE RP (de 1.944 cartones) 07032019

	decltype	Variadic templates		Move semantics		array tuple		Ref-qualified member functions
Generic lambdas	Lambda capture express-ions			Return type deduction	index_ sequence		plus<>	
CTAD			apply		variant	optional		Variadic fold express-ions

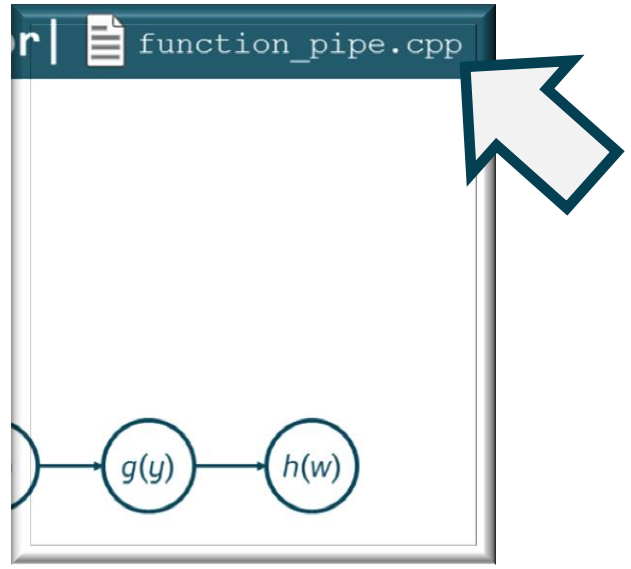
EXTRACTO DE LAS REGLAS AL DORSO

Enter µrp implementation

- We won't cover 100% of µrp
  - Goal is to develop a taste for RP and appreciate C++17 expressive power
- Bumps ahead! Look for these visual aids to the slide flow

C++{11|14|17} exotic features

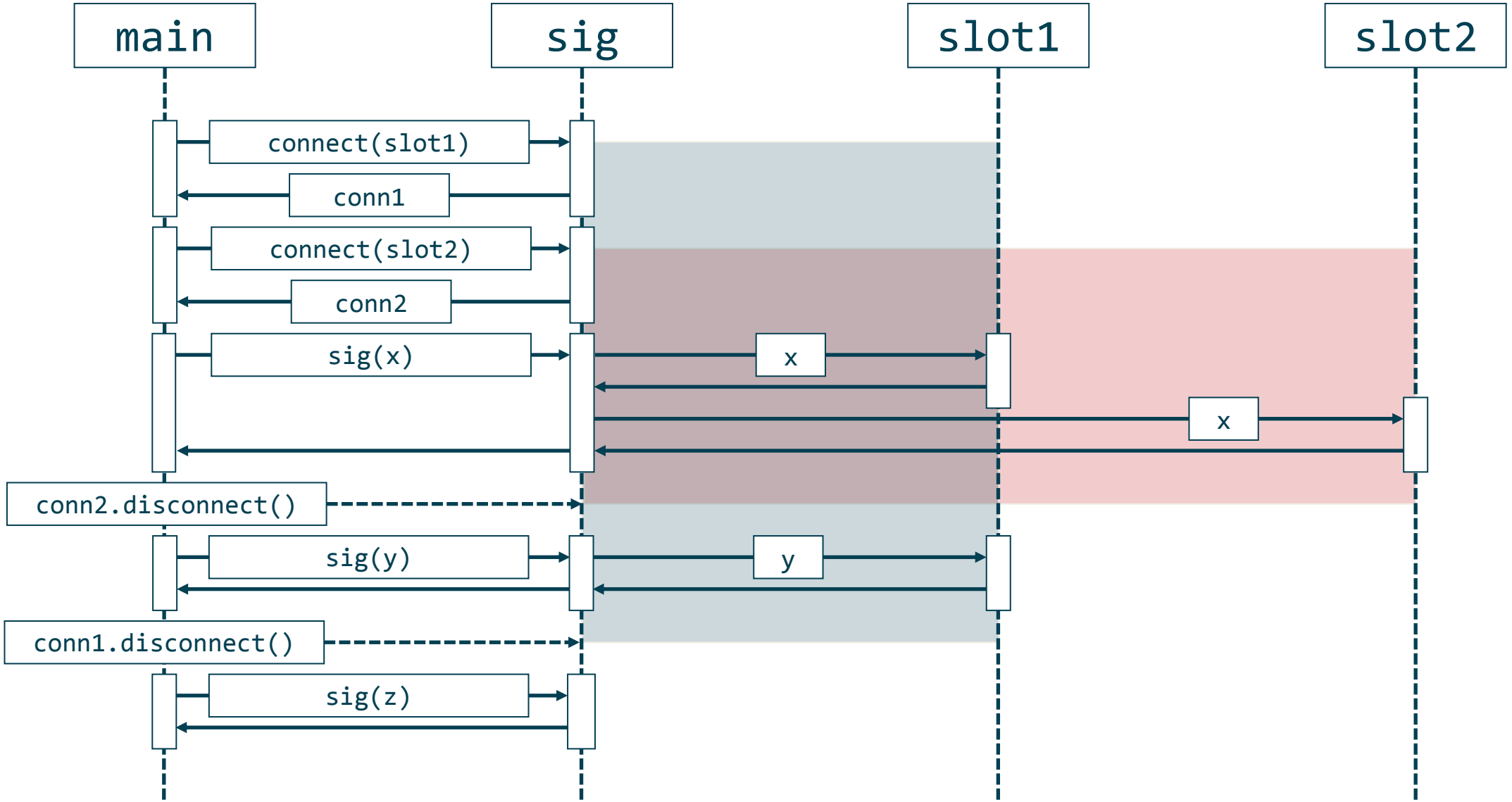
Design insights



Downloadable example code @
github.com/joaquintides/usingstdcpp2019

- First things first: introducing **Boost.Signals2** as µrp's tracking backbone

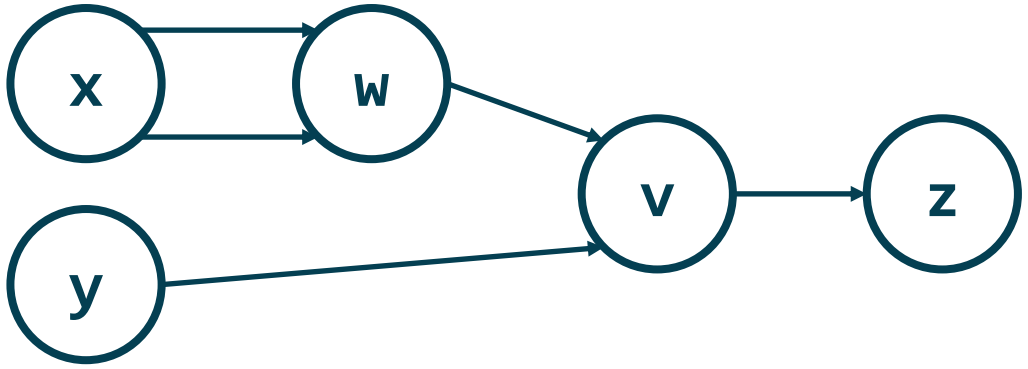
Boost.Signals2 in one slide



Decomposing the snippet

function_decomposed.cpp

```cpp
#include <functional>
#include <iostream>
#include "urp.hpp"

int main()
{
  using namespace usingstdcpp2019::urp;

  value    x=0,y=0;
  function w={std::multiplies<>{},x,x};
  function v={std::plus<>{},w,y};
  function z={[](int v){return v+1;},v};

  // this is functionally equivalent
  function z1={[](int x,int y){return x*x+y+1;},x,y};

  x=6;
  y=5;
  std::cout<<"z ="<<z.get() <<"\n"
           <<"z1="<<z1.get()<<"\n";
}
```



