Non-Allocating Future/Promise

Tony Van Eerd, BlackBerry, Inc.

C++Now, Aspen Colorado

May 13, 2013

Could do != Should do

```
template <class R>
class future {
public:
  future() noexcept;
  future(future &&) noexcept;
  future(const future& rhs) = delete;
  ~future();
  future& operator=(const future& rhs) = delete;
  future& operator=(future&&) noexcept;
  shared_future<R> share();

  // retrieving the value
  see below get();

  // functions to check state
  bool valid() const noexcept;
  void wait() const;
  template <class Rep, class Period>
   future_status wait_for(const chrono::duration<Rep, Period>& rel_time) const;
 template <class Clock, class Duration>
   future_status wait_until(const chrono::time_point<Clock, Duration>& abs_time) const;
};
```

```
template <class R>
class future {
public:
  future() noexcept;
  future(future &&) noexcept;
  future(const future& rhs) = delete;
  ~future();
  future& operator=(const future& rhs) = delete;
  future& operator=(future&&) noexcept;
  shared_future<R> share();

  // retrieving the value
  see below get();

  // functions to check state
  bool valid() const noexcept;
  void wait() const;
  template <class Rep, class Period>
   future_status wait_for(const chrono::duration<Rep, Period>& rel_time) const;
 template <class Clock, class Duration>
   future_status wait_until(const chrono::time_point<Clock, Duration>& abs_time) const;
};
```

```
template <class R>
class future {
public:
  future() noexcept;
  future(future &&) noexcept;
  future(const future& rhs) = delete;
  ~future();
  future& operator=(const future& rhs) = delete;
  future& operator=(future&&) noexcept;
  shared_future<R> share();

  // retrieving the value
  see below get();

  // functions to check state
  bool valid() const noexcept;
  void wait() const;
  template <class Rep, class Period>
   future_status wait_for(const chrono::duration<Rep, Period>& rel_time) const;
 template <class Clock, class Duration>
   future_status wait_until(const chrono::time_point<Clock, Duration>& abs_time) const;
};
```

```
template <class R>
class future {
public:
  future() noexcept;
  future(future &&) noexcept;
  future(const future& rhs) = delete;
  ~future();
  future& operator=(const future& rhs) = delete;
  future& operator=(future&&) noexcept;
  shared_future<R> share();

  // retrieving the value
  see below get();

  // functions to check state
  bool valid() const noexcept;
  void wait() const;
  template <class Rep, class Period>
   future_status wait_for(const chrono::duration<Rep, Period>& rel_time) const;
 template <class Clock, class Duration>
   future_status wait_until(const chrono::time_point<Clock, Duration>& abs_time) const;
};
```

MOVE = YES

COPY = NO

```
template <class R>
class future {
public:
  future() noexcept;
  future(future &&) noexcept;
  future(const future& rhs) = delete;
  ~future();  // ...
  future& operator=(const future& rhs) = delete;
  future& operator=(future&&) noexcept;
  shared_future<R> share();

  // retrieving the value
  see below get();

  // functions to check state
  bool valid() const noexcept;
  void wait() const;
  template <class Rep, class Period>
   future_status wait_for(const chrono::duration<Rep, Period>& rel_time) const;
 template <class Clock, class Duration>
   future_status wait_until(const chrono::time_point<Clock, Duration>& abs_time) const;
};
```

```
template <class R>
class future {
public:
  future() noexcept;
  future(future &&) noexcept;
  future(const future& rhs) = delete;
  ~future();  // “mostly harmless”
  future& operator=(const future& rhs) = delete;
  future& operator=(future&&) noexcept;
  shared_future<R> share();

  // retrieving the value
  see below get();

  // functions to check state
  bool valid() const noexcept;
  void wait() const;
  template <class Rep, class Period>
   future_status wait_for(const chrono::duration<Rep, Period>& rel_time) const;
 template <class Clock, class Duration>
   future_status wait_until(const chrono::time_point<Clock, Duration>& abs_time) const;
};
```

```
template <class R>
class future {
public:
  future() noexcept;
  future(future &&) noexcept;
  future(const future& rhs) = delete;
  ~future();
  future& operator=(const future& rhs) = delete;
  future& operator=(future&&) noexcept;
  shared_future<R> share();

  // retrieving the value
  see below get();

  // functions to check state
  bool valid() const noexcept;
  void wait() const;
  template <class Rep, class Period>
   future_status wait_for(const chrono::duration<Rep, Period>& rel_time) const;
 template <class Clock, class Duration>
   future_status wait_until(const chrono::time_point<Clock, Duration>& abs_time) const;
};
```

```
template <class R>
class future {
public:
  future() noexcept;
  future(future &&) noexcept;
  future(const future& rhs) = delete;
  ~future();
  future& operator=(const future& rhs) = delete;
  future& operator=(future&&) noexcept;
  shared_future<R> share();

  // retrieving the value
  see below get();

  // functions to check state
  bool valid() const noexcept;
  void wait() const;
  template <class Rep, class Period>
   future_status wait_for(const chrono::duration<Rep, Period>& rel_time) const;
 template <class Clock, class Duration>
   future_status wait_until(const chrono::time_point<Clock, Duration>& abs_time) const;
};
```

```
template <class R>
class future {
public:
  future() noexcept;
  future(future &&) noexcept;
  future(const future& rhs) = delete;
  ~future();
  future& operator=(const future& rhs) = delete;
  future& operator=(future&&) noexcept;
  shared_future<R> share();

  // retrieving the value
  R get();

  // functions to check state
  bool valid() const noexcept;
  void wait() const;
  template <class Rep, class Period>
   future_status wait_for(const chrono::duration<Rep, Period>& rel_time) const;
 template <class Clock, class Duration>
   future_status wait_until(const chrono::time_point<Clock, Duration>& abs_time) const;
};
```

```
template <class R>
class future {
public:
  future() noexcept;
  future(future &&) noexcept;
  future(const future& rhs) = delete;
  ~future();
  future& operator=(const future& rhs) = delete;
  future& operator=(future&&) noexcept;
  shared_future<R> share();

  // retrieving the value
  R get();

  // functions to check state
  bool valid() const noexcept;
  void wait() const;
  template <class Rep, class Period>
   future_status wait_for(const chrono::duration<Rep, Period>& rel_time) const;
  template <class Clock, class Duration>
   future_status wait_until(const chrono::time_point<Clock, Duration>& abs_time) const;
};
```

?

```
class LotteryNumbers {
  vector<int> numbes;
  ...
};

int main()
{
   // Step 1:
   future<LotteryNumbers> futureLotteryNumbers;
   // Step 2:
   LotteryNumbers numbers = futureLotteryNumbers.get();
   // Step 3: Profit
   cout << numbers;
};
```

```
template <class R>
class promise {
public:
  promise();
  template <class Allocator>
    promise(allocator_arg_t, const Allocator& a);
  promise(promise&& rhs) noexcept;
  promise(const promise& rhs) = delete;
  ~promise();

  promise& operator=(promise&& rhs) noexcept;
  promise& operator=(const promise& rhs) = delete;
  void swap(promise& other) noexcept;

  future<R> get_future();

  void set_value( see below );
  void set_exception(exception_ptr p);

  // setting the result with deferred notification
  void set_value_at_thread_exit(const R& r);
  void set_value_at_thread_exit(see below );
  void set_exception_at_thread_exit(exception_ptr p);
};
```

```cpp
template <class R>
class promise {
public:
  promise();
  template <class Allocator>
    promise(allocator_arg_t, const Allocator& a);
  promise(promise&& rhs) noexcept;
  promise(const promise& rhs) = delete;
  ~promise();

  promise& operator=(promise&& rhs) noexcept;
  promise& operator=(const promise& rhs) = delete;
  void swap(promise& other) noexcept;

  future<R> get_future();

  void set_value( see below );
  void set_exception(exception_ptr p);

  // setting the result with deferred notification
  void set_value_at_thread_exit(const R& r);
  void set_value_at_thread_exit(see below );
  void set_exception_at_thread_exit(exception_ptr p);
};
```

MOVE = YES

COPY = NO

```
template <class R>
class promise {
public:
  promise();
  template <class Allocator>
    promise(allocator_arg_t, const Allocator& a);
  promise(promise&& rhs) noexcept;
  promise(const promise& rhs) = delete;
  ~promise();

  promise& operator=(promise&& rhs) noexcept;
  promise& operator=(const promise& rhs) = delete;
  void swap(promise& other) noexcept;

  future<R> get_future();

  void set_value( see below );
  void set_exception(exception_ptr p);

  // setting the result with deferred notification
  void set_value_at_thread_exit(const R& r);
  void set_value_at_thread_exit(see below );
  void set_exception_at_thread_exit(exception_ptr p);
};
```

```
template <class R>
class promise {
public:
  promise();
  template <class Allocator>
    promise(allocator_arg_t, const Allocator& a);
  promise(promise&& rhs) noexcept;
  promise(const promise& rhs) = delete;
  ~promise();

  promise& operator=(promise&& rhs) noexcept;
  promise& operator=(const promise& rhs) = delete;
  void swap(promise& other) noexcept;

  future<R> get_future();

  void set_value( see below );
  void set_exception(exception_ptr p);

  // setting the result with deferred notification
  void set_value_at_thread_exit(const R& r);
  void set_value_at_thread_exit(see below );
  void set_exception_at_thread_exit(exception_ptr p);
};
```

```
template <class R>
class promise {
public:
  promise();
  template <class Allocator>
    promise(allocator_arg_t, const Allocator& a);
  promise(promise&& rhs) noexcept;
  promise(const promise& rhs) = delete;
  ~promise();

  promise& operator=(promise&& rhs) noexcept;
  promise& operator=(const promise& rhs) = delete;
  void swap(promise& other) noexcept;

  future<R> get_future();

  void set_value( see below );
  void set_exception(exception_ptr p);

  // setting the result with deferred notification
  void set_value_at_thread_exit(const R& r);
  void set_value_at_thread_exit(see below );
  void set_exception_at_thread_exit(exception_ptr p);
};
```

```
template <class R>
class promise {
public:
  promise();
  template <class Allocator>
    promise(allocator_arg_t, const Allocator& a);
  promise(promise&& rhs) noexcept;
  promise(const promise& rhs) = delete;
  ~promise();

  promise& operator=(promise&& rhs) noexcept;
  promise& operator=(const promise& rhs) = delete;
  void swap(promise& other) noexcept;

  future<R> get_future();

  void set_value(R const & value);
  void set_value(R & value);
  void set_value(R && value);
  void set_exception(exception_ptr p);

  // setting the result with deferred notification
  void set_value_at_thread_exit(const R& r);
  void set_value_at_thread_exit(see below );
  void set_exception_at_thread_exit(exception_ptr p);
};
```

```
template <class R>
class promise {
public:
  promise();
  template <class Allocator>
    promise(allocator_arg_t, const Allocator& a);
  promise(promise&& rhs) noexcept;
  promise(const promise& rhs) = delete;
  ~promise();

  promise& operator=(promise&& rhs) noexcept;
  promise& operator=(const promise& rhs) = delete;
  void swap(promise& other) noexcept;

  future<R> get_future();

  void set_value(R const & value);
  void set_value(R & value);
  void set_value(R && value);
  void set_exception(exception_ptr p);

  // setting the result with deferred notification
  void set_value_at_thread_exit(const R& r);
  void set_value_at_thread_exit(see below );
  void set_exception_at_thread_exit(exception_ptr p);
};
```

```
template <class R>
class promise {
public:
  promise();
  template <class Allocator>
    promise(allocator_arg_t, const Allocator& a);
  promise(promise&& rhs) noexcept;
  promise(const promise& rhs) = delete;
  ~promise();

  promise& operator=(promise&& rhs) noexcept;
  promise& operator=(const promise& rhs) = delete;
  void swap(promise& other) noexcept;

  future<R> get_future();

  void set_value(R * value);
  void set_exception(exception_ptr p);

  // setting the result with deferred notification
  void set_value_at_thread_exit(const R& r);
  void set_value_at_thread_exit(see below );
  void set_exception_at_thread_exit(exception_ptr p);
};
```

```
template <class R>
class promise {
public:
  promise();
  template <class Allocator>
    promise(allocator_arg_t, const Allocator& a);
  promise(promise&& rhs) noexcept;
  promise(const promise& rhs) = delete;
  ~promise();

  promise& operator=(promise&& rhs) noexcept;
  promise& operator=(const promise& rhs) = delete;
  void swap(promise& other) noexcept;

  future<R> get_future();

  void set_value(R value);
  void set_exception(exception_ptr p);

  // setting the result with deferred notification
  void set_value_at_thread_exit(const R& r);
  void set_value_at_thread_exit(see below );
  void set_exception_at_thread_exit(exception_ptr p);
};
```

```
template <class R>
class promise {
public:
  promise();
  template <class Allocator>
    promise(allocator_arg_t, const Allocator& a);
  promise(promise&& rhs) noexcept;
  promise(const promise& rhs) = delete;
  ~promise();

  promise& operator=(promise&& rhs) noexcept;
  promise& operator=(const promise& rhs) = delete;
  void swap(promise& other) noexcept;

  future<R> get_future();

  void set_value(R value);
  void set_exception(exception_ptr p);

  // setting the result with deferred notification
  void set_value_at_thread_exit(const R& r);
  void set_value_at_thread_exit(see below );
  void set_exception_at_thread_exit(exception_ptr p);
};
```

```
template <class R>
class future {
public:
  future();
  future(future &&) noexcept;
  ~future();
  future& operator=(future&&) noexcept;

  R get();
  void wait() const;
};


template <class R>
class promise {
public:
  promise();
  promise(promise&& rhs) noexcept;
  ~promise();
  promise& operator=(promise&& rhs) noexcept;

  future<R> get_future();
  void set_value(R value);
};
```

```
int main()
{
   promise<Numbers> ipromise;
   future<Numbers> thefuture = ipromise.get_future();
   ipromise.set_value(calculateLotteryNumbers());
   Numbers numbers = thefuture.get();
   // Step 3: Profit
   cout << numbers;
};
```

```
int main()
{
   promise<Numbers> ipromise;
   future<Numbers> thefuture = ipromise.get_future();
   ipromise.set_value(calculateLotteryNumbers());
   Numbers numbers = thefuture.get();
   // Step 3: Profit
   cout << numbers;
};
```

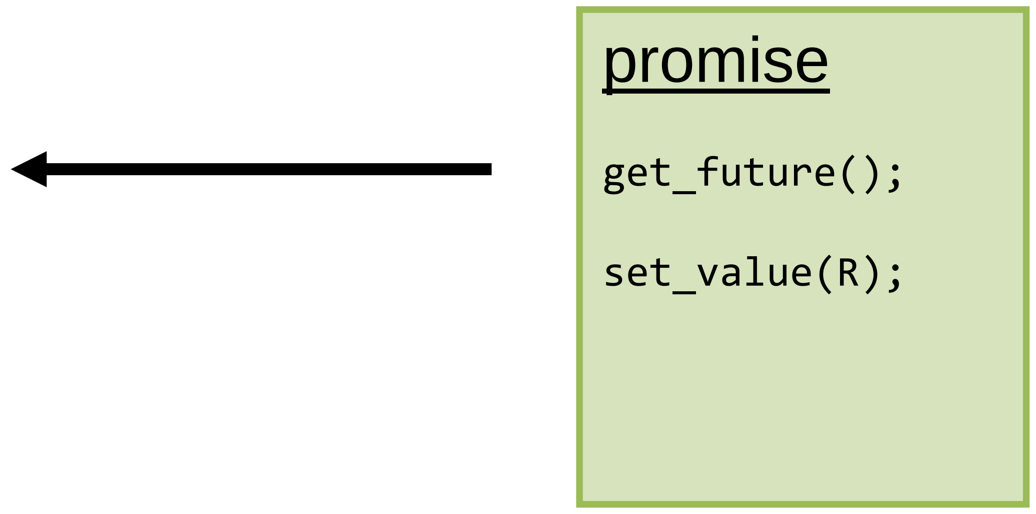
```
int main()
{
   // Step 1: Profit
   cout << calculateLotteryNumbers();
};
```

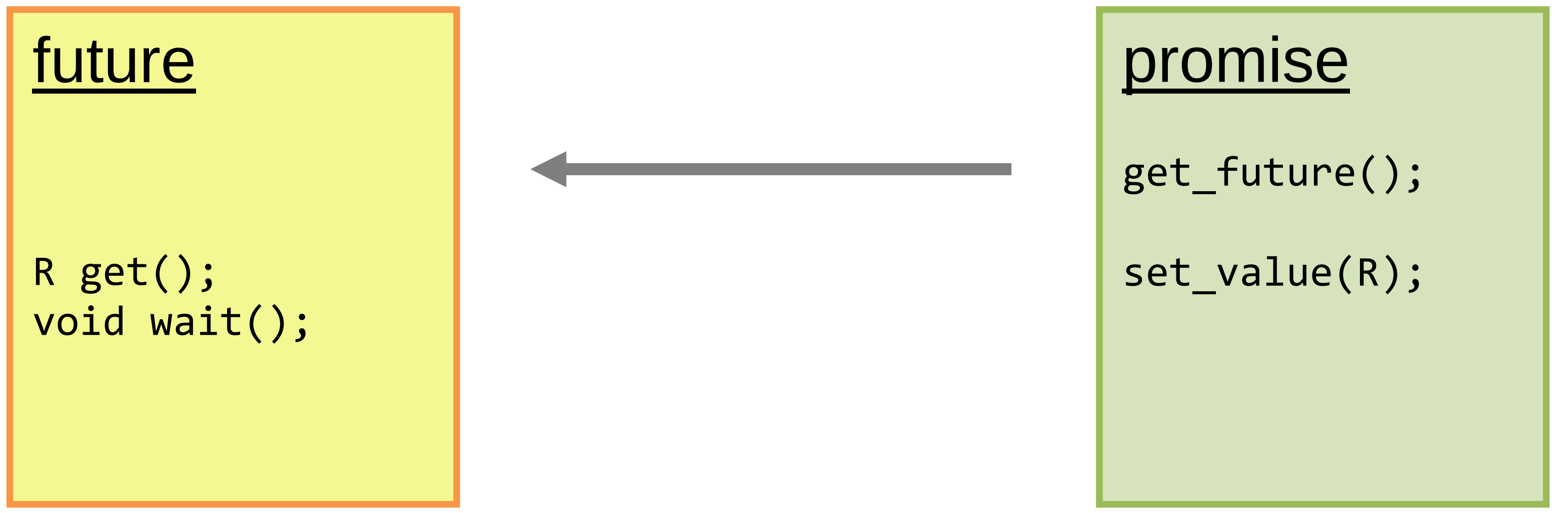
```
int main()
{
   promise<Numbers> ipromise;
   future<Numbers> thefuture = ipromise.get_future();
   ipromise.set_value(calculateLotteryNumbers());
   Numbers numbers = thefuture.get();
   // Step 3: Profit
   cout << numbers;
};
```

```
int main()
{
   // Step 1: Profit
   cout << calculateLotteryNumbers();
};
```

```
int main()
{
   promise<Numbers> ipromise;
   future<Numbers> thefuture = ipromise.get_future();

   concurrently
   {
      ipromise.set_value(calculateLotteryNumbers());
   }

   do_other_stuff();

   Numbers numbers = thefuture.get();
   // Step 3: Profit
   cout << numbers;
};
```

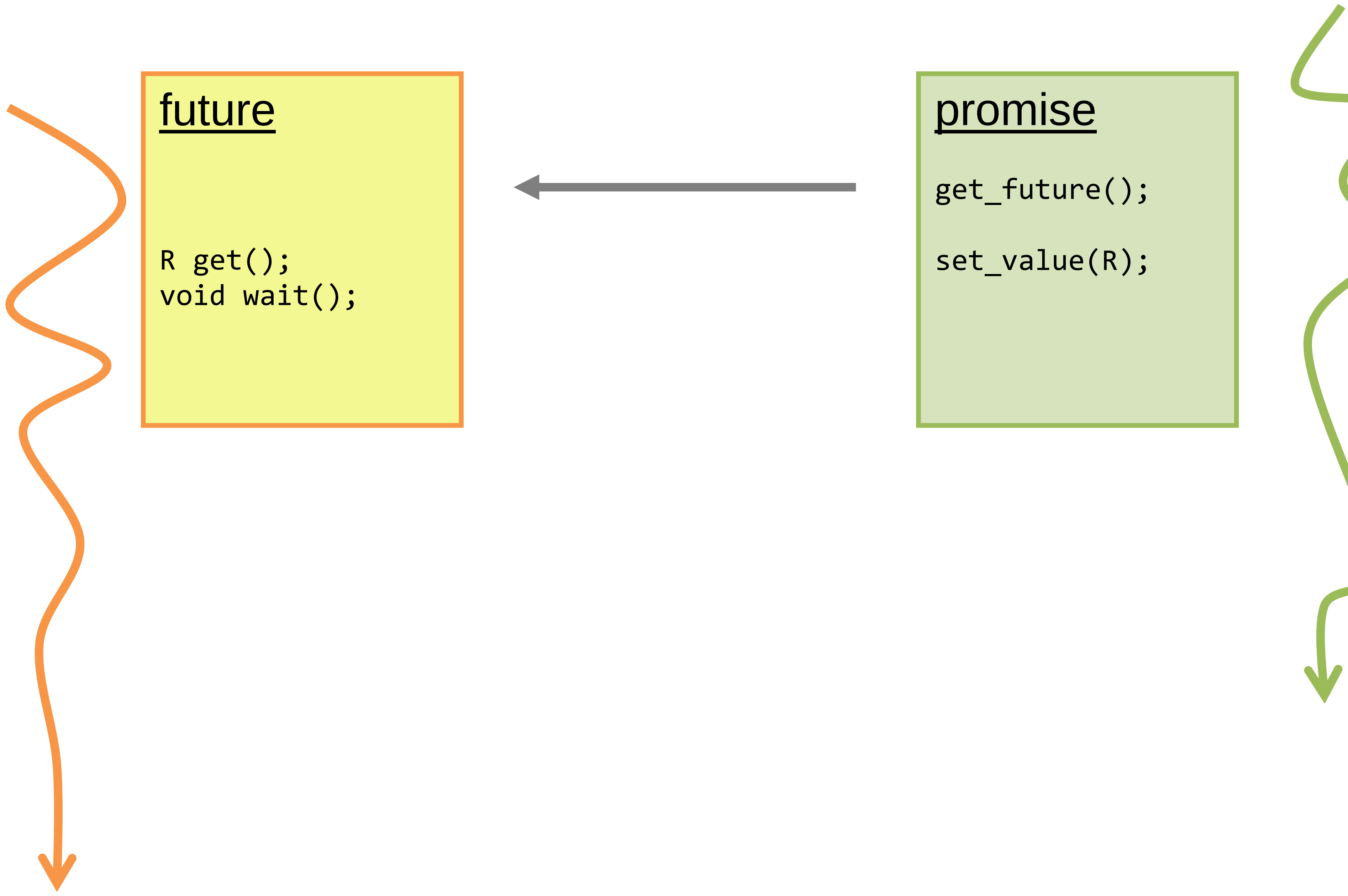
```
int main()
{
   promise<Numbers> ipromise;
   future<Numbers> thefuture = ipromise.get_future();

   while
   {
      ipromise.set_value(calculateLotteryNumbers());
   }
   do
   {
      other_stuff();
   };

   Numbers numbers = thefuture.get();
   // Step 3: Profit
   cout << numbers;
};
```

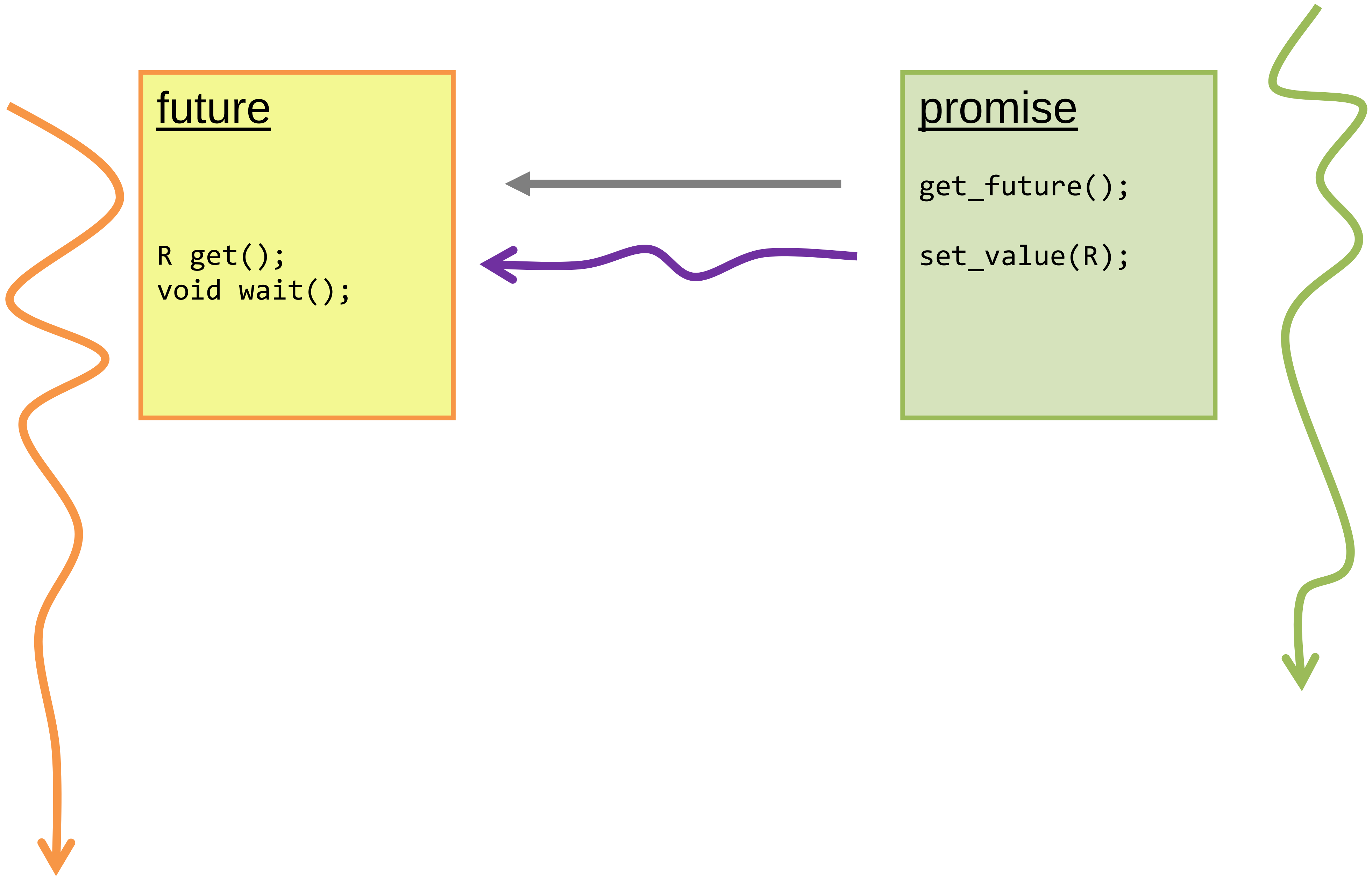

promise
get_future();
set_value(R);

