The Art of Writing Reasonable Concurrent Code

Pre-Conference Workshop ACCU 2017

Felix Petriconi

2017-04-25

The [C++] language is too large for *anyone* to master

The [C++] language is too large for *anyone* to master
So *everyone* lives within a subset

The [C++] language is too large for *anyone* to master
So *everyone* lives within a subset

Sean Parent, C++Now, 2012

Felix Petriconi

- School (UCSD Pascal, Turbo Pascal)
- Studied electrical engineering (Modula 2, Ada, C++)
- Student research assistant (1992-1996) (Turbo Pascal, C++, C)
- Freelance programmer 1996-2003 (Ericsson, Siemens-VDO, etc.)
  - Development of test software for embedded devices (Perl, C)
- Programmer and development manager 2003-today at MeVis Medical Solutions AG, Bremen, Germany
  - Development of medical devices in the area of mammography and radio therapy (C++, Ruby, Python)
- Programming activities:
  - Blog editor of ISO C++ website
  - Active member of C++ User Group Bremen
  - Contributor to Sean Parent's concurrency library
  - Member of ACCU conference committee
- Married with Nicole, having three children, living near Bremen, Germany
- Other interests: Classic film scores, composition

Why are we here?

Why am I here?

Why am I here?

I saw how we developed multi threaded code in the past.
I saw how easy it is to make mistakes.
I saw and still see how difficult it is to maintain this code.

Why am I here?

I saw how we developed multi threaded code in the past.
I saw how easy it is to make mistakes.
I saw and still see how difficult it is to maintain this code.

I watched recordings from Sean Parent's talks about "Better Code".
I was impressed.
I wanted to learn more.

Why am I here?

I saw how we developed multi threaded code in the past.
I saw how easy it is to make mistakes.
I saw and still see how difficult it is to maintain this code.

I watched recordings from Sean Parent's talks about "Better Code".
I was impressed.
I wanted to learn more.

I'm collaborating in his open source project for a new library.
I'm continuously learning there a lot.
I care about sharing my knowledge, here at the ACCU conference.

Why are you here?

What is your motivation to be here?

Problems from my domain

- Loading of huge images blocks UI
- Storing of files blocks UI
- Re-coding of huge images takes very long
- DB accesses takes too long
- ...

Why do we have to talk about concurrency?

The free lunch is over!

The free lunch is over!

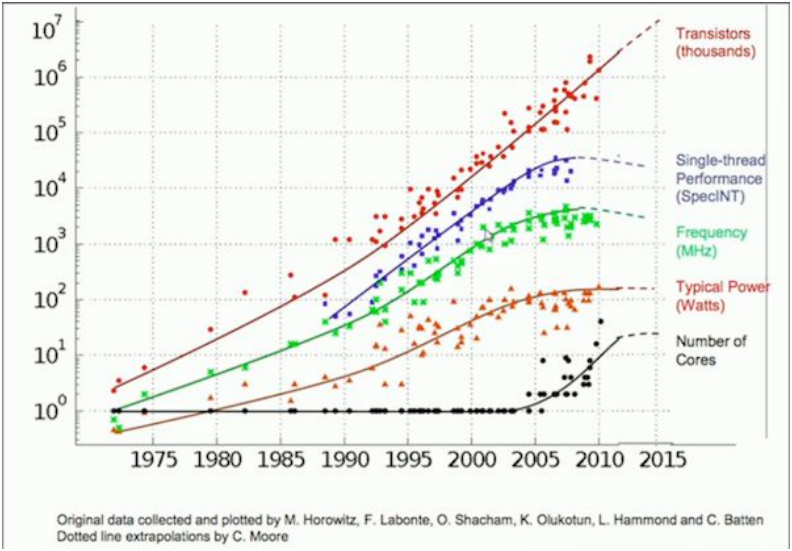
Herb Sutter, 2005[1]

The free lunch is over!

Herb Sutter, 2005[1]

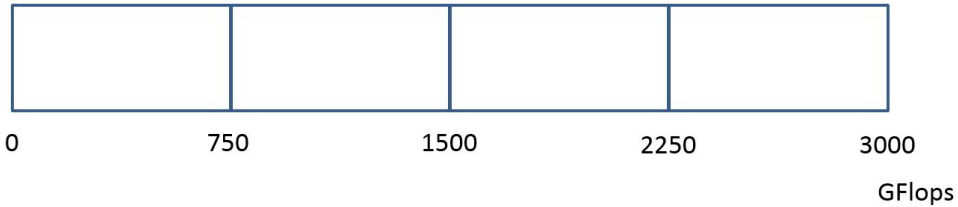
[1] The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software
http://www.gotw.ca/publications/concurrency-ddj.htm

The free lunch is over



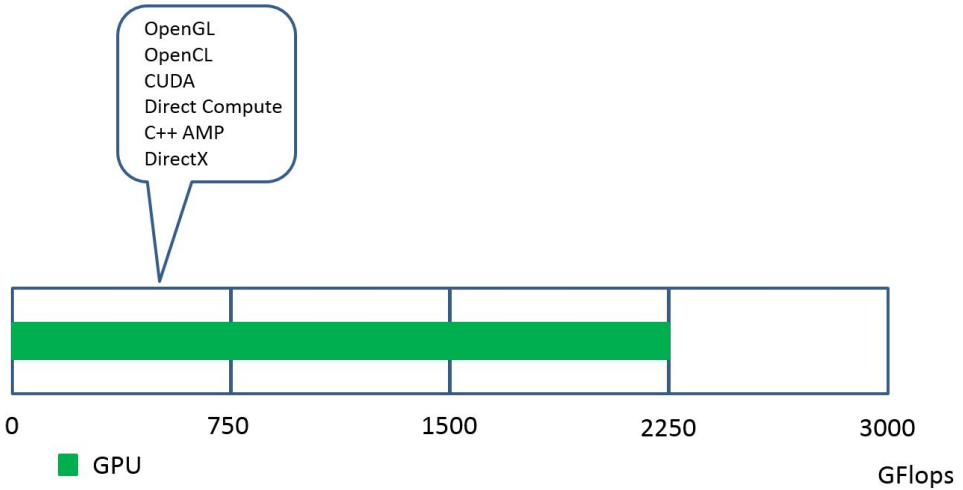
Desktop Compute Power

8-core 3.5GHz (Sandy Bridge + AMD Radeon 6950)



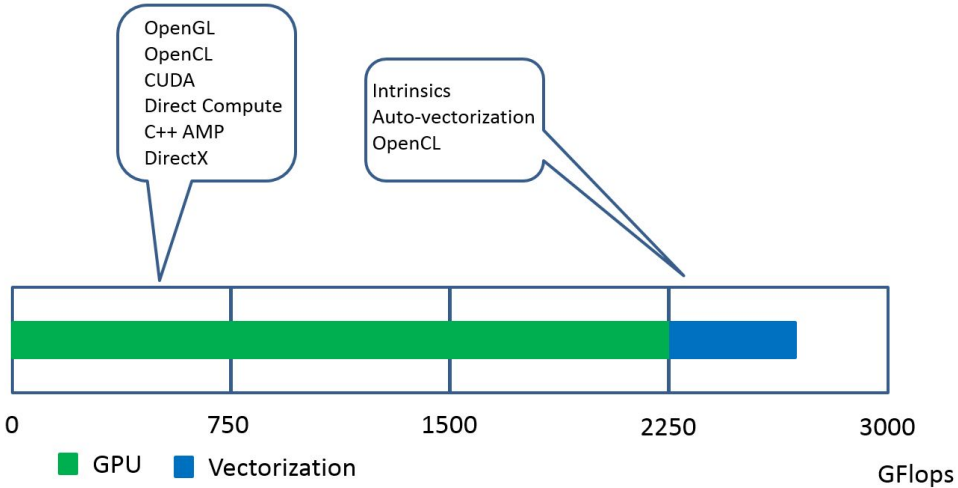
Desktop Compute Power

8-core 3.5GHz (Sandy Bridge + AMD Radeon 6950)



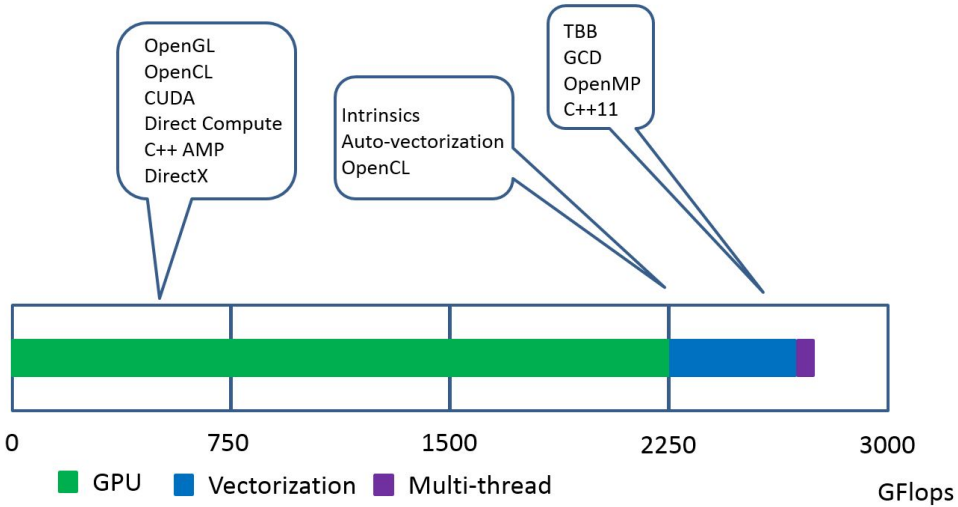
Desktop Compute Power

8-core 3.5GHz (Sandy Bridge + AMD Radeon 6950)



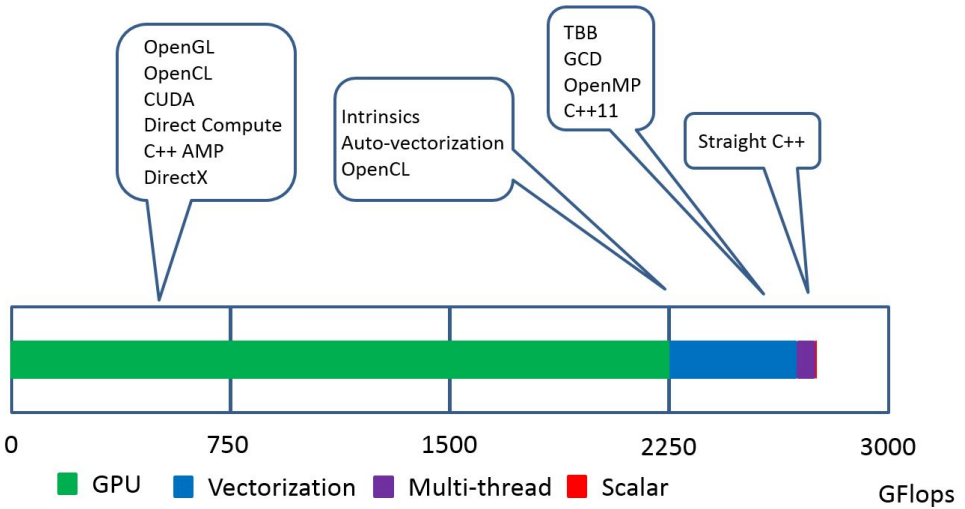
Desktop Compute Power

8-core 3.5GHz (Sandy Bridge + AMD Radeon 6950)



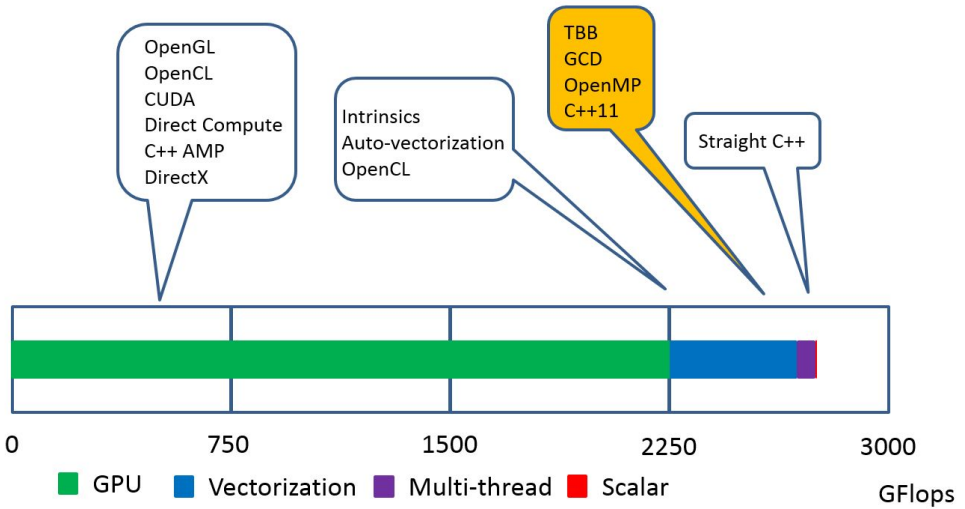
Desktop Compute Power

8-core 3.5GHz (Sandy Bridge + AMD Radeon 6950)



Desktop Compute Power

8-core 3.5GHz (Sandy Bridge + AMD Radeon 6950)



Amdahl's Law

$$S(N) = \frac{1}{(1-P)+\frac{P}{N}}$$

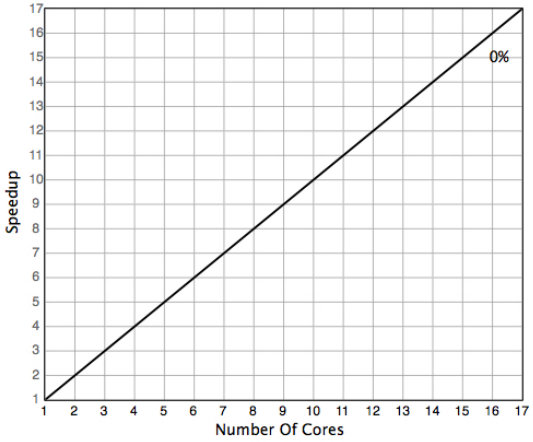
P : Synchronization $[0-1]$

N : Number of Cores

Amdahl's Law

0% Synchronization

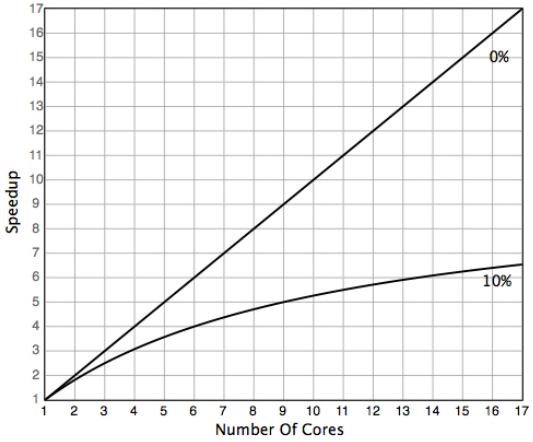
$$S(N) = \frac{1}{(1-P)+\frac{P}{N}}$$
$$P = 0$$



Amdahl's Law

10% Synchronization

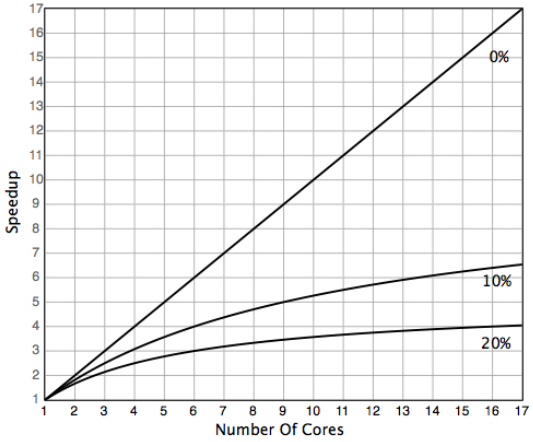
$$S(N) = \frac{1}{(1-P)+\frac{P}{N}}$$
$$P = 0.1$$



Amdahl's Law

20% Synchronization

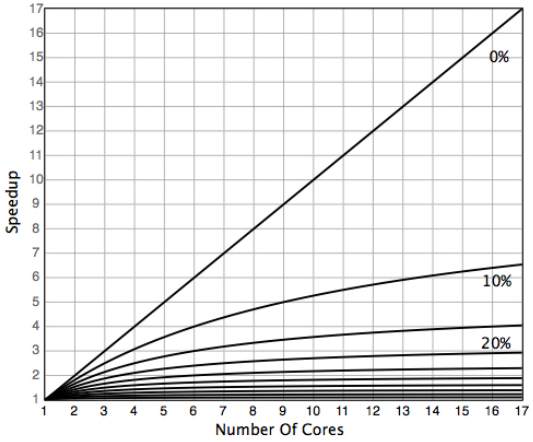
$$S(N) = \frac{1}{(1-P)+\frac{P}{N}}$$
$$P = 0.2$$



Amdahl's Law

90% Synchronization

$$S(N) = \frac{1}{(1-P)+\frac{P}{N}}$$
$$P = 0.9$$



Outline Futures

Why Futures?

