Department I - C Plus Plus

Modern and Lucid C++ Advanced
for Professional Programmers

Week 10 – Networking and Threads

Prof. Peter Sommerlad / Thomas Corbat

Rapperswil, 25.04.2019

FS2019

C++ Cevelop
Your C++ deserves it

HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL
FHO Fachhochschule Ostschweiz

IFS INSTITUTE FOR SOFTWARE

Recap Week 9

- **Create a tag type for the unit**

```cpp
struct Kph;
struct Mph;
struct Mps;
```

- **Create a quantity type template for speed**

```cpp
template <typename Unit>
struct Speed {
  constexpr explicit Speed(double value)
    : value{value}{};
  constexpr explicit operator double() const {
    return value;
  }
private:
  double value;
};
```

● **Add a `speedCast` function**

```cpp
template<typename Target, typename Source>
constexpr Speed<Target> speedCast(Speed<Source> const & source) {
  return Speed<Target>{ ConversionTraits<Target, Source>::convert(source) };
}
```

● **Create a `ConversionTraits` class template**

```cpp
template<typename Target, typename Source>
struct ConversionTraits {
  constexpr static Speed<Target> convert(Speed<Source> sourceValue) = delete;
};
```

```cpp
namespace velocity::literals {

constexpr inline Speed<Kph> operator"" _kph(unsigned long long value) {
  return Speed<Kph>{safeToDouble(value)};
}

constexpr inline Speed<Kph> operator"" _kph(long double value) {
  return Speed<Kph>{safeToDouble(value)};
}

//...

}
```

- **You know the basics of communication with sockets**

- **You can implement applications that communicate over a network**

- **You can implement network IO with synchronous and asynchronous operation**

- **You get familiar with the C++ thread API**

Networking in C++ (With ASIO)

- **Sockets are an abstraction of endpoints for communication**

- **TCP Sockets**

  - Reliable

  - Stream-oriented

  - Requires connection setup

- **UDP Sockets**

  - Packets might get lost or arrive out of order

  - Datagram-oriented (max 65k)

  - Sending/receiving possible without a connection

Primitive	Meaning
Socket	Create a new communication end point
Bind	Attach a local address to a socket
Listen	Announce willingness to accept connections
Accept	Block caller until a connection request arrives
Connect	Actively attempt to establish a connection
Send	Send some data over the connection
Receive	Receive some data over the connection
Close	Release the connection

- **Server**

| Socket | → | Bind | → | Listen | → | Accept | → | Read | → | Write | → | Close |

- **Client**

| Socket | → | Connect | → | Write | → | Read | → | Close |

Distributed Systems: Principles and Paradigms, 2nd Edition Andrew S. Tanenbaum
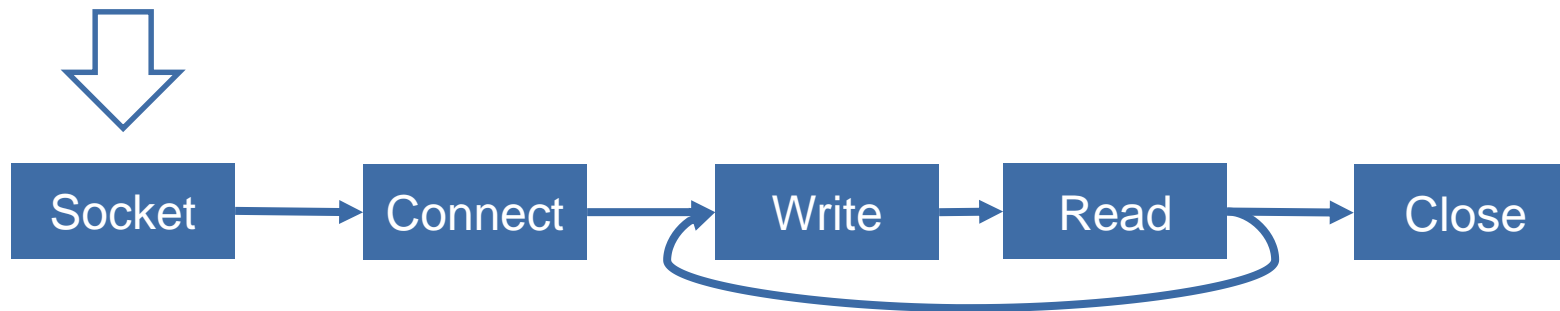
- **All ASIO operations require an I/O context (more on that later)**