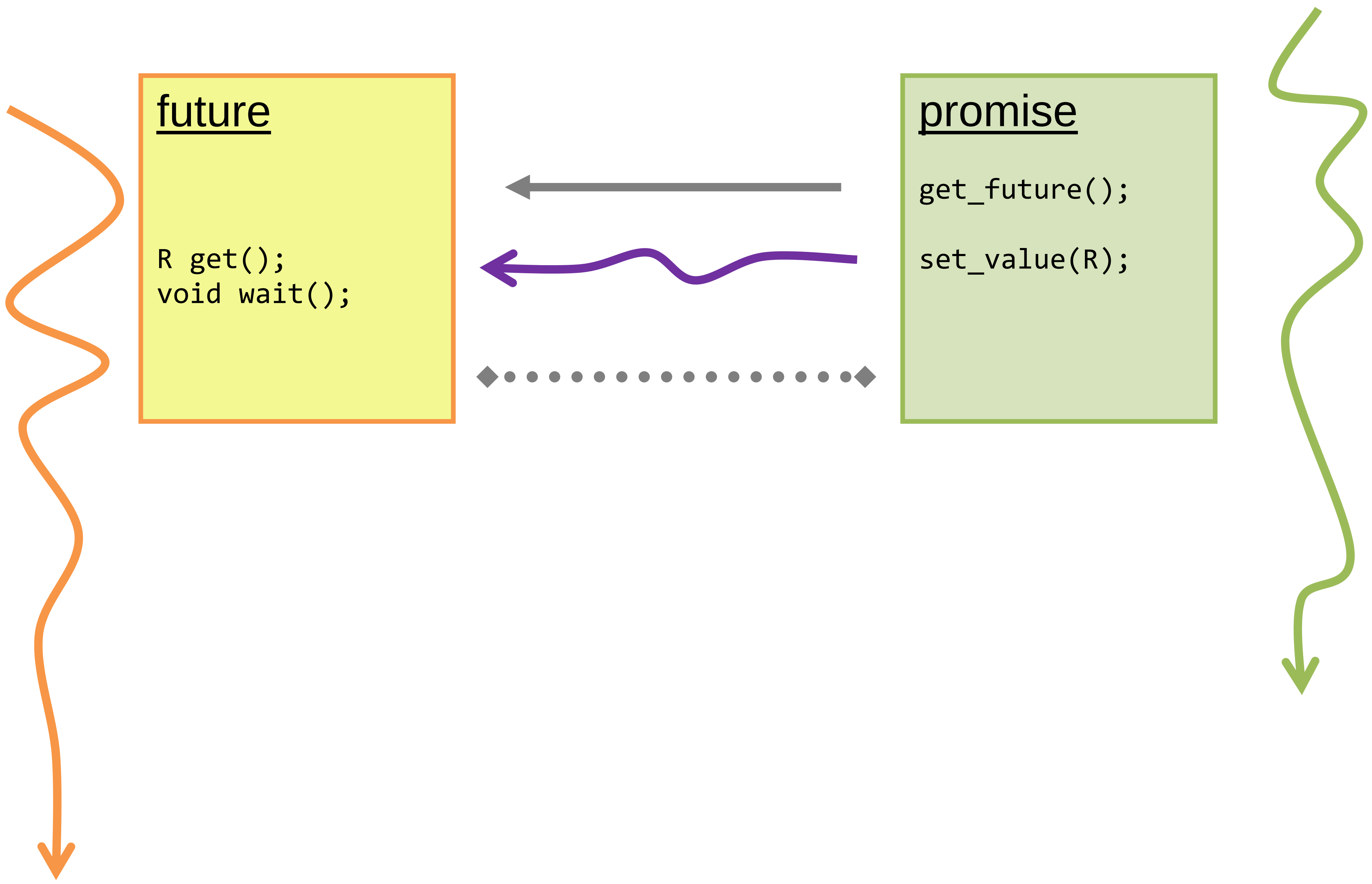


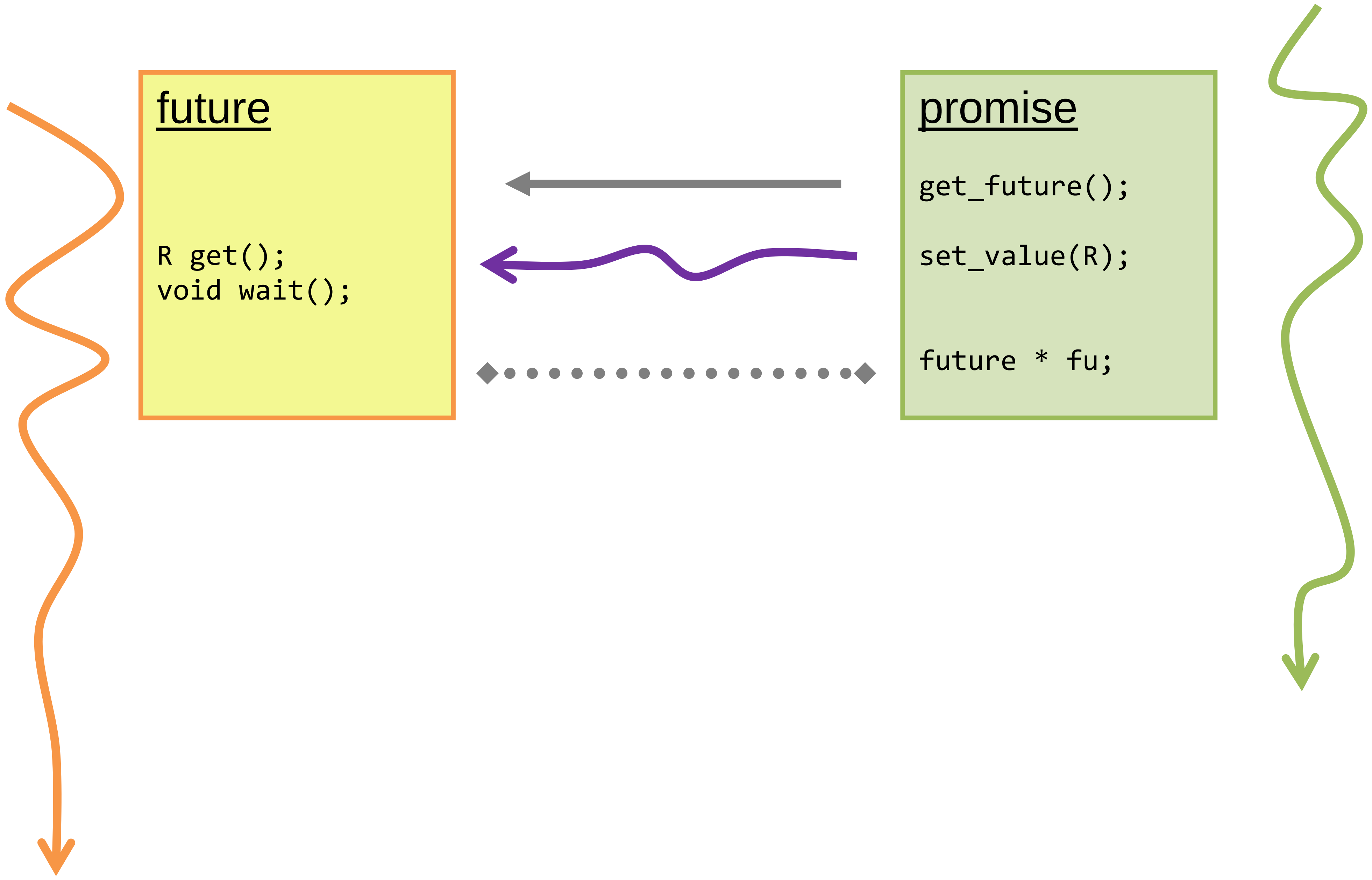
```
R get();
void wait();
```

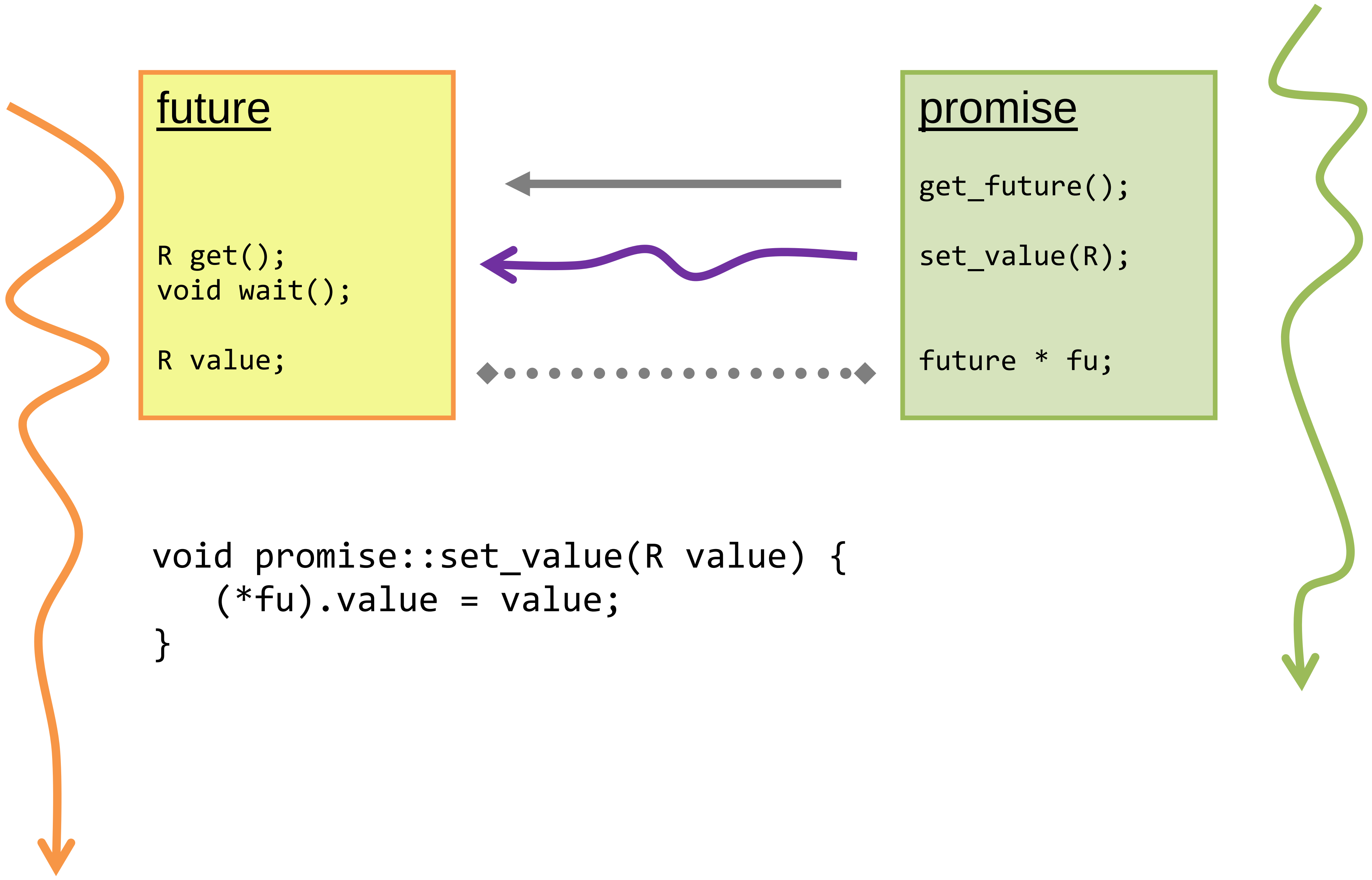
```
get_future();

set_value(R);
```


future
R get();
void wait();
promise
get_future();
set_value(R);

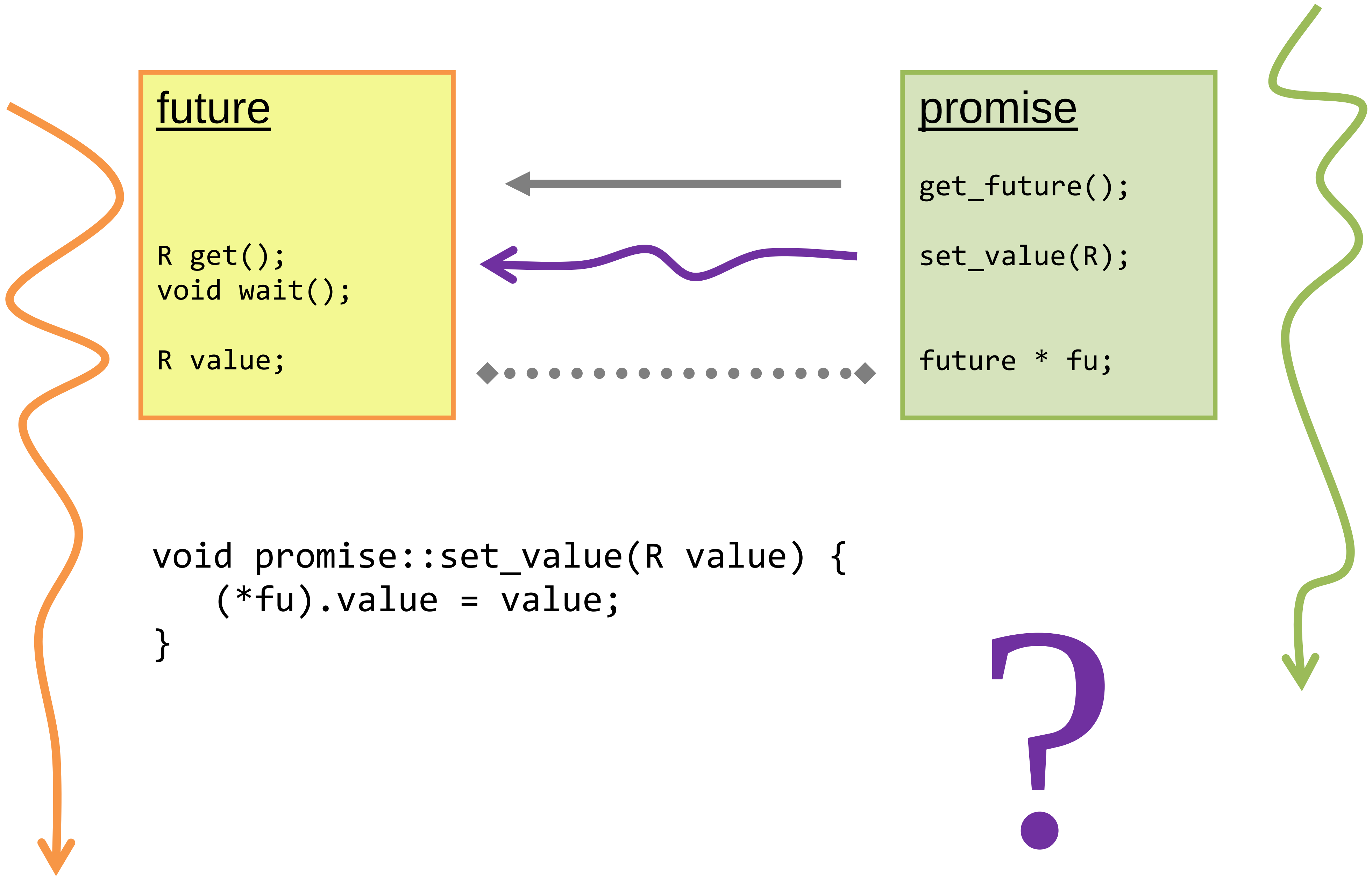

future
R get();
void wait();
promise
get_future();
set_value(R);


future
R get();
void wait();
promise
get_future();
set_value(R);


future
R get();
void wait();
promise
get_future();
set_value(R);
future * fu;



```
void promise::set_value(R value) {
   (*fu).value = value;
}
```

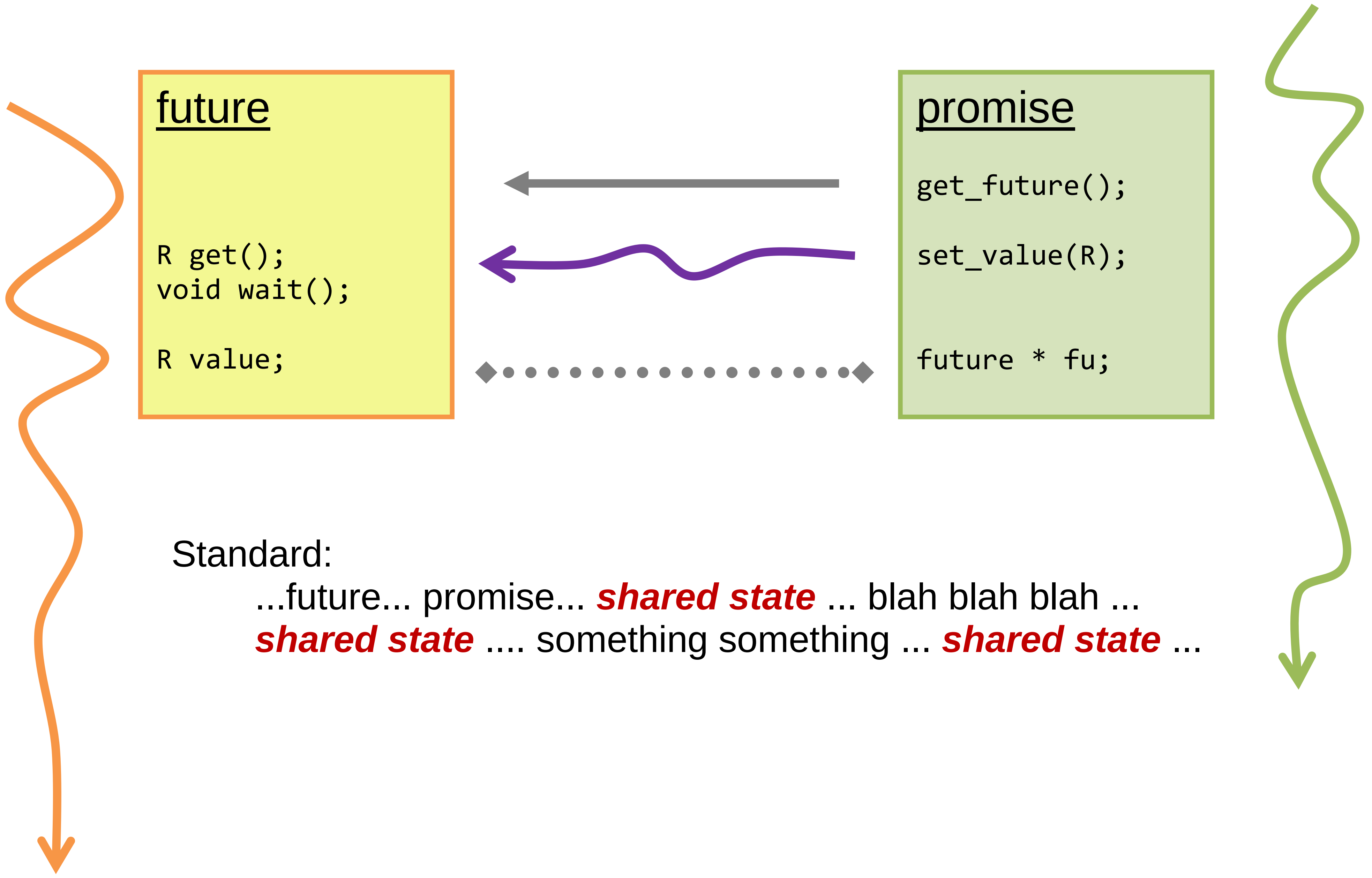


```
R get();
void wait();

R value;
```

```
get_future();

set_value(R);

future * fu;
```

```
void promise::set_value(R value) {
    (*fu).value = value;
}
```

?



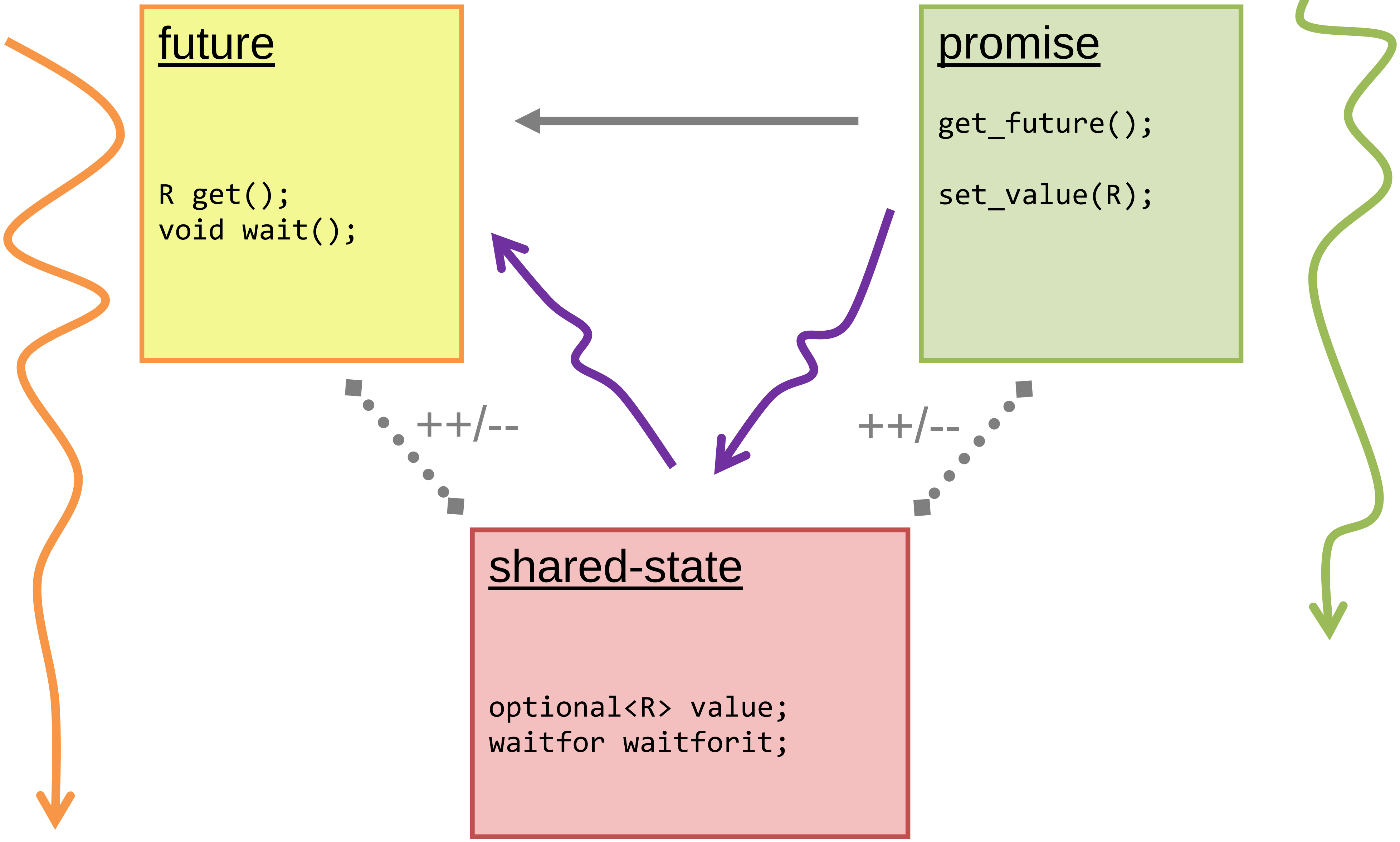
```
R get();
void wait();

R value;
```

```
get_future();

set_value(R);

future * fu;
```

Standard:

...future... promise... ***shared state*** ... blah blah blah ...
shared state .... something something ... ***shared state*** ...