Decomposing the snippet

function_decomposed.cpp

- Class template argument deduction (**CTAD**)

```
value<int>                                                x=0,y=0;
function<std::multiplies<>,value<int>,value<int>> w={std::multiplies<>{},x,x};
function<std::plus<>,decltype(w),decltype(y)>      v={std::plus<>{},w,y};
function<decltype(···),decltype(v)>                z={[](int v){return v+1;},v};
```

  - When deduction doesn't cut it, so-called **deduction guides** can be provided

```
std::vector x{first,last}; // what's value_type?
```

- `std::multiplies<>`? `std::plus<>`?
  - Instantiations of `<functional>` function objects with default `void` work generically for any type
  - Similar to **generic lambdas** with `auto` args

```
std::less<>{}  ↔  [](const auto& x,const auto& y){return x<y;}
```

`value`/`function` public interface (parts omitted)

Underlying value type

Underlying function type

Arguments to the function (can be values or functions)

```
template<typename T>
class value
{
public:
  using value_type=T;
  value(const T& t);

  value& operator=(const T& u);
  value& operator=(T&& u);

  const value_type& get()const;

  template<typename Slot>
  auto connect(const Slot& s);

  template<typename F>
  auto operator|(F f)&;


};
```

Assign value (and propagate change)

Get value

Connect external slots

Function chaining

```
template<typename F,typename... Args>
class function
{
public:
  using value_type=···
  function(F f,Args&... args);

  const value_type& get()const;

  template<typename Slot>
  auto connect(const Slot& s);

  template<typename G>
  auto operator|(G g)&;
  template<typename G>
  auto operator|(G g)&&;
};
```

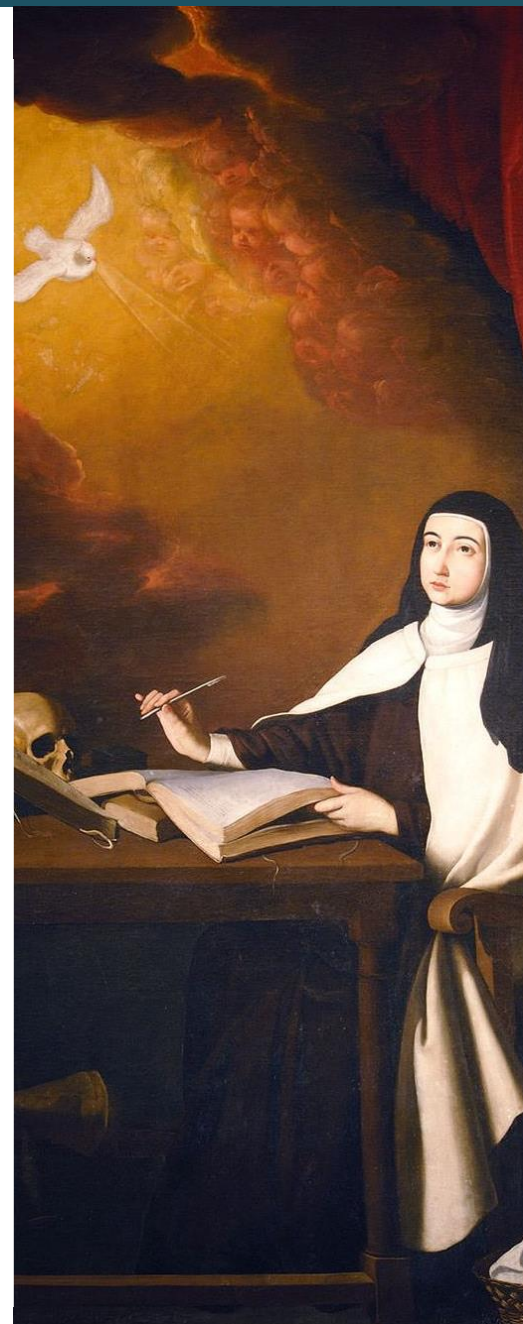
Under the hood

- Factor out DAG tracking and propagation into a **CRTP** base class

User class | Propagated info | Depended-on nodes

```
template<typename Derived,typename Signature,typename... Srcs>
class node
{
  // public connect interface for external slots
  // protected interface to fire propagation and access sources
  // internally calls Derived::callback to transform propagated info
};

template<typename T>
class value:public node<value<T>,void(const value<T>&)>;

template<typename F,typename... Args>
class function:public node<
  function<F,Args...>,void(const function<F,Args...>&),Args...
>;
```

- node is reused in the `trigger/event` part of µrp



Non-dependent (source) **node**

```
template<typename Derived,typename... SigArgs>
class node<Derived,void(SigArgs...)>
{
public:
  template<typename Slot>
  auto connect(const Slot& s)
  {
    return connect_node([=](auto arg){
      std::visit(overloaded{
        [](node*){},
        [=](auto& sigargs){std::apply(s,sigargs);}
      },arg);
    });
  }

protected:
  void signal(SigArgs... sigargs)
  {
    sig(std::forward_as_tuple(
      std::forward<SigArgs>(sigargs)...));
  }

  auto get_srcs()const{return std::tuple{};}

private:
  using extended_signature=void(
    std::variant<
      node*,
      std::tuple<SigArgs...>
    >
  );

  template<typename Slot>
  auto connect_node(const Slot& s)
  {
    return sig.connect(s);
  }

  boost::signals2::signal<
    extended_signature
  >                          sig;
};
```

Non-dependent (source) **node**

- Variadic template multiple inheritance

```
template<typename... Ts> struct overloaded:Ts...{using Ts::operator()...;};
template<typename... Ts> overloaded(Ts...)->overloaded<Ts...>; // ded. guide

auto f=overloaded{
 [](int x)   {std::cout<<"int: "   <<x<<"\n";},
 [](double x){std::cout<<"double: "<<x<<"\n";}
};
f(42);
f(3.14169265);
```

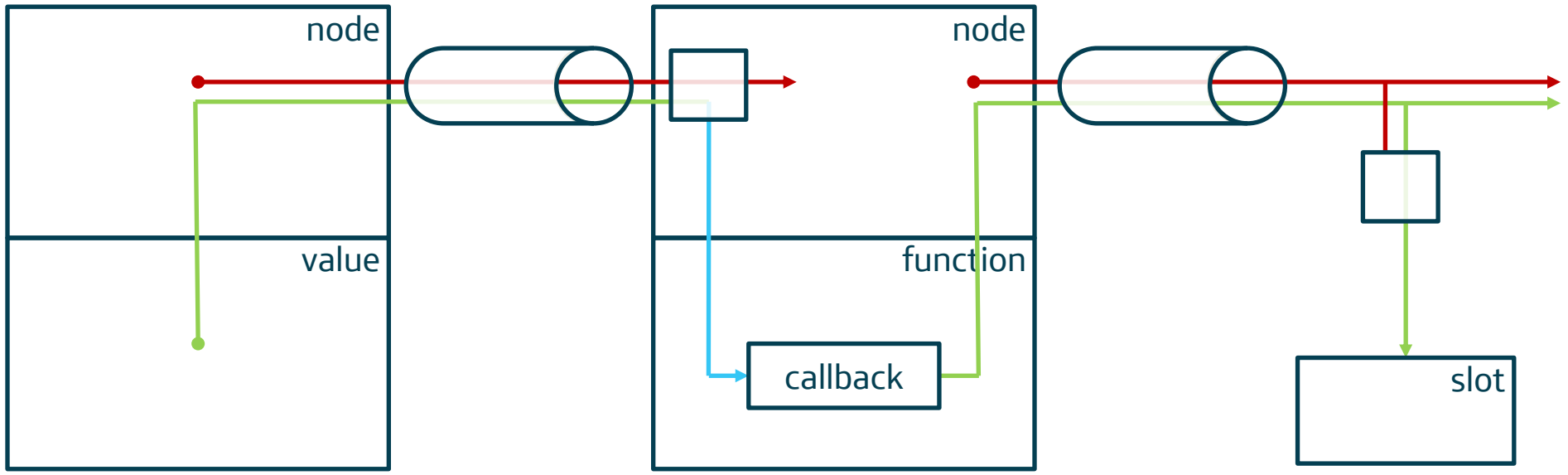
  - `overloaded` useful for `visitation`