Why using futures?

Why Futures?

Why using futures?
Aren't threads, mutex, atomics great?

Why Futures?

Why using futures?
Aren't threads, mutex, atomics great?
They are great tools "to shot yourself into the foot!"

Why Futures?

Why using futures?
Aren't threads, mutex, atomics great?
They are great tools "to shot yourself into the foot!"
It is so easy

- having race conditions
- having dead locks
- wasting CPU cycles through contention

Why Futures?

Why using futures?
Aren't threads, mutex, atomics great?
They are great tools "to shot yourself into the foot!"
It is so easy

- having race conditions
- having dead locks
- wasting CPU cycles through contention

Do you program your application in assembly?

Why Futures?

Why using futures?
Aren't threads, mutex, atomics great?
They are great tools "to shot yourself into the foot!"
It is so easy

- having race conditions
- having dead locks
- wasting CPU cycles through contention

Do you program your application in assembly?
Only if it absolute time critical.
Then don't use tools from the level of assembly!

Future Introduction

- Futures provide a mechanism to separate a function from its result

Future Introduction

- Futures provide a mechanism to separate a function from its result
- After the function is called the result appears "magically" in the future

Future Introduction

- Futures provide a mechanism to separate a function from its result
- After the function is called the result appears "magically" in the future
- A future is a token to the result of a function

Future Introduction

- Futures provide a mechanism to separate a function from its result
- After the function is called the result appears "magically" in the future
- A future is a token to the result of a function
- Added with C++11

Future Introduction

- Futures provide a mechanism to separate a function from its result
- After the function is called the result appears "magically" in the future
- A future is a token to the result of a function
- Added with C++11
- Futures, resp. promises where invented 1977/1978 by Daniel P. Friedman, David Wise, Henry Baker and Carl Hewitt

C++ Standard - Futures

```cpp
#include <future>
#include <iostream>

using namespace std;

int main() {

  auto getTheAnswer = [] {
    this_thread::sleep_for(chrono::milliseconds(815));
    return 42;
  };

  future<int> f2 = async(launch::async, getTheAnswer);

  // Do other stuff, getting the answer may take longer
  cout << f2.get() << '\n';        // access the value
}
```

C++ Standard - Futures

```cpp
#include <future>
#include <iostream>

using namespace std;

int main() {

  auto getTheAnswer = [] {
    this_thread::sleep_for(chrono::milliseconds(815));
    return 42;
  };

  future<int> f2 = async(launch::async, getTheAnswer);

  // Do other stuff, getting the answer may take longer
  cout << f2.get() << '\n';        // access the value
}
```

C++ Standard - Futures

```cpp
#include <future>
#include <iostream>

using namespace std;

int main() {

  auto getTheAnswer = [] {
    this_thread::sleep_for(chrono::milliseconds(815));
    return 42;
  };

  future<int> f2 = async(launch::async, getTheAnswer);

  // Do other stuff, getting the answer may take longer
  cout << f2.get() << '\n';        // access the value
}
```

C++ Standard - Futures

```
#include <future>
#include <iostream>

using namespace std;

int main() {

  auto getTheAnswer = [] {
    this_thread::sleep_for(chrono::milliseconds(815));
    return 42;
  };

  future<int> f2 = async(launch::async, getTheAnswer);

  // Do other stuff, getting the answer may take longer
  cout << f2.get() << '\n';        // access the value
}
```

C++ Standard - Futures

```cpp
#include <future>
#include <iostream>

using namespace std;

int main() {

  auto getTheAnswer = [] {
    this_thread::sleep_for(chrono::milliseconds(815));
    return 42;
  };

  future<int> f2 = async(launch::async, getTheAnswer);

  // Do other stuff, getting the answer may take longer
  cout << f2.get() << '\n';        // access the value
}
```

C++ Standard - Futures

```cpp
#include <future>
#include <iostream>

using namespace std;

int main() {

  auto getTheAnswer = [] {
    this_thread::sleep_for(chrono::milliseconds(815));
    return 42;
  };

  future<int> f2 = async(launch::async, getTheAnswer);

  // Do other stuff, getting the answer may take longer
  cout << f2.get() << '\n';         // access the value
}
```

Output

```
42
```

C++ Standard - Futures - Exceptions

```cpp
#include <future>
#include <iostream>
#include <exception>

using namespace std;

int main() {
  auto getTheAnswer = [] {
    throw runtime_error("Bad things happened: Vogons appeared!");
    return 42;
  };

  future<int> f2 = async(launch::async, getTheAnswer);

  // Do other stuff, getting the answer may take longer
  try {
    cout << f2.get() << '\n';  // try accessing the value
                               // rethrows the stored exception
  }
  catch (const runtime_error& ex) {
    cout << ex.what() << '\n';
  }
}
```

C++ Standard - Futures - Exceptions

```cpp
#include <future>
#include <iostream>
#include <exception>

using namespace std;

int main() {
  auto getTheAnswer = [] {
    throw runtime_error("Bad things happened: Vogons appeared!");
    return 42;
  };

  future<int> f2 = async(launch::async, getTheAnswer);

  // Do other stuff, getting the answer may take longer
  try {
    cout << f2.get() << '\n';  // try accessing the value
                               // rethrows the stored exception
  }
  catch (const runtime_error& ex) {
    cout << ex.what() << '\n';
  }
}
```

C++ Standard - Futures - Exceptions

```cpp
#include <future>
#include <iostream>
#include <exception>

using namespace std;

int main() {
  auto getTheAnswer = [] {
    throw runtime_error("Bad things happened: Vogons appeared!");
    return 42;
  };

  future<int> f2 = async(launch::async, getTheAnswer);

  // Do other stuff, getting the answer may take longer
  try {
    cout << f2.get() << '\n';  // try accessing the value
                               // rethrows the stored exception
  }
  catch (const runtime_error& ex) {
    cout << ex.what() << '\n';
  }
}
```

C++ Standard - Futures - Exceptions

```cpp
#include <future>
#include <iostream>
#include <exception>

using namespace std;

int main() {
  auto getTheAnswer = [] {
    throw runtime_error("Bad things happened: Vogons appeared!");
    return 42;
  };

  future<int> f2 = async(launch::async, getTheAnswer);

  // Do other stuff, getting the answer may take longer
  try {
    cout << f2.get() << '\n';  // try accessing the value
                               // rethrows the stored exception
  }
  catch (const runtime_error& ex) {
    cout << ex.what() << '\n';
  }
}
```

C++ Standard - Futures - Exceptions

```cpp
#include <future>
#include <iostream>
#include <exception>

using namespace std;

int main() {
  auto getTheAnswer = [] {
    throw runtime_error("Bad things happened: Vogons appeared!");
    return 42;
  };

  future<int> f2 = async(launch::async, getTheAnswer);

  // Do other stuff, getting the answer may take longer
  try {
    cout << f2.get() << '\n';  // try accessing the value
                               // rethrows the stored exception
  }
  catch (const runtime_error& ex) {
    cout << ex.what() << '\n';
  }
}
```

Output

Bad things happened: Vogons appeared!

C++ Standard - Futures - Exceptions

```cpp
#include <future>
#include <iostream>
#include <exception>

using namespace std;

int main() {
  auto getTheAnswer = [] {
    throw runtime_error("Bad things happened: Vogons appeared!");
    return 42;
  };

  future<int> f2 = async(launch::async, getTheAnswer);

  // Do other stuff, getting the answer may take longer
  try {
    cout << f2.get() << '\n';  // try accessing the value
                               // rethrows the stored exception
  }
  catch (const runtime_error& ex) {
    cout << ex.what() << '\n';
  }
}
```

Output

Bad things happened: Vogons appeared!

C++11/14 Future Deficiencies

C++11/14 Future Deficiencies

- No continuation – .then() ✗*

* Comes with C++17(TS)

C++11/14 Future Deficiencies

- No continuation – .then() ✗*
- No join – .when_all() and .when_any() ✗*

∗ Comes with C++17(TS)

C++11/14 Future Deficiencies

- No continuation – .then() ✗*
- No join – .when_all() and .when_any() ✗*
- No split – continuation in different directions ✗

∗ Comes with C++17(TS)

C++11/14 Future Deficiencies

- No continuation – .then() ✗*
- No join – .when_all() and .when_any() ✗*
- No split – continuation in different directions ✗
- No cancellation (but can be modelled) ✗

∗ Comes with C++17(TS)

C++11/14 Future Deficiencies

- No continuation – .then() ✗*
- No join – .when_all() and .when_any() ✗*
- No split – continuation in different directions ✗
- No cancellation (but can be modelled) ✗
- No progress monitoring (except ready) ✗

∗ Comes with C++17(TS)

C++11/14 Future Deficiencies

- No continuation – .then() ✗*
- No join – .when_all() and .when_any() ✗*
- No split – continuation in different directions ✗
- No cancellation (but can be modelled) ✗
- No progress monitoring (except ready) ✗
- No custom executor ✗

∗ Comes with C++17(TS)

C++11/14 Future Deficiencies

- No continuation – .then() ✗*
- No join – .when_all() and .when_any() ✗*
- No split – continuation in different directions ✗
- No cancellation (but can be modelled) ✗
- No progress monitoring (except ready) ✗
- No custom executor ✗
- Blocks on destruction (may even blocks until termination of used thread) ✗

∗ Comes with C++17(TS)

C++11/14 Future Deficiencies

- No continuation – .then() ✗*
- No join – .when_all() and .when_any() ✗*
- No split – continuation in different directions ✗
- No cancellation (but can be modelled) ✗
- No progress monitoring (except ready) ✗
- No custom executor ✗
- Blocks on destruction (may even blocks until termination of used thread) ✗
- .get() has two problems:

∗ Comes with C++17(TS)

C++11/14 Future Deficiencies

- No continuation – .then() ✗*
- No join – .when_all() and .when_any() ✗*
- No split – continuation in different directions ✗
- No cancellation (but can be modelled) ✗
- No progress monitoring (except ready) ✗
- No custom executor ✗
- Blocks on destruction (may even blocks until termination of used thread) ✗
- .get() has two problems:
  1. One thread resource is consumed which increases contention and possibly causing a deadlock ✗

∗ Comes with C++17(TS)

C++11/14 Future Deficiencies

- No continuation – .then() ✗*
- No join – .when_all() and .when_any() ✗*
- No split – continuation in different directions ✗
- No cancellation (but can be modelled) ✗
- No progress monitoring (except ready) ✗
- No custom executor ✗
- Blocks on destruction (may even blocks until termination of used thread) ✗
- .get() has two problems:
  1. One thread resource is consumed which increases contention and possibly causing a deadlock ✗
  2. Any subsequent non-dependent calculations on the task are also blocked ✗

∗ Comes with C++17(TS)

C++11/14 Future Deficiencies

- No continuation – .then() ✗*
- No join – .when_all() and .when_any() ✗*
- No split – continuation in different directions ✗
- No cancellation (but can be modelled) ✗
- No progress monitoring (except ready) ✗
- No custom executor ✗
- Blocks on destruction (may even blocks until termination of used thread) ✗
- .get() has two problems:
  1. One thread resource is consumed which increases contention and possibly causing a deadlock ✗
  2. Any subsequent non-dependent calculations on the task are also blocked ✗
- Don't behave as a regular type ✗

∗ Comes with C++17(TS)

Boost - Futures

Boost - Futures

- Continuation – .then() ✓
- Join – .when_all() and .when_any() ✓

Boost - Futures

- Continuation – .then() ✓
- Join – .when_all() and .when_any() ✓
- No split – continuation in different directions ✗
- No cancellation (but can be modelled) ✗
- No progress monitoring (except ready) ✗

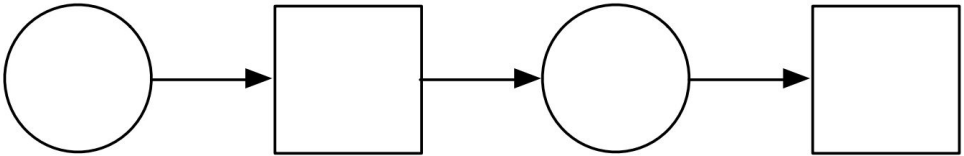
Boost - Futures

- Continuation – .then() ✓
- Join – .when_all() and .when_any() ✓
- No split – continuation in different directions ✗
- No cancellation (but can be modelled) ✗
- No progress monitoring (except ready) ✗
- Custom executor ✓

Boost - Futures

- Continuation – .then() ✓
- Join – .when_all() and .when_any() ✓
- No split – continuation in different directions ✗
- No cancellation (but can be modelled) ✗
- No progress monitoring (except ready) ✗
- Custom executor ✓
- Blocks on destruction (may even blocks until termination of used thread) ✗
- .get() has two problems:
  1. One thread resource is consumed which increases contention and possibly causing a deadlock ✗
  2. Any subsequent non-dependent calculations on the task are also blocked ✗
- Don't behave as a regular type ✗

Future Continuation



C++17(TS) / Boost - Continuation

```cpp
#include <iostream>
#include <boost/thread/future.hpp>

using namespace std;

int main() {
  boost::future<int> answer = boost::async([]{ return 42; });

  boost::future<void> done = answer.then(
    [](boost::future<int> a) { std::cout << a.get() << '\n';} );

  // do something else
  done.wait();      // waits until future done is fulfilled
}
```

C++17(TS) / Boost - Continuation

```cpp
#include <iostream>
#include <boost/thread/future.hpp>

using namespace std;

int main() {
  boost::future<int> answer = boost::async([]{ return 42; });

  boost::future<void> done = answer.then(
    [](boost::future<int> a) { std::cout << a.get() << '\n';} );

  // do something else
  done.wait();      // waits until future done is fulfilled
}
```

C++17(TS) / Boost - Continuation

```
1  #include <iostream>
2  #include <boost/thread/future.hpp>
3
4  using namespace std;
5
6  int main() {
7    boost::future<int> answer = boost::async([]{ return 42; });
8
9    boost::future<void> done = answer.then(
10     [](boost::future<int> a) { std::cout << a.get() << '\n';} );
11
12   // do something else
13   done.wait();       // waits until future done is fulfilled
14 }
```

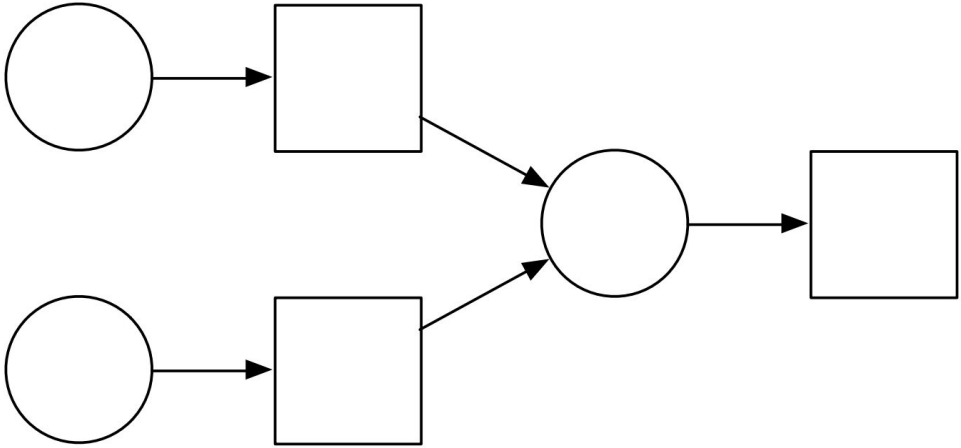
C++17(TS) / Boost - Continuation

```cpp
#include <iostream>
#include <boost/thread/future.hpp>

using namespace std;

int main() {
  boost::future<int> answer = boost::async([]{ return 42; });

  boost::future<void> done = answer.then(
    [](boost::future<int> a) { std::cout << a.get() << '\n';} );

  // do something else
  done.wait();       // waits until future done is fulfilled
}
```