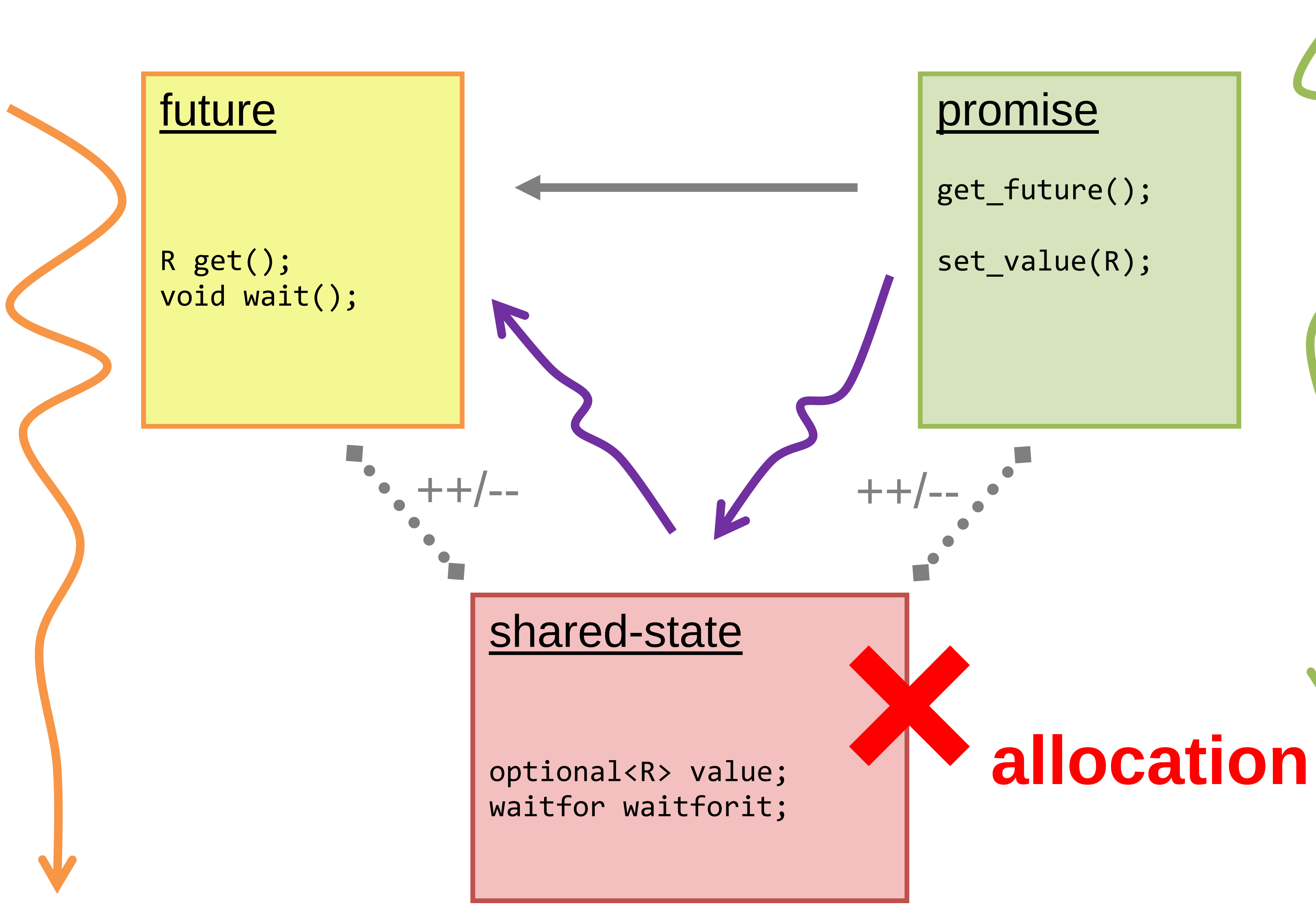
std::future & std::promise

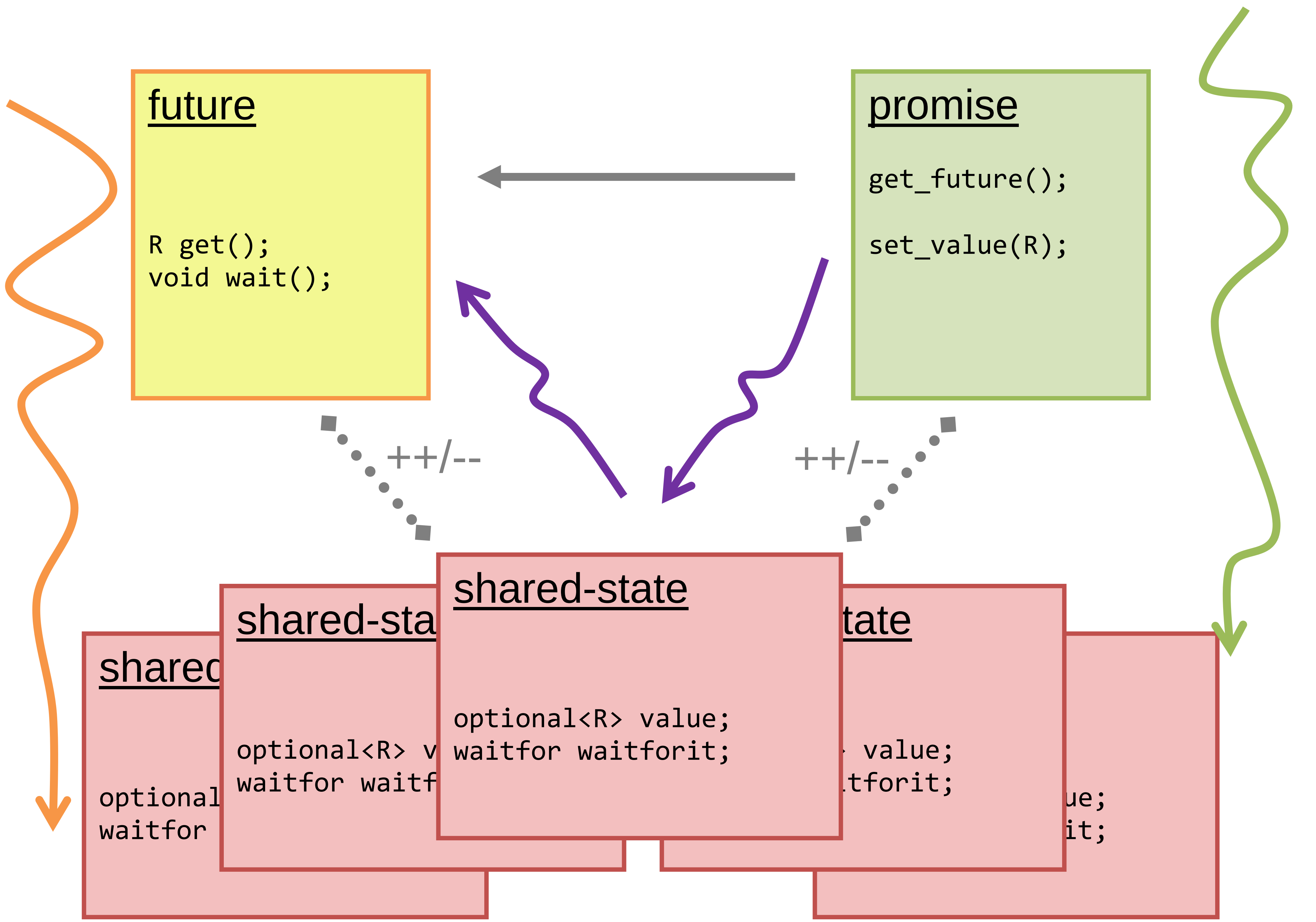


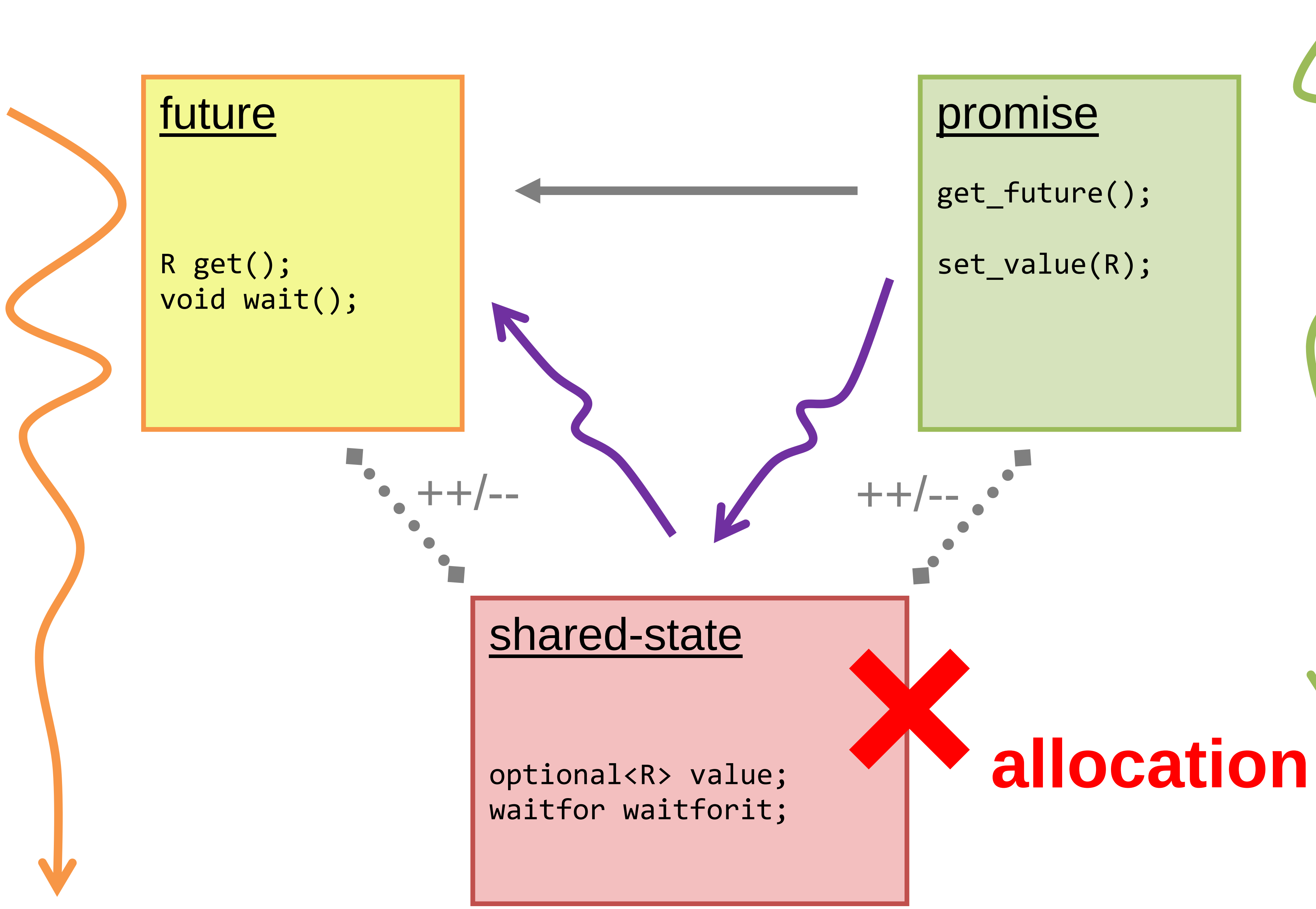
```
R get();
void wait();
```

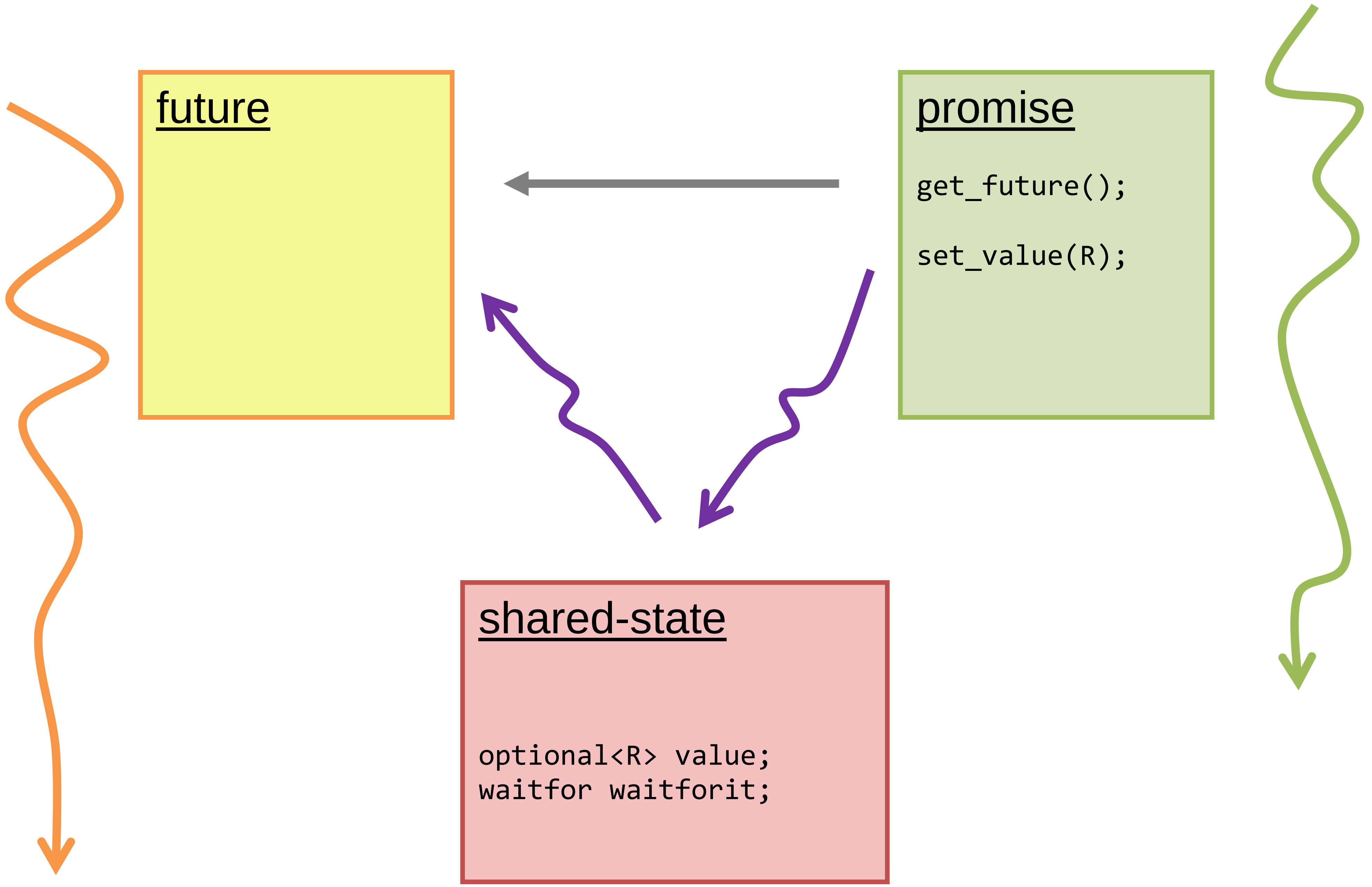
```
get_future();

set_value(R);
```

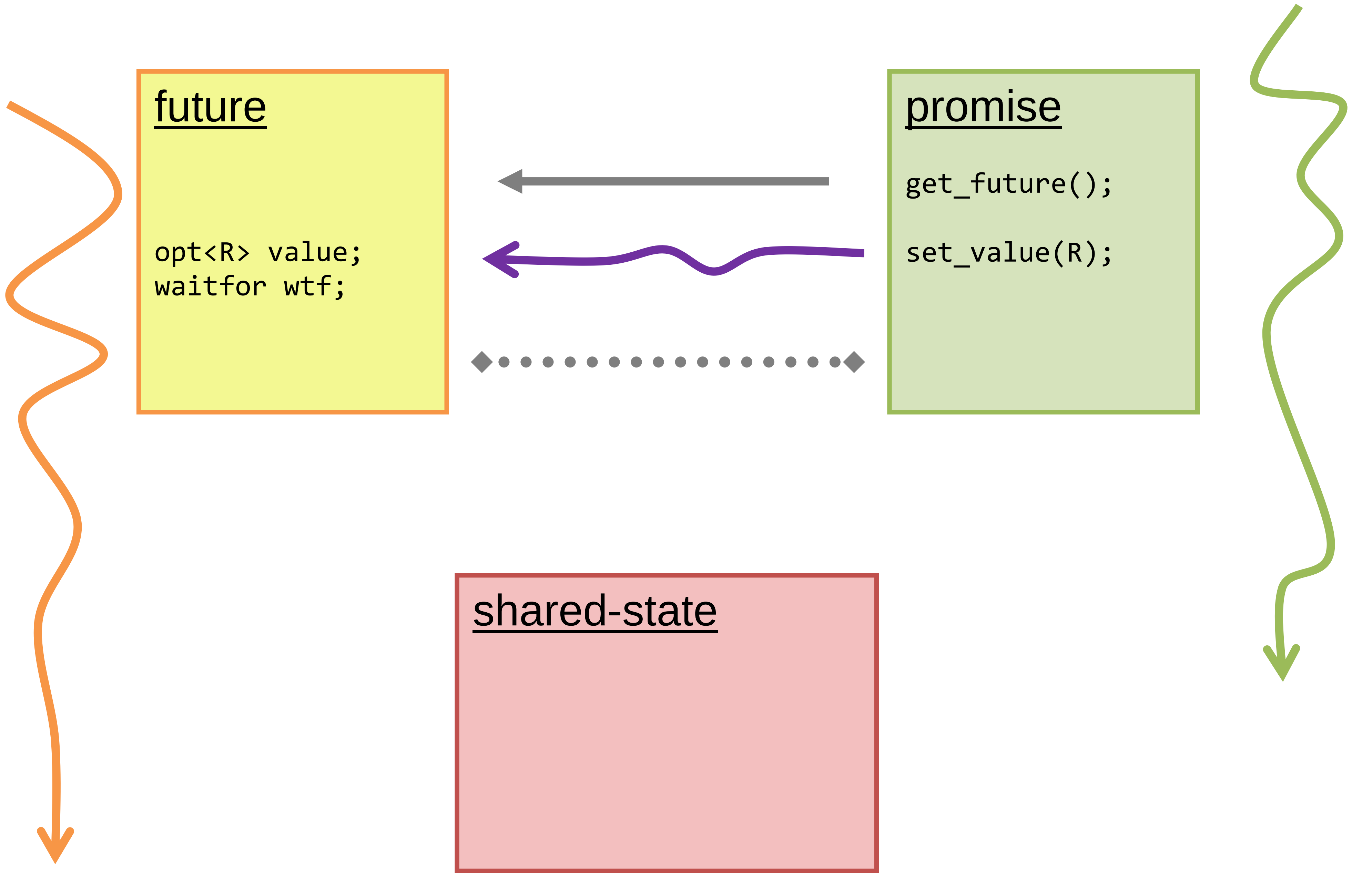
```
optional<R> value;
waitfor waitforit;
```


future
R get();
void wait();
promise
get_future();
set_value(R);
++/--
++/--
shared-state
optional<R> value;
waitfor waitforit;
allocation


future
R get();
void wait();
promise
get_future();
set_value(R);
++/--
++/--
shared-state
optional<R> value;
waitfor waitforit;
shared-sta
optional<R> v
waitfor waitf
shared
optional
waitfor
tate
> value;
itforit;
ue;
it;

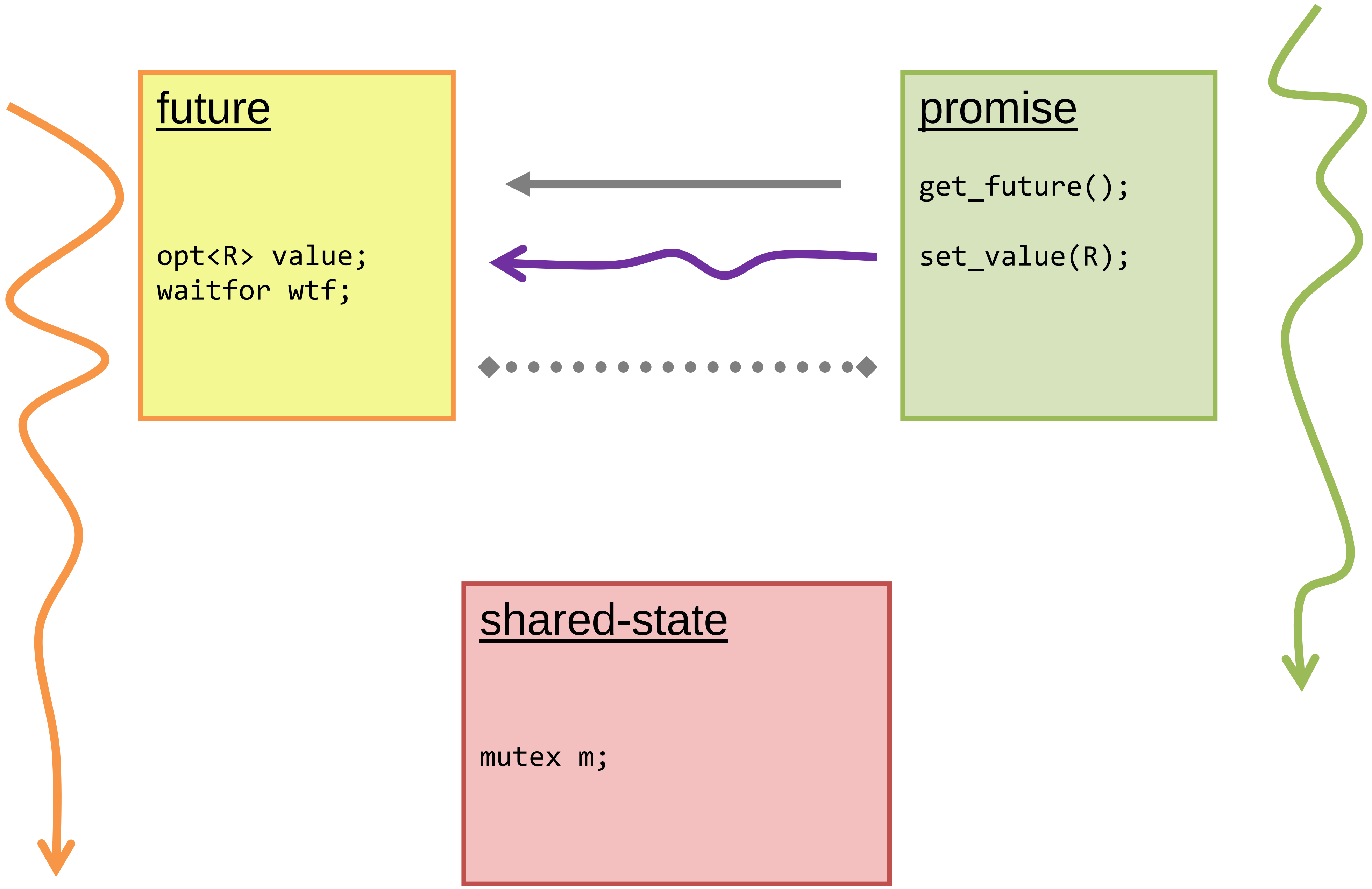

future
R get();
void wait();
promise
get_future();
set_value(R);
++/--
++/--
shared-state
optional<R> value;
waitfor waitforit;
allocation


future
promise
get_future();
set_value(R);
shared-state
optional<R> value;
waitfor waitforit;



```
opt<R> value;
waitfor wtf;
```

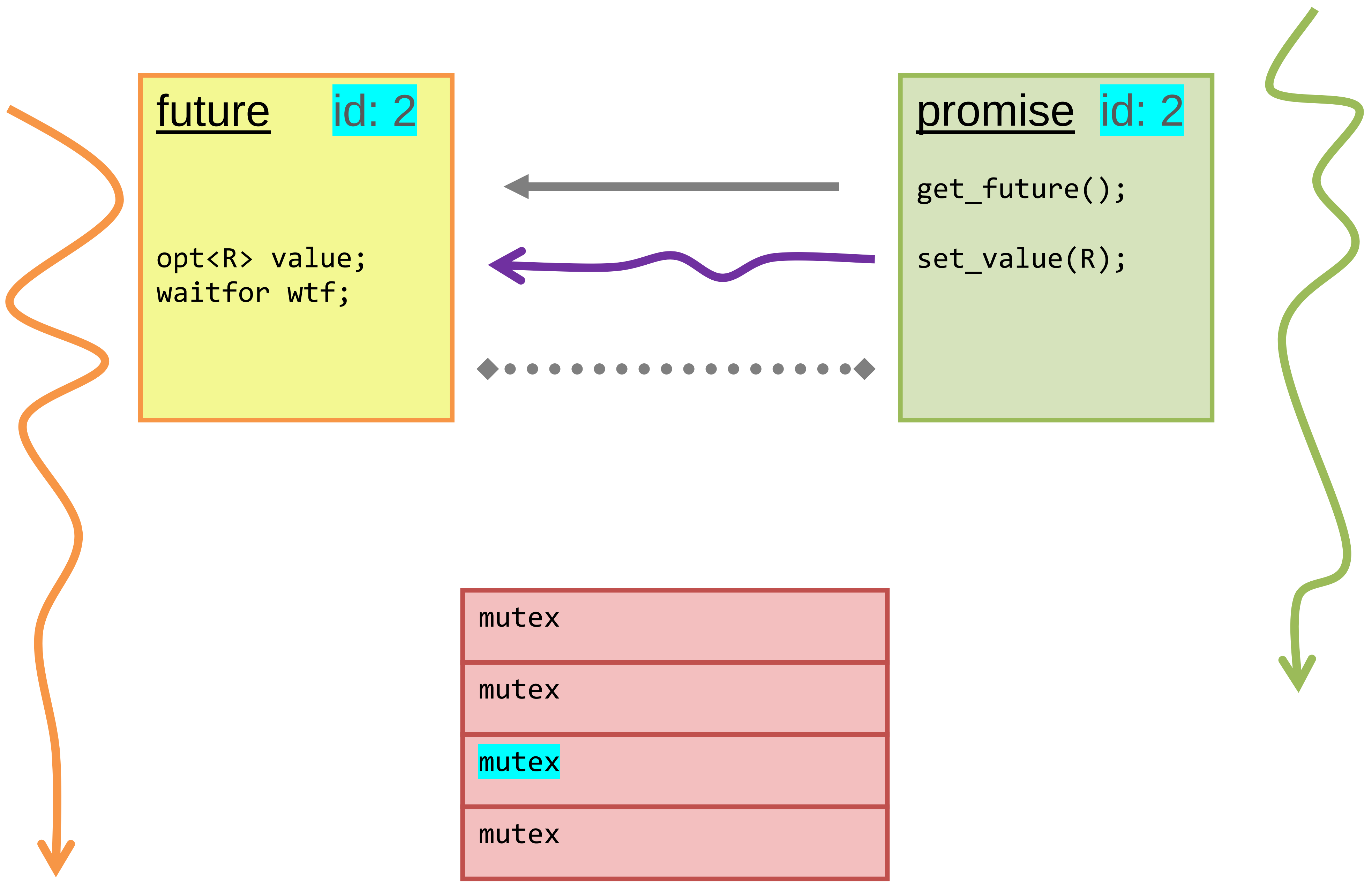
```
get_future();

set_value(R);
```

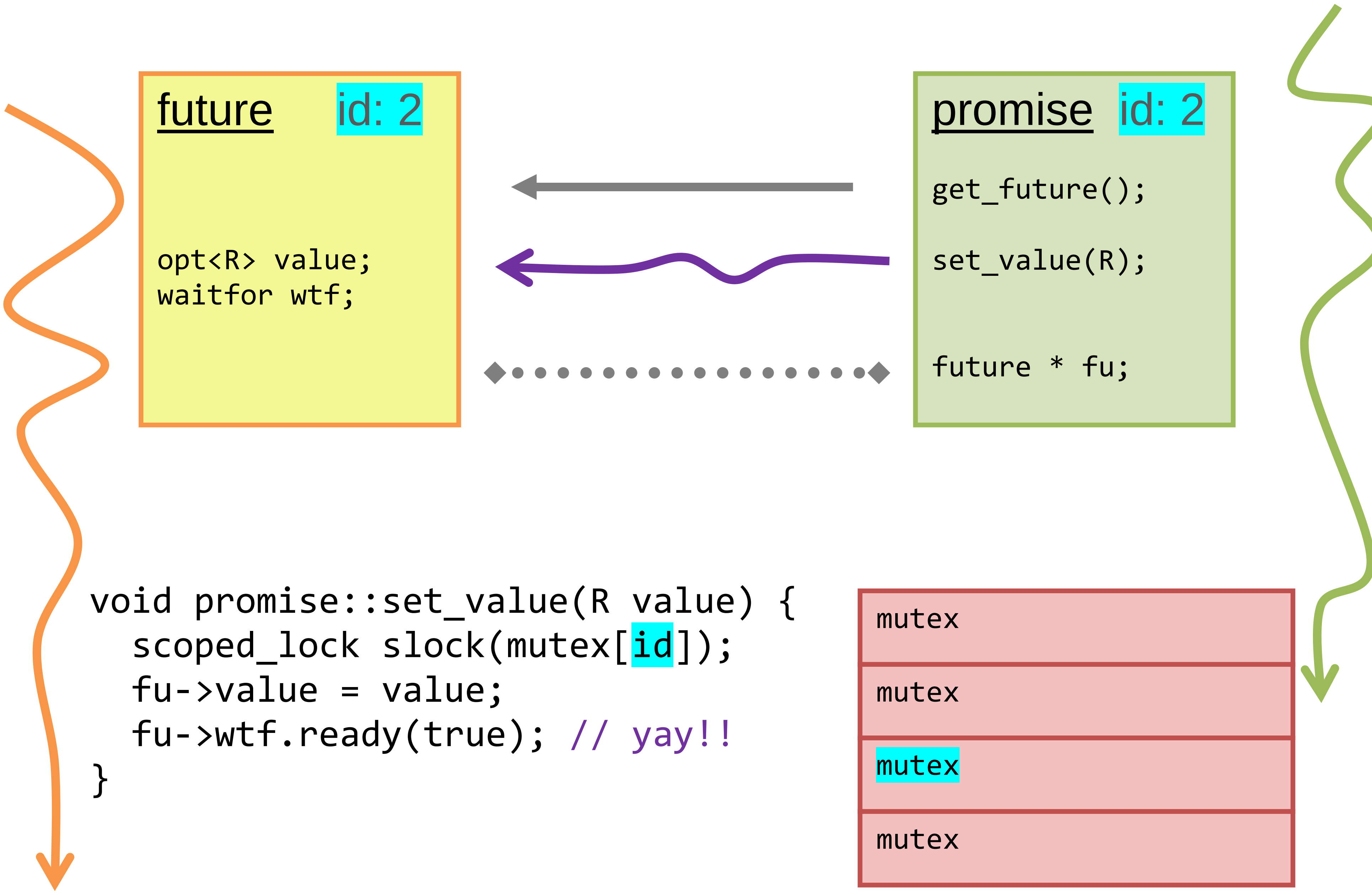


```
opt<R> value;
waitfor wtf;
```

```
get_future();

set_value(R);
```