- `std::apply`

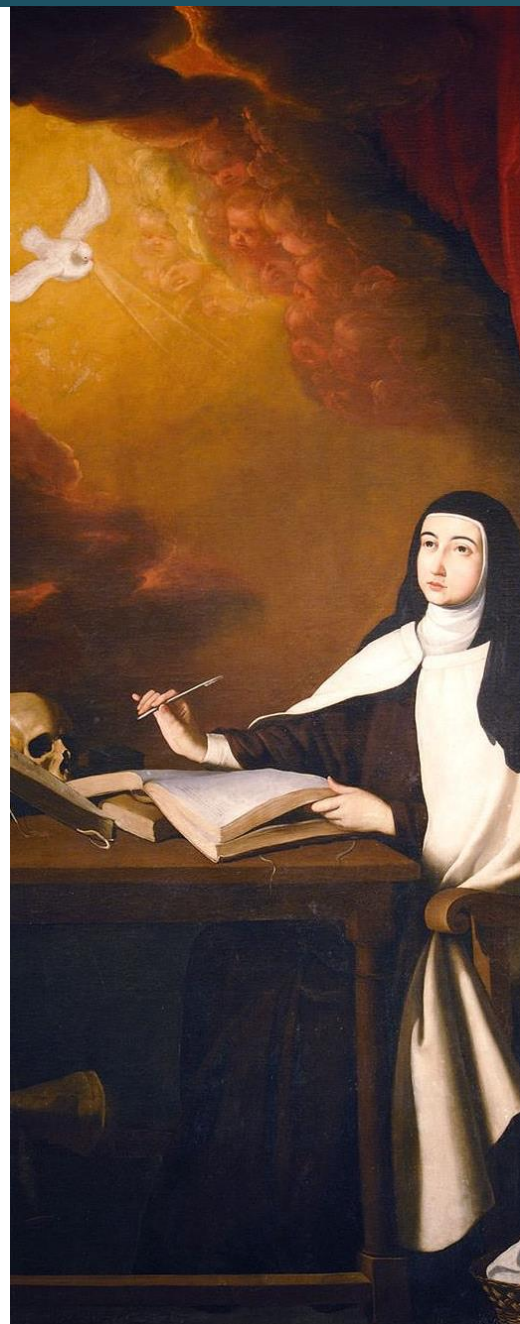
```
std::tuple t={"welcome to %s %d", "using std cpp", 2019};
std::apply(std::printf,t);
```

  - Standard way to pass variadic packs around

Interconnection signals as multilane highways



- Red is for internal node communication (movement signaling)
- Green to blue: adding positional info (source index)
  - Useful later for events
- Scheme can be easily augmented to support error and completion propagation



Dependent node

```
template<typename Derived,typename... SigArgs,typename... Srcs>
class node<Derived,void(SigArgs...),Srcs...>:
  public node<Derived,void(SigArgs...)>
{
public:
 node(Srcs&... srcs):srcs{&srcs...}{}
 ~node(){disconnect_srcs();}

protected:
  auto& get_srcs()const noexcept{return srcs;}

private:
  auto connect_srcs();

  void disconnect_srcs()
  {
    std::apply([](auto&&... conns){(conns.disconnect(),...);},conns);
  }

  std::tuple<Srcs*...>                                  srcs;
  std::array<boost::signals2::connection,sizeof...(Srcs)> conns=connect_srcs();
};
```

Dependent **node**: source management

```
auto connect_srcs(){
  return connect_srcs(std::make_index_sequence<sizeof...(Srcs)>{});
}

template<std::size_t... I>
auto connect_srcs(std::index_sequence<I...>){
  return std::array{
    std::get<I>(srcs)->connect_node([this](auto arg){
      std::visit(overloaded{
        [this](auto* p){
          auto& src=std::get<I>(srcs);
          src=static_cast<std::decay_t<decltype(src)>>(p);
        },
        [this](auto& sigargs){
          std::apply([this](auto&&... sigargs){
            derived().callback(
              node_index_type<I>{},
              std::forward<decltype(sigargs)>(sigargs)...);
          },sigargs);
        }
      },arg);
    })...
  };
}
```

Compile-time iteration

For each source:

Connect and reroute signal to:

Red lane
Source movement

Green lane
Add index and pass to callback

Dependent **node**: source management

- Compile-time iteration

```
template<typename T,std::size_t N,std::size_t... I>
auto inner_product(
  const std::array<T,N>& a1,const std::array<T,N>& a2,
  std::index_sequence<I...>
)
{
  return ((a1[I]*a2[I])+...);
}

template<typename T,std::size_t N>
auto inner_product(const std::array<T,N>& a1,const std::array<T,N>& a2)
{
  return inner_product(a1,a2,std::make_index_sequence<N>{});
}

std::array<int,6> a1={1, 2, 4, 8,16,32},
                  a2={0 ,1, 0, 1, 0, 1};

std::cout<<inner_product(a1,a2)<<"\n";
```

Dependent **node**: source management

- Variadic fold expressions
  - You just saw one

```
template<typename T,std::size_t N,std::size_t... I>
auto inner_product(
  const std::array<T,N>& a1,const std::array<T,N>& a2,
  std::index_sequence<I...>
)
{
  return ((a1[I]*a2[I])+...);
}
```

  - And, if you're really observant, you might have spotted this other one:

```
void disconnect_srcs()
{
  std::apply([](auto&&... conns){(conns.disconnect(),...);},conns);
}
```

  - Enclosing parentheses not optional

With **node** out of the way, **value** is a piece of cake...

```
template<typename T>
class value:public node<value<T>,void(const value<T>&)>;
{
public:
  value& operator=(const T& u)
  {
    if(!(t==u)){
      t=u;
      this->signal(*this);
    }
    return *this;
  }

  const T& get()const noexcept{return t;}

  template<typename F>
  auto operator|(F f)&{return function{f,*this};}

private:
  T t;
};
```

Signal only in case of change

`signal` inherited from node

We'll see this later

...and `function` is not much harder

```
template<typename F,typename... Args>
class function:public node<
  function<F,Args...>,
  void(const function<F,Args...>&),Args...
>
{
public:
  auto const& get()const noexcept{return t;}

  template<typename G>
  auto operator|(G g)&
  {return urp::function{g,*this};}

  template<typename G>
  auto operator|(G g)&&
  {return urp::function{g,std::move(*this)};}

private:
  template<typename Index,typename Arg>
  void callback(Index,const Arg&){update();}

  void update()
  {
    if(const auto& u=value();!(t==u)){
      t=u;
      this->signal(*this);
    }
  }

  auto value()const
  {
    return std::apply(
      [this](const auto&... args){
        return f(args->get()...);
      },
      this->get_srcs());
  }

  F           f;
  value_type t=value();
};
```

