Higher order functions for the rest of us

Björn Fahller

```
compose([](auto const& s) { return s == "foo";},
        std::mem_fn(&foo::name))
```

Definition

A **higher-order function** is a function that takes other functions as arguments or returns a function as result.

```
#include <algorithm>
#include <vector>

std::vector<int> v;

...

if (std::none_of(std::begin(v), std::end(v),
                 [](int x) { return x == 0; })) {
...
}
```

```cpp
#include <algorithm>
#include <vector>

std::vector<int> v;

...

if (std::none_of(std::begin(v), std::end(v),
                 [](int x) { return x == 0; })) {
```

```cpp
template <typename Iterator, typename Predicate>
bool none_of(Iterator i, Iterator e, Predicate predicate)
{
  while (i != e)
  {
    if (predicate(*i)) return false;
    i++;
  }
  return true;
}
```

```
#include <algorithm>
#include <vector>

std::vector<int> v;

...

if (std::none_of(std::begin(v), std::end(v),
                 [](int x) { return x == 0; })) {
...
}

...
int num;
...
while (std::any_of(std::begin(v), std::end(v),
                   [num](int x){ return x == num; })) {
...
}
```

```
[num](int x){ return x == num; }
```

```
auto equals(int num)
{
   return         [num](int x){ return x == num; };
}
```

```
template <typename T>
auto equals(T num)
{
   return [num](auto const& x){ return x == num; };
}
```

```
#include <algorithm>
#include <vector>

std::vector<int> v;

...

if (std::none_of(std::begin(v), std::end(v), equals(0)) {
...
}

...
int num;
...
while (std::any_of(std::begin(v), std::end(v), equals(num)) {
...
}
```

```
struct ip
{
  ip(uint8_t i1, uint8_t i2, uint8_t i3, uint8_t i4)
  : num((uint32_t(i1) << 24) | (uint32_t(i2) << 16) | (uint32_t(i3) << 8)  | i4) {}
  ip(uint32_t n) : num(n) {}

  bool operator==(ip rh) const { return num == rh.num;}
  bool operator!=(ip rh) const { return !(*this == rh);}

  uint32_t num;
};

struct netmask : ip
{
  using ip::ip;
};

inline ip operator&(ip lh, netmask rh)
{
  return {lh.num & rh.num};
};
```

```
struct ip
{
  ip(uint8_t i1, uint8_t i2, uint8_t i3, uint8_t i4)
  : num((uint32_t(i1) << 24) | (uint32_t(i2) << 16) | (uint32_t(i3) << 8)  | i4) {}
  ip(uint32_t n) : num(n) {}

  bool operator==(ip rh) const { return num == rh.num;}
  bool operator!=(ip rh) const { return !(*this == rh);}

  uint32_t num;
};

struct netmask : ip
{
  using ip::ip;
};

inline ip operator&(ip lh, netmask rh)
{
  return {lh.num & rh.num};
};
```

```
struct ip
{
  ip(uint8_t i1, uint8_t i2, uint8_t i3, uint8_t i4)
  : num((uint32_t(i1) << 24) | (uint32_t(i2) << 16) | (uint32_t(i3) << 8)  | i4) {}
  ip(uint32_t n) : num(n) {}

  bool operator==(ip rh) const { return num == rh.num;}
  bool operator!=(ip rh) const { return !(*this == rh);}