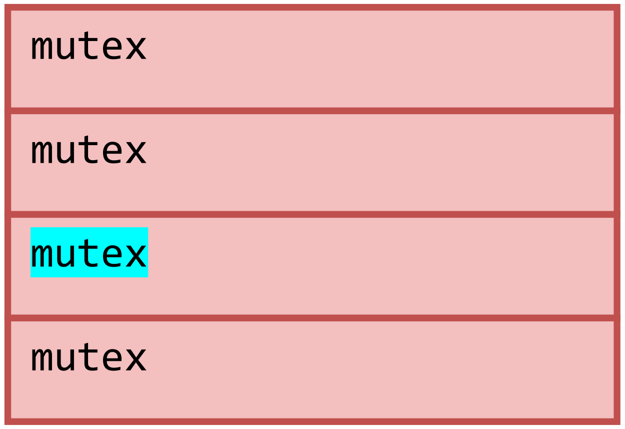
```
mutex m;
```

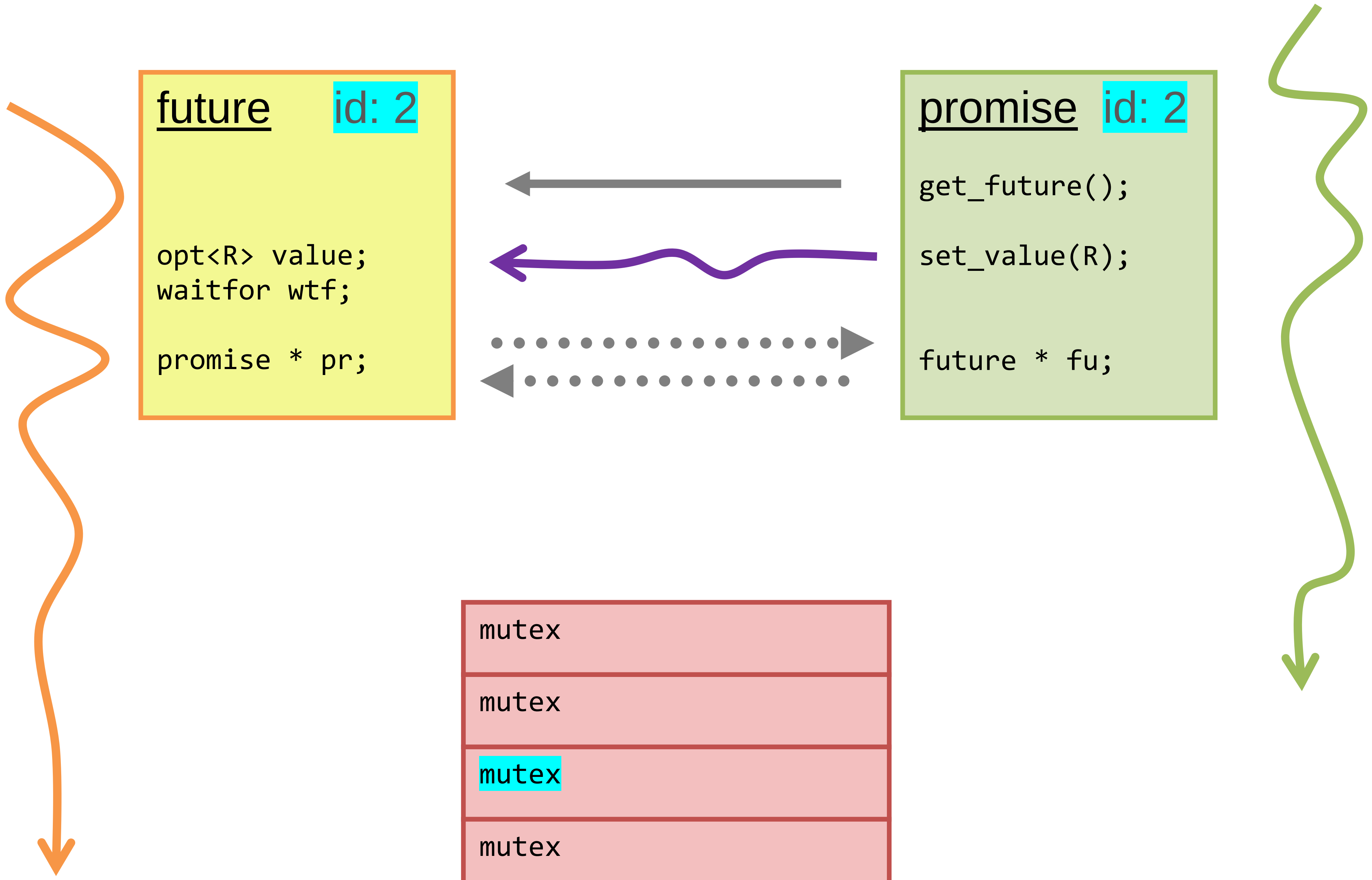

future
id: 2
opt<R> value;
waitfor wtf;
promise
id: 2
get_future();
set_value(R);
mutex
mutex
mutex
mutex

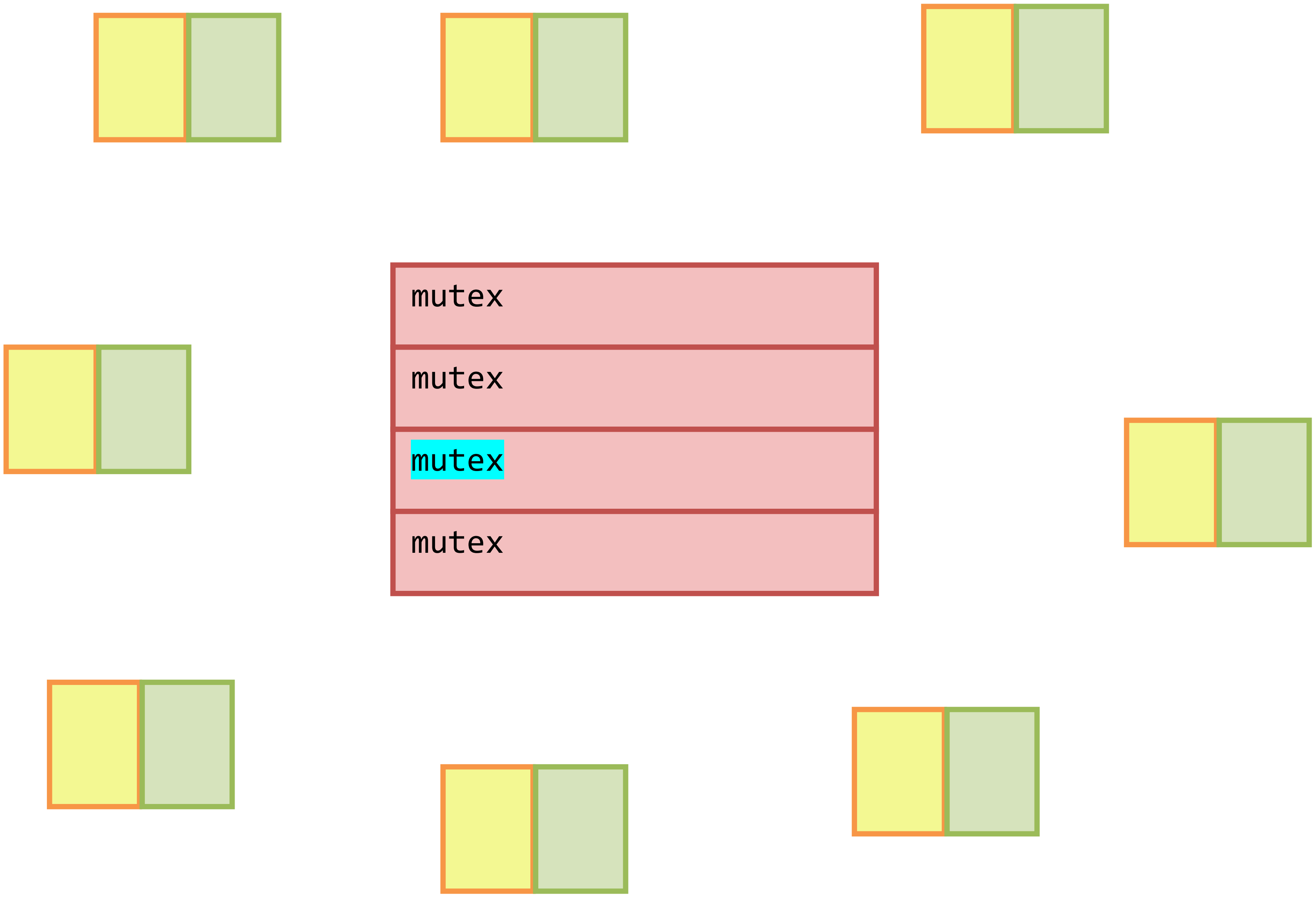


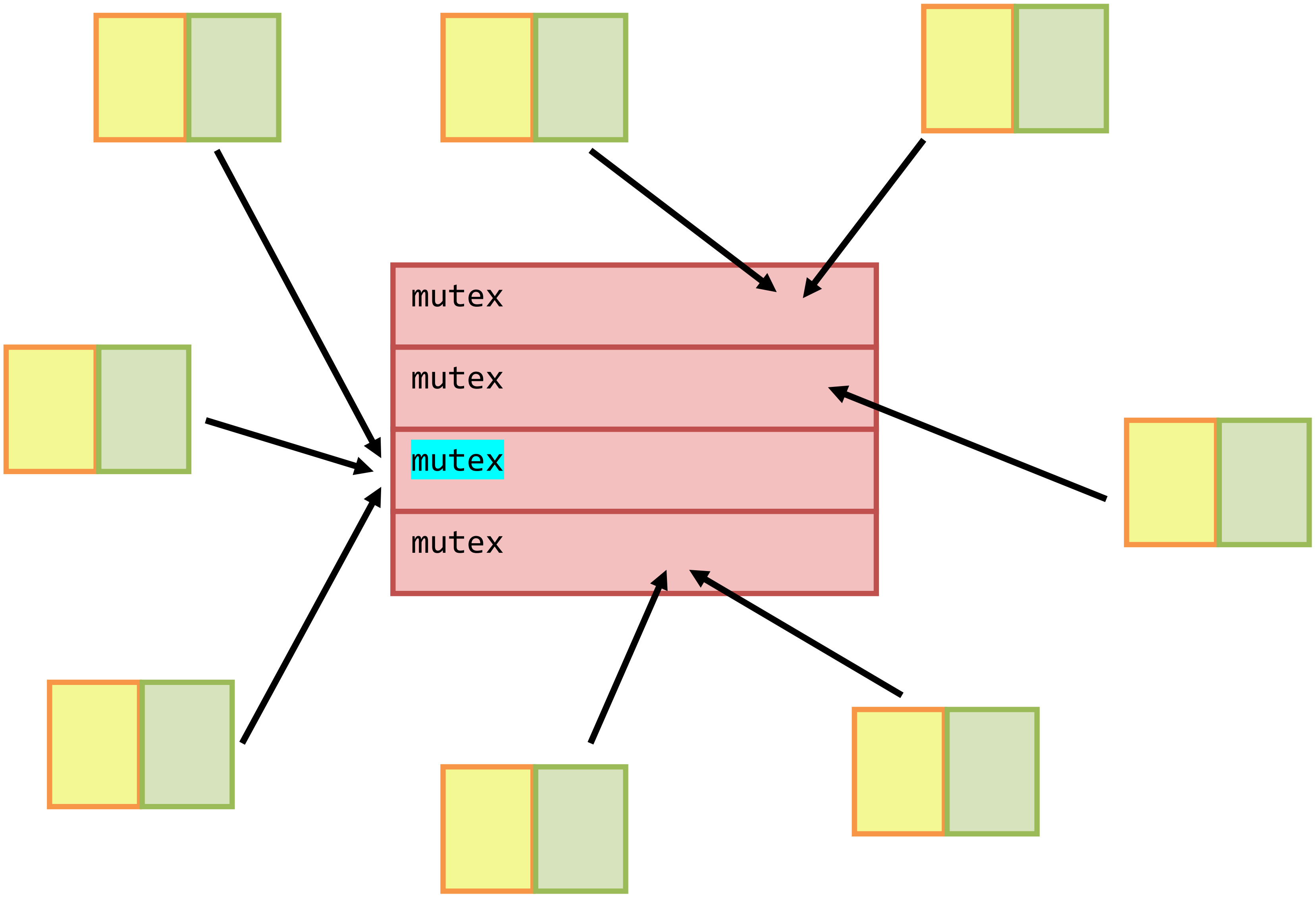
```
void promise::set_value(R value) {
  scoped_lock slock(mutex[id]);
  fu->value = value;
  fu->wtf.ready(true); // yay!!
}
```

```
void promise::set_value(R value) {
  scoped_lock slock(mutex[id]);
  fu->value = value;
  fu->wtf.ready(true); // yay!!
}

void future::mov(future * to) {
  scoped_lock slock(mutex[id]);
  pr->fu = to; // tell promise new address
  // move self...
}

void promise::mov(promise * to) {
  scoped_lock slock(mutex[id]);
  fu->pr = to; // tell future new address
  // move self...
}
```

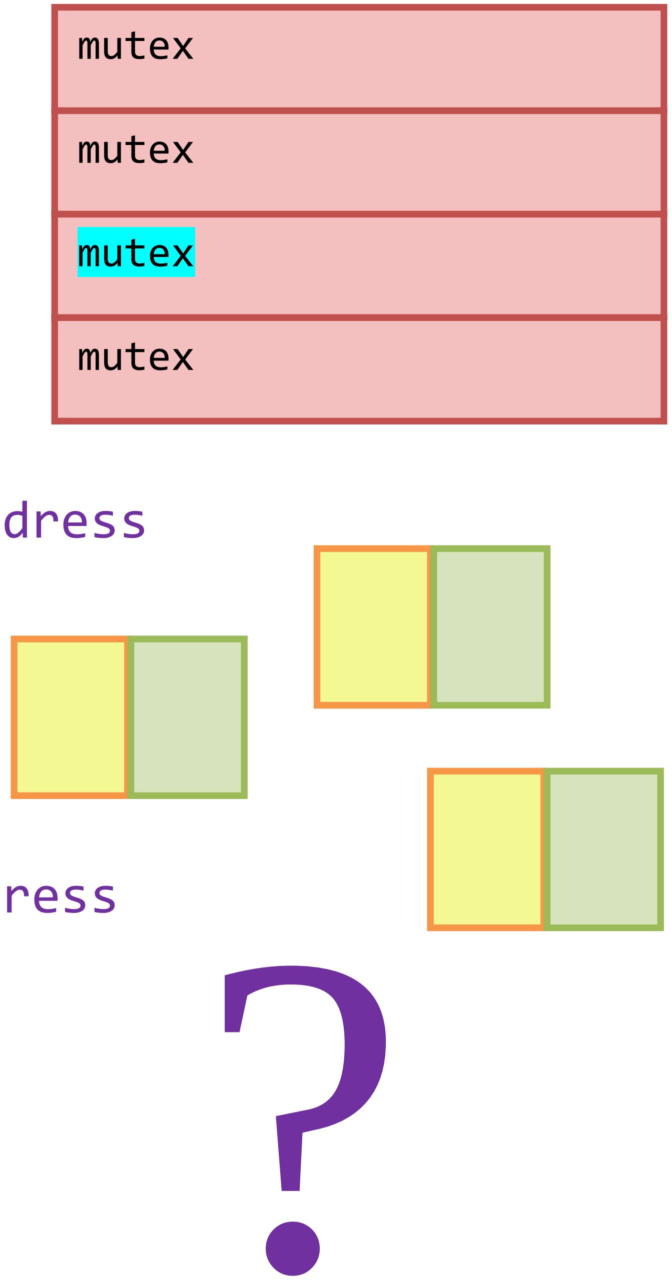



future
id: 2
opt<R> value;
waitfor wtf;
promise * pr;
promise
id: 2
get_future();
set_value(R);
future * fu;
mutex
mutex
mutex
mutex

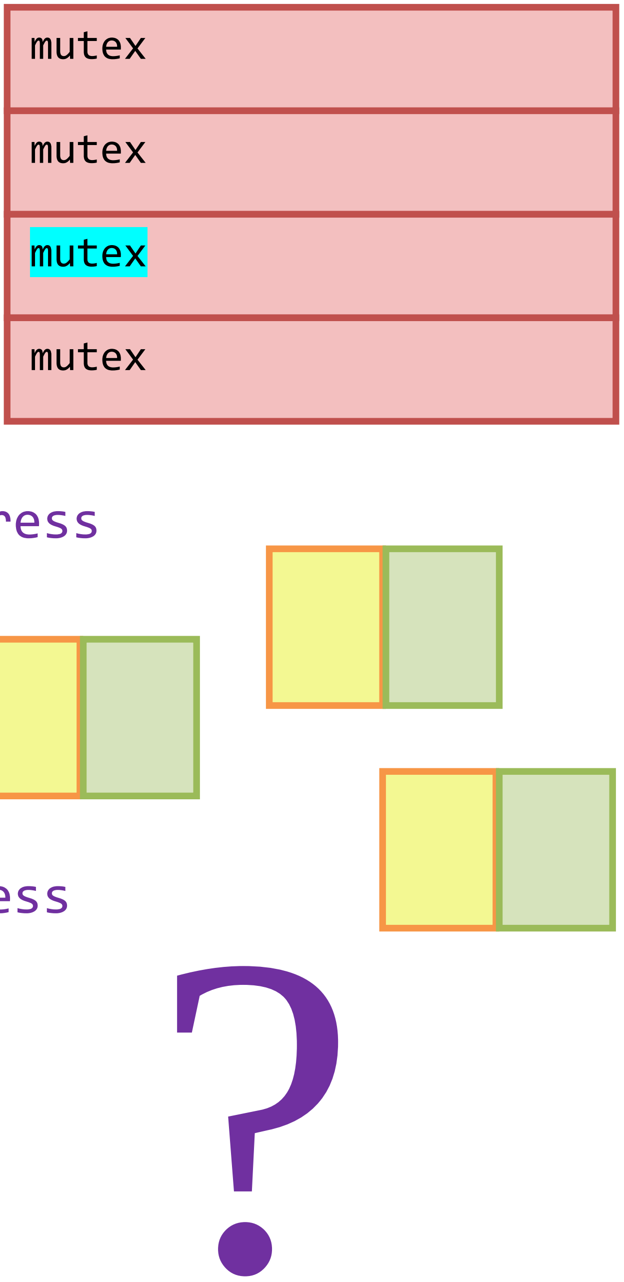

mutex
mutex
mutex
mutex


mutex
mutex
mutex
mutex

```
void promise::set_value(R value) {
  scoped_lock slock(mutex[id]);
  fu->value = value;
  fu->wtf.ready(true); // yay!!
}

void future::mov(future * to) {
  scoped_lock slock(mutex[id]);
  pr->fu = to; // tell promise new address
  // move self...
}

void promise::mov(promise * to) {
  scoped_lock slock(mutex[id]);
  fu->pr = to; // tell future new address
  // move self...
}
```

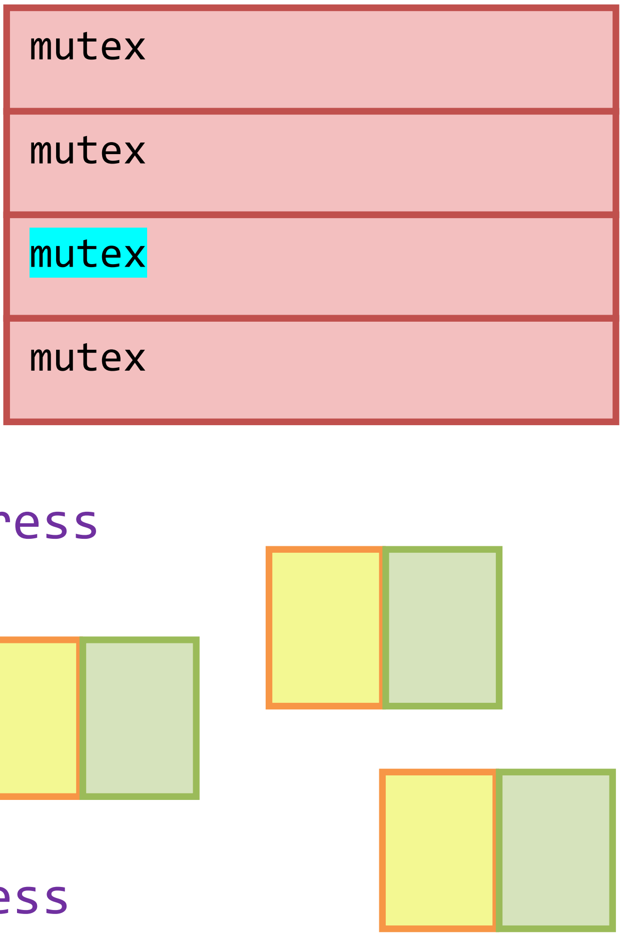


```
void promise::set_value(R value) {
  scoped_lock slock(mutex[id]);
  fu->value = value;
  fu->wtf.ready(true); // yay!!
}

void future::mov(future * to) {
  scoped_lock slock(mutex[id]);
  pr->fu = to; // tell promise new address
  // move self...
}

void promise::mov(promise * to) {
  scoped_lock slock(mutex[id]);
  fu->pr = to; // tell future new address
  // move self...
}
```

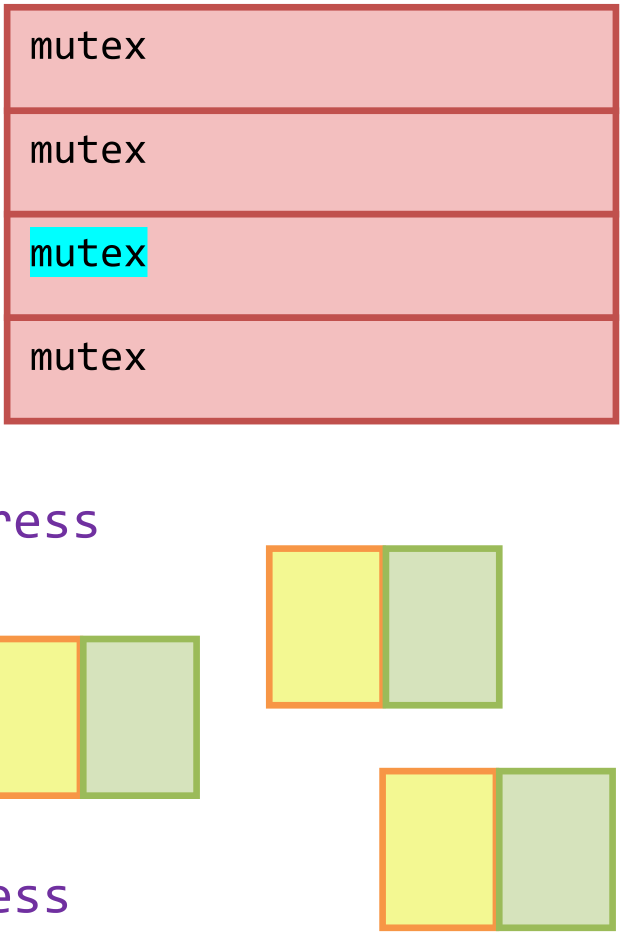


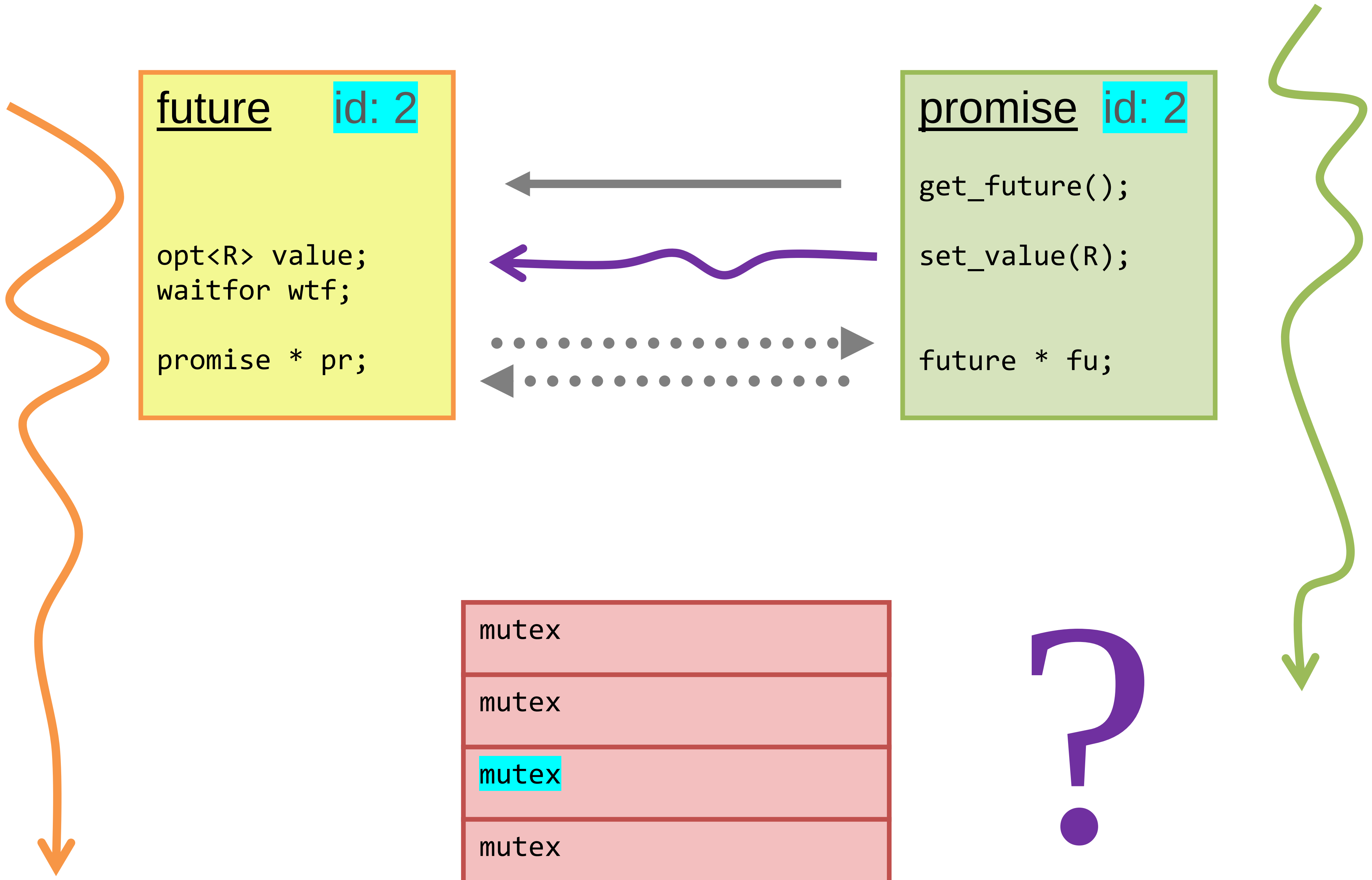
Rule #1: ***When holding a lock,***
DO NOT call unknown code.