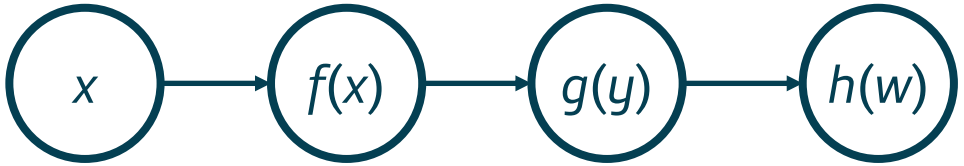
The long-ignored and as of lately much-abused operator|

function_pipe.cpp

```
value x=0;

auto f=[](int x){return x+1;};
auto g=[](int x){return 2*x;};
auto h=[](int x){return x*(x+1);};
```

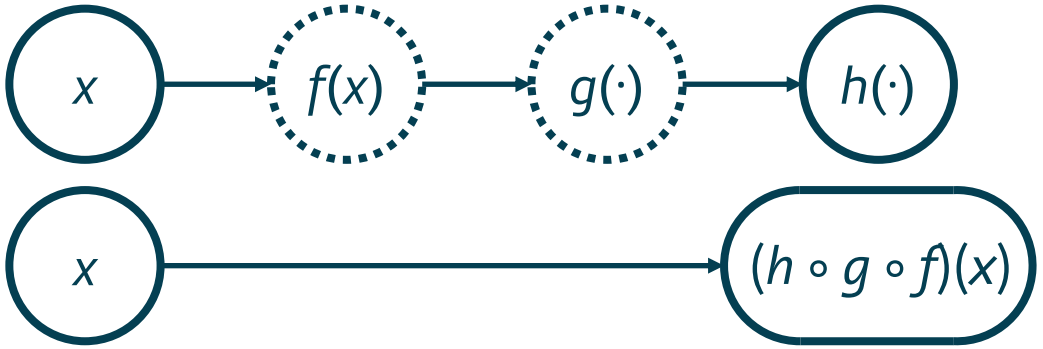
- operator| as syntactic convenience

```
auto  y=x|f;
auto  w=y|g;
auto  z=w|h;

auto  y1=function{f,x};
auto  w1=function{g,y1};
auto  z1=function{h,w1};
```



- Where have all the temporaries gone?

```
auto  z2=x|f|g|h;

auto  z3=function{h,function{g,function{f,x}}};
```



The long-ignored and as of lately much-abused **operator|** function_pipe.cpp

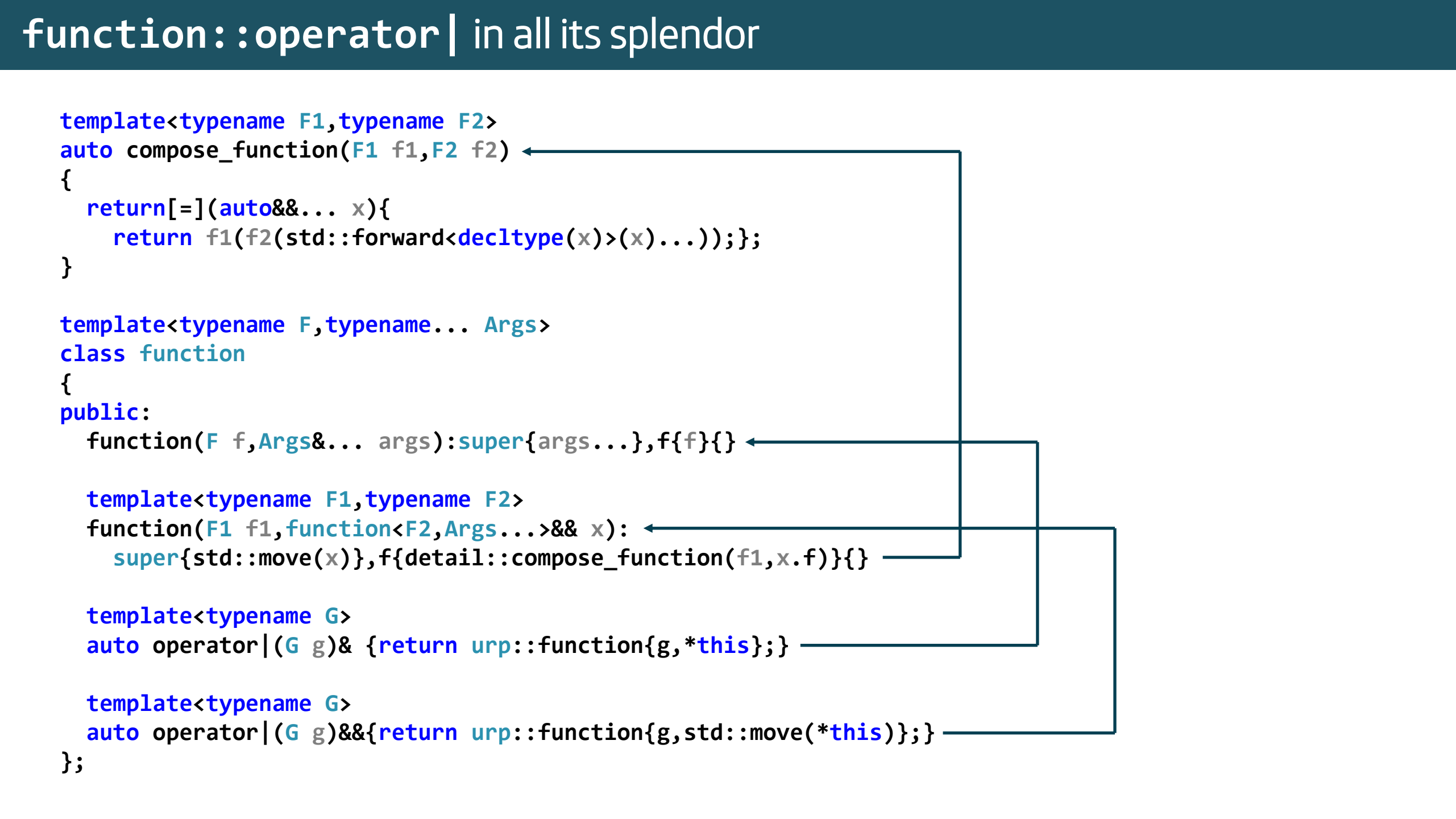
- Ref-qualified member functions

```
struct entity
{
  void manifest()&&{std::cout<<"I'm temporary yet\n";}
  void manifest()& {std::cout<<"I have a name\n";}
};

entity{}.manifest();
entity x;
x.manifest();
```

  - &&-qualified member functions not much useful unless they return `*this`...
  - ...or do a switcheroo