  uint32_t num;
};

struct netmask : ip
{
  using ip::ip;
};

inline ip operator&(ip lh, netmask rh)
{
  return {lh.num & rh.num};
};
```

```
struct ip
{
  ip(uint8_t i1, uint8_t i2, uint8_t i3, uint8_t i4)
  : num((uint32_t(i1) << 24) | (uint32_t(i2) << 16) | (uint32_t(i3) << 8)  | i4) {}
  ip(uint32_t n) : num(n) {}

  bool operator==(ip rh) const { return num == rh.num;}
  bool operator!=(ip rh) const { return !(*this == rh);}

  uint32_t num;
};

struct netmask : ip
{
  using ip::ip;
};

inline ip operator&(ip lh, netmask rh)
{
  return {lh.num & rh.num};
};
```

```
struct ip
{
  ip(uint8_t i1, uint8_t i2, uint8_t i3, uint8_t i4)
  :                                                                   ) {}
  ip

  bo
  bo

  ui
};
```

```
auto ip_matches(ip desired,
                netmask mask = netmask{255,255,255,255})
{
  return [desired, mask](ip actual)
         {
           return (desired & mask) == (actual & mask);
         };
}
```

```
struct netmask : ip
{
  using ip::ip;
};

inline ip operator&(ip lh, netmask rh)
{
  return {lh.num & rh.num};
};
```

```
struct ip
{
  ip(uint8_t i1, uint8_t i2, uint8_t i3, uint8_t i4)
  :                                                   ) {}
  ip

  bo
  bo

  ui
};

struct netmask : ip
{
  us
};

inline ip operator&(ip lh, netmask rh)
{
  return {lh.num & rh.num};
};
```

```
auto ip_matches(ip desired,
                netmask mask = netmask{255,255,255,255})
{
  return [desired, mask](ip actual)
         {
           return (desired & mask) == (actual & mask);
         };
}
```

```
std::vector<ip> v;
...
auto i = std::remove_if(v.begin(), v.end(),
                        ip_matches({192,168,1,1}, {255,255,0,0}));
```

```
class ipif
{
public:
  using state_type = enum { off, on };
...
  void       set_state(state_type);

  state_type state() const { return state_; }
  ip         addr() const  { return addr_;}
  netmask    mask() const  { return mask_; }
  ip         gw() const    { return gw_; }
private:
  ip addr_;
  netmask mask_;
  ip gw_;
  state_type state_;
};
```

```
class ipif
{
public:
  using state_type = enum { off, on };
...
  void       set_state(state_type);

  state_type state() const { return state_; }
  ip         addr() const  { return addr_;}
  netmask    mask() const  { return mask_; }
  ip         gw() const    { return gw_; }
private:
  ip addr_;
  netmask mask_;
  ip gw_;
  state_type state_;
};
```

```
class ipif
{
public:
  using state_type = enum { off, on };
...
  void       set_state(state_type);

  state_type state() const { return state_; }
  ip         addr() const  { return addr_;}
  netmask    mask() const  { return mask_; }
  ip         gw() const    { return gw_; }
private:
  ip addr_;
  netmask mask_;
  ip gw_;
  state_type state_;
};
```