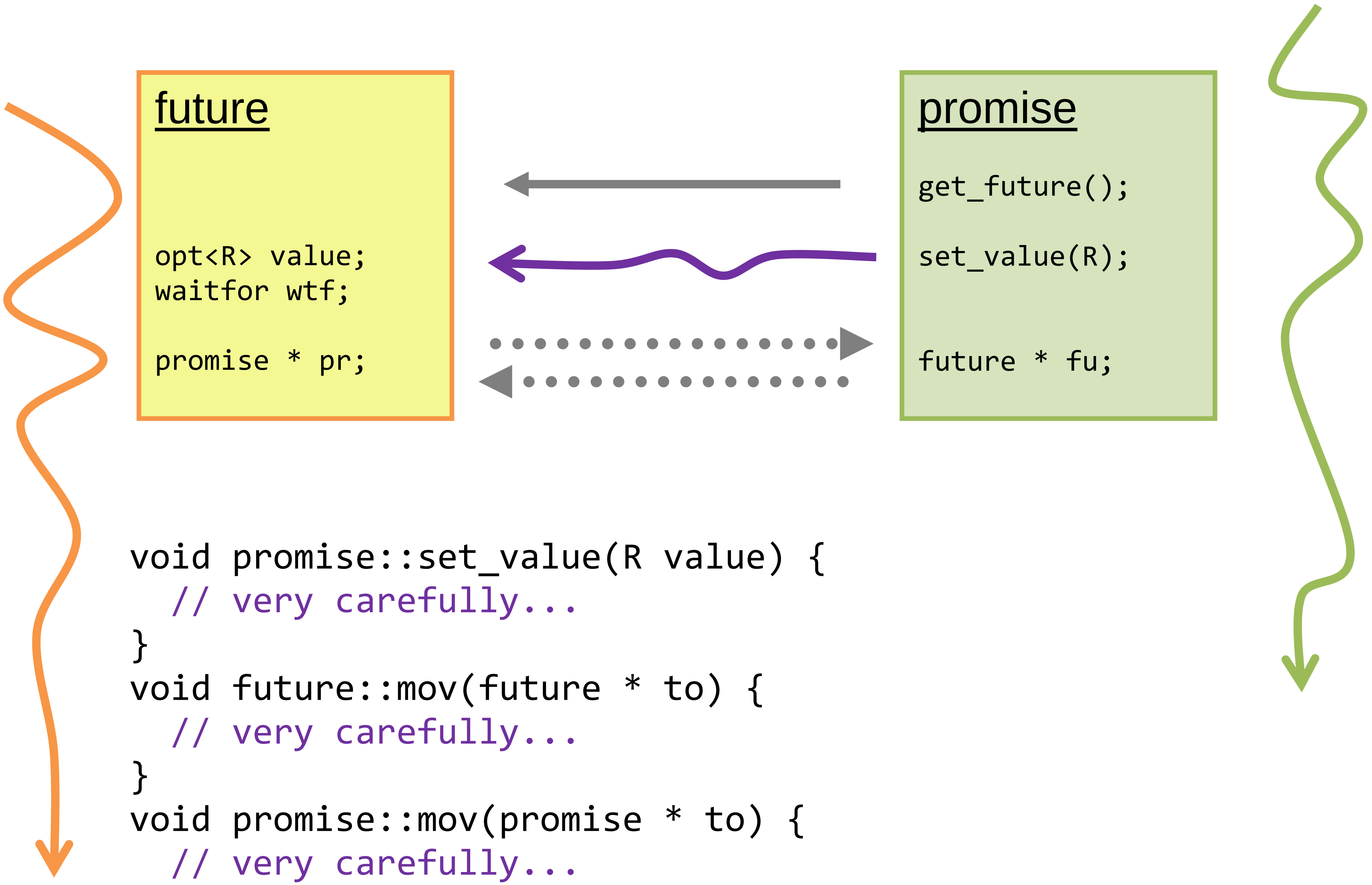
```
void promise::set_value(R value) {
  scoped_lock slock(mutex[id]);
  fu->value = value;
  fu->wtf.ready(true); // yay!!
}

void future::mov(future * to) {
  scoped_lock slock(mutex[id]);
  pr->fu = to; // tell promise new address
  // move self...
}

void promise::mov(promise * to) {
  scoped_lock slock(mutex[id]);
  fu->pr = to; // tell future new address
  // move self...
}
```



```
void promise::set_value(R value) {
  scoped_lock slock(mutex[id]);
  fu->value = value; // R(R&&)
  fu->wtf.ready(true); // yay!!
}

void future::mov(future * to) {
  scoped_lock slock(mutex[id]);
  pr->fu = to; // tell promise new address
  // move self...
}

void promise::mov(promise * to) {
  scoped_lock slock(mutex[id]);
  fu->pr = to; // tell future new address
  // move self...
}
```

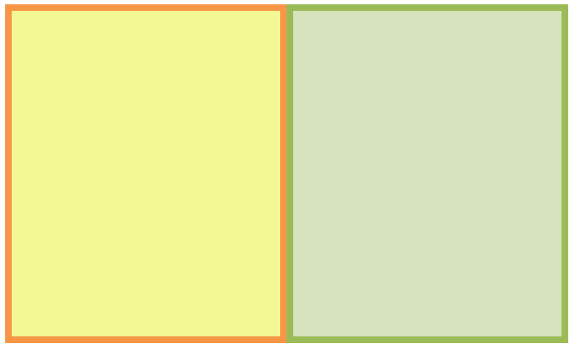



future
id: 2
opt<R> value;
waitfor wtf;
promise * pr;
promise
id: 2
get_future();
set_value(R);
future * fu;
mutex
mutex
mutex
mutex
?

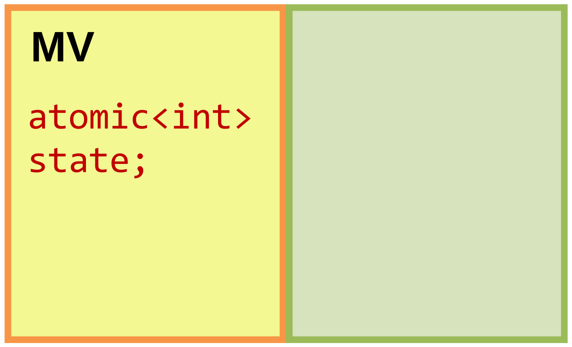


```
void promise::set_value(R value) {
  // very carefully...
}
void future::mov(future * to) {
  // very carefully...
}
void promise::mov(promise * to) {
  // very carefully...
}
```

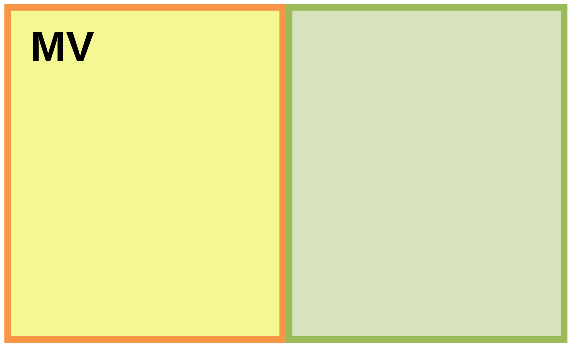
```
void promise::set_value(R value) {
  // very carefully...
  fu->value = value; // R(R&&)
  fu->wtf.ready(true); // yay!!
}
void future::mov(future * to) {
  // very carefully...
  pr->fu = to; // tell partner
  move_self(to);
}
void promise::mov(promise * to) {
  // very carefully...
  fu->pr = to; // tell partner
  move_self(to);
}
```



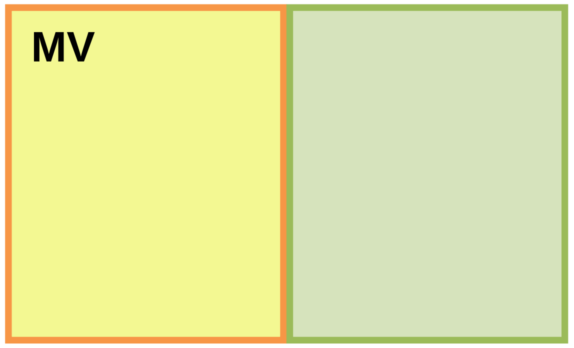
```
void promise::set_value(R value) {
  // very carefully...
  fu->value = value; // R(R&&)
  fu->wtf.ready(true); // yay!!
}
void future::mov(future * to) {
  state = MV;
  pr->fu = to; // tell partner
  move_self(to);
}
void promise::mov(promise * to) {
  // very carefully...
  fu->pr = to; // tell partner
  move_self(to);
}
```



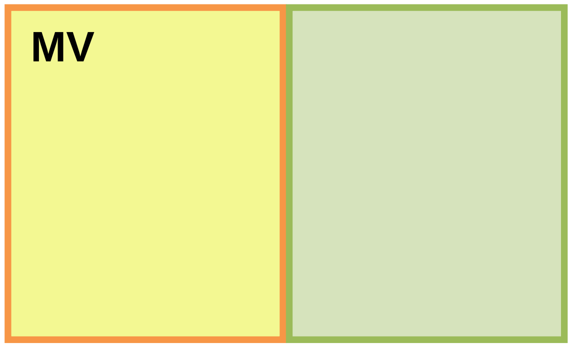
```
void promise::set_value(R value) {
  // very carefully...
  fu->value = value; // R(R&&)
  fu->wtf.ready(true); // yay!!
}
void future::mov(future * to) {
  state = MV;
  pr->fu = to; // tell partner
  move_self(to);
}
void promise::mov(promise * to) {
  while (fu->state != 0)
      ;
  fu->pr = to; // tell partner
  move_self(to);
}
```



```
void promise::set_value(R value) {
  // very carefully...
  fu->value = value; // R(R&&)
  fu->wtf.ready(true); // yay!!
}
void future::mov(future * to) {
  state = MV;
  pr->fu = to; // tell partner
  move_self(to);
}
void promise::mov(promise * to) {
  while (fu->state != 0)
      ;
  fu->pr = to; // tell partner
  move_self(to);
}
```

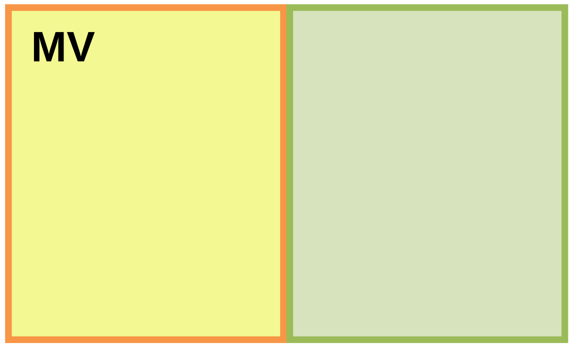


```
void promise::set_value(R value) {
  // very carefully...
  fu->value = value; // R(R&&)
  fu->wtf.ready(true); // yay!!
}
void future::mov(future * to) {
  state = MV;
  pr->fu = to; // tell partner
  move_self(to);
}
void promise::mov(promise * to) {
  while (fu->state != 0)
      ;
  fu->pr = to; // tell partner
  move_self(to);
}
```



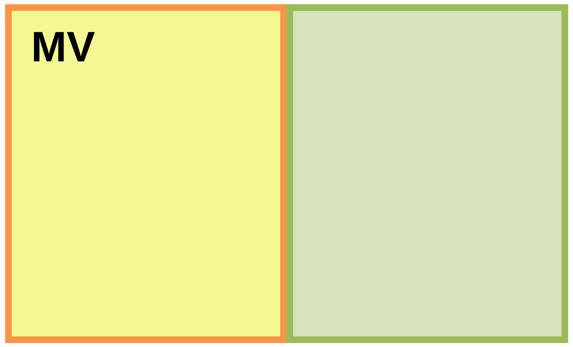
?

```
void promise::set_value(R value) {
  // very carefully...
  fu->value = value; // R(R&&)
  fu->wtf.ready(true); // yay!!
}
void future::mov(future * to) {
  state = MV;
  pr->fu = to; // tell partner
  move_self(to);
}
void promise::mov(promise * to) {
  while (fu->state != 0)
      ;
  fu->pr = to; // tell partner
  move_self(to);
}
```



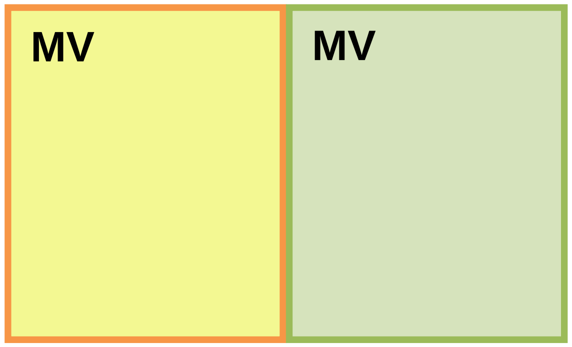
?

```
void promise::set_value(R value) {
  // very carefully...
  fu->value = value; // R(R&&)
  fu->wtf.ready(true); // yay!!
}
void future::mov(future * to) {
  state = MV;
  pr->fu = to; // tell partner
  move_self(to);
}
void promise::mov(promise * to) {
  while (fu->state != 0)
      ;
  fu->pr = to; // tell partner
  move_self(to);
}
```

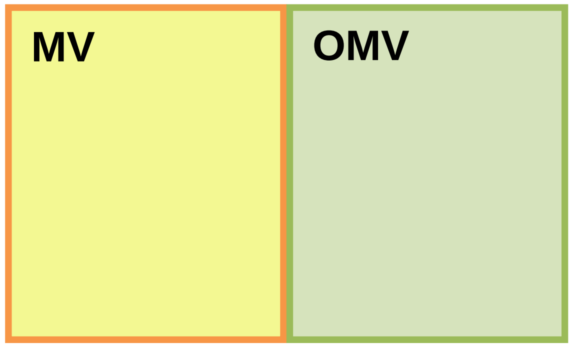


Safe or Safe not.
There is no
“Safer”.

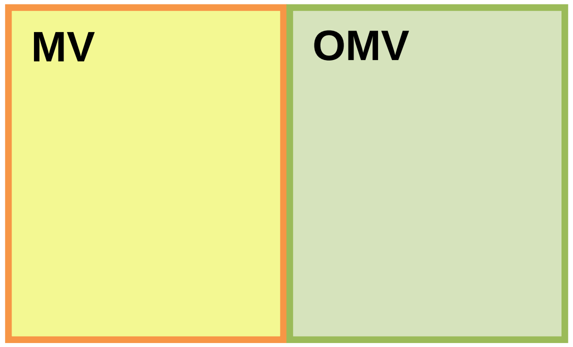
```
void promise::set_value(R value) {
   // very carefully...
   fu->value = value; // R(R&&)
   fu->wtf.ready(true); // yay!!
}
void future::mov(future * to) {
   state = MV;
   while (pr->state != 0)
         ;
   pr->fu = to; // tell partner
   move_self(to);
}
void promise::mov(promise * to) {
   state = MV;
   while (fu->state != 0)
         ;
   fu->pr = to; // tell partner
   move_self(to);
}
```



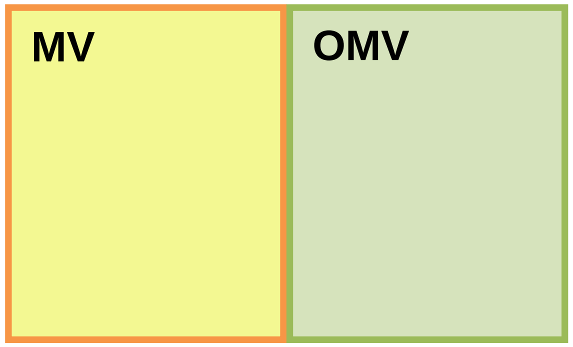
```
void promise::set_value(R value) {
 // very carefully...
 fu->value = value; // R(R&&)
 fu->wtf.ready(true); // yay!!
}
void future::mov(future * to) {
 state = MV;
 while (pr->state != 0)
   ;
 pr->fu = to; // tell partner
 move_self(to);
}
void promise::mov(promise * to) {
 state = MV;
 while (fu->state != 0)
   ;
 fu->pr = to; // tell partner
 move_self(to);
}
```



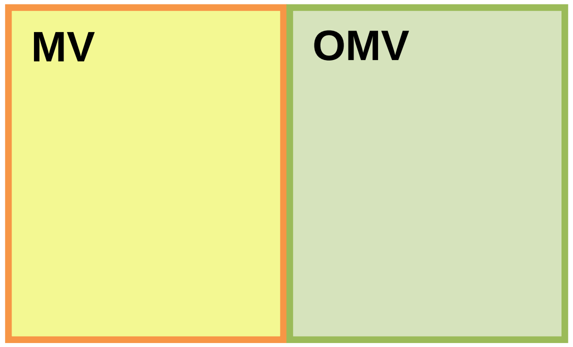
```
void promise::set_value(R value) {
  // very carefully...
  fu->value = value; // R(R&&)
  fu->wtf.ready(true); // yay!!
}
void future::mov(future * to) {
  while(!CAS(state, 0, MV)) pause();
  while(!CAS(pr->state, 0, OMV)) pause();
  pr->fu = to; // tell partner
  move_self(to);
}
void promise::mov(promise * to) {
  while(!CAS(state, 0, MV)) pause();
  while(!CAS(fu->state, 0, OMV)) pause();
  fu->pr = to; // tell partner
  move_self(to);
}
```



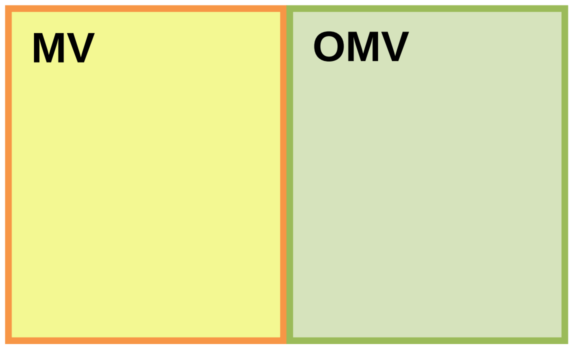
```
void promise::set_value(R value) {
  // very carefully...
  fu->value = value; // R(R&&)
  fu->wtf.ready(true); // yay!!
}
void future::mov(future * to) {
  while(!CAS(state, 0, MV)) pause();
  while(!CAS(pr->state, 0, OMV)) pause();
  pr->fu = to; // tell partner
  move_self(to);
}
void promise::mov(promise * to) {
  while(!CAS(state, 0, MV)) pause();
  while(!CAS(fu->state, 0, OMV)) pause();
  fu->pr = to; // tell partner
  move_self(to);
}
```



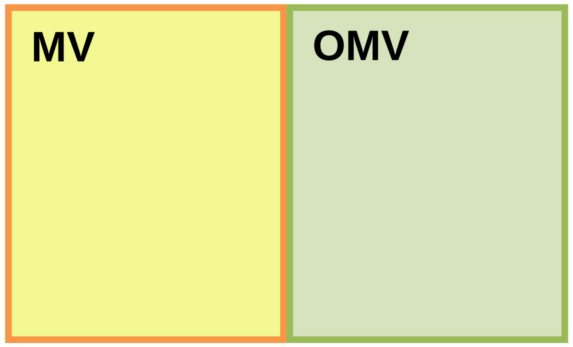
```
void promise::set_value(R value) {
  // very carefully...
  fu->value = value; // R(R&&)
  fu->wtf.ready(true); // yay!!
}
void future::mov(future * to) {
  while(!CAS(state, 0, MV)) pause();
  while(!CAS(pr->state, 0, OMV)) pause();
  pr->fu = to; // tell partner
  move_self(to);
}
void promise::mov(promise * to) {
  while(!CAS(state, 0, MV)) pause();
  while(!CAS(fu->state, 0, OMV)) pause();
  fu->pr = to; // tell partner
  move_self(to);
}
```



```
void promise::set_value(R value) {
  // very carefully...
  fu->value = value; // R(R&&)
  fu->wtf.ready(true); // yay!!
}
void future::mov(future * to) {
  while(!CAS(state, 0, MV)) pause();
  while(!CAS(pr->state, 0, OMV)) pause();
  pr->fu = to; // tell partner
  move_self(to);
}
void promise::mov(promise * to) {
  while(!CAS(state, 0, MV)) pause();
  while(!CAS(fu->state, 0, OMV)) pause();
  fu->pr = to; // tell partner
  move_self(to);
}
```

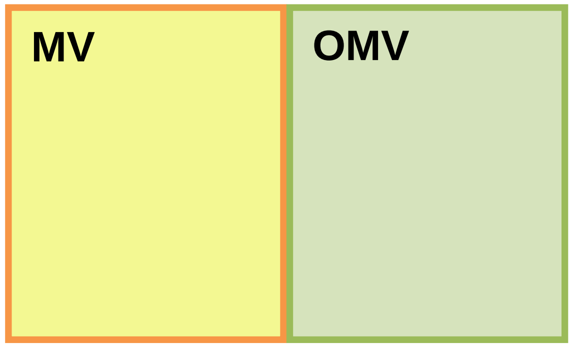


```
void promise::set_value(R value) {
  // very carefully...
  fu->value = value; // R(R&&)
  fu->wtf.ready(true); // yay!!
}
void future::mov(future * to) {
  while(!CAS(state, 0, MV)) pause();
  while(!CAS(pr->state, 0, OMV)) pause();
  pr->fu = to; // tell partner
  move_self(to);
}
void promise::mov(promise * to) {
  while(!CAS(state, 0, MV)) pause();
  while(!CAS(fu->state, 0, OMV)) pause();
  fu->pr = to; // tell partner
  move_self(to);
}
```



?

```
void promise::set_value(R value) {...}
void future::mov(future * to) {
  retry:
    while(!CAS(state, 0, MV)) pause();
    if (!CAS(pr->state, 0, OMV)) {
      state = 0;  pause();
      goto retry;
    }
  pr->fu = to; // tell partner
  move_self(to);
}
void promise::mov(promise * to) {
  retry:
    while(!CAS(state, 0, MV)) pause();
    if (!CAS(fu->state, 0, OMV)) {
      state = 0;  pause();
      goto retry;
    }
  fu->pr = to; // tell partner
  move_self(to);
}
```



```
void future::mov(future * to)
{
 retry:
   while(!CAS(state, 0, MV))
     pause();
   if(!CAS(pr->state, 0, OMV))
   {
     state = 0;
     pause();
     goto retry;
   }
 pr->fu = to; // tell partner
 move_self(to);
}
```

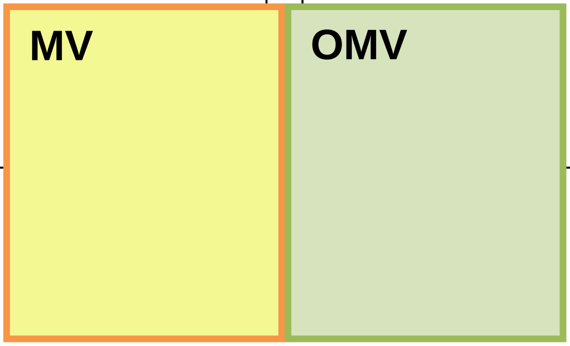
```
void promise::mov(promise * to)
{
 retry:
   while(!CAS(state, 0, MV))
     pause();
   if(!CAS(fu->state, 0, OMV))
   {
     state = 0;
     pause();
     goto retry;
   }
 fu->pr = to; // tell partner
 move_self(to);
}
```

```
void future::mov(future * to)
{
 retry:
   while(!CAS(state, 0, MV))
     pause();
   if(!CAS(pr->state, 0, OMV))
   {
     state = 0;
     pause();
     goto retry;
   }
 pr->fu = to; // tell partner
 move_self(to);
}
```

```
void promise::mov(promise * to)
{
 retry:
   while(!CAS(state, 0, MV))
     pause();
   if(!CAS(fu->state, 0, OMV))
   {
     state = 0;
     pause();        ?
     goto retry;
   }
 fu->pr = to; // tell partner
 move_self(to);
}
```

```
void future::mov(future * to)
{
 retry:
   while(!CAS(state, 0, MV))
     pause();
   if(!CAS(pr->state, 0, OMV))
   {
     state = 0;
     pause();
     goto retry;
   }
 pr->fu = to; // tell partner
 move_self(to);
}
```

```
void promise::mov(promise * to)
{
 retry:
   while(!CAS(state, 0, MV))
     pause();
   if(!CAS(fu->state, 0, OMV))
   {
     state = 0;
     pause();
     goto retry;
   }
 fu->pr = to; // tell partner
 move_self(to);
}
```



```
void future::mov(future * to)
{
 retry:
   while(!CAS(state, 0, MV))
     pause();
   if(!CAS(pr->state, 0, OMV))
   {
     state = 0;
     pause();
     goto retry;
   }
 pr->fu = to; // tell partner
 move_self(to);
}
```

```
void promise::mov(promise * to)
{
 retry:
   while(!CAS(state, 0, MV))
     pause();
   if(!CAS(fu->state, 0, OMV))
   {
     state = 0;
     pause();
     goto retry;
   }
 fu->pr = to; // tell partner
 move_self(to);
}
```

2 Rules:

- Ask before doing.
- Don’t leave before saying goodbye.