Output

```
42
```

Future Join



C++17(TS) / Boost - Join

```
#include <iostream>
#include <boost/thread/future.hpp>

using namespace std;

int main() {
  auto a = boost::async([]{ return 40; });
  auto b = boost::async([]{ return 2; });

  auto answer = boost::when_all(std::move(a), std::move(b)).then(
    [](auto f) {
      auto t = f.get();
      return get<0>(t).get() + get<1>(t).get();
    });

  // wait for the something else
  cout << answer.get() << '\n';
}
```

C++17(TS) / Boost - Join

```
#include <iostream>
#include <boost/thread/future.hpp>

using namespace std;

int main() {
  auto a = boost::async([]{ return 40; });
  auto b = boost::async([]{ return 2; });

  auto answer = boost::when_all(std::move(a), std::move(b)).then(
    [](auto f) {
      auto t = f.get();
      return get<0>(t).get() + get<1>(t).get();
    });

  // wait for the something else
  cout << answer.get() << '\n';
}
```

C++17(TS) / Boost - Join

```cpp
#include <iostream>
#include <boost/thread/future.hpp>

using namespace std;

int main() {
  auto a = boost::async([]{ return 40; });
  auto b = boost::async([]{ return 2; });

  auto answer = boost::when_all(std::move(a), std::move(b)).then(
    [](auto f) {
      auto t = f.get();
      return get<0>(t).get() + get<1>(t).get();
    });

  // wait for the something else
  cout << answer.get() << '\n';
}
```

C++17(TS) / Boost - Join

```
#include <iostream>
#include <boost/thread/future.hpp>

using namespace std;

int main() {
  auto a = boost::async([]{ return 40; });
  auto b = boost::async([]{ return 2; });

  auto answer = boost::when_all(std::move(a), std::move(b)).then(
    [](auto f) {
      auto t = f.get();
      return get<0>(t).get() + get<1>(t).get();
    });

  // wait for the something else
  cout << answer.get() << '\n';
}
```

C++17(TS) / Boost - Join

```cpp
#include <iostream>
#include <boost/thread/future.hpp>

using namespace std;

int main() {
  auto a = boost::async([]{ return 40; });
  auto b = boost::async([]{ return 2; });

  auto answer = boost::when_all(std::move(a), std::move(b)).then(
    [](auto f) {
      auto t = f.get();
      return get<0>(t).get() + get<1>(t).get();
    });

  // wait for the something else
  cout << answer.get() << '\n';
}
```

C++17(TS) / Boost - Join

```cpp
#include <iostream>
#include <boost/thread/future.hpp>

using namespace std;

int main() {
  auto a = boost::async([]{ return 40; });
  auto b = boost::async([]{ return 2; });

  auto answer = boost::when_all(std::move(a), std::move(b)).then(
    [](auto f) {
      auto t = f.get();
      return get<0>(t).get() + get<1>(t).get();
    });

  // wait for the something else
  cout << answer.get() << '\n';
}
```

Output

```
42
```

C++17(TS) / Boost - Join

```
1  #include <iostream>
2  #include <boost/thread/future.hpp>
3
4  using namespace std;
5
6  int main() {
7    auto a = boost::async([]{ return 40; });
8    auto b = boost::async([]{ return 2; });
9
10   auto answer = boost::when_all(std::move(a), std::move(b)).then(
11     [](auto f) {
12       auto t = f.get();
13       return get<0>(t).get() + get<1>(t).get();
14     });
15
16   // wait for the something else
17   cout << answer.get() << '\n';
18 }
```

What is the type of f?

C++17(TS) / Boost - Join

```cpp
#include <iostream>
#include <boost/thread/future.hpp>

using namespace std;

int main() {
  auto a = boost::async([]{ return 40; });
  auto b = boost::async([]{ return 2; });

  auto answer = boost::when_all(std::move(a), std::move(b)).then(
    [](auto f) {
      auto t = f.get();
      return get<0>(t).get() + get<1>(t).get();
    });

  // wait for the something else
  cout << answer.get() << '\n';
}
```

What is the type of f?

f is a future tuple of futures: `future<tuple<future<int>, future<int>>>`

stlab - Futures

stlab::future

Source: `https://github.com/stlab/libraries`

Documentation: `http://www.stlab.cc/libraries`

stlab - Futures

stlab - Futures

- Continuation – .then() ✓

stlab - Futures

- Continuation – .then() ✓
- Join – .when_all() and .when_any() ✓

stlab - Futures

- Continuation – .then() ✓
- Join – .when_all() and .when_any() ✓
- Split – continuation in different directions ✓

stlab - Futures

- Continuation – .then() ✓
- Join – .when_all() and .when_any() ✓
- Split – continuation in different directions ✓
- Cancellation ✓

stlab - Futures

- Continuation – .then() ✓
- Join – .when_all() and .when_any() ✓
- Split – continuation in different directions ✓
- Cancellation ✓
- No progress monitoring (except ready), more planned ✗

stlab - Futures

- Continuation – .then() ✓
- Join – .when_all() and .when_any() ✓
- Split – continuation in different directions ✓
- Cancellation ✓
- No progress monitoring (except ready), more planned ✗
- Custom executor ✓

stlab - Futures

- Continuation – .then() ✓
- Join – .when_all() and .when_any() ✓
- Split – continuation in different directions ✓
- Cancellation ✓
- No progress monitoring (except ready), more planned ✗
- Custom executor ✓
- Do not block on destruction ✓

stlab - Futures

- Continuation – .then() ✓
- Join – .when_all() and .when_any() ✓
- Split – continuation in different directions ✓
- Cancellation ✓
- No progress monitoring (except ready), more planned ✗
- Custom executor ✓
- Do not block on destruction ✓
- Behave as a regular type ✓

stlab - Futures

- Continuation – .then() ✓
- Join – .when_all() and .when_any() ✓
- Split – continuation in different directions ✓
- Cancellation ✓
- No progress monitoring (except ready), more planned ✗
- Custom executor ✓
- Do not block on destruction ✓
- Behave as a regular type ✓
- Additional dependencies:
  - C++14: boost (optional, variant)
  - C++17: none

stlab::future

```cpp
#include <stlab/future.hpp>
#include <stlab/default_executor.hpp>
#include <iostream>
using namespace std;
int main() {
  auto getTheAnswer = [] {
    this_thread::sleep_for(chrono::milliseconds(815));
    return 42;
  };
  stlab::future<int> f =
    stlab::async(
      stlab::default_executor,// default_executor
                              // uses platfrom thread pool on Win/OSX
                              // uses stlab thread pool on other OS
      getTheAnswer
    );

  while (!f.get_try()) {      // does not block
    // Do other stuff, getting the answer may take longer :-)
  }

  cout << f.get_try().value() << '\n'; // access the value
                           // throws exception .value() if not ready
}
```

stlab::future

```cpp
#include <stlab/future.hpp>
#include <stlab/default_executor.hpp>
#include <iostream>
using namespace std;
int main() {
  auto getTheAnswer = [] {
    this_thread::sleep_for(chrono::milliseconds(815));
    return 42;
  };
  stlab::future<int> f =
    stlab::async(
      stlab::default_executor,// default_executor
                              // uses platfrom thread pool on Win/OSX
                              // uses stlab thread pool on other OS
      getTheAnswer
    );

  while (!f.get_try()) {      // does not block
    // Do other stuff, getting the answer may take longer :-)
  }

  cout << f.get_try().value() << '\n'; // access the value
                           // throws exception .value() if not ready
}
```

stlab::future

```cpp
#include <stlab/future.hpp>
#include <stlab/default_executor.hpp>
#include <iostream>
using namespace std;
int main() {
  auto getTheAnswer = [] {
    this_thread::sleep_for(chrono::milliseconds(815));
    return 42;
  };
  stlab::future<int> f =
    stlab::async(
      stlab::default_executor,// default_executor
                              // uses platfrom thread pool on Win/OSX
                              // uses stlab thread pool on other OS
      getTheAnswer
    );

  while (!f.get_try()) {      // does not block
    // Do other stuff, getting the answer may take longer :-)
  }

  cout << f.get_try().value() << '\n'; // access the value
                           // throws exception .value() if not ready
}
```

stlab::future

```cpp
#include <stlab/future.hpp>
#include <stlab/default_executor.hpp>
#include <iostream>
using namespace std;
int main() {
  auto getTheAnswer = [] {
    this_thread::sleep_for(chrono::milliseconds(815));
    return 42;
  };
  stlab::future<int> f =
    stlab::async(
      stlab::default_executor,// default_executor
                              // uses platfrom thread pool on Win/OSX
                              // uses stlab thread pool on other OS
      getTheAnswer
    );

  while (!f.get_try()) {      // does not block
    // Do other stuff, getting the answer may take longer :-)
  }

  cout << f.get_try().value() << '\n'; // access the value
                            // throws exception .value() if not ready
}
```

stlab::future

```cpp
#include <stlab/future.hpp>
#include <stlab/default_executor.hpp>
#include <iostream>
using namespace std;
int main() {
  auto getTheAnswer = [] {
    this_thread::sleep_for(chrono::milliseconds(815));
    return 42;
  };
  stlab::future<int> f =
    stlab::async(
      stlab::default_executor,// default_executor
                              // uses platfrom thread pool on Win/OSX
                              // uses stlab thread pool on other OS
      getTheAnswer
    );

  while (!f.get_try()) {      // does not block
    // Do other stuff, getting the answer may take longer :-)
  }

  cout << f.get_try().value() << '\n'; // access the value
                           // throws exception .value() if not ready
}
```

stlab::future

```cpp
#include <stlab/future.hpp>
#include <stlab/default_executor.hpp>
#include <iostream>
using namespace std;
int main() {
  auto getTheAnswer = [] {
    this_thread::sleep_for(chrono::milliseconds(815));
    return 42;
  };
  stlab::future<int> f =
    stlab::async(
      stlab::default_executor,// default_executor
                              // uses platfrom thread pool on Win/OSX
                              // uses stlab thread pool on other OS
      getTheAnswer
    );

  while (!f.get_try()) {      // does not block
    // Do other stuff, getting the answer may take longer :-)
  }

  cout << f.get_try().value() << '\n'; // access the value
                           // throws exception .value() if not ready
}
```

stlab::future

```cpp
#include <stlab/future.hpp>
#include <stlab/default_executor.hpp>
#include <iostream>
using namespace std;
int main() {
  auto getTheAnswer = [] {
    this_thread::sleep_for(chrono::milliseconds(815));
    return 42;
  };
  stlab::future<int> f =
    stlab::async(
      stlab::default_executor,// default_executor
                              // uses platfrom thread pool on Win/OSX
                              // uses stlab thread pool on other OS
      getTheAnswer
    );

  while (!f.get_try()) {      // does not block
    // Do other stuff, getting the answer may take longer :-)
  }

  cout << f.get_try().value() << '\n'; // access the value
                            // throws exception .value() if not ready
}
```

stlab::future

```cpp
#include <stlab/future.hpp>
#include <stlab/default_executor.hpp>
#include <iostream>
using namespace std;
int main() {
  auto getTheAnswer = [] {
    this_thread::sleep_for(chrono::milliseconds(815));
    return 42;
  };
  stlab::future<int> f =
    stlab::async(
      stlab::default_executor,// default_executor
                              // uses platfrom thread pool on Win/OSX
                              // uses stlab thread pool on other OS
      getTheAnswer
    );

  while (!f.get_try()) {      // does not block
    // Do other stuff, getting the answer may take longer :-)
  }

  cout << f.get_try().value() << '\n'; // access the value
                           // throws exception .value() if not ready
}
```

Output

```
42
```

stlab::future - Exceptions

```cpp
#include <stlab/future.hpp>
#include <stlab/default_executor.hpp>
#include <iostream>
#include <exception>

int main() {
  auto getTheAnswer = [] {
    throw std::runtime_error("Bad thing happened: Vogons appeared!");
    return 42;
  };
  auto f = stlab::async(stlab::default_executor, getTheAnswer);

  try {
    while (!f.get_try()) { // try accessing the value
                           // may rethrow a stored exception
      // Do other stuff, getting the answer may take longer
    }

    std::cout << f.get_try().value() << '\n';
  }
  catch (const std::runtime_error& ex) {
    std::cout << ex.what() << '\n';
  }
}
```

stlab::future - Exceptions

```cpp
#include <stlab/future.hpp>
#include <stlab/default_executor.hpp>
#include <iostream>
#include <exception>

int main() {
  auto getTheAnswer = [] {
    throw std::runtime_error("Bad thing happened: Vogons appeared!");
    return 42;
  };
  auto f = stlab::async(stlab::default_executor, getTheAnswer);

  try {
    while (!f.get_try()) { // try accessing the value
                           // may rethrow a stored exception
      // Do other stuff, getting the answer may take longer
    }

    std::cout << f.get_try().value() << '\n';
  }
  catch (const std::runtime_error& ex) {
    std::cout << ex.what() << '\n';
  }
}
```

stlab::future - Exceptions

```cpp
#include <stlab/future.hpp>
#include <stlab/default_executor.hpp>
#include <iostream>
#include <exception>

int main() {
  auto getTheAnswer = [] {
    throw std::runtime_error("Bad thing happened: Vogons appeared!");
    return 42;
  };
  auto f = stlab::async(stlab::default_executor, getTheAnswer);

  try {
    while (!f.get_try()) { // try accessing the value
                           // may rethrow a stored exception
      // Do other stuff, getting the answer may take longer
    }

    std::cout << f.get_try().value() << '\n';
  }
  catch (const std::runtime_error& ex) {
    std::cout << ex.what() << '\n';
  }
}
```

stlab::future - Exceptions

```cpp
#include <stlab/future.hpp>
#include <stlab/default_executor.hpp>
#include <iostream>
#include <exception>

int main() {
  auto getTheAnswer = [] {
    throw std::runtime_error("Bad thing happened: Vogons appeared!");
    return 42;
  };
  auto f = stlab::async(stlab::default_executor, getTheAnswer);

  try {
    while (!f.get_try()) { // try accessing the value
                           // may rethrow a stored exception
      // Do other stuff, getting the answer may take longer
    }

    std::cout << f.get_try().value() << '\n';
  }
  catch (const std::runtime_error& ex) {
    std::cout << ex.what() << '\n';
  }
}
```

stlab::future - Exceptions

```cpp
#include <stlab/future.hpp>
#include <stlab/default_executor.hpp>
#include <iostream>
#include <exception>

int main() {
  auto getTheAnswer = [] {
    throw std::runtime_error("Bad thing happened: Vogons appeared!");
    return 42;
  };
  auto f = stlab::async(stlab::default_executor, getTheAnswer);

  try {
    while (!f.get_try()) { // try accessing the value
                           // may rethrow a stored exception
      // Do other stuff, getting the answer may take longer
    }

    std::cout << f.get_try().value() << '\n';
  }
  catch (const std::runtime_error& ex) {
    std::cout << ex.what() << '\n';
  }
}
```

Output

Bad things happened: Vogons appeared!

stlab::future - Continuation

```cpp
#include <stlab/future.hpp>
#include <stlab/default_executor.hpp>
#include <iostream>

int main() {
  auto answer =
    stlab::async(stlab::default_executor, []{ return 42; } );

  stlab::future<void> done = answer.then(
    [](int a)                       // pass by value and not by future
    {
      std::cout << a << '\n';
    });

  while (!done.get_try()) {
    // do something in the meantime
  }
}
```

stlab::future - Continuation

```
#include <stlab/future.hpp>
#include <stlab/default_executor.hpp>
#include <iostream>

int main() {
  auto answer =
    stlab::async(stlab::default_executor, []{ return 42; } );

  stlab::future<void> done = answer.then(
    [](int a)                       // pass by value and not by future
    {
      std::cout << a << '\n';
    });

  while (!done.get_try()) {
    // do something in the meantime
  }
}
```

stlab::future - Continuation

```
#include <stlab/future.hpp>
#include <stlab/default_executor.hpp>
#include <iostream>

int main() {
  auto answer =
    stlab::async(stlab::default_executor, []{ return 42; } );

  stlab::future<void> done = answer.then(
    [](int a)                       // pass by value and not by future
    {
      std::cout << a << '\n';
    });

  while (!done.get_try()) {
    // do something in the meantime
  }
}
```

stlab::future - Continuation

```
#include <stlab/future.hpp>
#include <stlab/default_executor.hpp>
#include <iostream>

int main() {
  auto answer =
    stlab::async(stlab::default_executor, []{ return 42; } );

  stlab::future<void> done = answer.then(
    [](int a)                       // pass by value and not by future
    {
      std::cout << a << '\n';
    });

  while (!done.get_try()) {
    // do something in the meantime
  }
}
```

stlab::future - Continuation

```cpp
#include <stlab/future.hpp>
#include <stlab/default_executor.hpp>
#include <iostream>

int main() {
  auto answer =
    stlab::async(stlab::default_executor, []{ return 42; } );

  stlab::future<void> done = answer.then(
    [](int a)                       // pass by value and not by future
    {
      std::cout << a << '\n';
    });

  while (!done.get_try()) {
    // do something in the meantime
  }
}
```