- **Create a TCP socket using the context**

```
asio::io_context context{};

asio::ip::tcp::socket socket{context};
```

Socket → Connect → Write → Read → Close

- **A resolver resolves host names to end points (for TCP IP address and port)**

- **`asio::connect()` tries to establish a connection to given end point(s)**

  - It tries `connect()` of the socket for each endpoint to establish a connection

```cpp
asio::ip::tcp::resolver resolver{context};
auto endpoints = resolver.resolve(domain, "80");

asio::connect(socket, endpoints);
```

```
Socket → Connect → Write → Read → Close
                    ↑_____|
```
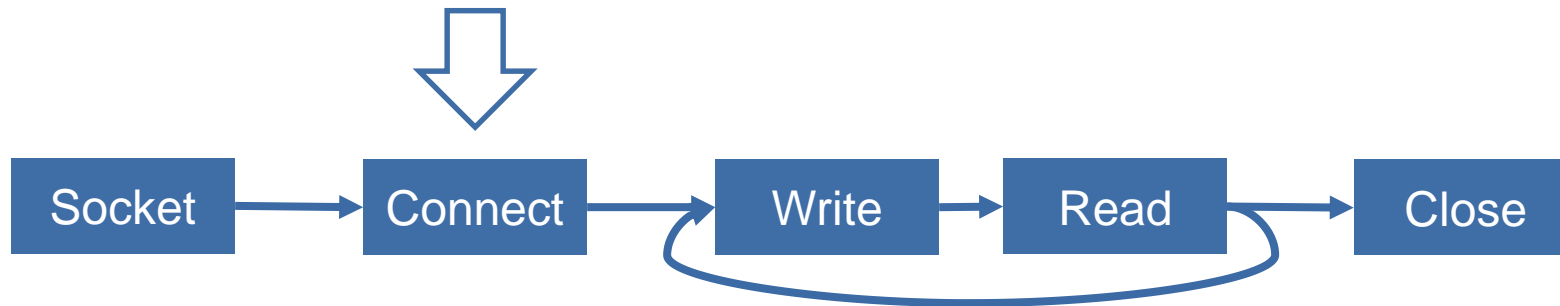
- **`asio::write()` sends the data to the peer the socket is connected to**

  - ▪ It returns when all data is sent or an error occurred (exception: `asio::system_error`)

```cpp
std::ostringstream request{};
request << "GET / HTTP/1.1\r\n";
request << "Host: " << domain << "\r\n";
request << "\r\n";

asio::write(socket, asio::buffer(request.str()));
```

⬇

| Socket | → | Connect | → | Write | → | Read | → | Close |
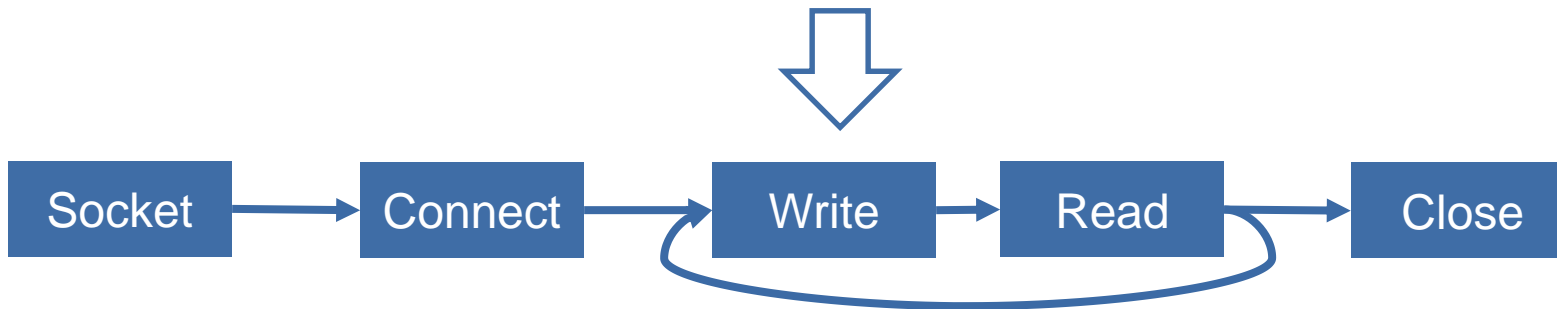
- **`asio::read()` receives data sent by the peer the socket is connected to**