2 Rules:

- Ask before doing.
- Don’t leave before saying goodbye.

(be polite, eh?)

```
void future::mov(future * to)
{
 retry:
   while(!CAS(state, 0, MV))
     pause();
   if(!CAS(pr->state, 0, OMV))
   {
     state = 0;
     pause();
     goto retry;
   }
 pr->fu = to; // tell partner
 move_self(to);
}
```

```
void promise::mov(promise * to)
{
 retry:
   while(!CAS(state, 0, MV))
     pause();
   if(!CAS(fu->state, 0, OMV))
   {
     state = 0;
     pause();
     goto retry;
   }
 fu->pr = to; // tell partner
 move_self(to);
}
```

```
void future::mov(future * to)
{
 retry:
 while(!CAS(state, 0, MV))
   pause();
 if(!CAS(pr->state, 0, OMV)) {
   state = 0;  pause();
   goto retry;
 }
 pr->fu = to; // tell partner
 move_self(to);
}
```

```
void promise::mov(promise * to)
{
 retry:
 while(!CAS(state, 0, MV))
   pause();
 if(!CAS(fu->state, 0, OMV)) {
   state = 0;  pause();
   goto retry;
 }
 fu->pr = to; // tell partner
 move_self(to);
}
```

```
void future::mov(future * to)
{
 retry:
 while(!CAS(state, 0, MV))
   pause();
 if (pr) {
   if(!CAS(pr->state, 0, OMV)) {
     state = 0;  pause();
     goto retry;
   }
   pr->fu = to; // tell partner
 }
 move_self(to);
}
```

```
void promise::mov(promise * to)
{
 retry:
 while(!CAS(state, 0, MV))
   pause();
 if (fu) {
   if(!CAS(fu->state, 0, OMV)) {
     state = 0;  pause();
     goto retry;
   }
   fu->pr = to; // tell partner
 }
 move_self(to);
}
```

```
void future::mov(future * to)
{
 retry:
 while(!CAS(state, 0, MV))
   pause();
 if (pr) {
   if(!CAS(pr->state, 0, OMV)) {
     state = 0;  pause();
     goto retry;
   }
   pr->fu = to; // tell partner
 }
 move_self(to);
}
```

```
void promise::mov(promise * to)
{
 retry:
 while(!CAS(state, 0, MV))
   pause();
 if (fu) {
   if(!CAS(fu->state, 0, OMV)) {
     state = 0;  pause();
     goto retry;
   }
   fu->pr = to; // tell partner
 }
 move_self(to);
}
```

```
void future::mov(future * to)
{
 retry:
 while(!CAS(state, 0, MV))
   pause();
 if (pr) {
   if(!CAS(pr->state, 0, OMV)) {
     state = 0;  pause();
     goto retry;
   }
   pr->fu = to; // tell partner
 }
 move_self(to);
}
```

```
void promise::mov(promise * to)
{
 retry:
 while(!CAS(state, 0, MV))
   pause();
 if (fu) {
   if(!CAS(fu->state, 0, OMV)) {
     state = 0;  pause();
     goto retry;
   }
   fu->pr = to; // tell partner
 }
 move_self(to);
}
```

```
void future::mov(future * to)
{
 retry:
 while(!CAS(state, 0, MV))
   pause();
 if (pr) {
   if(!CAS(pr->state, 0, OMV)) {
     state = 0;  pause();
     goto retry;
   }
   pr->fu = to; // tell partner
 }
 move_self(to);
}
```

```
void promise::mov(promise * to)
{
 retry:
 while(!CAS(state, 0, MV))
   pause();
 if (fu) {
   if(!CAS(fu->state, 0, OMV)) {
     state = 0;  pause();
     goto retry;
   }
   fu->pr = to; // tell partner
 }
 move_self(to);
}
```

```
void future::mov(future * to)
{
 retry:
 while(!CAS(state, 0, MV))
   pause();
 if (pr) {
   if(!CAS(pr->state, 0, OMV)) {
     state = 0;  pause();
     goto retry;
   }
   pr->fu = to; // tell partner
 }
 move_self(to);
}
```

```
void promise::mov(promise * to)
{
 retry:
 while(!CAS(state, 0, MV))
   pause();
 if (fu) {
   if(!CAS(fu->state, 0, OMV)) {
     state = 0;  pause();
     goto retry;
   }
   fu->pr = to; // tell partner
 }
 move_self(to);
}
```

```
void future::mov(future * to)
{
 retry:
 while(!CAS(state, 0, MV))
   pause();
 if (pr) {
   if(!CAS(pr->state, 0, OMV)) {
     state = 0;  pause();
     goto retry;
   }
   pr->fu = to; // tell partner
 }
 move_self(to);
}
```

```
void promise::mov(promise * to)
{
 retry:
 while(!CAS(state, 0, MV))
   pause();
 if (fu) {
   if(!CAS(fu->state, 0, OMV)) {
     state = 0;  pause();
     goto retry;
   }
   fu->pr = to; // tell partner
 }
 move_self(to);
}
```

```
void future::mov(future * to)
{
 retry:
 while(!CAS(state, 0, MV))
   pause();
 if (pr) {
   if(!CAS(pr->state, 0, OMV)) {
     state = 0;  pause();
     goto retry;
   }
   pr->fu = to; // tell partner
 }
 move_self(to);
}
```

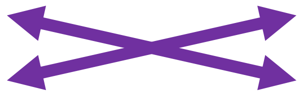
```
void promise::mov(promise * to)
{
 retry:
 while(!CAS(state, 0, MV))
   pause();
 if (fu) {
   if(!CAS(fu->state, 0, OMV)) {
     state = 0;  pause();
     goto retry;
   }
   fu->pr = to; // tell partner
 }
 move_self(to);
}
```

?

```
void future::mov(future * to)
{
 retry:
 while(!CAS(state, 0, MV))
   pause();
 if (pr) {
   if(!CAS(pr->state, 0, OMV)) {
     state = 0;  pause();
     goto retry;
   }
   pr->fu = to; // tell partner
 }
 move_self(to);
 pr->state = 0;
 state = 0;
}
```

```
void promise::mov(promise * to)
{
 retry:
 while(!CAS(state, 0, MV))
   pause();
 if (fu) {
   if(!CAS(fu->state, 0, OMV)) {
     state = 0;  pause();
     goto retry;
   }
   fu->pr = to; // tell partner
 }
 move_self(to);
 state = 0;
 fu->state = 0;
}
```

```
void future::mov(future * to)
{
 retry:
 while(!CAS(state, 0, MV))
   pause();
 if (pr) {
   if(!CAS(pr->state, 0, OMV)) {
     state = 0;  pause();
     goto retry;
   }
   pr->fu = to; // tell partner
 }
 move_self(to);
 pr->state = 0;
 state = 0;
}
```



```
void promise::mov(promise * to)
{
 retry:
 while(!CAS(state, 0, MV))
   pause();
 if (fu) {
   if(!CAS(fu->state, 0, OMV)) {
     state = 0;  pause();
     goto retry;
   }
   fu->pr = to; // tell partner
 }
 move_self(to);
 state = 0;
 fu->state = 0;
}
```

```
void future::mov(future * to)
{
 retry:
 while(!CAS(state, 0, MV))
   pause();
 if (pr) {
   if(!CAS(pr->state, 0, OMV)) {
     state = 0;  pause();
     goto retry;
   }
   pr->fu = to; // tell partner
 }
 move_self(to);
 pr->state = 0;
 state = 0;
}
```

```
void promise::mov(promise * to)
{
 retry:
 while(!CAS(state, 0, MV))
   pause();
 if (fu) {
   if(!CAS(fu->state, 0, OMV)) {
     state = 0;  pause();
     goto retry;
   }
   fu->pr = to; // tell partner
 }
 move_self(to);
 state = 0;
 fu->state = 0;
}
```

```
void future::mov(future * to)
{
 retry:
 while(!CAS(state, 0, MV))
   pause();
 if (pr) {
   if(!CAS(pr->state, 0, OMV)) {
     state = 0;  pause();
     goto retry;
   }
   pr->fu = to; // tell partner
 }
 move_self(to);
 pr->state = 0;
 state = 0;
}
```

```
void promise::mov(promise * to)
{
 retry:
 while(!CAS(state, 0, MV))
   pause();
 if (fu) {
   if(!CAS(fu->state, 0, OMV)) {
     state = 0;  pause();
     goto retry;
   }
   fu->pr = to; // tell partner
 }
 move_self(to);
 state = 0;
 fu->state = 0;
}
```

```
void future::mov(future * to)
{
 retry:
 while(!CAS(state, 0, MV))
   pause();
 if (pr) {
   if(!CAS(pr->state, 0, OMV)) {
     state = 0;  pause();
     goto retry;
   }
   pr->fu = to; // tell partner
 }
 move_self(to);
 pr->state = 0;
 state = 0;
}
```

```
void promise::mov(promise * to)
{
 retry:
 while(!CAS(state, 0, MV))
   pause();
 if (fu) {
   if(!CAS(fu->state, 0, OMV)) {
     state = 0;  pause();
     goto retry;
   }
   fu->pr = to; // tell partner
 }
 move_self(to);
 state = 0;
 fu->state = 0;
}
```

```
void future::mov(future * to)
{
 retry:
 while(!CAS(state, 0, MV))
   pause();
 if (pr) {
   if(!CAS(pr->state, 0, OMV)) {
     state = 0;  pause();
     goto retry;
   }
   pr->fu = to; // tell partner
 }
 move_self(to);
 pr->state = 0;
 state = 0;
}
```

```
void promise::mov(promise * to)
{
 retry:
 while(!CAS(state, 0, MV))
   pause();
 if (fu) {
   if(!CAS(fu->state, 0, OMV)) {
     state = 0;  pause();
     goto retry;
   }
   fu->pr = to; // tell partner
 }
 move_self(to);
 state = 0;
 fu->state = 0;
}
```

```
void future::mov(future * to)
{
 retry:
 while(!CAS(state, 0, MV))
   pause();
 if (pr) {
   if(!CAS(pr->state, 0, OMV)) {
     state = 0;  pause();
     goto retry;
   }
   pr->fu = to; // tell partner
 }
 move_self(to);
 pr->state = 0;
 state = 0;
}
```

```
void promise::mov(promise * to)
{
 retry:
 while(!CAS(state, 0, MV))
   pause();
 if (fu) {
   if(!CAS(fu->state, 0, OMV)) {
     state = 0;  pause();
     goto retry;
   }
   fu->pr = to; // tell partner
 }
 move_self(to);
 state = 0;
 fu->state = 0;
}
```

```
void future::mov(future * to)
{
 retry:
 while(!CAS(state, 0, MV))
   pause();
 if (pr) {
   if(!CAS(pr->state, 0, OMV)) {
     state = 0;  pause();
     goto retry;
   }
   pr->fu = to; // tell partner
 }
 move_self(to);
 pr->state = 0;
 state = 0;
}
```

```
void promise::mov(promise * to)
{
 retry:
 while(!CAS(state, 0, MV))
   pause();
 if (fu) {
   if(!CAS(fu->state, 0, OMV)) {
     state = 0;  pause();
     goto retry;
   }
   fu->pr = to; // tell partner
 }
 move_self(to);
 state = 0;
 fu->state = 0;
}
```

```
void future::mov(future * to)
{
 retry:
 while(!CAS(state, 0, MV))
   pause();
 if (pr) {
   if(!CAS(pr->state, 0, OMV)) {
     state = 0;  pause();
     goto retry;
   }
   pr->fu = to; // tell partner
 }
 move_self(to);
 pr->state = 0;
 state = 0;
}
```

```
void promise::mov(promise * to)
{
 retry:
 while(!CAS(state, 0, MV))
   pause();
 if (fu) {
   if(!CAS(fu->state, 0, OMV)) {
     state = 0;  pause();
     goto retry;
   }
   fu->pr = to; // tell partner
 }
 move_self(to);
 state = 0;
 fu->state = 0;
}
```

```
void future::mov(future * to)
{
 retry:
 while(!CAS(state, 0, MV))
   pause();
 if (pr) {
   if(!CAS(pr->state, 0, OMV)) {
     state = 0;  pause();
     goto retry;
   }
   pr->fu = to; // tell partner
 }
 move_self(to);
 pr->state = 0;
 state = 0;
}
```

✓

```
void promise::mov(promise * to)
{
 retry:
 while(!CAS(state, 0, MV))
   pause();
 if (fu) {
   if(!CAS(fu->state, 0, OMV)) {
     state = 0;  pause();
     goto retry;
   }
   fu->pr = to; // tell partner
 }
 move_self(to);
 state = 0;
 fu->state = 0;
}
```

```
void future::mov(future * to)
{
 retry:
 while(!CAS(state, 0, MV))
   pause();
 if (pr) {
   if(!CAS(pr->state, 0, OMV)) {
     state = 0;  pause();
     goto retry;
   }
   pr->fu = to; // tell partner
 }
 move_self(to);
 pr->state = 0;
 state = 0;
}
```

```
void promise::mov(promise * to)
{
 retry:
 while(!CAS(state, 0, MV))
   pause();
 if (fu) {
   if(!CAS(fu->state, 0, OMV)) {
     state = 0;  pause();
     goto retry;
   }
   fu->pr = to; // tell partner
 }
 move_self(to);
 state = 0;
 fu->state = 0;
}
```

```
void future::mov(future * to)
{
 retry:
 while(!CAS(state, 0, MV))
   pause();
 if (pr) {
   if(!CAS(pr->state, 0, OMV)) {
     state = 0;  pause();
     goto retry;
   }
   pr->fu = to; // tell partner
 }
 move_self(to);
 pr->state = 0;
 state = 0;
}
```

```
void promise::mov(promise * to)
{
 retry:
 while(!CAS(state, 0, MV))
   pause();
 if (fu) {
   if(!CAS(fu->state, 0, OMV)) {
     state = 0;  pause();
     goto retry;
   }
   fu->pr = to; // tell partner
 }
 move_self(to);
 state = 0;
 fu->state = 0;
}
```

```
void future::mov(future * to)
{
 retry:
 while(!CAS(state, 0, MV))
   pause();
 if (pr) {
   if(!CAS(pr->state, 0, OMV)) {
     state = 0;  pause();
     goto retry;
   }
   pr->fu = to; // tell partner
 }
 move_self(to);
 pr->state = 0;
 state = 0;
}
```

```
void promise::mov(promise * to)
{
 retry:
 while(!CAS(state, 0, MV))
   pause();
 if (fu) {
   if(!CAS(fu->state, 0, OMV)) {
     state = 0;  pause();
     goto retry;
   }
   fu->pr = to; // tell partner
 }
 move_self(to);
 state = 0;
 fu->state = 0;
}
```

```
void future::mov(future * to)
{
 retry:
 while(!CAS(state, 0, MV))
   pause();
 if (pr) {
   if(!CAS(pr->state, 0, OMV)) {
     state = 0;  pause();
     goto retry;
   }
   pr->fu = to; // tell partner
 }
 move_self(to);
 pr->state = 0;
 state = 0;
}
```

```
void promise::mov(promise * to)
{
 retry:
 while(!CAS(state, 0, MV))
   pause();
 if (fu) {
   if(!CAS(fu->state, 0, OMV)) {
     state = 0;  pause();
     goto retry;
   }
   fu->pr = to; // tell partner
 }
 move_self(to);
 state = 0;
 fu->state = 0;
}
```

```
void future::mov(future * to)
{
 retry:
 while(!CAS(state, 0, MV))
   pause();
 if (pr) {
   if(!CAS(pr->state, 0, OMV)) {
     state = 0;  pause();
     goto retry;
   }
   pr->fu = to; // tell partner
 }
 move_self(to);
 pr->state = 0;
 state = 0;
}
```

```
void promise::mov(promise * to)
{
 retry:
 while(!CAS(state, 0, MV))
   pause();
 if (fu) {
   if(!CAS(fu->state, 0, OMV)) {
     state = 0;  pause();
     goto retry;
   }
   fu->pr = to; // tell partner
 }
 move_self(to);
 state = 0;
 fu->state = 0;       ✓✓
}
```

```
void future::mov(future * to)
{
 retry:
 while(!CAS(state, 0, MV))
   pause();
 if (pr) {
   if(!CAS(pr->state, 0, OMV)) {
     state = 0;  pause();
     goto retry;
   }
   pr->fu = to; // tell partner
 }
 move_self(to);
 pr->state = 0;
 state = 0;                ✓
}
```

```
void promise::mov(promise * to)
{
 retry:
 while(!CAS(state, 0, MV))
   pause();
 if (fu) {
   if(!CAS(fu->state, 0, OMV)) {
     state = 0;  pause();
     goto retry;
   }
   fu->pr = to; // tell partner
 }
 move_self(to);
 state = 0;
 fu->state = 0;            ✓
}
```

```
void future::mov(future * to)
{
 retry:
 while(!CAS(state, 0, MV))
   pause();
 if (pr) {
   if(!CAS(pr->state, 0, OMV)) {
     state = 0;  pause();
     goto retry;
   }
   pr->fu = to; // tell partner
 }
 move_self(to);
 pr->state = 0;
 state = 0;
}
```

```
void promise::mov(promise * to)
{
 retry:
 while(!CAS(state, 0, MV))
   pause();
 if (fu) {
   if(!CAS(fu->state, 0, OMV)) {
     state = 0;  pause();
     goto retry;
   }
   fu->pr = to; // tell partner
 }
 move_self(to);
 state = 0;
 fu->state = 0;
}
```

```
void future::mov(future * to)
{
 retry:
 while(!CAS(state, 0, MV))
   pause();
 if (pr) {
   if(!CAS(pr->state, 0, OMV)) {
     state = 0;  pause();
     goto retry;
   }
   pr->fu = to; // tell partner
 }
 move_self(to);
 pr->state = 0;
 state = 0;
}
```

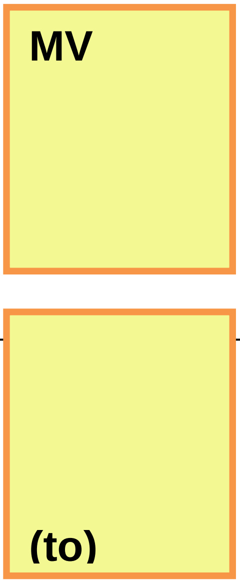
```
void promise::mov(promise * to)
{
 retry:
 while(!CAS(state, 0, MV))
   pause();
 if (fu) {
   if(!CAS(fu->state, 0, OMV)) {
     state = 0;  pause();
     goto retry;
   }
   fu->pr = to; // tell partner
 }
 move_self(to);
 state = 0;
 fu->state = 0;
```

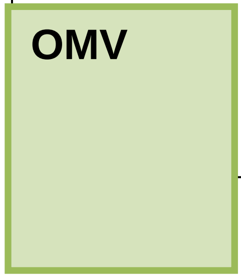
MV

OMV

```
void future::mov(future * to)
{
 retry:
 while(!CAS(state, 0, MV))
   pause();
 if (pr) {
   if(!CAS(pr->state, 0, OMV)) {
     state = 0;  pause();
     goto retry;
   }
   pr->fu = to; // tell partner
 }
 move_self(to);
 pr->state = 0;
 state = 0;
}
```

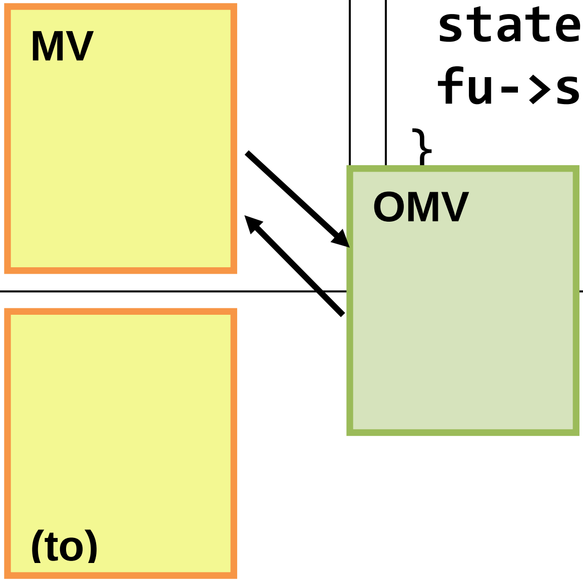
```
void promise::mov(promise * to)
{
 retry:
 while(!CAS(state, 0, MV))
   pause();
 if (fu) {
   if(!CAS(fu->state, 0, OMV)) {
     state = 0;  pause();
     goto retry;
   }
   fu->pr = to; // tell partner
 }
 move_self(to);
 state = 0;
 fu->state = 0;
```





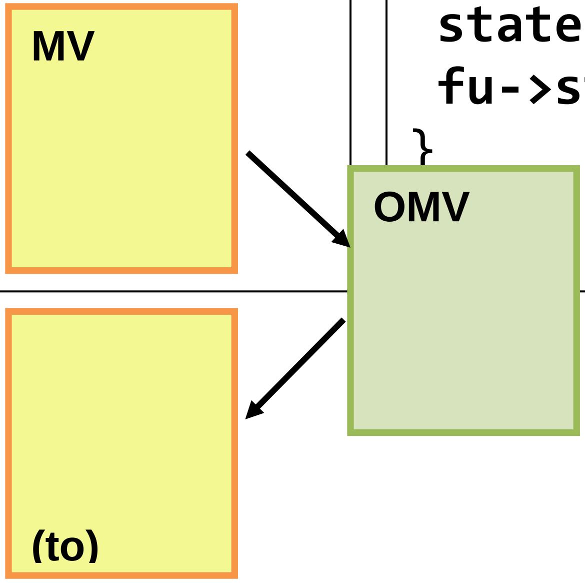
```
void future::mov(future * to)
{
 retry:
 while(!CAS(state, 0, MV))
   pause();
 if (pr) {
   if(!CAS(pr->state, 0, OMV)) {
     state = 0;  pause();
     goto retry;
   }
   pr->fu = to; // tell partner
 }
 move_self(to);
 pr->state = 0;
 state = 0;
}
```

```
void promise::mov(promise * to)
{
 retry:
 while(!CAS(state, 0, MV))
   pause();
 if (fu) {
   if(!CAS(fu->state, 0, OMV)) {
     state = 0;  pause();
     goto retry;
   }
   fu->pr = to; // tell partner
 }
 move_self(to);
 state = 0;
 fu->state = 0;
}
```



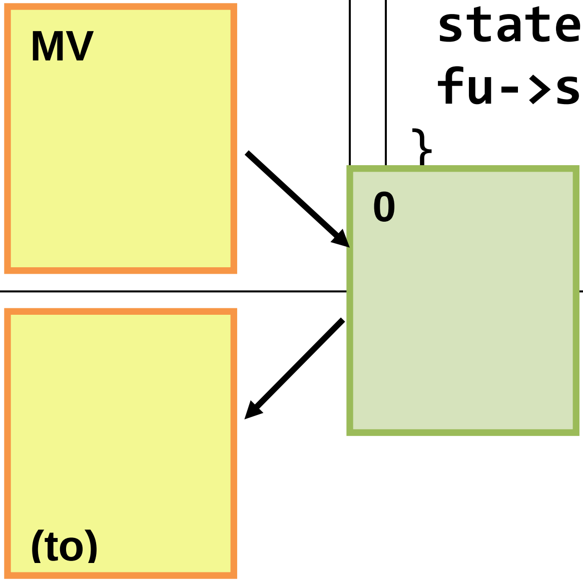
```
void future::mov(future * to)
{
 retry:
 while(!CAS(state, 0, MV))
   pause();
 if (pr) {
   if(!CAS(pr->state, 0, OMV)) {
     state = 0;  pause();
     goto retry;
   }
   pr->fu = to; // tell partner
 }
 move_self(to);
 pr->state = 0;
 state = 0;
}
```

```
void promise::mov(promise * to)
{
 retry:
 while(!CAS(state, 0, MV))
   pause();
 if (fu) {
   if(!CAS(fu->state, 0, OMV)) {
     state = 0;  pause();
     goto retry;
   }
   fu->pr = to; // tell partner
 }
 move_self(to);
 state = 0;
 fu->state = 0;
}
```



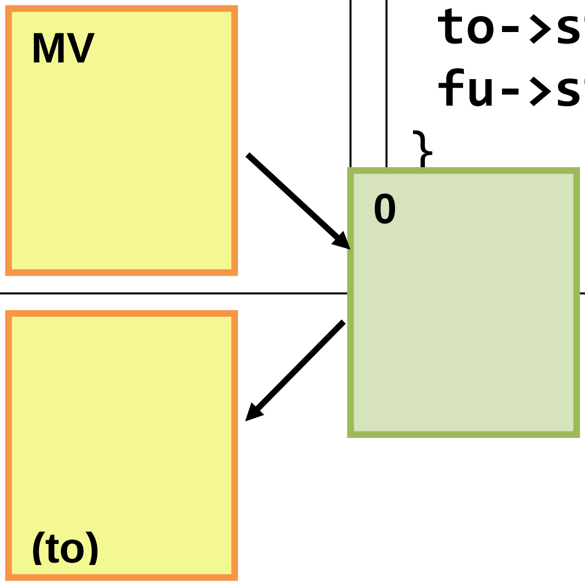
```
void future::mov(future * to)
{
 retry:
 while(!CAS(state, 0, MV))
   pause();
 if (pr) {
   if(!CAS(pr->state, 0, OMV)) {
     state = 0;  pause();
     goto retry;
   }
   pr->fu = to; // tell partner
 }
 move_self(to);
 pr->state = 0;
 state = 0;
}
```

```
void promise::mov(promise * to)
{
 retry:
 while(!CAS(state, 0, MV))
   pause();
 if (fu) {
   if(!CAS(fu->state, 0, OMV)) {
     state = 0;  pause();
     goto retry;
   }
   fu->pr = to; // tell partner
 }
 move_self(to);
 state = 0;
 fu->state = 0;
}
```