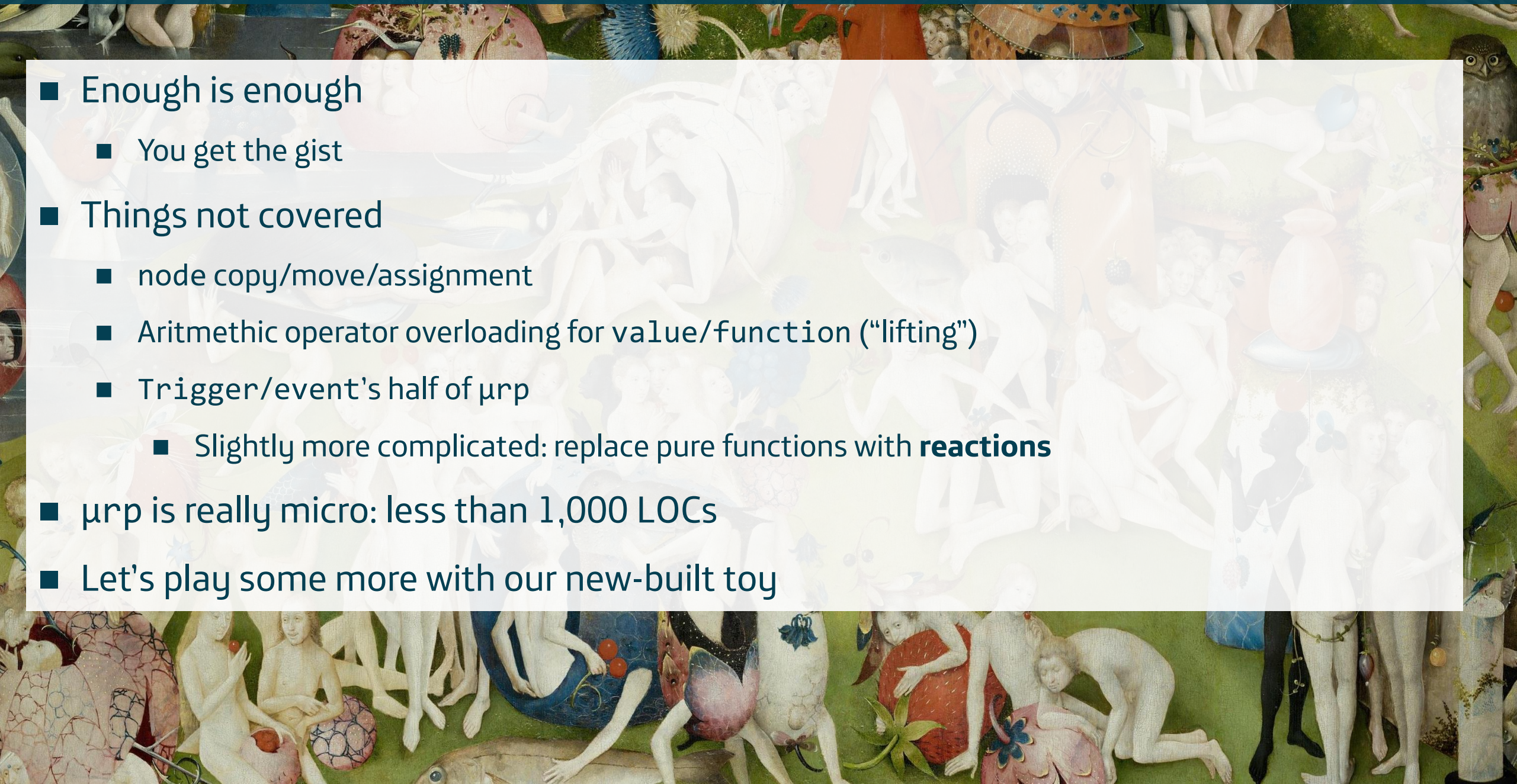
function::operator| in all its splendor

```
template<typename F1,typename F2>
auto compose_function(F1 f1,F2 f2)
{
  return[=](auto&&... x){
    return f1(f2(std::forward<decltype(x)>(x)...));};
}

template<typename F,typename... Args>
class function
{
public:
  function(F f,Args&... args):super{args...},f{f}{}

  template<typename F1,typename F2>
  function(F1 f1,function<F2,Args...>&& x):
    super{std::move(x)},f{detail::compose_function(f1,x.f)}{}

  template<typename G>
  auto operator|(G g)& {return urp::function{g,*this};;}

  template<typename G>
  auto operator|(G g)&&{return urp::function{g,std::move(*this)};;}
};
```



Back from `µrp` implementation

Back from `µrp` implementation

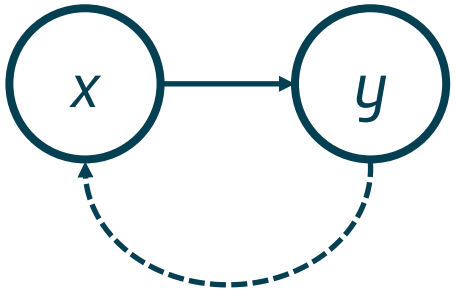


- Enough is enough
  - You get the gist
- Things not covered
  - node copy/move/assignment
  - Aritmethic operator overloading for `value/function` (“lifting”)
  - `Trigger/event`’s half of `µrp`
    - Slightly more complicated: replace pure functions with **reactions**
- `µrp` is really micro: less than 1,000 LOCs
- Let’s play some more with our new-built toy

Using **connect** for devious purposes

newton_raphson.cpp

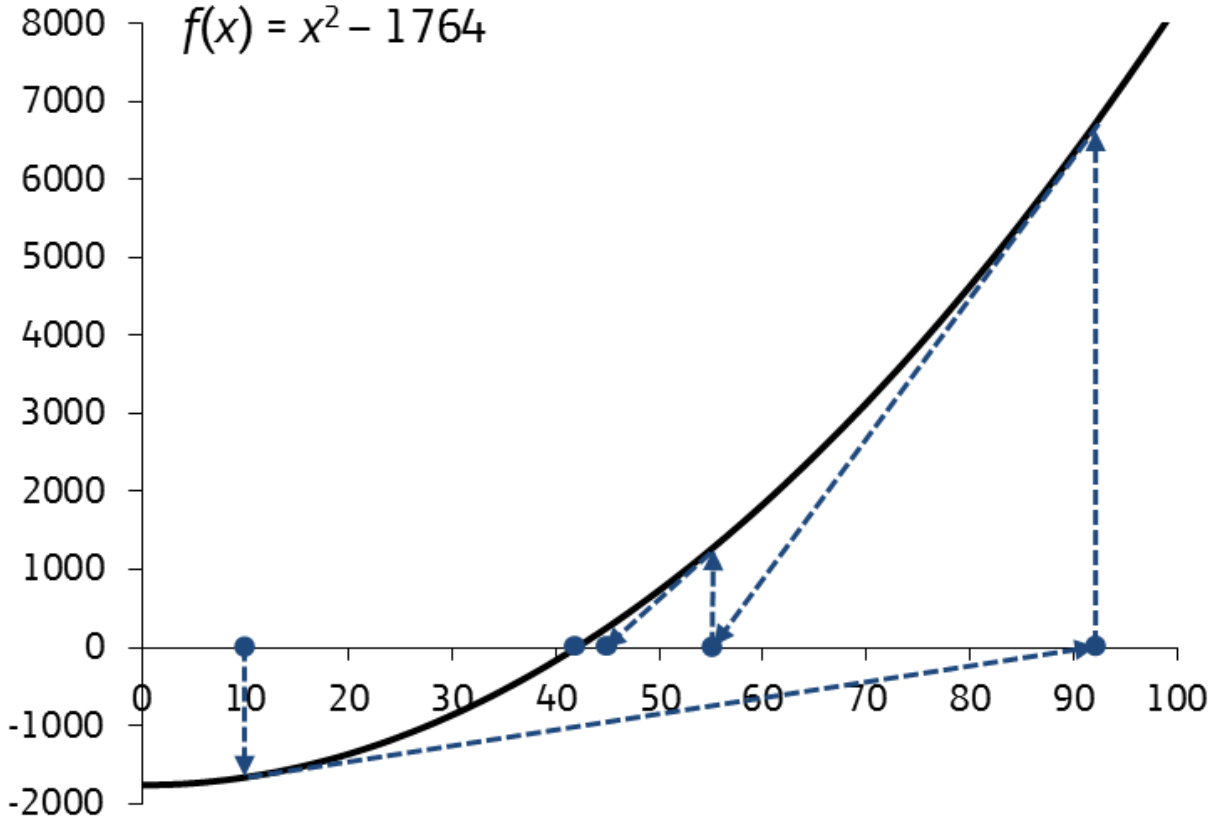
```
#include <iostream>
#include "urp.hpp"

int main()
{
  using namespace usingstdcpp2019::urp;

  value x=1.0;
  auto  y=x/2+882/x;

  // create a cycle
  y.connect([&](const auto& y){x=y.get();});

  x=10.0; // should segfault, right?
  std::cout<<"y="<<y.get()<<"\n";
}
```