Output

```
42
```

stlab::future - Continuation

```
1  #include <stlab/future.hpp>
2  #include <stlab/default_executor.hpp>
3  #include <iostream>
4
5  int main() {
6    auto answer =
7      stlab::async(stlab::default_executor, []{ return 42; } );
8
9    stlab::future<void> done = answer.then(
10     [](int a)                       // pass by value and not by future
11     {
12       std::cout << a << '\n';
13     });
14
15   while (!done.get_try()) {
16     // do something in the meantime
17   }
18 }
```

Output

```
42
```

Executors

Executors

- Executors are needed to customize where the task shall be executed

Executors

- Executors are needed to customize where the task shall be executed
- Executors can be general thread pools, serial queues, main queues, dedicated task groups, etc.

stlab::future - Continuation with Custom Executor

```
#include <stlab/future.hpp>
#include <stlab/default_executor.hpp>
#include <iostream>
#include <QLineEdit>
#include "QtScheduler.h"

int main() {
  QLineEdit theAnswerEdit;

  auto answer =
    stlab::async(stlab::default_executor, []{ return 42; } );

  stlab::future<void> done = answer.then(
    QtScheduler(),                      // different scheduler
    [&](int a) { theAnswerEdit.setValue(a); }// here update in main thread
  );

  while (!done.get_try()) {
    // do something in the meantime
  }
}
```

stlab::future - Continuation with Custom Executor

```cpp
#include <stlab/future.hpp>
#include <stlab/default_executor.hpp>
#include <iostream>
#include <QLineEdit>
#include "QtScheduler.h"

int main() {
  QLineEdit theAnswerEdit;

  auto answer =
    stlab::async(stlab::default_executor, []{ return 42; } );

  stlab::future<void> done = answer.then(
    QtScheduler(),                      // different scheduler
    [&](int a) { theAnswerEdit.setValue(a); }// here update in main thread
  );

  while (!done.get_try()) {
    // do something in the meantime
  }
}
```

stlab::future - Continuation with Custom Executor

```
#include <stlab/future.hpp>
#include <stlab/default_executor.hpp>
#include <iostream>
#include <QLineEdit>
#include "QtScheduler.h"

int main() {
  QLineEdit theAnswerEdit;

  auto answer =
    stlab::async(stlab::default_executor, []{ return 42; } );

  stlab::future<void> done = answer.then(
    QtScheduler(),                          // different scheduler
    [&](int a) { theAnswerEdit.setValue(a); }// here update in main thread
  );

  while (!done.get_try()) {
    // do something in the meantime
  }
}
```

stlab::future - Continuation with Custom Executor

```
1  #include <stlab/future.hpp>
2  #include <stlab/default_executor.hpp>
3  #include <iostream>
4  #include <QLineEdit>
5  #include "QtScheduler.h"
6
7  int main() {
8    QLineEdit theAnswerEdit;
9
10   auto answer =
11     stlab::async(stlab::default_executor, []{ return 42; } );
12
13   stlab::future<void> done = answer.then(
14     QtScheduler(),                          // different scheduler
15     [&](int a) { theAnswerEdit.setValue(a); }// here update in main thread
16   );
17
18   while (!done.get_try()) {
19     // do something in the meantime
20   }
21 }
```

stlab::future - Custom Executor

stlab::future - Custom Executor

- In boost, executors derive from a common base class

stlab::future - Custom Executor

- In boost, executors derive from a common base class
- In stlab the executors must provide
  `template <typename F> void operator()(F f)`

stlab::future - Custom Executor

- In boost, executors derive from a common base class
- In stlab the executors must provide
  `template <typename F> void operator()(F f)`
- Let's build exemplary a custom executor for the Qt GUI, that allows to perform updates in the Qt main event loop

stlab::future - Custom Executor - Qt

```cpp
#include <QApplication>
#include <Event>

class QtExecutor
{
  using result_type = void;

  class ExecutorEvent : public QEvent
  {
  };

public:
  template <typename F>
  void operator()(F f) {
    auto event = new ExecutorEvent(std::move(f));
    QApplication::postEvent(event->receiver(), event);
  }
};
```

stlab::future - Custom Executor - Qt

```cpp
#include <QApplication>
#include <Event>

class QtExecutor
{
  using result_type = void;

  class ExecutorEvent : public QEvent
  {
  };

public:
  template <typename F>
  void operator()(F f) {
    auto event = new ExecutorEvent(std::move(f));
    QApplication::postEvent(event->receiver(), event);
  }
};
```

stlab::future - Custom Executor - Qt

```cpp
#include <QApplication>
#include <Event>

class QtExecutor
{
  using result_type = void;

  class ExecutorEvent : public QEvent
  {
  };

public:
  template <typename F>
  void operator()(F f) {
    auto event = new ExecutorEvent(std::move(f));
    QApplication::postEvent(event->receiver(), event);
  }
};
```

stlab::future - Custom Executor - Qt

```cpp
#include <QApplication>
#include <Event>

class QtExecutor
{
  using result_type = void;

  class ExecutorEvent : public QEvent
  {
  };

public:
  template <typename F>
  void operator()(F f) {
    auto event = new ExecutorEvent(std::move(f));
    QApplication::postEvent(event->receiver(), event);
  }
};
```

stlab::future - Custom Executor - Qt

```cpp
#include <QApplication>
#include <Event>

class QtExecutor
{
  using result_type = void;

  class ExecutorEvent : public QEvent
  {
  };

public:
  template <typename F>
  void operator()(F f) {
    auto event = new ExecutorEvent(std::move(f));
    QApplication::postEvent(event->receiver(), event);
  }
};
```

stlab::future - Custom Executor - Qt

```cpp
#include <QApplication>
#include <Event>

class QtExecutor
{
  using result_type = void;

  class ExecutorEvent : public QEvent
  {
  };

public:
  template <typename F>
  void operator()(F f) {
    auto event = new ExecutorEvent(std::move(f));
    QApplication::postEvent(event->receiver(), event);
  }
};
```

stlab::future - Custom Executor - Qt cont. I

```
#include <QApplication>
#include <Event>

class QtExecutor
{
  using result_type = void;

  class EventReceiver;

  class ExecutorEvent : public QEvent
  {
    std::function<void()> _f;
    std::unique_ptr<EventReceiver> _receiver;

  public:
    ExecutorEvent(std::function<void()> f)
      : QEvent(QEvent::User)
      , _f(std::move(f))
      , _receiver(new EventReceiver()) {
      _receiver()->moveToThread(QApplication::instance()->thread());
    }

    void execute() { _f(); }

    QObject *receiver() const { return _receiver.get(); }
  };

public:
};
```

stlab::future - Custom Executor - Qt cont. I

```cpp
#include <QApplication>
#include <Event>

class QtExecutor
{
  using result_type = void;

  class EventReceiver;

  class ExecutorEvent : public QEvent
  {
    std::function<void()> _f;
    std::unique_ptr<EventReceiver> _receiver;

  public:
    ExecutorEvent(std::function<void()> f)
      : QEvent(QEvent::User)
      , _f(std::move(f))
      , _receiver(new EventReceiver()) {
      _receiver()->moveToThread(QApplication::instance()->thread());
    }

    void execute() { _f(); }

    QObject *receiver() const { return _receiver.get(); }
  };

public:
};
```

stlab::future - Custom Executor - Qt cont. I

```
#include <QApplication>
#include <Event>

class QtExecutor
{
  using result_type = void;

  class EventReceiver;

  class ExecutorEvent : public QEvent
  {
    std::function<void()> _f;
    std::unique_ptr<EventReceiver> _receiver;

  public:
    ExecutorEvent(std::function<void()> f)
      : QEvent(QEvent::User)
      , _f(std::move(f))
      , _receiver(new EventReceiver()) {
      _receiver()->moveToThread(QApplication::instance()->thread());
    }

    void execute() { _f(); }

    QObject *receiver() const { return _receiver.get(); }
  };

public:
};
```

stlab::future - Custom Executor - Qt cont. I

```
#include <QApplication>
#include <Event>

class QtExecutor
{
  using result_type = void;

  class EventReceiver;

  class ExecutorEvent : public QEvent
  {
    std::function<void()> _f;
    std::unique_ptr<EventReceiver> _receiver;

  public:
    ExecutorEvent(std::function<void()> f)
      : QEvent(QEvent::User)
      , _f(std::move(f))
      , _receiver(new EventReceiver()) {
      _receiver()->moveToThread(QApplication::instance()->thread());
    }

    void execute() { _f(); }

    QObject *receiver() const { return _receiver.get(); }
  };

public:
};
```

stlab::future - Custom Executor - Qt cont. I

```
#include <QApplication>
#include <Event>

class QtExecutor
{
  using result_type = void;

  class EventReceiver;

  class ExecutorEvent : public QEvent
  {
    std::function<void()> _f;
    std::unique_ptr<EventReceiver> _receiver;

  public:
    ExecutorEvent(std::function<void()> f)
      : QEvent(QEvent::User)
      , _f(std::move(f))
      , _receiver(new EventReceiver()) {
      _receiver()->moveToThread(QApplication::instance()->thread());
    }

    void execute() { _f(); }

    QObject *receiver() const { return _receiver.get(); }
  };

public:
};
```

stlab::future - Custom Executor - Qt cont. I

```
#include <QApplication>
#include <Event>

class QtExecutor
{
  using result_type = void;

  class EventReceiver;

  class ExecutorEvent : public QEvent
  {
    std::function<void()> _f;
    std::unique_ptr<EventReceiver> _receiver;

  public:
    ExecutorEvent(std::function<void()> f)
      : QEvent(QEvent::User)
      , _f(std::move(f))
      , _receiver(new EventReceiver()) {
      _receiver()->moveToThread(QApplication::instance()->thread());
    }

    void execute() { _f(); }

    QObject *receiver() const { return _receiver.get(); }
  };

public:
};
```

stlab::future - Custom Executor - Qt cont. I

```
#include <QApplication>
#include <Event>

class QtExecutor
{
  using result_type = void;

  class EventReceiver;

  class ExecutorEvent : public QEvent
  {
    std::function<void()> _f;
    std::unique_ptr<EventReceiver> _receiver;

  public:
    ExecutorEvent(std::function<void()> f)
      : QEvent(QEvent::User)
      , _f(std::move(f))
      , _receiver(new EventReceiver()) {
      _receiver()->moveToThread(QApplication::instance()->thread());
    }

    void execute() { _f(); }

    QObject *receiver() const { return _receiver.get(); }
  };

public:
};
```

stlab::future - Custom Executor - Qt cont. II

```cpp
#include <QApplication>
#include <Event>

class QtExecutor
{
  class ExecutorEvent : public QEvent
  {
    QObject *receiver() const { return _receiver.get(); }
  };

  class EventReceiver : public QObject
  {
  public:
    bool event(QEvent *event) override {
      auto myEvent = dynamic_cast<ExecutorEvent*>(event);
      if (myEvent) {
        myEvent->execute();
        return true;
      }
      return false;
    }
  };

public:
  template <typename F>
  void operator()(F f) {
    auto event = new ExecutorEvent(std::move(f));
    QApplication::postEvent(event->receiver(), event);
  }
};
```

stlab::future - Custom Executor - Qt cont. II

```cpp
#include <QApplication>
#include <Event>

class QtExecutor
{
  class ExecutorEvent : public QEvent
  {
    QObject *receiver() const { return _receiver.get(); }
  };

  class EventReceiver : public QObject
  {
  public:
    bool event(QEvent *event) override {
      auto myEvent = dynamic_cast<ExecutorEvent*>(event);
      if (myEvent) {
        myEvent->execute();
        return true;
      }
      return false;
    }
  };

public:
  template <typename F>
  void operator()(F f) {
    auto event = new ExecutorEvent(std::move(f));
    QApplication::postEvent(event->receiver(), event);
  }
};
```

stlab::future - Custom Executor - Qt cont. II

```cpp
#include <QApplication>
#include <Event>

class QtExecutor
{
  class ExecutorEvent : public QEvent
  {
    QObject *receiver() const { return _receiver.get(); }
  };

  class EventReceiver : public QObject
  {
  public:
    bool event(QEvent *event) override {
      auto myEvent = dynamic_cast<ExecutorEvent*>(event);
      if (myEvent) {
        myEvent->execute();
        return true;
      }
      return false;
    }
  };

public:
  template <typename F>
  void operator()(F f) {
    auto event = new ExecutorEvent(std::move(f));
    QApplication::postEvent(event->receiver(), event);
  }
};
```

stlab::future - Custom Executor - Qt cont. II

```cpp
#include <QApplication>
#include <Event>

class QtExecutor
{
  class ExecutorEvent : public QEvent
  {
    QObject *receiver() const { return _receiver.get(); }
  };

  class EventReceiver : public QObject
  {
  public:
    bool event(QEvent *event) override {
      auto myEvent = dynamic_cast<ExecutorEvent*>(event);
      if (myEvent) {
        myEvent->execute();
        return true;
      }
      return false;
    }
  };

public:
  template <typename F>
  void operator()(F f) {
    auto event = new ExecutorEvent(std::move(f));
    QApplication::postEvent(event->receiver(), event);
  }
};
```

stlab::future - Custom Executor - Qt cont. II

```cpp
#include <QApplication>
#include <Event>

class QtExecutor
{
  class ExecutorEvent : public QEvent
  {
    QObject *receiver() const { return _receiver.get(); }
  };

  class EventReceiver : public QObject
  {
  public:
    bool event(QEvent *event) override {
      auto myEvent = dynamic_cast<ExecutorEvent*>(event);
      if (myEvent) {
        myEvent->execute();
        return true;
      }
      return false;
    }
  };

public:
  template <typename F>
  void operator()(F f) {
    auto event = new ExecutorEvent(std::move(f));
    QApplication::postEvent(event->receiver(), event);
  }
};
```

stlab::future - Custom Executor - Qt cont. II

```cpp
#include <QApplication>
#include <Event>

class QtExecutor
{
  class ExecutorEvent : public QEvent
  {
    QObject *receiver() const { return _receiver.get(); }
  };

  class EventReceiver : public QObject
  {
  public:
    bool event(QEvent *event) override {
      auto myEvent = dynamic_cast<ExecutorEvent*>(event);
      if (myEvent) {
        myEvent->execute();
        return true;
      }
      return false;
    }
  };

public:
  template <typename F>
  void operator()(F f) {
    auto event = new ExecutorEvent(std::move(f));
    QApplication::postEvent(event->receiver(), event);
  }
};
```

stlab::future - Error Recovery

```
1  int main() {
2    auto getTheAnswer = [] {
3      throw std::runtime_error("Bad thing happened: Vogons appeared");
4      std::cout << "I have got the answer\n"; return 42;
5    };
6    auto handleTheAnswer = [](int v) {
7      if (v == 0) std::cout << "We have a problem!\n";
8      else std::cout << "The answer is " << v << '\n';
9    };
10
11   auto f = stlab::async(stlab::default_executor, getTheAnswer)
12     .recover([](stlab::future<int> result) {
13       if (result.error()) {
14         std::cout << "Listen to Vogon poetry!\n";
15         return 0;
16       }
17       return result.get_try().value();
18   }).then(handleTheAnswer);
19
20   while (!f.get_try());
21 }
```

stlab::future - Error Recovery

```
1  int main() {
2    auto getTheAnswer = [] {
3      throw std::runtime_error("Bad thing happened: Vogons appeared");
4      std::cout << "I have got the answer\n"; return 42;
5    };
6    auto handleTheAnswer = [](int v) {
7      if (v == 0) std::cout << "We have a problem!\n";
8      else std::cout << "The answer is " << v << '\n';
9    };
10
11   auto f = stlab::async(stlab::default_executor, getTheAnswer)
12     .recover([](stlab::future<int> result) {
13       if (result.error()) {
14         std::cout << "Listen to Vogon poetry!\n";
15         return 0;
16       }
17       return result.get_try().value();
18   }).then(handleTheAnswer);
19
20   while (!f.get_try());
21 }
```