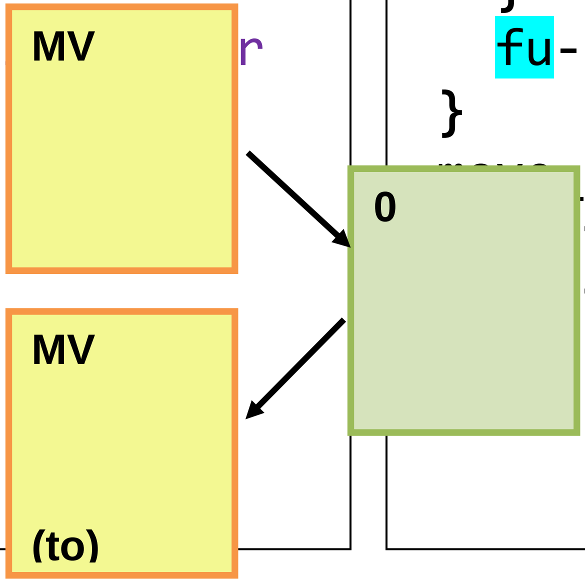
```
void future::mov(future * to)
{
 retry:
 while(!CAS(state, 0, MV))
   pause();
 if (pr) {
   if(!CAS(pr->state, 0, OMV)) {
     state = 0;  pause();
     goto retry;
   }
   pr->fu = to; // tell partner
 }
 move_self(to);
 pr->state = 0;
 to->state = 0;
}
```

```
void promise::mov(promise * to)
{
 retry:
 while(!CAS(state, 0, MV))
   pause();
 if (fu) {
   if(!CAS(fu->state, 0, OMV)) {
     state = 0;  pause();
     goto retry;
   }
   fu->pr = to; // tell partner
 }
 move_self(to);
 to->state = 0;
 fu->state = 0;
}
```



```
void future::mov(future * to)
{
 to->state = MV;
 to->pr = pr;
 prep(to);

 retry:
 while(!CAS(state, 0, MV))
   pause();
 if (pr) {
   if(!CAS(pr->state, 0, OMV)) {
     state = 0;  pause();
     goto retry;
   }
   pr->fu = to; // tel          r
 }
 move_self(to);
 pr->state = 0;
 to->state = 0;
}
```

```
void promise::mov(promise * to)
{
 to->state = MV;
 to->fu = fu;
 prep(to);

 retry:
 while(!CAS(state, 0, MV))
   pause();
 if (fu) {
   if(!CAS(fu->state, 0, OMV)) {
     state = 0;  pause();
     goto retry;
   }
   fu->pr = to; // tell partner
 }
        self(to);
        tate = 0;
        tate = 0;
```



```
void future::mov(future * to)
{
 to->state = MV;
 to->pr = pr;
 prep(to);

 retry:
 while(!CAS(state, 0, MV))
   pause();
 if (pr) {
   if(!CAS(pr->state, 0, OMV)) {
     state = 0;  pause();
     goto retry;
   }
   pr->fu = to; // tell partner
 }
 move_self(to);
 pr->state = 0;
 to->state = 0;
}
```

```
void promise::mov(promise * to)
{
 to->state = MV;
 to->fu = fu;
 prep(to);

 retry:
 while(!CAS(state, 0, MV))
   pause();
 if (fu) {
   if(!CAS(fu->state, 0, OMV)) {
     state = 0;  pause();
     goto retry;
   }
   fu->pr = to; // tell partner
 }
 move_self(to);
 to->state = 0;
 fu->state = 0;
}
```

```
void future::mov(future * to)
{
 to->state = MV;
 to->pr = pr;
 prep(to);

 retry:
 while(!CAS(state, 0, MV))
   pause();
 if (pr) {
   if(!CAS(pr->state, 0, OMV)) {
     state = 0;  pause();
     goto retry;
   }
   pr->fu = to; // tell partner
 }
 move_self(to);
 pr->state = 0;
 to->state = 0;
}
```

```
void promise::mov(promise * to)
{
 to->state = MV;
 to->fu = fu;
 prep(to);

 retry:
 while(!CAS(state, 0, MV))
   pause();
 if (fu) {
   if(!CAS(fu->state, 0, OMV)) {
     state = 0;  pause();
     goto retry;
   }
   fu->pr = to; // tell partner
 }
 move_self(to);
 to->state = 0;
 fu->state = 0;
}
```

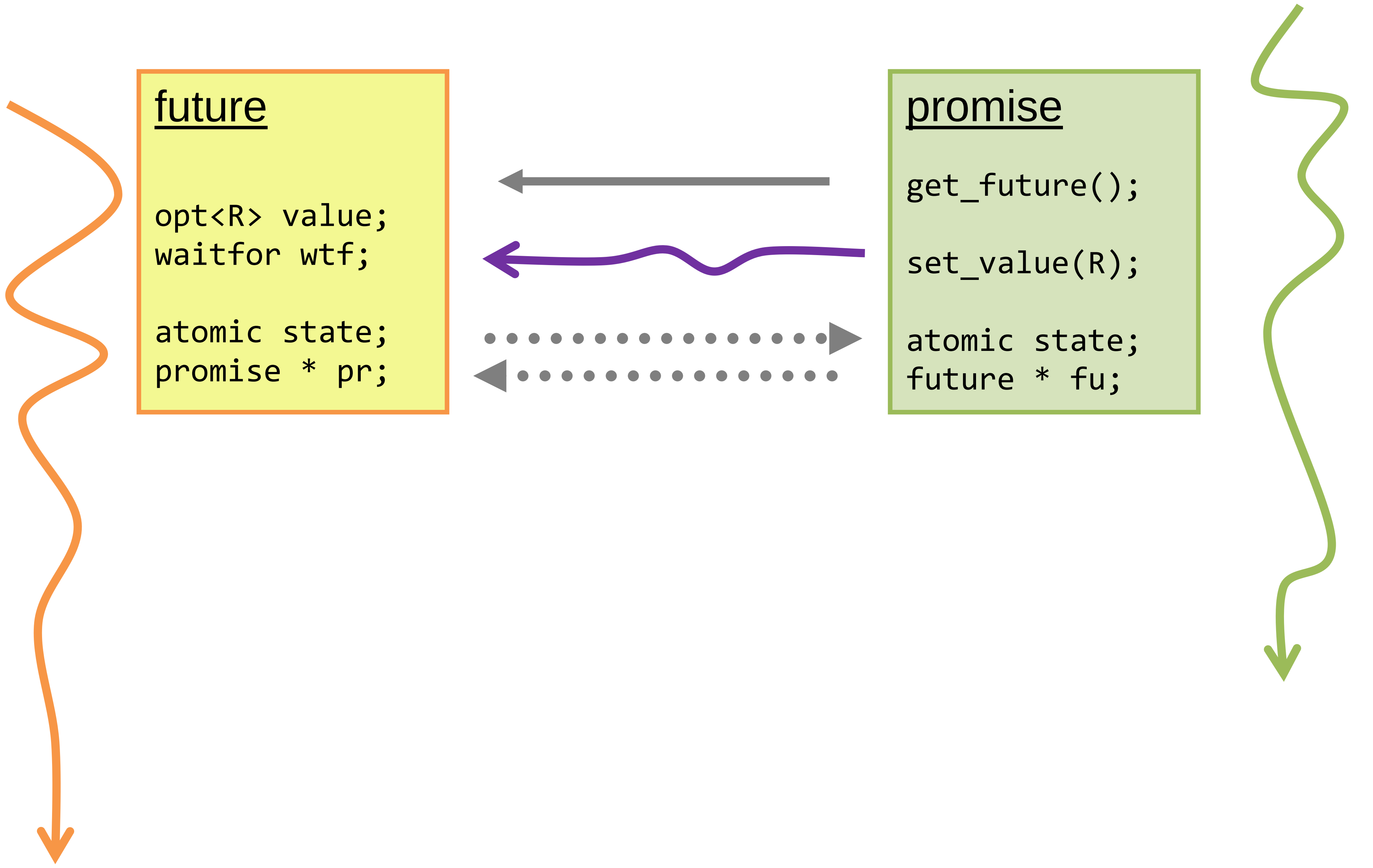
```
void future::mov(future * to)
{
 to->state = MV;
 to->pr = pr;
 prep(to);

 retry:
 while(!CAS(state, 0, MV))
   pause();
 if (pr) {
   if(!CAS(pr->state, 0, OMV)) {
     state = 0;  pause();
     goto retry;
   }
   pr->fu = to; // tell partner
 }
 move_self(to);
 pr->state = 0;
 to->state = 0;
}
```

```
to->mov(0);
```

```
void promise::mov(promise * to)
{
 to->state = MV;
 to->fu = fu;
 prep(to);

 retry:
 while(!CAS(state, 0, MV))
   pause();
 if (fu) {
   if(!CAS(fu->state, 0, OMV)) {
     state = 0;  pause();
     goto retry;
   }
   fu->pr = to; // tell partner
 }
 move_self(to);
 to->state = 0;
 fu->state = 0;
}
```

```
void future::mov(future * to)
{
 to->state = MV;
 to->pr = pr;
 prep(to);     <-

 retry:
 while(!CAS(state, 0, MV))
   pause();
 if (pr) {
   if(!CAS(pr->state, 0, OMV)) {
     state = 0;  pause();
     goto retry;
   }
   pr->fu = to; // tell partner
 }
 move_self(to);     <-
 pr->state = 0;
 to->state = 0;
}
```

```
void promise::mov(promise * to)
{
 to->state = MV;
 to->fu = fu;
 prep(to);     <-

 retry:
 while(!CAS(state, 0, MV))
   pause();
 if (fu) {
   if(!CAS(fu->state, 0, OMV)) {
     state = 0;  pause();
     goto retry;
   }
   fu->pr = to; // tell partner
 }
 move_self(to);     <-
 to->state = 0;
 fu->state = 0;
}
```


future
opt<R> value;
waitfor wtf;
atomic state;
promise * pr;
promise
get_future();
set_value(R);
atomic state;
future * fu;

```
atomic state;
future * fu;
```

```
void future::mov(future * to)
{
 to->state = MV;
 to->pr = pr;
 prep(to);

 retry:
 while(!CAS(state, 0, MV))
   pause();
 if (pr) {
   if(!CAS(pr->state, 0, OMV)) {
     state = 0;  pause();
     goto retry;
   }
   pr->fu = to; // tell partner
 }
 move_self(to);
 pr->state = 0;
 to->state = 0;
}
```

```
void promise::mov(promise * to)
{
 to->state = MV;
 to->fu = fu;
 prep(to);       ←

 retry:
 while(!CAS(state, 0, MV))
   pause();
 if (fu) {
   if(!CAS(fu->state, 0, OMV)) {
     state = 0;  pause();
     goto retry;
   }
   fu->pr = to; // tell partner
 }
 move_self(to);    ←
 to->state = 0;
 fu->state = 0;
}
```

```
atomic state;
future * fu;
```

```
void future::mov(future * to)
{
 to->state = MV;
 to->pr = pr;
 prep(to);

 retry:
 while(!CAS(state, 0, MV))
   pause();
 if (pr) {
   if(!CAS(pr->state, 0, OMV)) {
     state = 0;  pause();
     goto retry;
   }
   pr->fu = to; // tell partner
 }
 move_self(to);
 pr->state = 0;
 to->state = 0;
}
```

```
void promise::mov(promise * to)
{
 to->state = MV;
 to->fu = fu;

 retry:
 while(!CAS(state, 0, MV))
   pause();
 if (fu) {
   if(!CAS(fu->state, 0, OMV)) {
     state = 0;  pause();
     goto retry;
   }
   fu->pr = to; // tell partner
 }

 to->state = 0;
 fu->state = 0;
}
```

?

```
atomic state;
future * fu;
```

```
void future::mov(future * to)
{
 to->state = MV;
 to->pr = pr;
 prep(to);

 retry:
 while(!CAS(state, 0, MV))
   pause();
 if (pr) {
   if(!CAS(pr->state, 0, OMV)) {
     state = 0;  pause();
     goto retry;
   }
   pr->fu = to; // tell partner
 }
 move_self(to);
 pr->state = 0;
 to->state = 0;
}
```

```
void promise::mov(promise * to)
{
 to->state = MV;
 to->fu = fu;

 retry:
 while(!CAS(state, 0, MV))
   pause();
 if (fu) {
   if(!CAS(fu->state, 0, OMV)) {
     state = 0;  pause();
     goto retry;
   }
   fu->pr = to; // tell partner
 }

 to->state = 0;
 fu->state = 0;
}
```

```
atomic state;
future * fu;
```

```
void future::mov(future * to)
{
 to->state = MV;
 to->pr = pr;
 prep(to);

 retry:
 while(!CAS(state, 0, MV))
   pause();
 if (pr) {
   if(!CAS(pr->state, 0, OMV)) {
     state = 0;  pause();
     goto retry;
   }
   pr->fu = to; // tell partner
 }
 move_self(to);
 pr->state = 0;
 to->state = 0;
}
```

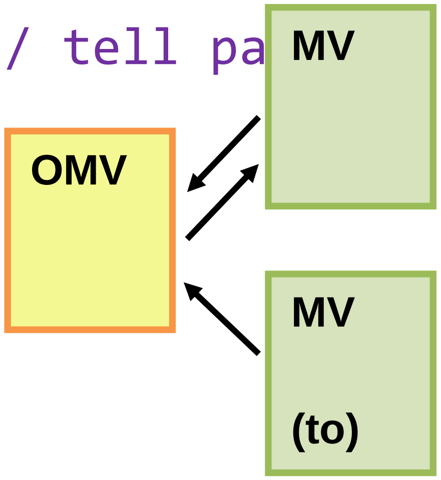
```
void promise::mov(promise * to)
{
 to->state = MV;

 retry:
 while(!CAS(state, 0, MV))
   pause();
 if (fu) {
   if(!CAS(fu->state, 0, OMV)) {
     state = 0;  pause();
     goto retry;
   }
   fu->pr = to; // tell partner
 }

 to->fu = fu;
 to->state = 0;
 fu->state = 0;
}
```

?

```
atomic state;
future * fu;
```

```
void future::mov(future * to)
{
 to->state = MV;
 to->pr = pr;
 prep(to);

 retry:
 while(!CAS(state, 0, MV))
   pause();
 if (pr) {
   if(!CAS(pr->state, 0, OMV)) {
     state = 0;  pause();
     goto retry;
   }
   pr->fu = to; // tell partner
 }
 move_self(to);
 pr->state = 0;
 to->state = 0;
}
```

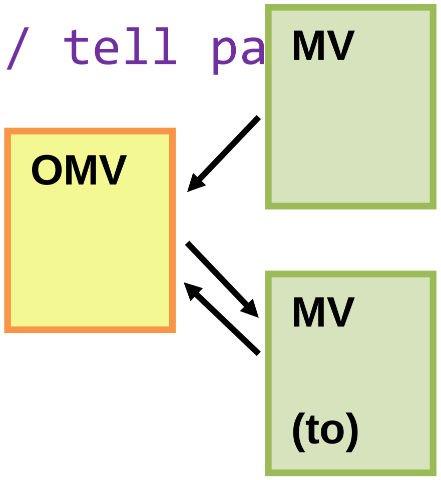
```
void promise::mov(promise * to)
{
 to->state = MV;

 retry:
 while(!CAS(state, 0, MV))
   pause();
 if (fu) {
   if(!CAS(fu->state, 0, OMV)) {
     state = 0;  pause();
     goto retry;
   }
   fu->pr = to; // tell pa
 }

 to->fu = fu;
 to->state = 0;
 fu->state = 0;
}
```



```
atomic state;
future * fu;
```

```
void future::mov(future * to)
{
 to->state = MV;
 to->pr = pr;
 prep(to);

 retry:
 while(!CAS(state, 0, MV))
   pause();
 if (pr) {
   if(!CAS(pr->state, 0, OMV)) {
     state = 0;  pause();
     goto retry;
   }
   pr->fu = to; // tell partner
 }
 move_self(to);
 pr->state = 0;
 to->state = 0;
}
```

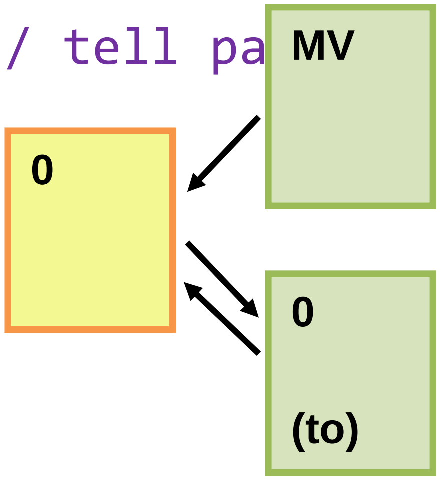
```
void promise::mov(promise * to)
{
 to->state = MV;

 retry:
 while(!CAS(state, 0, MV))
   pause();
 if (fu) {
   if(!CAS(fu->state, 0, OMV)) {
     state = 0;  pause();
     goto retry;
   }
   fu->pr = to; // tell pa
 }

 to->fu = fu;
 to->state = 0;
 fu->state = 0;
}
```



```
atomic state;
future * fu;
```

```
void future::mov(future * to)
{
 to->state = MV;
 to->pr = pr;
 prep(to);

 retry:
 while(!CAS(state, 0, MV))
   pause();
 if (pr) {
   if(!CAS(pr->state, 0, OMV)) {
     state = 0;  pause();
     goto retry;
   }
   pr->fu = to; // tell partner
 }
 move_self(to);
 pr->state = 0;
 to->state = 0;
}
```

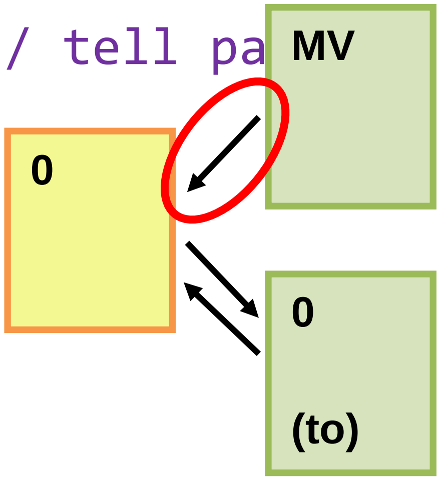
```
void promise::mov(promise * to)
{
 to->state = MV;

 retry:
 while(!CAS(state, 0, MV))
   pause();
 if (fu) {
   if(!CAS(fu->state, 0, OMV)) {
     state = 0;  pause();
     goto retry;
   }
   fu->pr = to; // tell pa
 }

 to->fu = fu;
 to->state = 0;
 fu->state = 0;
}
```



```
atomic state;
future * fu;
```

```
void future::mov(future * to)
{
 to->state = MV;
 to->pr = pr;
 prep(to);

 retry:
 while(!CAS(state, 0, MV))
   pause();
 if (pr) {
   if(!CAS(pr->state, 0, OMV)) {
     state = 0;  pause();
     goto retry;
   }
   pr->fu = to; // tell partner
 }
 move_self(to);
 pr->state = 0;
 to->state = 0;
}
```

```
void promise::mov(promise * to)
{
 to->state = MV;

 retry:
 while(!CAS(state, 0, MV))
   pause();
 if (fu) {
   if(!CAS(fu->state, 0, OMV)) {
     state = 0;  pause();
     goto retry;
   }
   fu->pr = to; // tell pa
 }

 to->fu = fu;
 to->state = 0;
 fu->state = 0;
}
```



```
atomic state;
future * fu;
```

```
void future::mov(future * to)
{
 to->state = MV;
 to->pr = pr;
 prep(to);

 retry:
 while(!CAS(state, 0, MV))
   pause();
 if (pr) {
   if(!CAS(pr->state, 0, OMV)) {
     state = 0;  pause();
     goto retry;
   }
   pr->fu = to; // tell partner
 }
 move_self(to);
 pr->state = 0;
 to->state = 0;
}
```

```
void promise::mov(promise * to)
{
 to->state = MV;

 retry:
 while(!CAS(state, 0, MV))
   pause();
 if (fu) {
   if(!CAS(fu->state, 0, OMV)) {
     state = 0;  pause();
     goto retry;
   }
   fu->pr = to; // tell partner
 }

 to->fu = fu;
 to->state = 0;
 fu->state = 0;
}
```

```
fu = 0;
```

```
atomic state;
future * fu;
```

```
void future::mov(future * to)
{
 to->state = MV;
 to->pr = pr;
 prep(to);

 retry:
 while(!CAS(state, 0, MV))
   pause();
 if (pr) {
   if(!CAS(pr->state, 0, OMV)) {
     state = 0;  pause();
     goto retry;
   }
   pr->fu = to; // tell partner
 }
 move_self(to);
 pr->state = 0;
 to->state = 0;
}
```

```
void promise::mov(promise * to)
{
 to->state = MV;

 retry:
 while(!CAS(state, 0, MV))
   pause();
 if (fu) {
   if(!CAS(fu->state, 0, OMV)) {
     state = 0;  pause();
     goto retry;
   }
   fu->pr = to; // tell partner
 }

 to->fu = fu;
 to->state = 0;
 fu->state = 0;
 fu = 0;
}
```

```
atomic state;
future * fu;
```

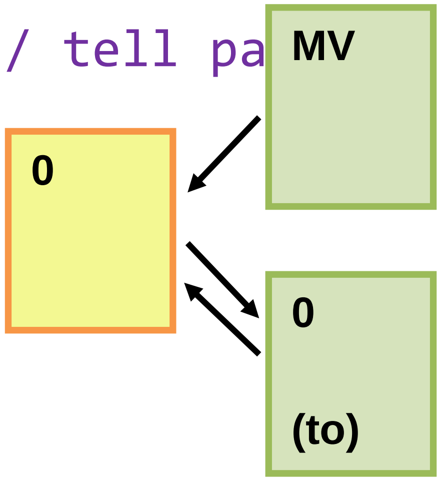
```
void future::mov(future * to)
{
 to->state = MV;
 to->pr = pr;
 prep(to);

 retry:
 while(!CAS(state, 0, MV))
   pause();
 if (pr) {
   if(!CAS(pr->state, 0, OMV)) {
     state = 0;  pause();
     goto retry;
   }
   pr->fu = to; // tell partner
 }
 move_self(to);
 pr->state = 0;
 to->state = 0;
}
```

```
void promise::mov(promise * to)
{
 to->state = MV;

 retry:
 while(!CAS(state, 0, MV))
   pause();
 if (fu) {
   if(!CAS(fu->state, 0, OMV)) {
     state = 0;  pause();
     goto retry;
   }
   fu->pr = to; // tell partner
 }

 to->fu = fu;
 to->state = 0;
 fu->state = 0;
 fu = 0;
}
```

```
atomic state;
future * fu;
```

```
void future::mov(future * to)
{
 to->state = MV;
 to->pr = pr;
 prep(to);

 retry:
 while(!CAS(state, 0, MV))
   pause();
 if (pr) {
   if(!CAS(pr->state, 0, OMV)) {
     state = 0;  pause();
     goto retry;
   }
   pr->fu = to; // tell partner
 }
 move_self(to);
 pr->state = 0;
 to->state = 0;
}
```

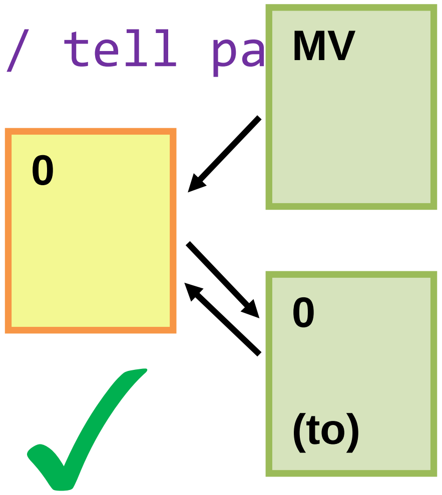
```
void promise::mov(promise * to)
{
 to->state = MV;

 retry:
 while(!CAS(state, 0, MV))
   pause();
 if (fu) {
   if(!CAS(fu->state, 0, OMV)) {
     state = 0;  pause();
     goto retry;
   }
   fu->pr = to; // tell pa
 }

 to->fu = fu;
 to->state = 0;
 fu->state = 0;
 fu = 0;
}
```



```
atomic state;
future * fu;
```

```
void future::mov(future * to)
{
 to->state = MV;
 to->pr = pr;
 prep(to);

 retry:
 while(!CAS(state, 0, MV))
   pause();
 if (pr) {
   if(!CAS(pr->state, 0, OMV)) {
     state = 0;  pause();
     goto retry;
   }
   pr->fu = to; // tell partner
 }
 move_self(to);
 pr->state = 0;
 to->state = 0;
}
```

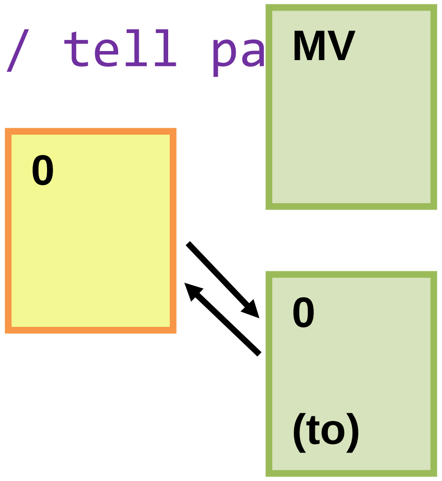
```
void promise::mov(promise * to)
{
 to->state = MV;

 retry:
 while(!CAS(state, 0, MV))
   pause();
 if (fu) {
   if(!CAS(fu->state, 0, OMV)) {
     state = 0;  pause();
     goto retry;
   }
   fu->pr = to; // tell pa
 }

 to->fu = fu;
 to->state = 0;
 fu->state = 0;
 fu = 0;
}
```



```
atomic state;
future * fu;
```

```
void future::mov(future * to)
{
 to->state = MV;
 to->pr = pr;
 prep(to);

 retry:
 while(!CAS(state, 0, MV))
   pause();
 if (pr) {
   if(!CAS(pr->state, 0, OMV)) {
     state = 0;  pause();
     goto retry;
   }
   pr->fu = to; // tell partner
 }
 move_self(to);
 pr->state = 0;
 to->state = 0;
}
```