Using **connect** for devious purposes

newton_raphson.cpp

```cpp
#include <iostream>
#include "urp.hpp"

int main()
{
  using namespace usingstdcpp2019::urp;

  value x=1.0;
  auto  y=x/2+882/x;

  // create a cycle
  y.connect([&](const auto& y){x=y.get();});

  x=10.0; // should segfault, right?
  std::cout<<"y="<<y.get()<<"\n";
}
```




```
joaquin@machine:~$ ./newton_raphson
y=42
joaquin@machine:~$
```

Complex DAGs

matrix.cpp

```
using matrix2x2=std::array<std::array<value<double>,2>,2>;

auto mult=[](auto& m,auto& n){
  auto& [m00,m01]=std::get<0>(m);
  auto& [m10,m11]=std::get<1>(m);
  auto& [n00,n01]=std::get<0>(n);
  auto& [n10,n11]=std::get<1>(n);

  return std::tuple{
    std::tuple{m00*n00+m01*n10,m00*n01+m01*n11},
    std::tuple{m10*n00+m11*n10,m10*n01+m11*n11}
  };
};

auto print=[](const auto& m){
  auto& [m00,m01]=std::get<0>(m);
  auto& [m10,m11]=std::get<1>(m);
  std::cout<<
    "/ " <<m00.get()<<"\t"<<m01.get()<<"\n"<<
    "\\ "<<m10.get()<<"\t"<<m11.get()<<"\n";
};
```

$$\begin{bmatrix} m_{00} & m_{01} \\ m_{10} & m_{11} \end{bmatrix} \times \begin{bmatrix} n_{00} & n_{01} \\ n_{10} & n_{11} \end{bmatrix} =$$

$$\begin{bmatrix} m_{00}n_{00} + m_{01}n_{10} & m_{00}n_{01} + m_{01}n_{11} \\ m_{10}n_{00} + m_{11}n_{10} & m_{10}n_{01} + m_{11}n_{11} \end{bmatrix}$$

Complex DAGs

matrix.cpp

```
using matrix2x2=std::array<std::array<value<double>,2>,2>;

auto mult=[](auto& m,auto& n){···};

auto print=[](const auto& m){···};

matrix2x2 m={{ {  6.0, 14.0},
              { 21.0, 28.0} }},
          n={{ {-12.0,  6.0},
              {  9.0, -3.0} }};
auto      p=mult(m,n);

print(p);
m[0][0]=7.0;
print(p);
```

$$\begin{bmatrix} m_{00} & m_{01} \\ m_{10} & m_{11} \end{bmatrix} \times \begin{bmatrix} n_{00} & n_{01} \\ n_{10} & n_{11} \end{bmatrix} =$$

$$\begin{bmatrix} m_{00}n_{00} + m_{01}n_{10} & m_{00}n_{01} + m_{01}n_{11} \\ m_{10}n_{00} + m_{11}n_{10} & m_{10}n_{01} + m_{11}n_{11} \end{bmatrix}$$

Complex DAGs

matrix.cpp

```
using matrix2x2=std::array<std::array<value<double>,2>,2>;

auto mult=[](auto& m,auto& n){···};

auto print=[](const auto& m){···};

matrix2x2 m={{ {  6.0, 14.0},
               { 21.0, 28.0} }},
          n={{ {-12.0,  6.0},
               {  9.0, -3.0} }};
auto      p=mult(m,n);

print(p);
m[0][0]=7.0;
print(p);
```

$$\begin{bmatrix} m_{00} & m_{01} \\ m_{10} & m_{11} \end{bmatrix} \times \begin{bmatrix} n_{00} & n_{01} \\ n_{10} & n_{11} \end{bmatrix} =$$

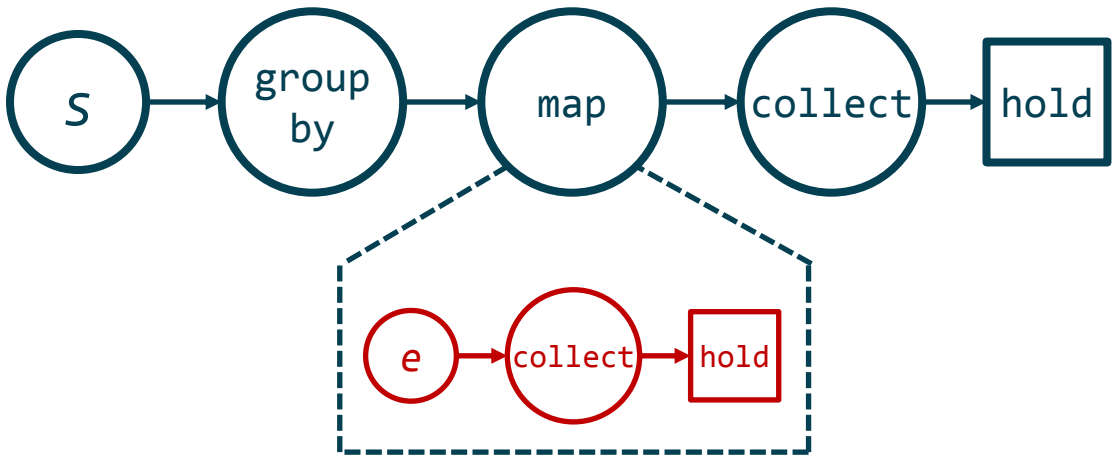
$$\begin{bmatrix} m_{00}n_{00} + m_{01}n_{10} & m_{00}n_{01} + m_{01}n_{11} \\ m_{10}n_{00} + m_{11}n_{10} & m_{10}n_{01} + m_{11}n_{11} \end{bmatrix}$$

```
joaquin@machine:~$ ./matrix
/ 54  -6
\ 0   42
/ 42  0
\ 0   42

joaquin@machine:~$
```

Higher-order events

classify.cpp

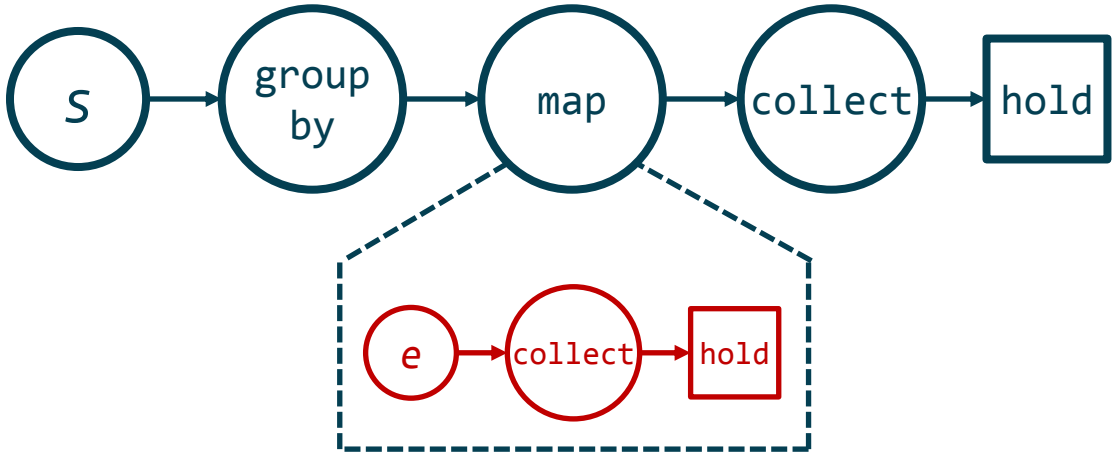
```
trigger<std::string> s;
auto res=hold(
  s|group_by([](const auto& str){return str[0];})
   |map([](auto e){
     return hold(std::move(e)|collect());
   })
   |collect()
);

auto names={
  "John","Jack","Susan","Mary","Anne","Anthony",
  "Bjarne","Margaret","George","Barack","Sarah",
  "Peter","Hillary","Ronda","Alice","Herbert",
};
for(const auto& str:names)s=str;

for(const auto& e:res.get()){
  for(const auto& str:e.get())std::cout<<str<<" ";
  std::cout<<"\n";
}
```