To match, for example the address of an **`ipif`**, we need to make the **`ip_matches()`** predicate work on a member.

```
class ipif
{
public:
  using s
...
  void

  state_t
  ip
  netmask
  ip
private:
  ip addr
  netmask
  ip gw_
  state_type state_;
};
```

Given:

f1(y) -> z

and

f2(x) -> y

We want a composition f(x)->z as f1(f2(x))

```
class ipif
{
public:
  using
...
  void

  state_
  ip
  netmas
  ip
private:
  ip add
  netmas
  ip gw_
  state_type state_;
};
```

Given:

f1(y) -> z   ip_matches(ip) -> bool

and

f2(x) -> y   select_addr(ipif) -> ip

We want a composition f(x)->z as f1(f2(x))

```
class ipif
{
public:
  using s
...
  void

  state_t
  ip
  netmask
  ip
private:
  ip addr
  netmask
  ip gw_
  state_type state_;
};
```

Given:

f1(y) -> z   ip_matches(ip) -> bool

and

f2(x) -> y   select_addr(ipif) -> ip

We want a composition f(x)->z as f1(f2(x))

```
template <typename F1, typename F2>
auto compose(F1 f1, F2 f2)
{
  return [=](const auto& x) { return f1(f2(x)); };
}
```

```
class ipif
{
public:
...
  ip          addr() const { return addr_;}
...
private:
  ip addr;
  ...
};
```

```
std::vector<ipif> interfaces;

auto i = std::find_if(interfaces.begin(), interfaces.end(),
                      compose(ip_matches({192,168,1,1}),
                              select_addr));
```

```
class ipif
{
public:
...
  ip         addr() const { return addr_;}
...
private:
  ip addr;
  ...
};
```

```
ip select_addr(ipif const& interface)
{
  return interface.addr();
}
```

```
std::vector<ipif> interfaces;

auto i = std::find_if(interfaces.begin(), interfaces.end(),
                      compose(ip_matches({192,168,1,1}),
                              select_addr));
```

```
class ipif
{
public:
...
  ip         addr() const { return addr_;}
...
private:
  ip addr;
  ...
};

auto addr_matches(ip addr, netmask mask = {255,255,255,255})
{
  return compose(ip_matches(addr, mask),
                 select_addr);
}

std::vector<ipif> interfaces;

auto i = std::find_if(interfaces.begin(), interfaces.end(),
                      compose(ip_matches({192,168,1,1}),
                              select_addr);
```

```
class ipif
{
public:
...
  ip         addr() const { return addr_;}
...
private:
  ip addr;
  ...
};
```

```
auto addr_matches(ip addr, netmask mask = {255,255,255,255})
{
  return compose(ip_matches(addr, mask),
                 select_addr);
}
```

```
std::vector<ipif> interfaces;

auto i = std::find_if(interfaces.begin(), interfaces.end(),
                      addr_matches({192,168,1,1}));
```

```
class ipif
{
public:
  using state_type = enum { off, on };
...
  void         set_state(state_type);

  state_type state()       { return state_; }
  ip         addr() const { return addr_;}
  netmask    mask() const { return mask_; }
  ip         gw() const   { return gw_; }
  ...
};
```

```
inline ip select_gw(ipif const& interface)
{
  return interface.gw();
}

inline ipif::state_type select_state(ipif const& interface)
{
  return interface.state();
}
```

```
auto i = find_if(v.begin(), v.end(),
                 when_all(addr_matches({192,168,1,1},{255,255,0,0}),
                          state_is(ipif::off)));
```

```
template <typename ... Predicates>
auto when_all(Predicates ... ps)
{
  return [=](auto const& x)
         {
           return (ps(x) && ...);
         };
}
```

```
auto i = find_if(v.begin(), v.end(),
                 when_all(addr_matches({192,168,1,1},{255,255,0,0}),
                          state_is(ipif::off)));
```

```
template <typename ... Predicates>
auto when_all(Predicates ... ps)
{
  return [=](auto const& x)
         {
           return (ps(x) && ...);
         };
}
```

```
auto addr_matches(ip addr, netmask mask=netmask{255,255,255,255})
{
  return compose(match_ip(addr, mask),
                 select_addr);
}
```

```
auto i = find_if(v.begin(), v.end(),
                 when_all(addr_matches({192,168,1,1},{255,255,0,0}),
                          state_is(ipif::off)));
```

```
template <typename ... Predicates>
auto when_all(Predicates ... ps)
{
  return [=](auto const& x)
         {
           return (ps(x) && ...);
         };
}

auto addr_matches(ip addr, netmask mask=netmask{255,255,255,255})
{
  return compose(match_ip(addr, mask),
                 select_addr);
}

auto state_is(ipif::state_type state)
{
  return compose(equals(state),
                 select_state);
}

auto i = find_if(v.begin(), v.end(),
                 when_all(addr_matches({192,168,1,1},{255,255,0,0}),
                          state_is(ipif::off)));
```

```
for_each(v.begin(), v.end(),
         if_then(when_all(addr_matches({192,168,1,1}, {255,255,0,0}),
                          state_is(ipif::off)),
                 set_state(ipif::on)));
```