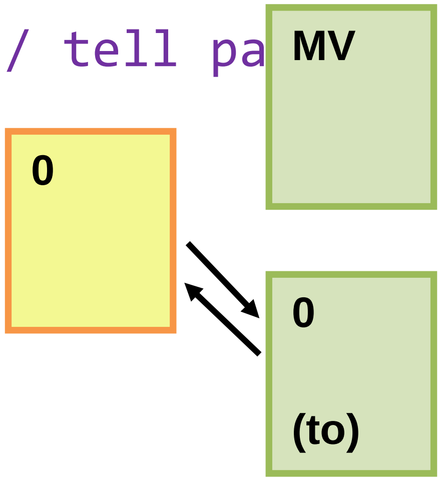
```
void promise::mov(promise * to)
{
 to->state = MV;

 retry:
 while(!CAS(state, 0, MV))
   pause();
 if (fu) {
   if(!CAS(fu->state, 0, OMV)) {
     state = 0;  pause();
     goto retry;
   }
   fu->pr = to; // tell pa
 }

 to->fu = fu;
 to->state = 0;
 fu->state = 0;
 fu = 0;
}
```



```
atomic state;
future * fu;
```

```
void future::mov(future * to)
{
 to->state = MV;
 to->pr = pr;
 prep(to);

 retry:
 while(!CAS(state, 0, MV))
   pause();
 if (pr) {
   if(!CAS(pr->state, 0, OMV)) {
     state = 0;  pause();
     goto retry;
   }
   pr->fu = to; // tell partner
 }
 move_self(to);
 pr->state = 0;
 to->state = 0;
}
```

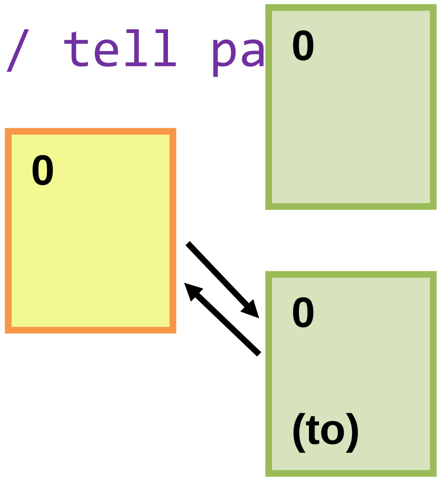
```
void promise::mov(promise * to)
{
 to->state = MV;

 retry:
 while(!CAS(state, 0, MV))
   pause();
 if (fu) {
   if(!CAS(fu->state, 0, OMV)) {
     state = 0;  pause();
     goto retry;
   }
   fu->pr = to; // tell pa
 }

 to->fu = fu;
 to->state = 0;
 fu->state = 0;
 fu = 0;
 state = 0;
}
```



```
atomic state;
future * fu;
```

```
void future::mov(future * to)
{
 to->state = MV;
 to->pr = pr;
 prep(to);

 retry:
 while(!CAS(state, 0, MV))
   pause();
 if (pr) {
   if(!CAS(pr->state, 0, OMV)) {
     state = 0;  pause();
     goto retry;
   }
   pr->fu = to; // tell partner
 }
 move_self(to);
 pr->state = 0;
 to->state = 0;
}
```

```
void promise::mov(promise * to)
{
 to->state = MV;

 retry:
 while(!CAS(state, 0, MV))
   pause();
 if (fu) {
   if(!CAS(fu->state, 0, OMV)) {
     state = 0;  pause();
     goto retry;
   }
   fu->pr = to; // tell pa
 }

 to->fu = fu;
 to->state = 0;
 fu->state = 0;
 fu = 0;
 state = 0;
}
```



```
atomic state;
future * fu;
```

```
void future::mov(future * to)
{
 to->state = MV;
 to->pr = pr;
 prep(to);

 retry:
 while(!CAS(state, 0, MV))
   pause();
 if (pr) {
   if(!CAS(pr->state, 0, OMV)) {
     state = 0;  pause();
     goto retry;
   }
   pr->fu = to; // tell partner
 }
 move_self(to);
 pr->state = 0;
 to->state = 0;
}
```

```
void promise::mov(promise * to)
{
 if (to) to->state = MV;

 retry:
 while(!CAS(state, 0, MV))
   pause();
 if (fu) {
   if(!CAS(fu->state, 0, OMV)) {
     state = 0;  pause();
     goto retry;
   }
   fu->pr = to; // tell partner
 }

 if (to) { to->fu = fu;
           to->state = 0; }
 fu->state = 0;
 fu = 0;
 state = 0;
}
```

```
atomic state;
future * fu;
```

```
void future::mov(future * to)
{
 to->state = MV;
 to->pr = pr;
 prep(to);

 retry:
 while(!CAS(state, 0, MV))
   pause();
 if (pr) {
   if(!CAS(pr->state, 0, OMV)) {
     state = 0;  pause();
     goto retry;
   }
   pr->fu = to; // tell partner
 }
 move_self(to);
 pr->state = 0;
 to->state = 0;
}
```

```
void promise::mov(promise * to)
{
 if (to) to->state = MV;

 retry:
 while(!CAS(state, 0, MV))
   pause();
 if (fu) {
   if(!CAS(fu->state, 0, OMV)) {
     state = 0;  pause();
     goto retry;
   }
   fu->pr = to; // tell partner
   fu->state = 0;
 }

 if (to) { to->fu = fu;
           to->state = 0; }
 fu = 0;
 state = 0;
}
```

```
optional<R> value;     atomic state;
waitfor wtf;           promise * pr;
```

```
void future::mov(future * to)
{
 to->state = MV;
 to->pr = pr;
 prep(to);

 retry:
 while(!CAS(state, 0, MV))
   pause();
 if (pr) {
   if(!CAS(pr->state, 0, OMV)) {
     state = 0;  pause();
     goto retry;
   }
   pr->fu = to; // tell partner
 }
 move_self(to);
 pr->state = 0;
 to->state = 0;
}
```

```
void promise::mov(promise * to)
{
 if (to) to->state = MV;

 retry:
 while(!CAS(state, 0, MV))
   pause();
 if (fu) {
   if(!CAS(fu->state, 0, OMV)) {
     state = 0;  pause();
     goto retry;
   }
   fu->pr = to; // tell partner
   fu->state = 0;
 }

 if (to) { to->fu = fu;
           to->state = 0; }
 fu = 0;
 state = 0;
}
```

```
optional<R> value;    atomic state;
waitfor wtf;          promise * pr;
```

```
void future::mov(future * to)
{
 if (to) to->state = MV;
 prep(to);
 retry:
 while(!CAS(state, 0, MV))
   pause();
 if (pr) {
   if(!CAS(pr->state, 0, OMV)) {
     state = 0;  pause();
     goto retry;
   }
   pr->fu = to; // tell partner
   pr->state = 0;
 }
 move_self(to);
 if (to) { to->pr = pr;
           to->state = 0; }
 pr = 0;
 state = 0;
}
```

```
void promise::mov(promise * to)
{
 if (to) to->state = MV;

 retry:
 while(!CAS(state, 0, MV))
   pause();
 if (fu) {
   if(!CAS(fu->state, 0, OMV)) {
     state = 0;  pause();
     goto retry;
   }
   fu->pr = to; // tell partner
   fu->state = 0;
 }

 if (to) { to->fu = fu;
           to->state = 0; }
 fu = 0;
 state = 0;
}
```

```
optional<R> value;?  atomic state;
waitfor wtf;         promise * pr; ✓
```

```
void future::mov(future * to)
{
 if (to) to->state = MV;
 prep(to);     ←
 retry:
 while(!CAS(state, 0, MV))
   pause();
 if (pr) {
   if(!CAS(pr->state, 0, OMV)) {
     state = 0;  pause();
     goto retry;
   }
   pr->fu = to; // tell partner
   pr->state = 0;
 }
 move_self(to);    ←
 if (to) { to->pr = pr;
           to->state = 0; }
 pr = 0;
 state = 0;
}
```

```
void promise::mov(promise * to)
{
 if (to) to->state = MV;

 retry:
 while(!CAS(state, 0, MV))
   pause();
 if (fu) {
   if(!CAS(fu->state, 0, OMV)) {
     state = 0;  pause();
     goto retry;
   }
   fu->pr = to; // tell partner
   fu->state = 0;
 }

 if (to) { to->fu = fu;
           to->state = 0; }
 fu = 0;
 state = 0;
}
```

```
optional<R> value; ? atomic state;
waitfor wtf;         promise * pr;
```

```
void future::mov(future * to)
{
 if (to) to->state = MV;
 prep(to);      ←  ✗
 retry:
 while(!CAS(state, 0, MV))
   pause();
 if (pr) {
   if(!CAS(pr->state, 0, OMV)) {
     state = 0;  pause();
     goto retry;
   }
   pr->fu = to; // tell partner
   pr->state = 0;
 }
 move_self(to);  ←
 if (to) { to->pr = pr;
           to->state = 0; }
 pr = 0;
 state = 0;
}
```

```
void promise::mov(promise * to)
{
 if (to) to->state = MV;

 retry:
 while(!CAS(state, 0, MV))
   pause();
 if (fu) {
   if(!CAS(fu->state, 0, OMV)) {
     state = 0;  pause();
     goto retry;
   }
   fu->pr = to; // tell partner
   fu->state = 0;
 }

 if (to) { to->fu = fu;
           to->state = 0; }
 fu = 0;
 state = 0;
}
```

```
optional<R> value;     atomic state;
waitfor wtf;           promise * pr;
```

```
void future::mov(future * to)
{
 if (to) to->state = MV;

 retry:
 while(!CAS(state, 0, MV))
   pause();
 if (pr) {
   if(!CAS(pr->state, 0, OMV)) {
     state = 0;  pause();
     goto retry;
   }
   pr->fu = to; // tell partner
   pr->state = 0;
 }
 if (to) {
   to->value = move(value);
   to->wtf.ready(wtf.ready());
   to->pr = pr;
   to->state = 0;
 }
 value = nullopt; wtf.ready(0);
 pr = 0;
 state = 0;
}
```

```
void promise::mov(promise * to)
{
 if (to) to->state = MV;

 retry:
 while(!CAS(state, 0, MV))
   pause();
 if (fu) {
   if(!CAS(fu->state, 0, OMV)) {
     state = 0;  pause();
     goto retry;
   }
   fu->pr = to; // tell partner
   fu->state = 0;
 }

 if (to) { to->fu = fu;
           to->state = 0; }
 fu = 0;
 state = 0;
}
```

```
void future::mov(future * to)
{
 if (to) to->state = MV;

 retry:
 while(!CAS(state, 0, MV))
   pause();
 if (pr) {
   if(!CAS(pr->state, 0, OMV)) {
     state = 0;  pause();
     goto retry;
   }
   pr->fu = to; // tell partner
   pr->state = 0;
 }
 if (to) {
   to->value = move(value);
   to->wtf.ready(wtf.ready());
   to->pr = pr;
   to->state = 0;
 }
 value = nullopt; wtf.ready(0);
 pr = 0;
 state = 0;
}
```

?

```
optional<R> value;    atomic state;
waitfor wtf;          promise * pr;
```

```
void promise::mov(promise * to)
{
 if (to) to->state = MV;

 retry:
 while(!CAS(state, 0, MV))
   pause();
 if (fu) {
   if(!CAS(fu->state, 0, OMV)) {
     state = 0;  pause();
     goto retry;
   }
   fu->pr = to; // tell partner
   fu->state = 0;
 }

 if (to) { to->fu = fu;
           to->state = 0; }
 fu = 0;
 state = 0;
}
```

```
optional<R> value;     atomic state;
waitfor wtf;           promise * pr;
```

```
void future::mov(future * to)
{
 if (to) to->state = MV;

 retry:
 while(!CAS(state, 0, MV))
   pause();
 if (pr) {
   if(!CAS(pr->state, 0, OMV)) {
     state = 0;  pause();
     goto retry;
   }
   pr->fu = to; // tell partner
   pr->state = 0;
}
 if (to) {
   to->value = move(value);
   to->wtf.ready(wtf.ready());
   to->pr = pr;
   to->state = 0;
 }
 value = nullopt; wtf.ready(0);
 pr = 0;
 state = 0;
}
```

R(R&&)
(maybe)

?

```
void promise::mov(promise * to)
{
 if (to) to->state = MV;

 retry:
 while(!CAS(state, 0, MV))
   pause();
 if (fu) {
   if(!CAS(fu->state, 0, OMV)) {
     state = 0;  pause();
     goto retry;
   }
   fu->pr = to; // tell partner
   fu->state = 0;
 }

 if (to) { to->fu = fu;
           to->state = 0; }
 fu = 0;
 state = 0;
}
```

```
optional<R> value;     atomic state;
waitfor wtf;           promise * pr;
```

```
void future::mov(future * to)
{
 if (to) to->state = MV;

 retry:
 while(!CAS(state, 0, MV))
   pause();
 if (pr) {
   if(!CAS(pr->state, 0, OMV)) {
     state = 0;  pause();
     goto retry;
   }
   pr->fu = to; // tell partner
   pr->state = 0;
 }
 if (to) {
   to->value = move(value);
   to->wtf.ready(wtf.ready());
   to->pr = pr;
   to->state = 0;
 }
 value = nullopt; wtf.ready(0);
 pr = 0;
 state = 0;
}
```

R(R&&)
(maybe)

?

```
void promise::mov(promise * to)
{
 if (to) to->state = MV;

 retry:
 while(!CAS(state, 0, MV))
   pause();
 if (fu) {
   if(!CAS(fu->state, 0, OMV)) {
     state = 0;  pause();
     goto retry;
   }
   fu->pr = to; // tell partner
   fu->state = 0;
 }

 if (to) { to->fu = fu;
           to->state = 0; }
 fu = 0;
 state = 0;
}
```

```
void future::mov(future * to)
{
 if (!pr) {  easymov<0>(to);
             return;  }

 if (to) to->state = MV;

 retry:
 while(!CAS(state, 0, MV))
   pause();
 if (pr) {
   if(!CAS(pr->state, 0, OMV)) {
     state = 0;  pause();
     goto retry;
   }
   pr->fu = to; // tell partner
   pr->state = 0;
 }

 easymov<plusStateAndPr>(to);
 pr = 0;
 state = 0;
}
```

```
optional<R> value;    atomic state;
waitfor wtf;          promise * pr;
```

```
enum move_what
{
  justValue = 0,
  plusState = 1,
  plusStateAndPr = 2
};

template<int i>
void future::easymov(future *to)
{
 if (to) {
   to->value = move(value);
   to->wtf = wtf; //(flag part)
   if (i > 1) to->pr = pr;
   if (i > 0) to->state = 0;
 }
 value = nullopt;
 wtf.ready(false);
}
```

```
optional<R> value;    atomic state;
waitfor wtf;          promise * pr;
```

```
void future::mov(future * to)
{
 if (!pr) {  easymov<0>(to);
             return;  }

 if (to) to->state = MV;

 retry:  ←
 while(!CAS(state, 0, MV))
   pause();
 if (pr) {
   if(!CAS(pr->state, 0, OMV)) {
     state = 0;  pause();
     goto retry;
   }
   pr->fu = to; // tell partner
   pr->state = 0;
 }

 easymov<plusStateAndPr>(to);
 pr = 0;
 state = 0;
}
```

```
void promise::mov(promise * to)
{
 if (to) to->state = MV;

 retry:
 while(!CAS(state, 0, MV))
   pause();
 if (fu) {
   if(!CAS(fu->state, 0, OMV)) {
     state = 0;  pause();
     goto retry;
   }
   fu->pr = to; // tell partner
   fu->state = 0;
 }

 if (to) { to->fu = fu;
           to->state = 0; }
 fu = 0;
 state = 0;
}
```

```
optional<R> value;    atomic state;
waitfor wtf;          promise * pr;
```

```
void future::mov(future * to)
{
 if (!pr) {  easymov<0>(to);
             return;  }

 if (to) to->state = MV;

 retry:
 while(!CAS(state, 0, MV))
   pause();
 if (pr) {
   if(!CAS(pr->state, 0, OMV)) {
     state = 0;
     if (!pr) {  easymov<1>(to);
                 return;  }
     pause(); goto retry;
   }
   pr->fu = to; // tell partner
   pr->state = 0;
 }
 easymov<plusStateAndPr>(to);
 pr = 0;
 state = 0;
}
```

```
void promise::mov(promise * to)
{
 if (to) to->state = MV;

 retry:
 while(!CAS(state, 0, MV))
   pause();
 if (fu) {
   if(!CAS(fu->state, 0, OMV)) {
     state = 0;  pause();
     goto retry;
   }
   fu->pr = to; // tell partner
   fu->state = 0;
 }

 if (to) { to->fu = fu;
           to->state = 0; }
 fu = 0;
 state = 0;
}
```

```
optional<R> value;     atomic state;
waitfor wtf;           promise * pr;
```

```
void future::mov(future * to)
{
 if (!pr) {  easymov<0>(to);
             return;  }

 if (to) to->state = MV;        ?

 retry:
 while(!CAS(state, 0, MV))
   pause();
 if (pr) {
   if(!CAS(pr->state, 0, OMV)) {
     state = 0;
     if (!pr) {  easymov<1>(to);
                 return;  }
     pause(); goto retry;
   }
   pr->fu = to; // tell partner
   pr->state = 0;
 }
 easymov<plusStateAndPr>(to);
 pr = 0;
 state = 0;
}
```

```
void promise::mov(promise * to)
{
 if (to) to->state = MV;

 retry:
 while(!CAS(state, 0, MV))
   pause();
 if (fu) {
   if(!CAS(fu->state, 0, OMV)) {
     state = 0;  pause();
     goto retry;
   }
   fu->pr = to; // tell partner
   fu->state = 0;
 }

 if (to) { to->fu = fu;
           to->state = 0; }
 fu = 0;
 state = 0;
}
```

```
void future::mov(future * to)
{
 if (!pr) {  easymov<0>(to);
             return;  }

 if (to) to->state = MV;

 retry:
 while(!CAS(state, 0, MV))
   pause();
 if (pr) {
   if(!CAS(pr->state, 0, OMV)) {
     state = 0;
     if (!pr) {  easymov<1>(to);
                 return;  }
     pause(); goto retry;
   }
   pr->fu = to; // tell partner
   pr->state = 0;
 }
 easymov<plusStateAndPr>(to);
 pr = 0;
 state = 0;
}
```

```
optional<R> value;    atomic state;
waitfor wtf;          promise * pr;
```

```
void promise::mov(promise * to)
{
 if (to) to->state = MV;

 retry:
 while(!CAS(state, 0, MV))
   pause();
 if (fu) {
   if(!CAS(fu->state, 0, OMV)) {
     state = 0;  pause();
     goto retry;
   }
   fu->pr = to; // tell partner
   fu->state = 0;
 }

 if (to) { to->fu = fu;
           to->state = 0; }
 fu = 0;
 state = 0;
}
```

```
void future::mov(future * to)
{
 if (!pr) {  easymov<0>(to);
             return;  }

 if (to) to->state = MV;

 retry:
 while(!CAS(state, 0, MV))
   pause();
 if (pr) {
   if(!CAS(pr->state, 0, OMV)) {
     state = 0;
     if (!pr) {  easymov<1>(to);
                 return;  }
     pause(); goto retry;
   }
   pr->fu = to; // tell partner
   pr->state = 0;
 }
 easymov<plusStateAndPr>(to);
 pr = 0;
 state = 0;
}
```

```
void promise::set_value(R value)
{
  fu->value = value;
  fu->wtf.ready(true); // yay!!
}
```

```
void future::mov(future * to)
{
 if (!pr) {  easymov<0>(to);
             return;  }

 if (to) to->state = MV;

 retry:
 while(!CAS(state, 0, MV))
   pause();
 if (pr) {
   if(!CAS(pr->state, 0, OMV)) {
     state = 0;
     if (!pr) {  easymov<1>(to);
                 return;  }
     pause(); goto retry;
   }
   pr->fu = to; // tell partner
   pr->state = 0;
 }
 easymov<plusStateAndPr>(to);
 pr = 0;
 state = 0;
}
```

```
void promise::set_value(R value)
{
 retry:
 while(!CAS(state, 0, ST))
   pause();
 if (fu) {
   if(!CAS(fu->state, 0, OST)) {
     state = 0;  pause();
     goto retry;
   }
   fu->value = value;
   fu->wtf.ready(true); // yay!!
   fu->pr = 0;  // bye bye
   fu->state = 0;
   fu = 0;
 }
                            ?
 state = 0;
}
```

```
void future::mov(future * to)
{
 if (!pr) {  easymov<0>(to);
             return;  }

 if (to) to->state = MV;

 retry:
 while(!CAS(state, 0, MV))
   pause();
 if (pr) {
   if(!CAS(pr->state, 0, OMV)) {
     state = 0;
     if (!pr) {  easymov<1>(to);
                 return;  }
     pause(); goto retry;
   }
   pr->fu = to; // tell partner
   pr->state = 0;
 }
 easymov<plusStateAndPr>(to);
 pr = 0;
 state = 0;
}
```

```
void promise::set_value(R value)
{
 retry:
 while(!CAS(state, 0, ST))
   pause();
 if (fu) {
   if(!CAS(fu->state, 0, OST)) {
     state = 0;  pause();
     goto retry;
   }
   fu->value = value;
   fu->wtf.ready(true); // yay!!
   fu->pr = 0;  // bye bye
   fu->state = 0;
   fu = 0;
 }

 state = 0;
}
```

R(R&&)

```
void future::mov(future * to)
{
 if (!pr) {  easymov<0>(to);
             return;  }

 if (to) to->state = MV;

 retry:
 while(!CAS(state, 0, MV))
   pause();
 if (pr) {
   if(!CAS(pr->state, 0, OMV)) {
     state = 0;
     if (!pr) {  easymov<1>(to);
                 return;  }
     pause(); goto retry;
   }
   pr->fu = to; // tell partner
   pr->state = 0;
 }
 easymov<plusStateAndPr>(to);
 pr = 0;
 state = 0;
}
```

```
void promise::set_value(R value)
{
 retry:
 while(!CAS(state, 0, ST))
   pause();
 if (fu) {
   if(!CAS(fu->state, 0, OST)) {
     state = 0;  pause();
     goto retry;
   }
   fu->value = value;
   fu->wtf.ready(true); // yay!!
   fu->pr = 0;  // bye bye
   fu->state = 0;
   fu = 0;
 }

 state = 0;
}
```

R(R&&)

```
void future::mov(future * to)
{
 if (!pr) {  easymov<0>(to);
             return;  }
 if (to) to->state = MV;
 retry:
 State tmp = 0;
 while(!CAS(state, tmp, MV)) {
   if (tmp == OST) {
     wtf.wait();
     easymov<plusState>(to);
     return;
   }
   pause();
   tmp = 0;
 }
 if (pr) {
   if(!CAS(pr->state, 0, OMV)) {
     state = 0;
     if (!pr) {  easymov<1>(to);
                 return;  }
     pause(); goto retry;
   }
   pr->fu = to; // tell partner
```

```
void promise::set_value(R value)
{
 retry:
 while(!CAS(state, 0, ST))
   pause();
 if (fu) {
   if(!CAS(fu->state, 0, OST)) {
     state = 0;  pause();
     goto retry;
   }
   fu->value = value;
   fu->wtf.ready(true); // yay!!
   fu->pr = 0;   // bye bye
   fu->state = 0;
   fu = 0;
 }

 state = 0;
}
```

R(R&&)

```
void future::mov(future * to)
{
 if (!pr) {  easymov<0>(to);
             return;  }
 if (to) to->state = MV;
 retry:
 State tmp = MV;
 while(!CAS(state, 0, tmp)) {
   if (tmp == OST) {
     wtf.wait();
     easymov<plusState>(to);
     return;
   }
   pause();
   tmp = MV;
 }
 if (pr) {
   if(!CAS(pr->state, 0, OMV)) {
     state = 0;
     if (!pr) {  easymov<1>(to);
                 return;  }
     pause(); goto retry;
   }
   pr->fu = to; // tell partner
```

```
void promise::set_value(R value)
{
 retry:
 while(!CAS(state, 0, ST))
   pause();
 if (fu) {
   if(!CAS(fu->state, 0, OST)) {
     state = 0;  pause();
     goto retry;
   }
   fu->value = value;
   fu->wtf.ready(true); // yay!!
   fu->pr = 0;  // bye bye
   fu->state = 0;
   fu = 0;
 }

 state = 0;
}
```

```
void future::mov(future * to)
{
 if (!pr) {  easymov<0>(to);
             return;  }
 if (to) to->state = MV;
 retry:
 State tmp = MV;
 while(!CAS(state, 0, tmp)) {
   if (tmp == OST) {
     wtf.wait();
     easymov<plusState>(to);
     return;
   }
   pause();
   tmp = MV;
 }
 if (pr) {
   if(!CAS(pr->state, 0, OMV)) {
     state = 0;
     if (!pr) {  easymov<1>(to);
                 return;  }
     pause(); goto retry;
   }
   pr->fu = to; // tell partner
   pr->state = 0;
```

```
   pr->state = 0;
 }
 easymov<plusStateAndPr>(to);
 pr = 0;
 state = 0;
}
```

```
void promise::set_value(R value)
{
 retry:
 while(!CAS(state, 0, ST))
   pause();
 if (fu) {
   if(!CAS(fu->state, 0, OST)) {
     state = 0;  pause();
     goto retry;
   }
   fu->value = value;
   fu->wtf.ready(true); // yay!!
   fu->pr = 0;  // bye bye
   fu->state = 0;
   fu = 0;
 }
 state = 0;
}
```

Homework:

- promise::set_exception()
- future::valid(), wait()
- swap()
- waitfor wtf;
- pause()
- exception safety? // R(R&&) may throw
- memory_order_* ?
- shared_future<R> ?
- measure!!!
- optimize
- test
- prove correctness :-)

Technically...

- future(future &&) noexcept;
  Effects: move constructs a future object that refers to the shared state that was originally referred to by rhs (if any).

- future& operator=(future&&) noexcept;
  Effects:
  — releases any shared state (30.6.4).
  — move assigns the contents of rhs to *this.

Non-Allocating Future/Promise

Tony Van Eerd, BlackBerry, Inc.

C++Now, Aspen Colorado

May 13, 2013