Higher-order events

classify.cpp

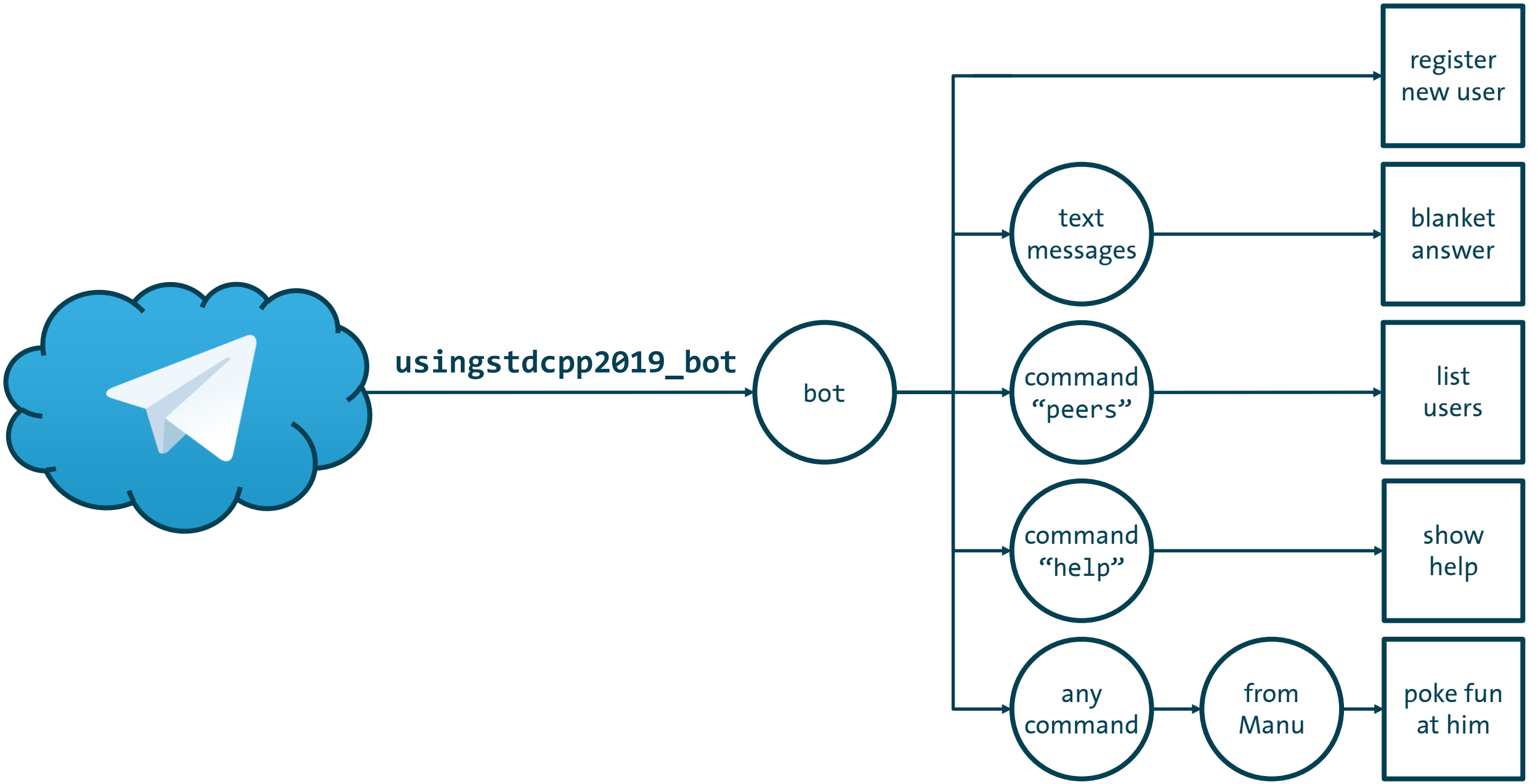
```
trigger<std::string> s;
auto res=hold(
  s|group_by([](const auto& str){return str[0];})
   |map([](auto e){
     return hold(std::move(e)|collect());
   })
   |collect()
);

auto names={
  "John","Jack","Susan","Mary","Anne","Anthony",
  "Bjarne","Margaret","George","Barack","Sarah",
  "Peter","Hillary","Ronda","Alice","Herbert",
};
for(const auto& str:names)s=str;

for(const auto& e:res.get()){
  for(const auto& str:e.get())std::cout<<str<<" ";
  std::cout<<"\n";
}
```




```
joaquin@machine:~$ ./classify

John Jack
Susan Sarah
Mary Margaret
Anne Anthony Alice
Bjarne Barack
George
Peter
Hillary Herbert
Ronda

joaquin@machine:~$
```

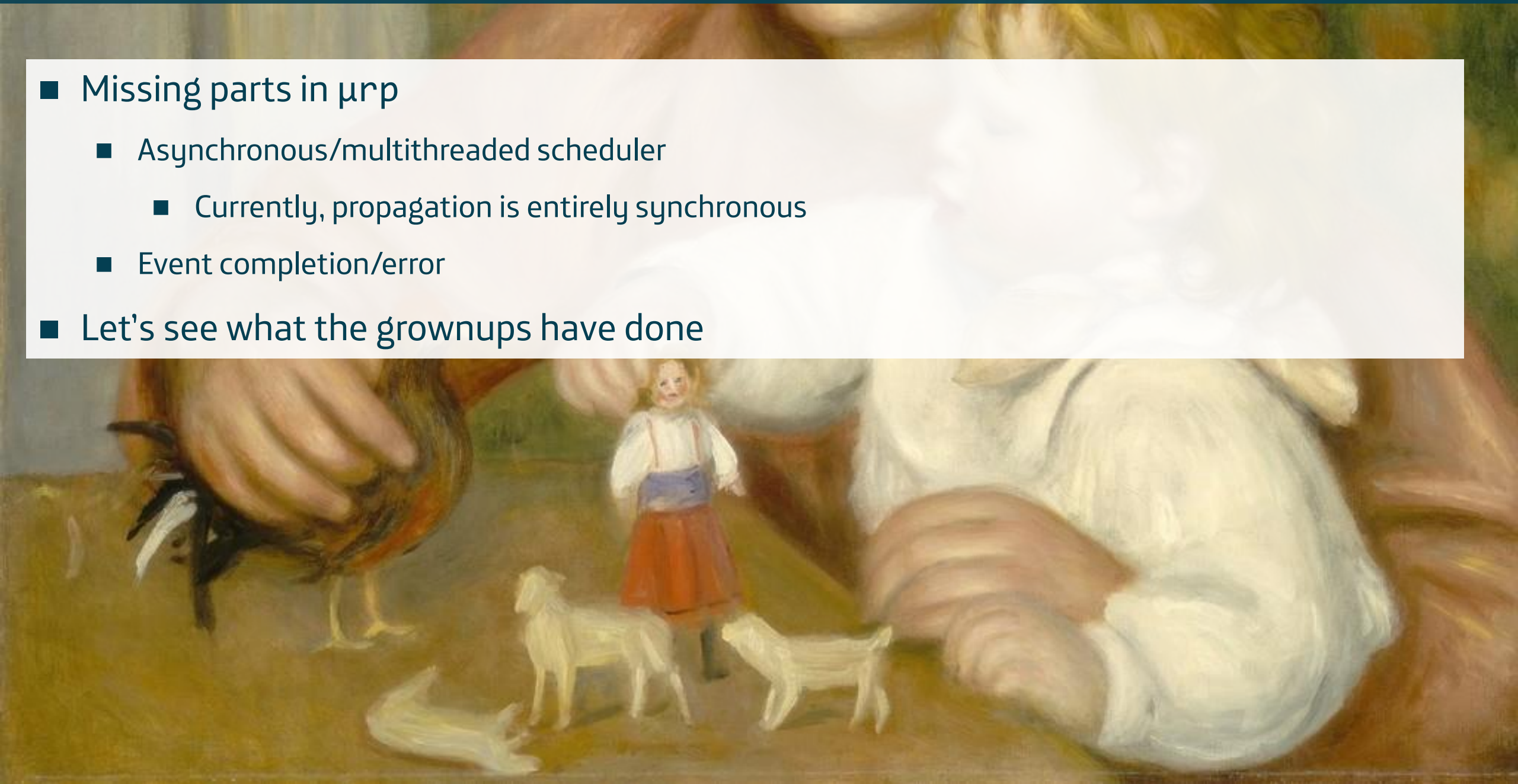
Reactive Telegram bot (from Manu343726)