stlab::future - Error Recovery

```
1  int main() {
2    auto getTheAnswer = [] {
3      throw std::runtime_error("Bad thing happened: Vogons appeared");
4      std::cout << "I have got the answer\n"; return 42;
5    };
6    auto handleTheAnswer = [](int v) {
7      if (v == 0) std::cout << "We have a problem!\n";
8      else std::cout << "The answer is " << v << '\n';
9    };
10
11   auto f = stlab::async(stlab::default_executor, getTheAnswer)
12     .recover([](stlab::future<int> result) {
13       if (result.error()) {
14         std::cout << "Listen to Vogon poetry!\n";
15         return 0;
16       }
17       return result.get_try().value();
18   }).then(handleTheAnswer);
19
20   while (!f.get_try());
21 }
```

stlab::future - Error Recovery

```
1  int main() {
2    auto getTheAnswer = [] {
3      throw std::runtime_error("Bad thing happened: Vogons appeared");
4      std::cout << "I have got the answer\n"; return 42;
5    };
6    auto handleTheAnswer = [](int v) {
7      if (v == 0) std::cout << "We have a problem!\n";
8      else std::cout << "The answer is " << v << '\n';
9    };
10
11   auto f = stlab::async(stlab::default_executor, getTheAnswer)
12     .recover([](stlab::future<int> result) {
13       if (result.error()) {
14         std::cout << "Listen to Vogon poetry!\n";
15         return 0;
16       }
17       return result.get_try().value();
18   }).then(handleTheAnswer);
19
20   while (!f.get_try());
21 }
```

stlab::future - Error Recovery

```
1  int main() {
2    auto getTheAnswer = [] {
3      throw std::runtime_error("Bad thing happened: Vogons appeared");
4      std::cout << "I have got the answer\n"; return 42;
5    };
6    auto handleTheAnswer = [](int v) {
7      if (v == 0) std::cout << "We have a problem!\n";
8      else std::cout << "The answer is " << v << '\n';
9    };
10
11   auto f = stlab::async(stlab::default_executor, getTheAnswer)
12     .recover([](stlab::future<int> result) {
13       if (result.error()) {
14         std::cout << "Listen to Vogon poetry!\n";
15         return 0;
16       }
17       return result.get_try().value();
18   }).then(handleTheAnswer);
19
20   while (!f.get_try());
21 }
```

stlab::future - Error Recovery

```
1  int main() {
2    auto getTheAnswer = [] {
3      throw std::runtime_error("Bad thing happened: Vogons appeared");
4      std::cout << "I have got the answer\n"; return 42;
5    };
6    auto handleTheAnswer = [](int v) {
7      if (v == 0) std::cout << "We have a problem!\n";
8      else std::cout << "The answer is " << v << '\n';
9    };
10
11   auto f = stlab::async(stlab::default_executor, getTheAnswer)
12     .recover([](stlab::future<int> result) {
13       if (result.error()) {
14         std::cout << "Listen to Vogon poetry!\n";
15         return 0;
16       }
17       return result.get_try().value();
18   }).then(handleTheAnswer);
19
20   while (!f.get_try());
21 }
```

stlab::future - Error Recovery

```
1  int main() {
2    auto getTheAnswer = [] {
3      throw std::runtime_error("Bad thing happened: Vogons appeared");
4      std::cout << "I have got the answer\n"; return 42;
5    };
6    auto handleTheAnswer = [](int v) {
7      if (v == 0) std::cout << "We have a problem!\n";
8      else std::cout << "The answer is " << v << '\n';
9    };
10
11   auto f = stlab::async(stlab::default_executor, getTheAnswer)
12     .recover([](stlab::future<int> result) {
13       if (result.error()) {
14         std::cout << "Listen to Vogon poetry!\n";
15         return 0;
16       }
17       return result.get_try().value();
18   }).then(handleTheAnswer);
19
20   while (!f.get_try());
21 }
```

stlab::future - Error Recovery

```
1  int main() {
2    auto getTheAnswer = [] {
3      throw std::runtime_error("Bad thing happened: Vogons appeared");
4      std::cout << "I have got the answer\n"; return 42;
5    };
6    auto handleTheAnswer = [](int v) {
7      if (v == 0) std::cout << "We have a problem!\n";
8      else std::cout << "The answer is " << v << '\n';
9    };
10
11   auto f = stlab::async(stlab::default_executor, getTheAnswer)
12     .recover([](stlab::future<int> result) {
13       if (result.error()) {
14         std::cout << "Listen to Vogon poetry!\n";
15         return 0;
16       }
17       return result.get_try().value();
18   }).then(handleTheAnswer);
19
20   while (!f.get_try());
21 }
```

stlab::future - Error Recovery

```
1  int main() {
2    auto getTheAnswer = [] {
3      throw std::runtime_error("Bad thing happened: Vogons appeared");
4      std::cout << "I have got the answer\n"; return 42;
5    };
6    auto handleTheAnswer = [](int v) {
7      if (v == 0) std::cout << "We have a problem!\n";
8      else std::cout << "The answer is " << v << '\n';
9    };
10
11   auto f = stlab::async(stlab::default_executor, getTheAnswer)
12     .recover([](stlab::future<int> result) {
13       if (result.error()) {
14         std::cout << "Listen to Vogon poetry!\n";
15         return 0;
16       }
17       return result.get_try().value();
18   }).then(handleTheAnswer);
19
20   while (!f.get_try());
21 }
```

Output

Listen to Vogon poetry!
We have a problem!

stlab::future - Join

```cpp
#include <stlab/future.hpp>
#include <stlab/default_executor.hpp>
#include <iostream>

using namespace stlab;

int main() {
  auto a = async(default_executor,[]{ return 40; });
  auto b = async(default_executor,[]{ return 2; });

  auto answer = when_all(
    default_executor,
    [](int x, int y) { return x + y; },
    a, b);

  while (!answer.get_try()) {
  // wait for something else
  }
  std::cout << answer.get_try().value() << '\n';
}
```

stlab::future - Join

```cpp
#include <stlab/future.hpp>
#include <stlab/default_executor.hpp>
#include <iostream>

using namespace stlab;

int main() {
  auto a = async(default_executor,[]{ return 40; });
  auto b = async(default_executor,[]{ return 2; });

  auto answer = when_all(
    default_executor,
    [](int x, int y) { return x + y; },
    a, b);

  while (!answer.get_try()) {
  // wait for something else
  }
  std::cout << answer.get_try().value() << '\n';
}
```

stlab::future - Join

```cpp
#include <stlab/future.hpp>
#include <stlab/default_executor.hpp>
#include <iostream>

using namespace stlab;

int main() {
  auto a = async(default_executor,[]{ return 40; });
  auto b = async(default_executor,[]{ return 2; });

  auto answer = when_all(
    default_executor,
    [](int x, int y) { return x + y; },
    a, b);

  while (!answer.get_try()) {
  // wait for something else
  }
  std::cout << answer.get_try().value() << '\n';
}
```

stlab::future - Join

```
1  #include <stlab/future.hpp>
2  #include <stlab/default_executor.hpp>
3  #include <iostream>
4
5  using namespace stlab;
6
7  int main() {
8    auto a = async(default_executor,[]{ return 40; });
9    auto b = async(default_executor,[]{ return 2; });
10
11   auto answer = when_all(
12     default_executor,
13     [](int x, int y) { return x + y; },
14     a, b);
15
16   while (!answer.get_try()) {
17   // wait for something else
18   }
19   std::cout << answer.get_try().value() << '\n';
20 }
```

stlab::future - Join

```cpp
#include <stlab/future.hpp>
#include <stlab/default_executor.hpp>
#include <iostream>

using namespace stlab;

int main() {
  auto a = async(default_executor,[]{ return 40; });
  auto b = async(default_executor,[]{ return 2; });

  auto answer = when_all(
    default_executor,
    [](int x, int y) { return x + y; },
    a, b);

  while (!answer.get_try()) {
  // wait for something else
  }
  std::cout << answer.get_try().value() << '\n';
}
```

stlab::future - Join

```cpp
#include <stlab/future.hpp>
#include <stlab/default_executor.hpp>
#include <iostream>

using namespace stlab;

int main() {
  auto a = async(default_executor,[]{ return 40; });
  auto b = async(default_executor,[]{ return 2; });

  auto answer = when_all(
    default_executor,
    [](int x, int y) { return x + y; },
    a, b);

  while (!answer.get_try()) {
  // wait for something else
  }
  std::cout << answer.get_try().value() << '\n';
}
```

stlab::future - Join

```cpp
#include <stlab/future.hpp>
#include <stlab/default_executor.hpp>
#include <iostream>

using namespace stlab;

int main() {
  auto a = async(default_executor,[]{ return 40; });
  auto b = async(default_executor,[]{ return 2; });

  auto answer = when_all(
    default_executor,
    [](int x, int y) { return x + y; },
    a, b);

  while (!answer.get_try()) {
  // wait for something else
  }
  std::cout << answer.get_try().value() << '\n';
}
```

stlab::future - Join

```cpp
#include <stlab/future.hpp>
#include <stlab/default_executor.hpp>
#include <iostream>

using namespace stlab;

int main() {
  auto a = async(default_executor,[]{ return 40; });
  auto b = async(default_executor,[]{ return 2; });

  auto answer = when_all(
    default_executor,
    [](int x, int y) { return x + y; },
    a, b);

  while (!answer.get_try()) {
  // wait for something else
  }
  std::cout << answer.get_try().value() << '\n';
}
```

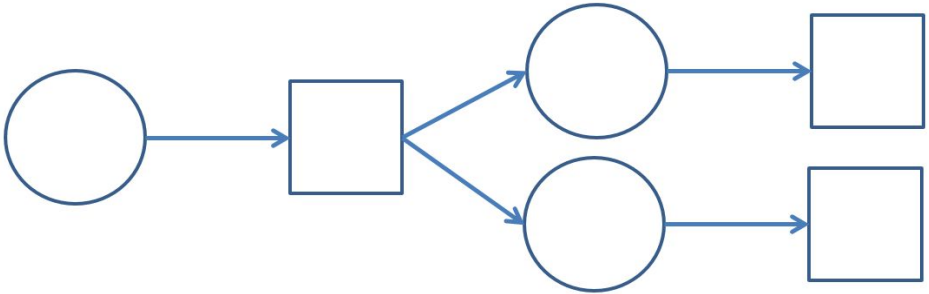
stlab::future - Join

```cpp
#include <stlab/future.hpp>
#include <stlab/default_executor.hpp>
#include <iostream>

using namespace stlab;

int main() {
  auto a = async(default_executor,[]{ return 40; });
  auto b = async(default_executor,[]{ return 2; });

  auto answer = when_all(
    default_executor,
    [](int x, int y) { return x + y; },
    a, b);

  while (!answer.get_try()) {
  // wait for something else
  }
  std::cout << answer.get_try().value() << '\n';
}
```

Output

```
42
```

future - Split



stlab::future - Split

```
1  #include <stlab/future.hpp>
2  #include <stlab/default_executor.hpp>
3  #include <iostream>
4
5  using namespace stlab;
6
7  int main() {
8    auto answer = async(default_executor,[]{ return 42; });
9
10   auto dent = answer.then([](int a) {
11     std::cout << "Tell the answer " << a << " Arthur Dent\n";
12   });
13
14   auto marvin = answer.then([](int a) {
15     std::cout << "May the answer " << a << " shear up Marvin\n";
16   });
17
18   while (!dent.get_try() && !marvin.get_try()) {
19   // wait for  something else
20   }
21 }
```

stlab::future - Split

```
#include <stlab/future.hpp>
#include <stlab/default_executor.hpp>
#include <iostream>

using namespace stlab;

int main() {
  auto answer = async(default_executor,[]{ return 42; });

  auto dent = answer.then([](int a) {
    std::cout << "Tell the answer " << a << " Arthur Dent\n";
  });

  auto marvin = answer.then([](int a) {
    std::cout << "May the answer " << a << " shear up Marvin\n";
  });

  while (!dent.get_try() && !marvin.get_try()) {
  // wait for  something else
  }
}
```

stlab::future - Split

```
1  #include <stlab/future.hpp>
2  #include <stlab/default_executor.hpp>
3  #include <iostream>
4
5  using namespace stlab;
6
7  int main() {
8    auto answer = async(default_executor,[]{ return 42; });
9
10   auto dent = answer.then([](int a) {
11     std::cout << "Tell the answer " << a << " Arthur Dent\n";
12   });
13
14   auto marvin = answer.then([](int a) {
15     std::cout << "May the answer " << a << " shear up Marvin\n";
16   });
17
18   while (!dent.get_try() && !marvin.get_try()) {
19   // wait for  something else
20   }
21 }
```

stlab::future - Split

```
1  #include <stlab/future.hpp>
2  #include <stlab/default_executor.hpp>
3  #include <iostream>
4
5  using namespace stlab;
6
7  int main() {
8    auto answer = async(default_executor,[]{ return 42; });
9
10   auto dent = answer.then([](int a) {
11     std::cout << "Tell the answer " << a << " Arthur Dent\n";
12   });
13
14   auto marvin = answer.then([](int a) {
15     std::cout << "May the answer " << a << " shear up Marvin\n";
16   });
17
18   while (!dent.get_try() && !marvin.get_try()) {
19   // wait for  something else
20   }
21 }
```

stlab::future - Split

```
1  #include <stlab/future.hpp>
2  #include <stlab/default_executor.hpp>
3  #include <iostream>
4
5  using namespace stlab;
6
7  int main() {
8    auto answer = async(default_executor,[]{ return 42; });
9
10   auto dent = answer.then([](int a) {
11     std::cout << "Tell the answer " << a << " Arthur Dent\n";
12   });
13
14   auto marvin = answer.then([](int a) {
15     std::cout << "May the answer " << a << " shear up Marvin\n";
16   });
17
18   while (!dent.get_try() && !marvin.get_try()) {
19   // wait for  something else
20   }
21 }
```

Output

Tell the answer May the answer 42 Arthur Dent

42 shear up Marvin

stlab::future - Split

```
#include <stlab/future.hpp>
#include <stlab/default_executor.hpp>
#include <iostream>

using namespace stlab;

int main() {
  auto answer = async(default_executor,[]{ return 42; });

  auto dent = answer.then([](int a) {
    std::cout << "Tell the answer " << a << " Arthur Dent\n";
  });

  auto marvin = answer.then([](int a) {
    std::cout << "May the answer " << a << " shear up Marvin\n";
  });

  while (!dent.get_try() && !marvin.get_try()) {
  // wait for  something else
  }
}
```

Output

Tell the answer May the answer 42 Arthur Dent

42 shear up Marvin ⇒ **Race condition by using std::cout**

Exercise 1

Demo Exercise 1

Exercise 1

Change the application in a way that

- using Start does not block the UI,
- it is possible to cancel the running operation,
- it is possible to restart it.

Conclusion

Futures are a great concept to structure the code so that it runs with minimal contention.

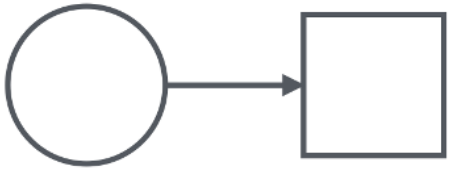
Conclusion

Futures are a great concept to structure the code so that it runs with minimal contention.

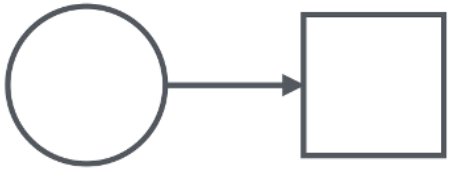
After a single execution the graph cannot be used any more.