  - It returns when all data is sent or an error occurred

- **The error code is set if a problem occurs or the stream has been closed (`asio::error::`*eof*)**

```cpp
constexpr size_t bufferSize = 1024;
std::array<char, bufferSize> reply{};
asio::error_code errorCode{};
auto readLength = asio::read(socket, asio::buffer(reply.data(), bufferSize), errorCode);
```

Socket → Connect → Write → Read → Close
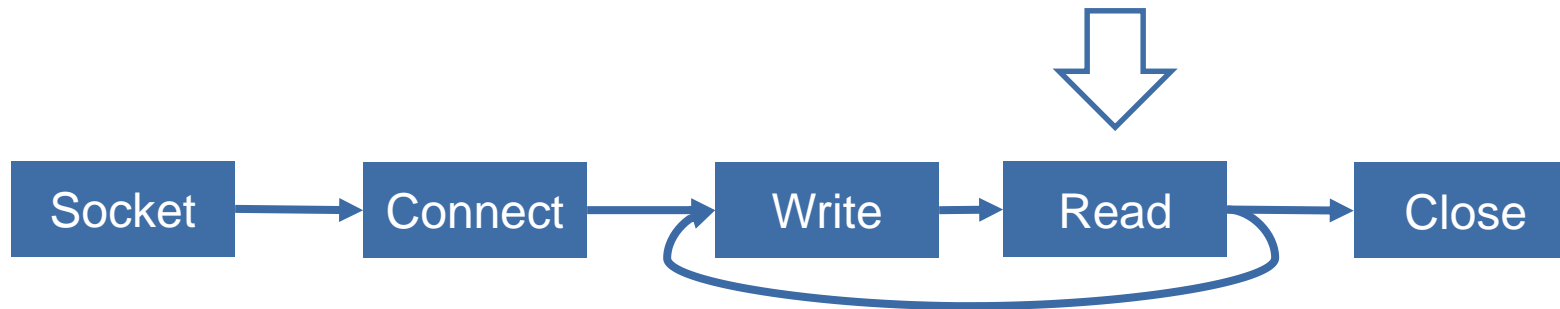
- **`shutdown()` closes the read/write stream associated with the socket, indicating to the peer that no more data will be received/sent**

- **The destructor of the socket cancels all pending operations and destroys the object**

```
socket.shutdown(asio::ip::tcp::socket::shutdown_both);

socket.close();
```
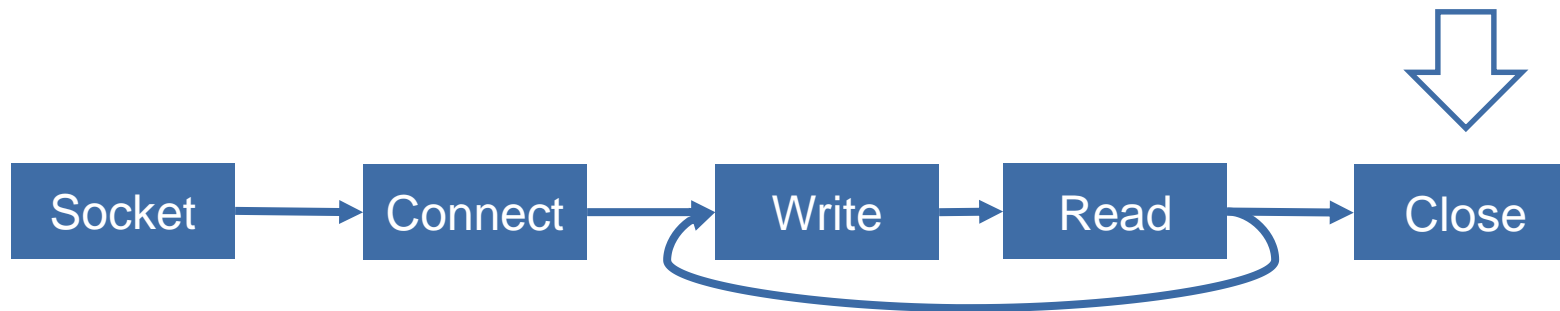
- **An acceptor is a special socket responsible for establishing incoming connections**