`telegram-reactive`



Toys are just for fun

Toys are just for fun



- Missing parts in μrp
  - Asynchronous/multithreaded scheduler
    - Currently, propagation is entirely synchronous
  - Event completion/error
- Let's see what the grownups have done

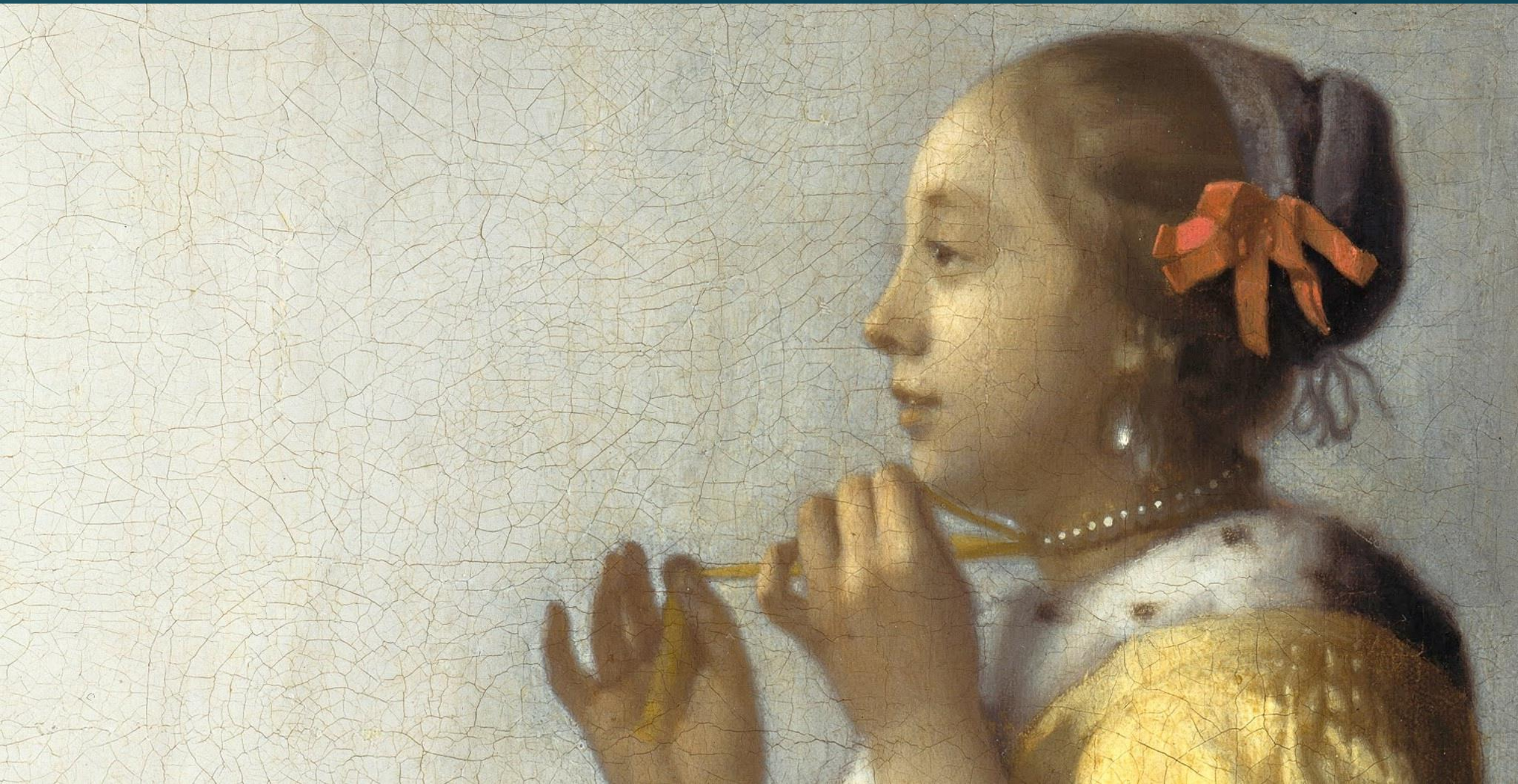
C++React

- C++11
- Concepts after Maier: *Deprecating the Observer Pattern with Scala.React* (2012)
- Supports functions (“signals”) and events (“event streams”)
  - Doesn’t support event termination/error
- Highly customizable per-domain scheduler
  - Uses Intel TBB for concurrency
- Looks semi-abandoned in the process of rewriting for C++14
  - Last commit Nov 2017

RxCpp

- C++11
- Part of ReactiveX multilanguage spec
  - Brainchild of Erik Meijer of LINQ fame, open sourced by Microsoft in 2012
- Only supports events (“observables”)
  - Full termination/error support
  - Conflates push and pull paradigms via “hot” vs. “cold” (~C++20 ranges) observables
  - Somewhat clunky emission start management (`publish`, `ref_count`, `share`)
- Opt-in scheduling on connection (“subscription”) time
- Feels a bit too .NETish and LINQish
  - Not a bad thing if you come from there
- Massively supported and documented

Recap and go



Recap and go

- RP is about incoming data streams propagating through a dependency graph
- Expressive power through reification
  - Graph nodes → composable functions and events
  - Control flow → declarative reactivity
- usingstdcpp2019::µrp as an experiment in lib prototyping with C++17
  - Boost.Signals2: reliable, undemanding, easy-to-use signal/slot library
  - Writing highly generic code in C++17 is pure joy
  - Automatic return type deduction, generic lambdas, CTAD, vocabulary types
- Industry-grade C++ libraries for RP: C++React, RxCpp
- Try RP yourself, go read some more
- Let's be careless out there

Some fun with Reactive Programming in C++17

Thank you

github.com/joaquintides/usingstdcpp2019

`using std::cpp 2019`
Joaquín M López Muñoz <joaquin.lopezmunoz@gmail.com>
Madrid, March 2019