Outline Channels

Channel Introduction

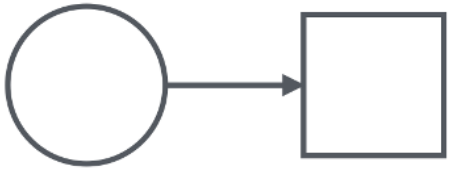


Channel Introduction



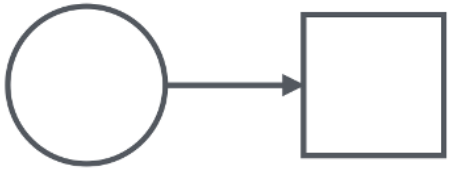
- Each change triggers a notification to the sink values

Channel Introduction



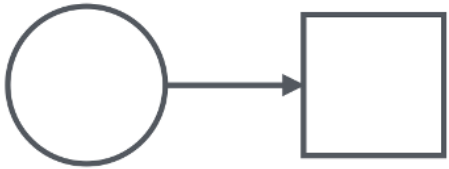
- Each change triggers a notification to the sink values
- Channels allow the creation of persistent execution graphs

Channel Introduction



- Each change triggers a notification to the sink values
- Channels allow the creation of persistent execution graphs
- This is also known as reactive programming model

Channel Introduction



- Each change triggers a notification to the sink values
- Channels allow the creation of persistent execution graphs
- This is also known as reactive programming model
- First published by Tony Hoare 1978

Channel - Stateless Process

```cpp
#include <stlab/channel.hpp>
#include <stlab/default_executor.hpp>
#include <iostream>
int main() {
  stlab::sender<int> send;        // sending part of the channel
  stlab::receiver<int> receiver; // receiving part of the channel
  std::tie(send, receiver) =      // combining both to a channel
    stlab::channel<int>(stlab::default_executor);

  auto printer =
    [](int x){ std::cout << x << '\n'; }; // stateless process

  auto printer_process =
    receiver | printer;           // attaching process to the receiving
                                  // part
  receiver.set_ready();           // no more processes will be attached
                                  // process starts to work
  send(1); send(2); send(3);      // start sending into the channel

  int end; std::cin >> end;       // simply wait to end application
}
```

Channel - Stateless Process

```cpp
#include <stlab/channel.hpp>
#include <stlab/default_executor.hpp>
#include <iostream>
int main() {
  stlab::sender<int> send;        // sending part of the channel
  stlab::receiver<int> receiver; // receiving part of the channel
  std::tie(send, receiver) =      // combining both to a channel
    stlab::channel<int>(stlab::default_executor);

  auto printer =
    [](int x){ std::cout << x << '\n'; }; // stateless process

  auto printer_process =
    receiver | printer;           // attaching process to the receiving
                                  // part
  receiver.set_ready();           // no more processes will be attached
                                  // process starts to work
  send(1); send(2); send(3);      // start sending into the channel

  int end; std::cin >> end;       // simply wait to end application
}
```

Channel - Stateless Process

```cpp
#include <stlab/channel.hpp>
#include <stlab/default_executor.hpp>
#include <iostream>
int main() {
  stlab::sender<int> send;        // sending part of the channel
  stlab::receiver<int> receiver; // receiving part of the channel
  std::tie(send, receiver) =      // combining both to a channel
    stlab::channel<int>(stlab::default_executor);

  auto printer =
    [](int x){ std::cout << x << '\n'; }; // stateless process

  auto printer_process =
    receiver | printer;           // attaching process to the receiving
                                  // part
  receiver.set_ready();           // no more processes will be attached
                                  // process starts to work
  send(1); send(2); send(3);      // start sending into the channel

  int end; std::cin >> end;       // simply wait to end application
}
```

Channel - Stateless Process

```cpp
#include <stlab/channel.hpp>
#include <stlab/default_executor.hpp>
#include <iostream>
int main() {
  stlab::sender<int> send;        // sending part of the channel
  stlab::receiver<int> receiver; // receiving part of the channel
  std::tie(send, receiver) =     // combining both to a channel
    stlab::channel<int>(stlab::default_executor);

  auto printer =
    [](int x){ std::cout << x << '\n'; }; // stateless process

  auto printer_process =
    receiver | printer;           // attaching process to the receiving
                                  // part
  receiver.set_ready();           // no more processes will be attached
                                  // process starts to work
  send(1); send(2); send(3);      // start sending into the channel

  int end; std::cin >> end;       // simply wait to end application
}
```

Channel - Stateless Process

```cpp
#include <stlab/channel.hpp>
#include <stlab/default_executor.hpp>
#include <iostream>
int main() {
  stlab::sender<int> send;        // sending part of the channel
  stlab::receiver<int> receiver; // receiving part of the channel
  std::tie(send, receiver) =      // combining both to a channel
    stlab::channel<int>(stlab::default_executor);

  auto printer =
    [](int x){ std::cout << x << '\n'; }; // stateless process

  auto printer_process =
    receiver | printer;           // attaching process to the receiving
                                  // part
  receiver.set_ready();           // no more processes will be attached
                                  // process starts to work
  send(1); send(2); send(3);      // start sending into the channel

  int end; std::cin >> end;       // simply wait to end application
}
```

Channel - Stateless Process

```
#include <stlab/channel.hpp>
#include <stlab/default_executor.hpp>
#include <iostream>
int main() {
  stlab::sender<int> send;        // sending part of the channel
  stlab::receiver<int> receiver; // receiving part of the channel
  std::tie(send, receiver) =      // combining both to a channel
    stlab::channel<int>(stlab::default_executor);

  auto printer =
    [](int x){ std::cout << x << '\n'; }; // stateless process

  auto printer_process =
    receiver | printer;           // attaching process to the receiving
                                  // part
  receiver.set_ready();           // no more processes will be attached
                                  // process starts to work
  send(1); send(2); send(3);      // start sending into the channel

  int end; std::cin >> end;       // simply wait to end application
}
```

Channel - Stateless Process

```cpp
#include <stlab/channel.hpp>
#include <stlab/default_executor.hpp>
#include <iostream>
int main() {
  stlab::sender<int> send;        // sending part of the channel
  stlab::receiver<int> receiver; // receiving part of the channel
  std::tie(send, receiver) =      // combining both to a channel
    stlab::channel<int>(stlab::default_executor);

  auto printer =
    [](int x){ std::cout << x << '\n'; }; // stateless process

  auto printer_process =
    receiver | printer;           // attaching process to the receiving
                                  // part
  receiver.set_ready();           // no more processes will be attached
                                  // process starts to work
  send(1); send(2); send(3);      // start sending into the channel

  int end; std::cin >> end;       // simply wait to end application
}
```

Channel - Stateless Process

```cpp
#include <stlab/channel.hpp>
#include <stlab/default_executor.hpp>
#include <iostream>
int main() {
  stlab::sender<int> send;        // sending part of the channel
  stlab::receiver<int> receiver; // receiving part of the channel
  std::tie(send, receiver) =      // combining both to a channel
    stlab::channel<int>(stlab::default_executor);

  auto printer =
    [](int x){ std::cout << x << '\n'; }; // stateless process

  auto printer_process =
    receiver | printer;           // attaching process to the receiving
                                  // part
  receiver.set_ready();           // no more processes will be attached
                                  // process starts to work
  send(1); send(2); send(3);      // start sending into the channel

  int end; std::cin >> end;       // simply wait to end application
}
```

Channel - Stateless Process

```cpp
#include <stlab/channel.hpp>
#include <stlab/default_executor.hpp>
#include <iostream>
int main() {
  stlab::sender<int> send;        // sending part of the channel
  stlab::receiver<int> receiver; // receiving part of the channel
  std::tie(send, receiver) =      // combining both to a channel
    stlab::channel<int>(stlab::default_executor);

  auto printer =
    [](int x){ std::cout << x << '\n'; }; // stateless process

  auto printer_process =
    receiver | printer;           // attaching process to the receiving
                                  // part
  receiver.set_ready();           // no more processes will be attached
                                  // process starts to work
  send(1); send(2); send(3);      // start sending into the channel

  int end; std::cin >> end;       // simply wait to end application
}
```

Channel - Stateless Process

```cpp
#include <stlab/channel.hpp>
#include <stlab/default_executor.hpp>
#include <iostream>
int main() {
  stlab::sender<int> send;        // sending part of the channel
  stlab::receiver<int> receiver; // receiving part of the channel
  std::tie(send, receiver) =      // combining both to a channel
    stlab::channel<int>(stlab::default_executor);

  auto printer =
    [](int x){ std::cout << x << '\n'; }; // stateless process

  auto printer_process =
    receiver | printer;           // attaching process to the receiving
                                  // part
  receiver.set_ready();           // no more processes will be attached
                                  // process starts to work
  send(1); send(2); send(3);      // start sending into the channel

  int end; std::cin >> end;       // simply wait to end application
}
```

Channel - Stateless Process

```cpp
#include <stlab/channel.hpp>
#include <stlab/default_executor.hpp>
#include <iostream>
int main() {
  stlab::sender<int> send;        // sending part of the channel
  stlab::receiver<int> receiver; // receiving part of the channel
  std::tie(send, receiver) =      // combining both to a channel
    stlab::channel<int>(stlab::default_executor);

  auto printer =
    [](int x){ std::cout << x << '\n'; }; // stateless process

  auto printer_process =
    receiver | printer;           // attaching process to the receiving
                                  // part
  receiver.set_ready();           // no more processes will be attached
                                  // process starts to work
  send(1); send(2); send(3);      // start sending into the channel

  int end; std::cin >> end;       // simply wait to end application
}
```

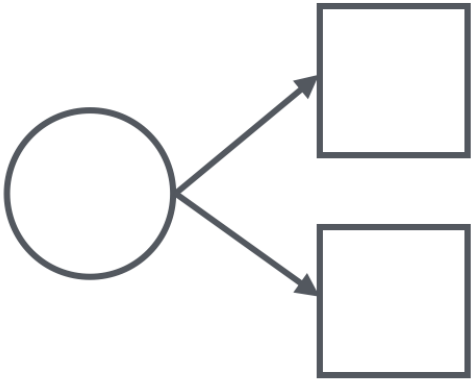
Channel - Stateless Process

```cpp
#include <stlab/channel.hpp>
#include <stlab/default_executor.hpp>
#include <iostream>
int main() {
  stlab::sender<int> send;        // sending part of the channel
  stlab::receiver<int> receiver; // receiving part of the channel
  std::tie(send, receiver) =      // combining both to a channel
    stlab::channel<int>(stlab::default_executor);

  auto printer =
    [](int x){ std::cout << x << '\n'; }; // stateless process

  auto printer_process =
    receiver | printer;           // attaching process to the receiving
                                  // part
  receiver.set_ready();           // no more processes will be attached
                                  // process starts to work
  send(1); send(2); send(3);      // start sending into the channel

  int end; std::cin >> end;       // simply wait to end application
}
```

Output

```
1
2
3
```

Channel - Split



Channel - Split Process

```
using namespace stlab;
int main() {
  sender<int> send;
  receiver<int> receiver;
  std::tie(send, receiver) = channel<int>(default_executor);

  auto printerA = [](int x){ printf("Process A %d\n", x); };
  auto printerB = [](int x){ printf("Process B %d\n", x); };

  auto printer_processA = receiver | printerA;
  auto printer_processB = receiver | printerB;

  receiver.set_ready();          // no more processes will be attached
                                 // process may start to work
  send(1); send(2); send(3);
  int end; std::cin >> end;
}
```

Channel - Split Process

```
using namespace stlab;
int main() {
  sender<int> send;
  receiver<int> receiver;
  std::tie(send, receiver) = channel<int>(default_executor);

  auto printerA = [](int x){ printf("Process A %d\n", x); };
  auto printerB = [](int x){ printf("Process B %d\n", x); };

  auto printer_processA = receiver | printerA;
  auto printer_processB = receiver | printerB;

  receiver.set_ready();          // no more processes will be attached
                                 // process may start to work
  send(1); send(2); send(3);
  int end; std::cin >> end;
}
```

Channel - Split Process

```
using namespace stlab;
int main() {
  sender<int> send;
  receiver<int> receiver;
  std::tie(send, receiver) = channel<int>(default_executor);

  auto printerA = [](int x){ printf("Process A %d\n", x); };
  auto printerB = [](int x){ printf("Process B %d\n", x); };

  auto printer_processA = receiver | printerA;
  auto printer_processB = receiver | printerB;

  receiver.set_ready();          // no more processes will be attached
                                 // process may start to work
  send(1); send(2); send(3);
  int end; std::cin >> end;
}
```

Channel - Split Process

```
using namespace stlab;
int main() {
  sender<int> send;
  receiver<int> receiver;
  std::tie(send, receiver) = channel<int>(default_executor);

  auto printerA = [](int x){ printf("Process A %d\n", x); };
  auto printerB = [](int x){ printf("Process B %d\n", x); };

  auto printer_processA = receiver | printerA;
  auto printer_processB = receiver | printerB;

  receiver.set_ready();          // no more processes will be attached
                                 // process may start to work
  send(1); send(2); send(3);
  int end; std::cin >> end;
}
```

Channel - Split Process

```
using namespace stlab;
int main() {
  sender<int> send;
  receiver<int> receiver;
  std::tie(send, receiver) = channel<int>(default_executor);

  auto printerA = [](int x){ printf("Process A %d\n", x); };
  auto printerB = [](int x){ printf("Process B %d\n", x); };

  auto printer_processA = receiver | printerA;
  auto printer_processB = receiver | printerB;

  receiver.set_ready();          // no more processes will be attached
                                 // process may start to work
  send(1); send(2); send(3);
  int end; std::cin >> end;
}
```

Channel - Split Process

```
using namespace stlab;
int main() {
  sender<int> send;
  receiver<int> receiver;
  std::tie(send, receiver) = channel<int>(default_executor);

  auto printerA = [](int x){ printf("Process A %d\n", x); };
  auto printerB = [](int x){ printf("Process B %d\n", x); };

  auto printer_processA = receiver | printerA;
  auto printer_processB = receiver | printerB;

  receiver.set_ready();          // no more processes will be attached
                                 // process may start to work
  send(1); send(2); send(3);
  int end; std::cin >> end;
}
```

Channel - Split Process

```
using namespace stlab;
int main() {
  sender<int> send;
  receiver<int> receiver;
  std::tie(send, receiver) = channel<int>(default_executor);

  auto printerA = [](int x){ printf("Process A %d\n", x); };
  auto printerB = [](int x){ printf("Process B %d\n", x); };

  auto printer_processA = receiver | printerA;
  auto printer_processB = receiver | printerB;

  receiver.set_ready();          // no more processes will be attached
                                 // process may start to work
  send(1); send(2); send(3);
  int end; std::cin >> end;
}
```

Channel - Split Process

```
using namespace stlab;
int main() {
  sender<int> send;
  receiver<int> receiver;
  std::tie(send, receiver) = channel<int>(default_executor);

  auto printerA = [](int x){ printf("Process A %d\n", x); };
  auto printerB = [](int x){ printf("Process B %d\n", x); };

  auto printer_processA = receiver | printerA;
  auto printer_processB = receiver | printerB;

  receiver.set_ready();          // no more processes will be attached
                                 // process may start to work
  send(1); send(2); send(3);
  int end; std::cin >> end;
}
```

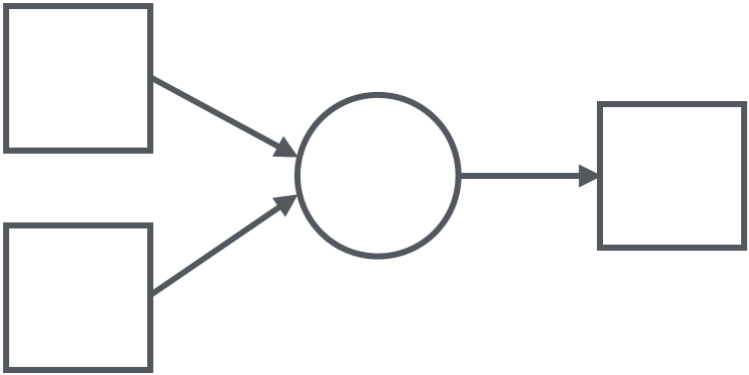
Channel - Split Process

```
using namespace stlab;
int main() {
  sender<int> send;
  receiver<int> receiver;
  std::tie(send, receiver) = channel<int>(default_executor);

  auto printerA = [](int x){ printf("Process A %d\n", x); };
  auto printerB = [](int x){ printf("Process B %d\n", x); };

  auto printer_processA = receiver | printerA;
  auto printer_processB = receiver | printerB;

  receiver.set_ready();          // no more processes will be attached
                                 // process may start to work
  send(1); send(2); send(3);
  int end; std::cin >> end;
}
```

Output

```
Process A 1
Process B 1
Process A 2
Process B 2
Process B 3
Process A 3
```

Channel - Join



Channel - Joined Processes

```
1  using namespace stlab;
2
3  int main() {
4    sender<int> sendA, sendB;
5    receiver<int> receiverA, receiverB;
6    std::tie(sendA, receiverA) = channel<int>(default_executor);
7    std::tie(sendB, receiverB) = channel<int>(default_executor);
8
9    auto printer = [](int x, int y){ printf("Process %d %d\n", x, y); };
10
11   auto printProcess = join(default_executor, printer,
12     receiverA, receiverB);
13
14   receiverA.set_ready();
15   receiverB.set_ready();
16
17   sendA(1); sendA(2); sendB(3); sendA(4); sendB(5); sendB(6);
18
19   int end; std::cin >> end;
20 }
```