- **In ASIO the acceptor is bound to a given local end point and starts listening automatically**

```
asio::io_context context{};
asio::ip::tcp::endpoint localEndpoint{asio::ip::tcp::v4(), port};
asio::ip::tcp::acceptor acceptor{context, localEndpoint};
```

| Socket | → | Bind | → | Listen | → | Accept | ⋮ | Read | → | Write | → | Close |
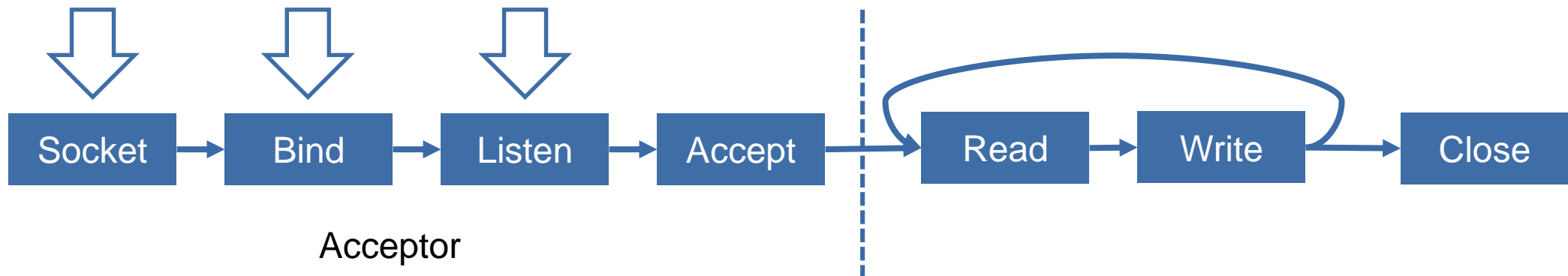
Acceptor

- **The `accept()` member function blocks until a client tries to establish a connection (with connect)**

- **It returns a new socket through which the connected client can be reached**

```
asio::ip::tcp::endpoint peerEndpoint{};
asio::ip::tcp::socket peerSocket = acceptor.accept(peerEndpoint);
```


Socket → Bind → Listen → Accept → Read → Write → Close

Acceptor   Socket
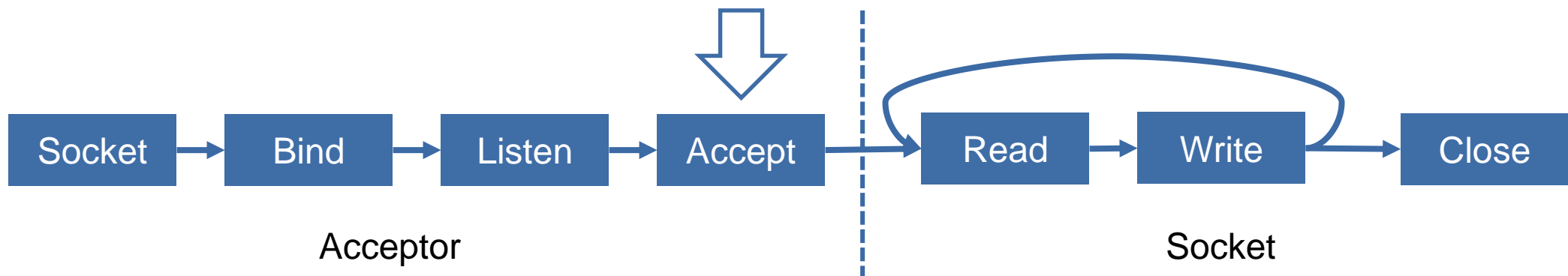
- **Using synchronous operations blocks the current thread**

- **No other operations can be executed while being blocked**

- **Asynchronous operations allow further processing of other requests while the async operation is executed (being blocked or making progress)**

- **Most operating systems support asynchronous IO operations**

BlockingOperation()

Result

AsyncOperation()

doOtherStuff()