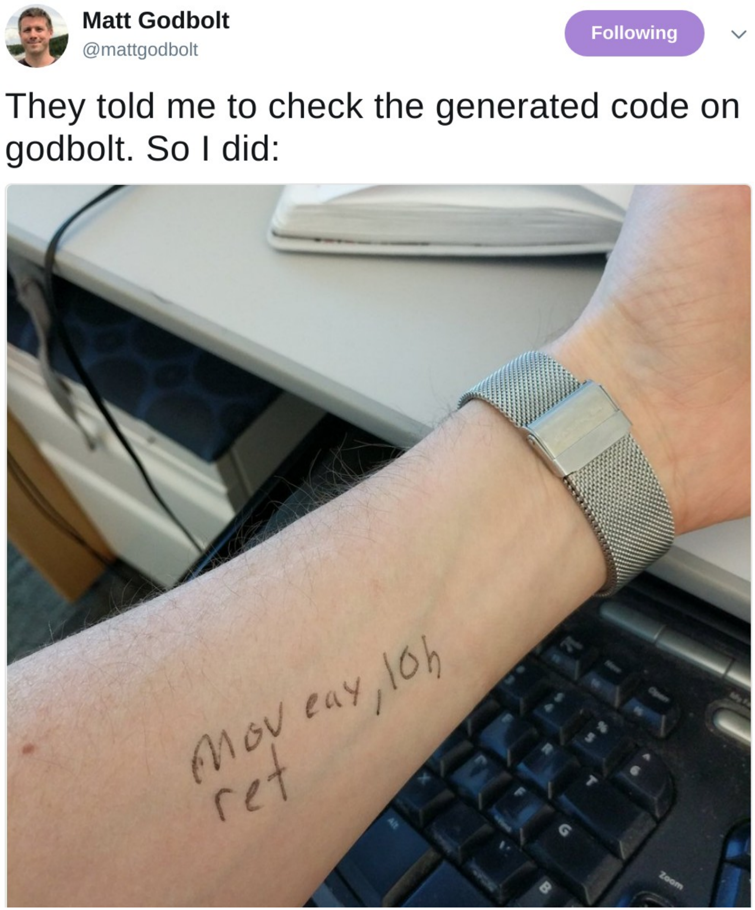
```
template <typename Predicate, typename Action>
auto if_then(Predicate predicate, Action action)
{
  return [=](auto&& x)
         {
           if (predicate(x)) {
             action(std::forward<decltype(x)>(x));
           }
         };
}
```

```
for_each(v.begin(), v.end(),
         if_then(when_all(addr_matches({192,168,1,1}, {255,255,0,0}),
                          state_is(ipif::off)),
                 set_state(ipif::on)));
```

```
template <typename Predicate, typename Action>
auto if_then(Predicate predicate, Action action)
{
  return [=](auto&& x)
         {
           if (predicate(x)) {
             action(std::forward<decltype(x)>(x));
           }
         };
}
```

```
auto set_state(ipif::state_type state)
{
  return [=](ipif& interface) { interface.set_state(state); };
}
```

```
for_each(v.begin(), v.end(),
         if_then(when_all(addr_matches({192,168,1,1}, {255,255,0,0}),
                          state_is(ipif::off)),
                 set_state(ipif::on)));
```



Matt Godbolt
@mattgodbolt

They told me to check the generated code on godbolt. So I did:

```
mov eax,10h
ret
```

Generic library functions

- `equals(value) ...`
- `compose(function...)`
- `if_then(pred,action)`
- `when_all(predicate...)`
- `when_none(predicate...)`
- `when_any(predicate...)`
- `do_all(action...)`

Domain specific functions

- `ip_matches(ip, mask)`
- `select_addr(ipif)`
- `select_gw(ipif)`
- `select_state(ipif)`
- `set_state(ipif&)`

Composed domain specific functions

- `addr_matches(ip)`
- `state_is(state_type)`

Generic library functions

- equals(value) ...
- compose(function...)
- if_then(pred,action)
- when_all(predicate...)
- when_none(predicate...)
- when_any(predicate...)
- do_all(action...)

Domain specific functions

- ip_matches(ip, mask)
- select_addr(ipif)
- select_gw(ipif)
- select_state(ipif)
- set_state(ipif&)

Composed domain specific functions

- addr_matches(ip)
- state_is(state_type)

https://github.com/rollbear/lift

Take away messages

- Write functions that uses auto return type to create lambdas

Take away messages

- Write functions that uses auto return type to create lambdas
- Write functions that access or modify your state

Take away messages

- Write functions that uses auto return type to create lambdas
- Write functions that access or modify your state
- Compose functions

Take away messages

- Write functions that uses auto return type to create lambdas
- Write functions that access or modify your state
- Compose functions
  - And give names to the compositions

Higher order functions for the rest of us

`https://github.com/rollbear/lift`

Björn Fahller

✉ bjorn@fahller.se

@bjorn_fahller

@rollbear *cpplang, swedencpp*