Channel - Joined Processes

```
1  using namespace stlab;
2
3  int main() {
4    sender<int> sendA, sendB;
5    receiver<int> receiverA, receiverB;
6    std::tie(sendA, receiverA) = channel<int>(default_executor);
7    std::tie(sendB, receiverB) = channel<int>(default_executor);
8
9    auto printer = [](int x, int y){ printf("Process %d %d\n", x, y); };
10
11   auto printProcess = join(default_executor, printer,
12     receiverA, receiverB);
13
14   receiverA.set_ready();
15   receiverB.set_ready();
16
17   sendA(1); sendA(2); sendB(3); sendA(4); sendB(5); sendB(6);
18
19   int end; std::cin >> end;
20 }
```

Channel - Joined Processes

```
1  using namespace stlab;
2
3  int main() {
4    sender<int> sendA, sendB;
5    receiver<int> receiverA, receiverB;
6    std::tie(sendA, receiverA) = channel<int>(default_executor);
7    std::tie(sendB, receiverB) = channel<int>(default_executor);
8
9    auto printer = [](int x, int y){ printf("Process %d %d\n", x, y); };
10
11   auto printProcess = join(default_executor, printer,
12     receiverA, receiverB);
13
14   receiverA.set_ready();
15   receiverB.set_ready();
16
17   sendA(1); sendA(2); sendB(3); sendA(4); sendB(5); sendB(6);
18
19   int end; std::cin >> end;
20 }
```

Channel - Joined Processes

```
1  using namespace stlab;
2
3  int main() {
4    sender<int> sendA, sendB;
5    receiver<int> receiverA, receiverB;
6    std::tie(sendA, receiverA) = channel<int>(default_executor);
7    std::tie(sendB, receiverB) = channel<int>(default_executor);
8
9    auto printer = [](int x, int y){ printf("Process %d %d\n", x, y); };
10
11   auto printProcess = join(default_executor, printer,
12     receiverA, receiverB);
13
14   receiverA.set_ready();
15   receiverB.set_ready();
16
17   sendA(1); sendA(2); sendB(3); sendA(4); sendB(5); sendB(6);
18
19   int end; std::cin >> end;
20 }
```

Channel - Joined Processes

```
1  using namespace stlab;
2
3  int main() {
4    sender<int> sendA, sendB;
5    receiver<int> receiverA, receiverB;
6    std::tie(sendA, receiverA) = channel<int>(default_executor);
7    std::tie(sendB, receiverB) = channel<int>(default_executor);
8
9    auto printer = [](int x, int y){ printf("Process %d %d\n", x, y); };
10
11   auto printProcess = join(default_executor, printer,
12     receiverA, receiverB);
13
14   receiverA.set_ready();
15   receiverB.set_ready();
16
17   sendA(1); sendA(2); sendB(3); sendA(4); sendB(5); sendB(6);
18
19   int end; std::cin >> end;
20 }
```

Channel - Joined Processes

```
1  using namespace stlab;
2
3  int main() {
4    sender<int> sendA, sendB;
5    receiver<int> receiverA, receiverB;
6    std::tie(sendA, receiverA) = channel<int>(default_executor);
7    std::tie(sendB, receiverB) = channel<int>(default_executor);
8
9    auto printer = [](int x, int y){ printf("Process %d %d\n", x, y); };
10
11   auto printProcess = join(default_executor, printer,
12     receiverA, receiverB);
13
14   receiverA.set_ready();
15   receiverB.set_ready();
16
17   sendA(1); sendA(2); sendB(3); sendA(4); sendB(5); sendB(6);
18
19   int end; std::cin >> end;
20 }
```

Channel - Joined Processes

```
1  using namespace stlab;
2
3  int main() {
4    sender<int> sendA, sendB;
5    receiver<int> receiverA, receiverB;
6    std::tie(sendA, receiverA) = channel<int>(default_executor);
7    std::tie(sendB, receiverB) = channel<int>(default_executor);
8
9    auto printer = [](int x, int y){ printf("Process %d %d\n", x, y); };
10
11   auto printProcess = join(default_executor, printer,
12     receiverA, receiverB);
13
14   receiverA.set_ready();
15   receiverB.set_ready();
16
17   sendA(1); sendA(2); sendB(3); sendA(4); sendB(5); sendB(6);
18
19   int end; std::cin >> end;
20 }
```

Output

```
Process 1 3
Process 2 5
Process 4 6
```

Channel operators

Beside `join()` there are:

- `zip()` The process takes the passed values in a round-robin manner, starting with the result from the first receiver.
- `merge()` The process takes the values in an arbitrary order.

Exercise 2

Demo Exercise 2

Exercise 2

Create a process chain with

- the inputs
  - one `int` input
  - one `std::string` input
  - one `double` input
- all inputs are joined to a process that concatenates all the results into a string and
- the result is split into
  - one process that prints the result into console,
  - one process that stores the result into a file
- show with two value triplets, that the implementation works
- don't use any synchronization primitive

Conclusion

- Stateless processes (from the point of view of the channel) have a 1:1 relationship from input to output

Channel Stateful Process - Motivation

- Some problems need a processor with state
- Some problems have an n : m relationship from input to output
- The picture becomes more complicated with states:
  - When to proceed?
  - How to handle situations when less than expected values come downstream?

Channel - Stateful Process Signature

```cpp
#include <stlab/channel.hpp>

using process_state_scheduled =
  std::pair<process_state, std::chrono::system_clock::time_point>;

struct process_signature
{
    void await(T... val);

    U yield();

    process_state_scheduled state() const;

    void close();                        // optional

    void set_error(std::exception_ptr); // optional
};
```

Stateful Process Signature - await

```
#include <stlab/channel.hpp>

using process_state_scheduled =
  std::pair<process_state, std::chrono::system_clock::time_point>;

struct process_signature
{
    void await(T... val);

    U yield();

    process_state_scheduled state() const;

    void close();                        // optional

    void set_error(std::exception_ptr); // optional
};
```

The `await` method is called on the process whenever a new value was received from upstream. The type T stands here for any semi regular or move-only type. The number of arguments depends on the number of attached upstream sender. Potential state changes from awaitable to yieldable should happen while this method is invoked.

Stateful Process Signature - yield

```
#include <stlab/channel.hpp>

using process_state_scheduled =
  std::pair<process_state, std::chrono::system_clock::time_point>;

struct process_signature
{
    void await(T... val);

    U yield();

    process_state_scheduled state() const;

    void close();                       // optional

    void set_error(std::exception_ptr); // optional
};
```

The `yield` method is called on the process whenever the `process_state_scheduled.first` is `process_state::yield` or a timeout was provided with the recent call to `state()` and that has elapsed.

Stateful Process Signature - state

```cpp
#include <stlab/channel.hpp>

using process_state_scheduled =
  std::pair<process_state, std::chrono::system_clock::time_point>;

struct process_signature
{
    void await(T... val);

    U yield();

    process_state_scheduled state() const;

    void close();                       // optional

    void set_error(std::exception_ptr); // optional
};
```

This method must return the current state of the process. Typical return values are `await_forever` and `yield_immediate`. By explicit using the second part of the return type, one can set a possible timeout. Subsequent calls *without* an intermittent `await(), close(), or yield()` must return the same values. Otherwise the result is undefined.

Stateful Process Signature - close

```cpp
#include <stlab/channel.hpp>

using process_state_scheduled =
  std::pair<process_state, std::chrono::system_clock::time_point>;

struct process_signature
{
    void await(T... val);

    U yield();

    process_state_scheduled state() const;

    void close();                          // optional

    void set_error(std::exception_ptr); // optional
};
```

The optional `close()` method is called on the process whenever the process state is `await_forever` and the incoming queue went dry. As well it is called when an exception is thrown while calling `await()` or `yield()` and no `set_error()` is available.

Stateful Process Signature - set_error

```cpp
#include <stlab/channel.hpp>

using process_state_scheduled =
  std::pair<process_state, std::chrono::system_clock::time_point>;

struct process_signature
{
    void await(T... val);

    U yield();

    process_state_scheduled state() const;

    void close();                          // optional

    void set_error(std::exception_ptr); // optional
};
```

The method `set_error()` is optional. It is called if either on calling `await()` or `yield()` an exception was thrown. The pointer of the caught exception is passed. In case that the process does not provide this method, `close()` is called instead of.

Channel - Stateful Process Example

```cpp
#include <stlab/channel.hpp>
#include <stlab/default_executor.hpp>
#include <iostream>
using namespace stlab;

struct adder
{
};

int main() {
  sender<int> send;
  receiver<int> receiver;
  std::tie(send, receiver) = channel<int>(default_executor);

  auto calculator = receiver | adder{} |
    [](int x) { std::cout << x << '\n'; };

  receiver.set_ready();

  while (true) {
    int x;
    std::cin >> x;
    send(x);
  }
}
```

Channel - Stateful Process Example

```cpp
#include <stlab/channel.hpp>
#include <stlab/default_executor.hpp>
#include <iostream>
using namespace stlab;

struct adder
{
};

int main() {
  sender<int> send;
  receiver<int> receiver;
  std::tie(send, receiver) = channel<int>(default_executor);

  auto calculator = receiver | adder{} |
    [](int x) { std::cout << x << '\n'; };

  receiver.set_ready();

  while (true) {
    int x;
    std::cin >> x;
    send(x);
  }
}
```

Channel - Stateful Process Example

```cpp
#include <stlab/channel.hpp>
#include <stlab/default_executor.hpp>
#include <iostream>
using namespace stlab;

struct adder
{
};

int main() {
  sender<int> send;
  receiver<int> receiver;
  std::tie(send, receiver) = channel<int>(default_executor);

  auto calculator = receiver | adder{} |
    [](int x) { std::cout << x << '\n'; };

  receiver.set_ready();

  while (true) {
    int x;
    std::cin >> x;
    send(x);
  }
}
```

Channel - Stateful Process Example

```cpp
#include <stlab/channel.hpp>
#include <stlab/default_executor.hpp>
#include <iostream>
using namespace stlab;

struct adder
{
};

int main() {
  sender<int> send;
  receiver<int> receiver;
  std::tie(send, receiver) = channel<int>(default_executor);

  auto calculator = receiver | adder{} |
    [](int x) { std::cout << x << '\n'; };

  receiver.set_ready();

  while (true) {
    int x;
    std::cin >> x;
    send(x);
  }
}
```

Channel - Stateful Process Example

```cpp
#include <stlab/channel.hpp>
#include <stlab/default_executor.hpp>
#include <iostream>
using namespace stlab;

struct adder
{
};

int main() {
  sender<int> send;
  receiver<int> receiver;
  std::tie(send, receiver) = channel<int>(default_executor);

  auto calculator = receiver | adder{} |
    [](int x) { std::cout << x << '\n'; };

  receiver.set_ready();

  while (true) {
    int x;
    std::cin >> x;
    send(x);
  }
}
```

Channel - Stateful Process Example cont.

```
1  struct adder
2  {
3    int _sum = 0;
4    process_state_scheduled _state = await_forever;
5
6    void await(int x) {
7      _sum += x;
8      if (x == 0) {
9        _state = yield_immediate;
10     }
11   }
12
13   int yield() {
14     int result = _sum;
15     _sum = 0;
16     _state = await_forever;
17     return result;
18   }
19
20   auto state() const { return _state; }
21 };
22
23 int main() {
24   auto calculator = receiver | adder{} |
25     [](int x) { std::cout << x << '\n'; };
26   while (true) {
27     int x;
28     std::cin >> x;
29     send(x);
30   }
31 }
```

Channel - Stateful Process Example cont.

```
1  struct adder
2  {
3    int _sum = 0;
4    process_state_scheduled _state = await_forever;
5
6    void await(int x) {
7      _sum += x;
8      if (x == 0) {
9        _state = yield_immediate;
10     }
11   }
12
13   int yield() {
14     int result = _sum;
15     _sum = 0;
16     _state = await_forever;
17     return result;
18   }
19
20   auto state() const { return _state; }
21 };
22
23 int main() {
24   auto calculator = receiver | adder{} |
25     [](int x) { std::cout << x << '\n'; };
26   while (true) {
27     int x;
28     std::cin >> x;
29     send(x);
30   }
31 }
```

Channel - Stateful Process Example cont.

```
1  struct adder
2  {
3    int _sum = 0;
4    process_state_scheduled _state = await_forever;
5
6    void await(int x) {
7      _sum += x;
8      if (x == 0) {
9        _state = yield_immediate;
10     }
11   }
12
13   int yield() {
14     int result = _sum;
15     _sum = 0;
16     _state = await_forever;
17     return result;
18   }
19
20   auto state() const { return _state; }
21 };
22
23 int main() {
24   auto calculator = receiver | adder{} |
25     [](int x) { std::cout << x << '\n'; };
26   while (true) {
27     int x;
28     std::cin >> x;
29     send(x);
30   }
31 }
```

Channel - Stateful Process Example cont.

```
1  struct adder
2  {
3    int _sum = 0;
4    process_state_scheduled _state = await_forever;
5
6    void await(int x) {
7      _sum += x;
8      if (x == 0) {
9        _state = yield_immediate;
10     }
11   }
12
13   int yield() {
14     int result = _sum;
15     _sum = 0;
16     _state = await_forever;
17     return result;
18   }
19
20   auto state() const { return _state; }
21 };
22
23 int main() {
24   auto calculator = receiver | adder{} |
25     [](int x) { std::cout << x << '\n'; };
26   while (true) {
27     int x;
28     std::cin >> x;
29     send(x);
30   }
31 }
```

Channel - Stateful Process Example cont.

```
1  struct adder
2  {
3    int _sum = 0;
4    process_state_scheduled _state = await_forever;
5
6    void await(int x) {
7      _sum += x;
8      if (x == 0) {
9        _state = yield_immediate;
10     }
11   }
12
13   int yield() {
14     int result = _sum;
15     _sum = 0;
16     _state = await_forever;
17     return result;
18   }
19
20   auto state() const { return _state; }
21 };
22
23 int main() {
24   auto calculator = receiver | adder{} |
25     [](int x) { std::cout << x << '\n'; };
26   while (true) {
27     int x;
28     std::cin >> x;
29     send(x);
30   }
31 }
```

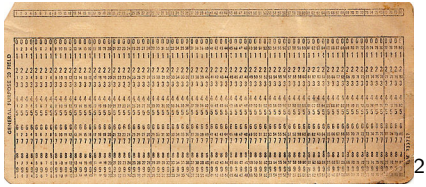
Channel - Stateful Process Example cont.

```
1  struct adder
2  {
3    int _sum = 0;
4    process_state_scheduled _state = await_forever;
5
6    void await(int x) {
7      _sum += x;
8      if (x == 0) {
9        _state = yield_immediate;
10     }
11   }
12
13   int yield() {
14     int result = _sum;
15     _sum = 0;
16     _state = await_forever;
17     return result;
18   }
19
20   auto state() const { return _state; }
21 };
22
23 int main() {
24   auto calculator = receiver | adder{} |
25     [](int x) { std::cout << x << '\n'; };
26   while (true) {
27     int x;
28     std::cin >> x;
29     send(x);
30   }
31 }
```

Exercise 3

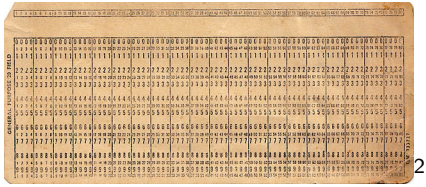
Demo Exercise 3

Exercise 3

 [2]

[2] By Ventriloquist - Own work, CC BY-SA 3.0, https://en.wikipedia.org/w/index.php?curid=32753387

Exercise 3

 [2]

A process which inputs cards of eighty characters and outputs their text, tightly packed into lines of 125 characters each.

- Write one process - unpack - that collect 80 chars in a bunch and yields them one after the other
- Write one process - pack - that packs 125 chars and yields them.
- Concatenate unpack - pack as a process chain.
- In a next step write one process - filter - that drops all newlines from the stream
- Concatenate now unpack - filter - pack as a process chain.