Result
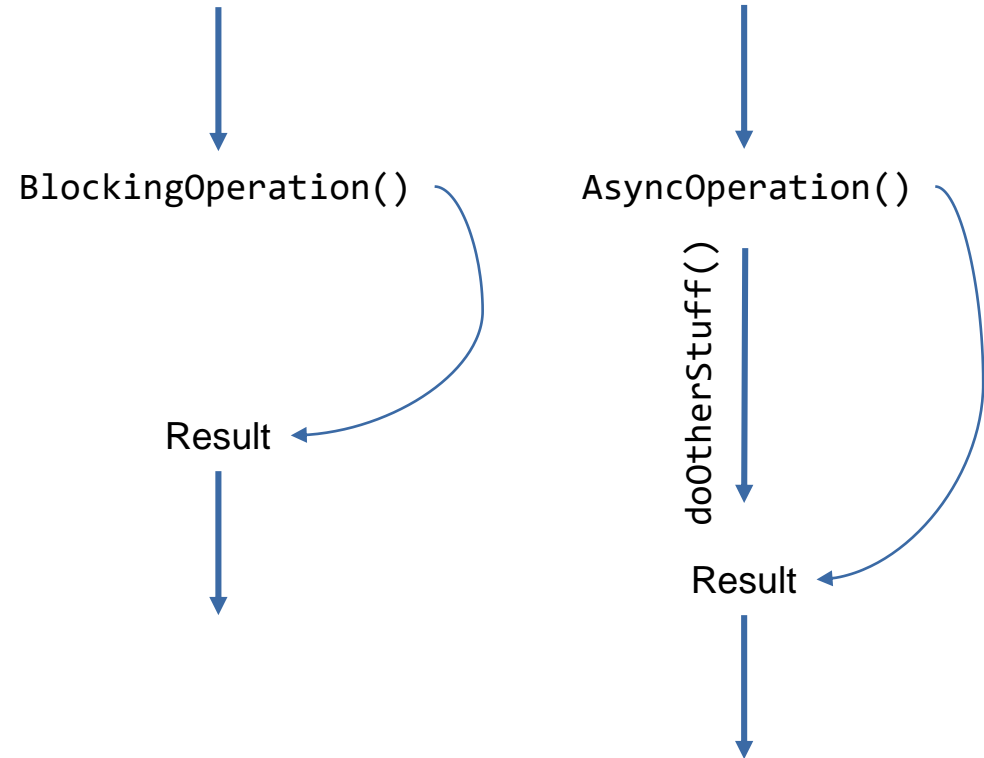
1. **The program invokes an async operation on an I/O object and passes a completion handler as a callback**

2. **The I/O object delegates the operation and the callback to its `io_context`**

3. **The operating system performs the asynchronous operation**

https://think-async.com/Asio/asio-1.12.2/doc/asio/overview/core/basics.html
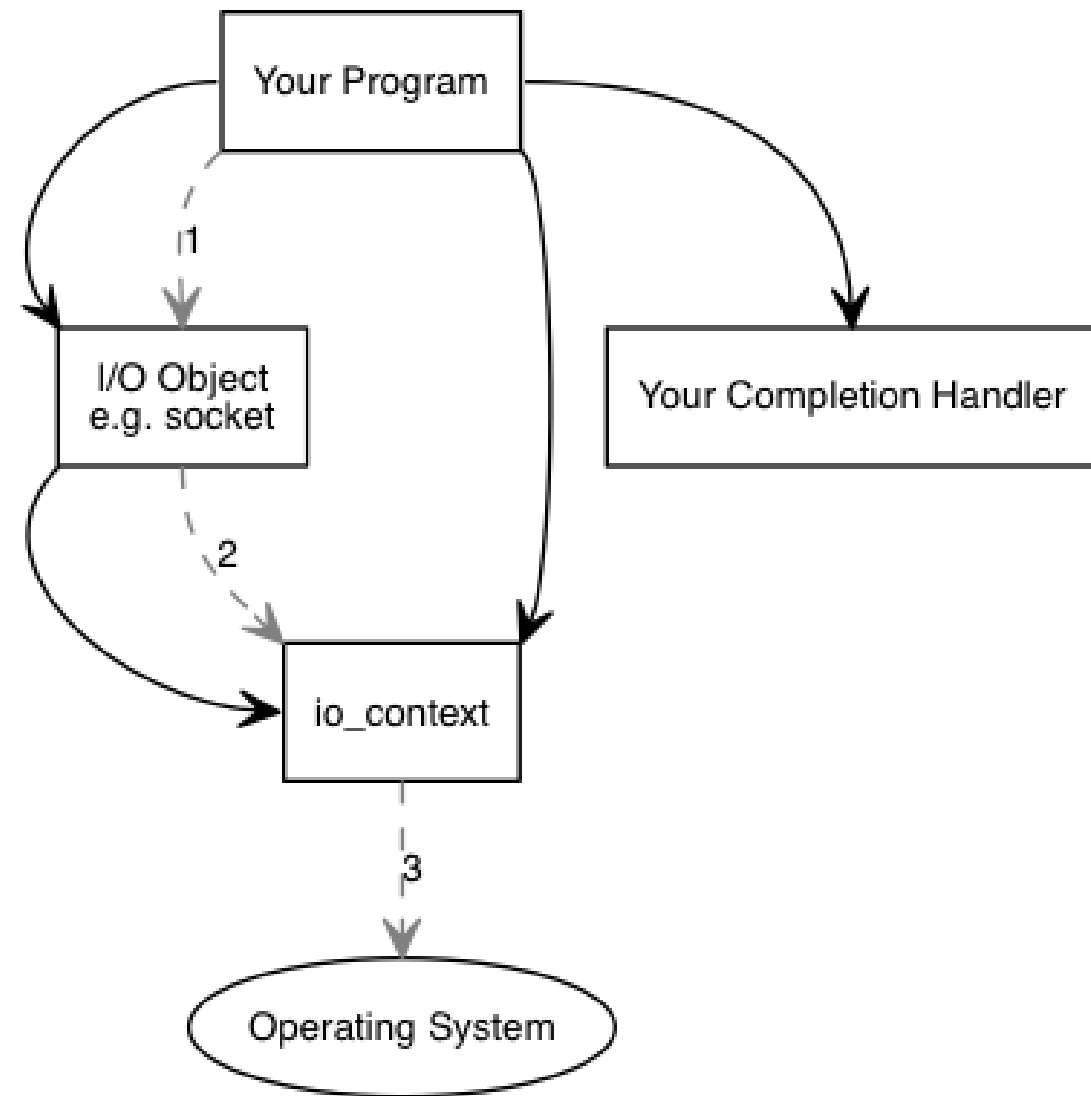
4. **The operating system signals the `io_context` that the operation has been completed**