[2] By Ventriloquist - Own work, CC BY-SA 3.0, https://en.wikipedia.org/w/index.php?curid=32753387

Process Analysis

- Are there performance or usability problems?
- Identify the overall critical part
- Disassemble this part into individual processes
- Chain the processes with futures or channels

Use case example I

Problem within our mammography application:

Use case example I

Problem within our mammography application:

- Medical device shall open every case in < 1 s

Use case example I

Problem within our mammography application:

- Medical device shall open every case in < 1 s
- Loading of patient data and first images takes about 0.6 s

Use case example I

Problem within our mammography application:

- Medical device shall open every case in < 1 s
- Loading of patient data and first images takes about 0.6 s
- Reading of additional data structures (CAD[3]reports) may take more than 0.4 s

[3] **C**omputer **A**ided **D**etection

Use case example I

Problem within our mammography application:

- Medical device shall open every case in < 1 s
- Loading of patient data and first images takes about 0.6 s
- Reading of additional data structures (CAD[3]reports) may take more than 0.4 s
- Direct access to any CAD report might be required

[3] **C**omputer **A**ided **D**etection

Use case example I

Problem within our mammography application:

- Medical device shall open every case in < 1 s
- Loading of patient data and first images takes about 0.6 s
- Reading of additional data structures (CAD[3]reports) may take more than 0.4 s
- Direct access to any CAD report might be required
- If the user skips this case and advances to the next one, outstanding load operations should be cancelled or at least be ignored

[3] **C**omputer **A**ided **D**etection

Exercise 2

Demo Exercise 4

Exercise 4

Improve the application that the UI is always responsible

- On Reset the reports are newly read
- If one presses 1 or 2 while the reset is running, the reports shall be displayed as soon as they become available.

Next Steps?

High level concurrency sessions at ACCU 2017:

- Thinking Outside the Synchronisation Quadrant by Kevlin Henney (Wed.)
- Coroutines in Python by Robert Smallshire (Thur.)
- Coroutines in C++ by Dominic Robinson (Fr.)
- Concurrency / Coroutines by Anthony Williams (Sat.)

Synchronization Motivation

Why do we have to synchronize?

Synchronization Motivation

Why do we have to synchronize?
Because we have to ensure sequential consistency.

Synchronization Motivation

Why do we have to synchronize?
Because we have to ensure sequential consistency.
What synchronization mechanism do you know?

Synchronization Motivation

Why do we have to synchronize?
Because we have to ensure sequential consistency.
What synchronization mechanism do you know?

- *Synchronization primitives (mutex, atomic, memory fence, ...)*

Synchronization Motivation

Why do we have to synchronize?
Because we have to ensure sequential consistency.
What synchronization mechanism do you know?

- *Synchronization primitives (mutex, atomic, memory fence, ...)*
- *Guaranteed sequential access*

Synchronization with Mutex

```
1  template <typename K, typename V>
2  class registry
3  {
4    map<K, V> _data;
5    mutex     _guard;
6  public:
7    void insert(const K& key, const V& value) {
8      unique_lock<mutex> lock(_guard);
9      _data.insert(
10       make_pair("What is the answer?", 42)
11     );
12   }
13
14   V operator[](const K& key) {
15     unique_lock<mutex> lock(_guard);
16     return _data[key];
17   }
18 };
19
20 int main() {
21   registry<string, int> my_registry;
22   auto work = [&] { my_registry.insert("What is the answer?", 42); };
23   auto f1 = async(launch::async, work);
24   auto f2 = async(launch::async, work);
25   f1.get(); f2.get();
26   cout << "What is the answer? " << my_registry["What is the answer?"] << '\n';
27 }
```

Synchronization with Mutex

```
1  template <typename K, typename V>
2  class registry
3  {
4    map<K, V> _data;
5    mutex     _guard;
6  public:
7    void insert(const K& key, const V& value) {
8      unique_lock<mutex> lock(_guard);
9      _data.insert(
10       make_pair("What is the answer?", 42)
11     );
12   }
13
14   V operator[](const K& key) {
15     unique_lock<mutex> lock(_guard);
16     return _data[key];
17   }
18 };
19
20 int main() {
21   registry<string, int> my_registry;
22   auto work = [&] { my_registry.insert("What is the answer?", 42); };
23   auto f1 = async(launch::async, work);
24   auto f2 = async(launch::async, work);
25   f1.get(); f2.get();
26   cout << "What is the answer? " << my_registry["What is the answer?"] << '\n';
27 }
```

Synchronization with Mutex

```
1  template <typename K, typename V>
2  class registry
3  {
4    map<K, V> _data;
5    mutex     _guard;
6  public:
7    void insert(const K& key, const V& value) {
8      unique_lock<mutex> lock(_guard);
9      _data.insert(
10       make_pair("What is the answer?", 42)
11     );
12   }
13
14   V operator[](const K& key) {
15     unique_lock<mutex> lock(_guard);
16     return _data[key];
17   }
18 };
19
20 int main() {
21   registry<string, int> my_registry;
22   auto work = [&] { my_registry.insert("What is the answer?", 42); };
23   auto f1 = async(launch::async, work);
24   auto f2 = async(launch::async, work);
25   f1.get(); f2.get();
26   cout << "What is the answer? " << my_registry["What is the answer?"] << '\n';
27 }
```

Synchronization with Mutex

```
1  template <typename K, typename V>
2  class registry
3  {
4    map<K, V> _data;
5    mutex     _guard;
6  public:
7    void insert(const K& key, const V& value) {
8      unique_lock<mutex> lock(_guard);
9      _data.insert(
10       make_pair("What is the answer?", 42)
11     );
12   }
13
14   V operator[](const K& key) {
15     unique_lock<mutex> lock(_guard);
16     return _data[key];
17   }
18 };
19
20 int main() {
21   registry<string, int> my_registry;
22   auto work = [&] { my_registry.insert("What is the answer?", 42); };
23   auto f1 = async(launch::async, work);
24   auto f2 = async(launch::async, work);
25   f1.get(); f2.get();
26   cout << "What is the answer? " << my_registry["What is the answer?"] << '\n';
27 }
```

Synchronization with Mutex

```
1  template <typename K, typename V>
2  class registry
3  {
4    map<K, V> _data;
5    mutex     _guard;
6  public:
7    void insert(const K& key, const V& value) {
8      unique_lock<mutex> lock(_guard);
9      _data.insert(
10       make_pair("What is the answer?", 42)
11     );
12   }
13
14   V operator[](const K& key) {
15     unique_lock<mutex> lock(_guard);
16     return _data[key];
17   }
18 };
19
20 int main() {
21   registry<string, int> my_registry;
22   auto work = [&] { my_registry.insert("What is the answer?", 42); };
23   auto f1 = async(launch::async, work);
24   auto f2 = async(launch::async, work);
25   f1.get(); f2.get();
26   cout << "What is the answer? " << my_registry["What is the answer?"] << '\n';
27 }
```

Synchronization with Mutex

```
1  template <typename K, typename V>
2  class registry
3  {
4    map<K, V> _data;
5    mutex     _guard;
6  public:
7    void insert(const K& key, const V& value) {
8      unique_lock<mutex> lock(_guard);
9      _data.insert(
10       make_pair("What is the answer?", 42)
11     );
12   }
13
14   V operator[](const K& key) {
15     unique_lock<mutex> lock(_guard);
16     return _data[key];
17   }
18 };
19
20 int main() {
21   registry<string, int> my_registry;
22   auto work = [&] { my_registry.insert("What is the answer?", 42); };
23   auto f1 = async(launch::async, work);
24   auto f2 = async(launch::async, work);
25   f1.get(); f2.get();
26   cout << "What is the answer? " << my_registry["What is the answer?"] << '\n';
27 }
```

Synchronization with Mutex

```
1  template <typename K, typename V>
2  class registry
3  {
4    map<K, V> _data;
5    mutex     _guard;
6  public:
7    void insert(const K& key, const V& value) {
8      unique_lock<mutex> lock(_guard);
9      _data.insert(
10       make_pair("What is the answer?", 42)
11     );
12   }
13
14   V operator[](const K& key) {
15     unique_lock<mutex> lock(_guard);
16     return _data[key];
17   }
18 };
19
20 int main() {
21   registry<string, int> my_registry;
22   auto work = [&] { my_registry.insert("What is the answer?", 42); };
23   auto f1 = async(launch::async, work);
24   auto f2 = async(launch::async, work);
25   f1.get(); f2.get();
26   cout << "What is the answer? " << my_registry["What is the answer?"] << '\n';
27 }
```

Synchronization with Mutex

```
1  template <typename K, typename V>
2  class registry
3  {
4    map<K, V> _data;
5    mutex     _guard;
6  public:
7    void insert(const K& key, const V& value) {
8      unique_lock<mutex> lock(_guard);
9      _data.insert(
10       make_pair("What is the answer?", 42)
11     );
12   }
13
14   V operator[](const K& key) {
15     unique_lock<mutex> lock(_guard);
16     return _data[key];
17   }
18 };
19
20 int main() {
21   registry<string, int> my_registry;
22   auto work = [&] { my_registry.insert("What is the answer?", 42); };
23   auto f1 = async(launch::async, work);
24   auto f2 = async(launch::async, work);
25   f1.get(); f2.get();
26   cout << "What is the answer? " << my_registry["What is the answer?"] << '\n';
27 }
```

Where are the problems in the code?

Synchronization with Mutex

```
1  template <typename K, typename V>
2  class registry
3  {
4    map<K, V> _data;
5    mutex     _guard;
6  public:
7    void insert(const K& key, const V& value) {
8      unique_lock<mutex> lock(_guard);
9      _data.insert(
10       make_pair("What is the answer?", 42)
11     );
12   }
13
14   V operator[](const K& key) {
15     unique_lock<mutex> lock(_guard);
16     return _data[key];
17   }
18 };
19
20 int main() {
21   registry<string, int> my_registry;
22   auto work = [&] { my_registry.insert("What is the answer?", 42); };
23   auto f1 = async(launch::async, work);
24   auto f2 = async(launch::async, work);
25   f1.get(); f2.get();
26   cout << "What is the answer? " << my_registry["What is the answer?"] << '\n';
27 }
```

Where are the problems in the code?

Synchronization with Mutex

```
1  template <typename K, typename V>
2  class registry
3  {
4    map<K, V> _data;
5    mutex     _guard;
6  public:
7    void insert(const K& key, const V& value) {
8      unique_lock<mutex> lock(_guard);
9      _data.insert(
10       make_pair("What is the answer?", 42)
11     );
12   }
13
14   V operator[](const K& key) {
15     unique_lock<mutex> lock(_guard);
16     return _data[key];
17   }
18 };
19
20 int main() {
21   registry<string, int> my_registry;
22   auto work = [&] { my_registry.insert("What is the answer?", 42); };
23   auto f1 = async(launch::async, work);
24   auto f2 = async(launch::async, work);
25   f1.get(); f2.get();
26   cout << "What is the answer? " << my_registry["What is the answer?"] << '\n';
27 }
```

Where are the problems in the code?

Synchronization with Mutex

Mutex - What would be a better name for it?

Synchronization with Mutex

Mutex - What would be a better name for it?

Bottleneck![4]

[4] Kevlin Henney, NDC London 2017

Synchronization without Mutex

How can the code be transformed into something without a mutex in the client code?

Synchronization without Mutex

How can the code be transformed into something without a mutex in the client code?
What is needed to perform that transformation? Which tools do we have in our tool box?

Synchronization without Mutex

```
1  template <typename K, typename V>
2  class registry
3  {
4    std::shared_ptr<map<K, V>>  _data;
5    serial_queue                _queue;
6  public:
7    void insert(K key, V val) {
8      _queue.async([_d = _data,
9                    _key = std::move(key),
10                   _val = std::move(val)] {
11       d->emplace(std::move(key), std::move(val));
12     });
13   }
14
15   future<V> operator[](K key) {
16     return _queue.async([_d = _data,
17                          _key = std::move(key)] {
18       return _d->at(key);
19     });
20   }
21 };
```

Synchronization without Mutex

```
1  template <typename K, typename V>
2  class registry
3  {
4    std::shared_ptr<map<K, V>>  _data;
5    serial_queue                _queue;
6  public:
7    void insert(K key, V val) {
8      _queue.async([_d = _data,
9                    _key = std::move(key),
10                   _val = std::move(val)] {
11       d->emplace(std::move(key), std::move(val));
12     });
13   }
14
15   future<V> operator[](K key) {
16     return _queue.async([_d = _data,
17                          _key = std::move(key)] {
18       return _d->at(key);
19     });
20   }
21 };
```

Synchronization without Mutex

```
1  template <typename K, typename V>
2  class registry
3  {
4    std::shared_ptr<map<K, V>>  _data;
5    serial_queue                _queue;
6  public:
7    void insert(K key, V val) {
8      _queue.async([_d = _data,
9                    _key = std::move(key),
10                   _val = std::move(val)] {
11       d->emplace(std::move(key), std::move(val));
12     });
13   }
14
15   future<V> operator[](K key) {
16     return _queue.async([_d = _data,
17                          _key = std::move(key)] {
18       return _d->at(key);
19     });
20   }
21 };
```

Synchronization without Mutex

```
1  template <typename K, typename V>
2  class registry
3  {
4    std::shared_ptr<map<K, V>>  _data;
5    serial_queue                _queue;
6  public:
7    void insert(K key, V val) {
8      _queue.async([_d = _data,
9                    _key = std::move(key),
10                   _val = std::move(val)] {
11       d->emplace(std::move(key), std::move(val));
12     });
13   }
14
15   future<V> operator[](K key) {
16     return _queue.async([_d = _data,
17                          _key = std::move(key)] {
18       return _d->at(key);
19     });
20   }
21 };
```

Synchronization without Mutex

```
1  template <typename K, typename V>
2  class registry
3  {
4    std::shared_ptr<map<K, V>>  _data;
5    serial_queue                _queue;
6  public:
7    void insert(K key, V val) {
8      _queue.async([_d = _data,
9                    _key = std::move(key),
10                   _val = std::move(val)] {
11       d->emplace(std::move(key), std::move(val));
12     });
13   }
14
15   future<V> operator[](K key) {
16     return _queue.async([_d = _data,
17                          _key = std::move(key)] {
18       return _d->at(key);
19     });
20   }
21 };
```

Synchronization without Mutex

```
1  template <typename K, typename V>
2  class registry
3  {
4    std::shared_ptr<map<K, V>>  _data;
5    serial_queue                _queue;
6  public:
7    void insert(K key, V val) {
8      _queue.async([_d = _data,
9                    _key = std::move(key),
10                   _val = std::move(val)] {
11       d->emplace(std::move(key), std::move(val));
12     });
13   }
14
15   future<V> operator[](K key) {
16     return _queue.async([_d = _data,
17                          _key = std::move(key)] {
18       return _d->at(key);
19     });
20   }
21 };
```

Synchronization Epilogue

So we try to avoid mutexes wherever it is possible.

Synchronization Epilogue

All computer wait at the same speed

Use case example II

Image Preparation Pipeline

- A medial device shall display multi-frame image data sets
- Each incoming data set is JPEG 2000 compressed
- The slices must be decompressed and then compressed in FELICS[5] format for fast decompression and display
- Reading and writing to disk takes a reasonable amount of time

[5] Special compression algorithm for 16bit grayscale images

Reference

- Concurrency library `https://github.com/stlab/libraries`
- Documentation `http://www.stlab.cc/libraries`
- Communicating Sequential Processes by C. A. R. Hoare `http://usingcsp.com/cspbook.pdf`
- The Theory and Practice of Concurrency by A.W. Roscoe`http://www.cs.ox.ac.uk/people/bill.roscoe/publications/68b.pdf`

Further reading I

Software Principles and Algorithms

- Elements of Programming by Alexander Stepanov, Paul McJones, Addison Wesley
- From Mathematics to Generic Programming by Alexander Stepanov, Daniel Rose, Addison Wesley

Further reading II

Concurrency and Parallelism

- HPX `http://stellar-group.org/libraries/hpx/`
- C++CSP `https://www.cs.kent.ac.uk/projects/ofa/c++csp`
- CAF_C++ Actor Framework `http://actor-framework.org/`
- C++ Concurrency In Action by Anthony Williams, Manning (2nd edition coming soon)

Further viewing

- Goals for better code by Sean Parent: `http://sean-parent.stlab.cc/papers-and-presentations`
- Goals for better code by Sean Parent: Concurrency: `https://youtu.be/au0xX4h8SCI?t=16354`
- Thinking Outside the Synchronization Quadrant by Kevlin Henney: `https://vimeo.com/205806162`

Acknowledgement

- My family, who gave me the freedom to develop over months the library, prepare this tutorial and let me travel to the ACCU.
- Sean Parent, who taught me over time lots about concurrency and abstraction. He gave me the permission to use whatever I needed from his presentations for my own.
- My company MeVis Medical Solutions AG, that released me from work during the ACCU.

Contact

- Mail: felix@petriconi.net
- Web: `https://petriconi.net`
- Twitter: @FelixPetriconi