5. **When the program calls `io_context::run()` the remaining asynchronous operations are performed (wait for the result of the operating system)**

6. **Still inside `io_context::run()` the completion handler is called to handle the result (or error) of the asynchronous operation**
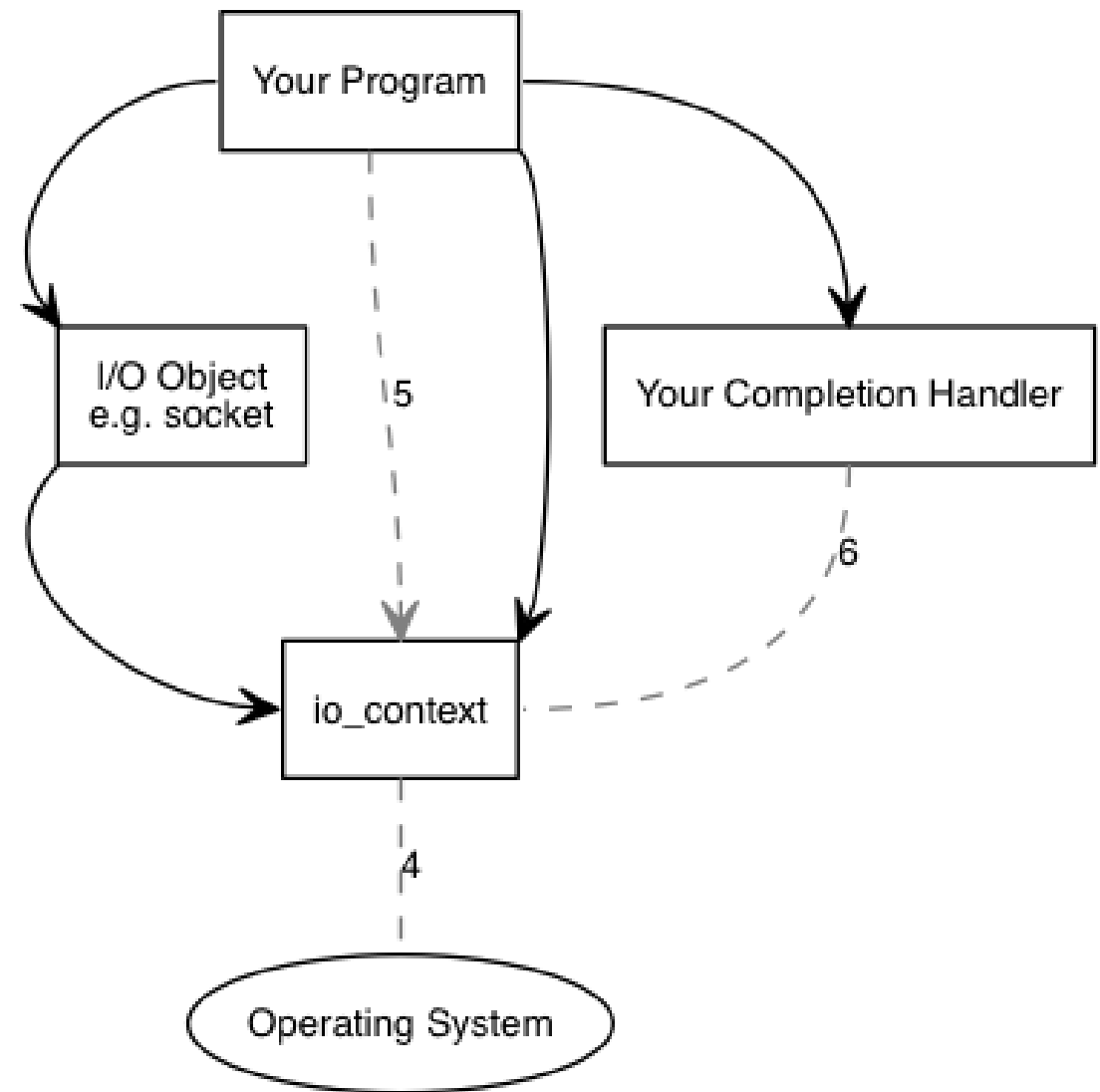
- **Async read operations**

  - ◼ `asio::async_read`

  - ◼ `asio::async_read_until`

  - ◼ `asio::async_read_at`

- **Async write operations**

  - ◼ `asio::async_write`

  - ◼ `asio::async_write_at`

- **They return immediately**

- **The operations is processed by the executer associated with the stream's `asio::io_context`**

- **A completion handler is called when the operation is done**

```
        async call
    <──────────────────>   ┌──────────────┐
                           │              │
                           │  io_context  │
                           │              │──────────────────>
                           └──────────────┘    completion
```

- **`asio::async_read_until` (the call returns immediately)**

  - ◼ Reads from asynchronous stream

  - ◼ Into a buffer

  - ◼ Until a specific character is encountered

  - ◼ Then it calls the completion handler

- **The completion handler is a callable taking an `asio::error_code` and a `std::size_t` as arguments**

```cpp
auto readCompletionHandler = [] (asio::error_code ec, std::size_t length) {
//...
};

asio::async_read_until(socket, buffer, '\n', readCompletionHandler);
```

- **`asio::async_write` (the call returns immediately)**

  - Writes to an asynchronous stream

  - The data from a buffer

  - Until all data has been written or an error occurs

  - Then it calls the completion handler

- **The completion handler is a callable taking an `asio::error_code` and a `std::size_t` as arguments**

```cpp
auto writeCompletionHandler = [] (asio::error_code ec, std::size_t length) {
//...
};

asio::async_write(socket, buffer, writeCompletionHandler);
```

```cpp
struct Server {
  using tcp = asio::ip::tcp;
  Server(asio::io_context & context, unsigned short port)
      : acceptor{context, tcp::endpoint{tcp::v4(), port}}{
    accept();
  }

private:
  void accept() {
    auto acceptHandler = [this] (asio::error_code ec, tcp::socket peer) {
      if (!ec) {
        auto session = std::make_shared<Session>(std::move(peer));
        session->start();
      }
      accept();
    };
    acceptor.async_accept(acceptHandler);
  }
  tcp::acceptor acceptor;
};
```

```cpp
void accept() {
  auto acceptHandler = [this] (asio::error_code ec, tcp::socket peer) {
    if (!ec) {
      auto session = std::make_shared<Session>(std::move(peer));
      session->start();
    }
    accept();
  };
  acceptor.async_accept(acceptHandler);
}
```

● **Creates an accept handler that is called when an incoming connection has been established**

  ▪ The second parameter is the socket of the newly connected client

  ▪ A Session object is created (on the heap) to handle all communication with the client

  ▪ `accept()` is called to continue accepting new inbound connection attempts

● **The accept handler is registered to handle the next accept asynchronously**

```cpp
Server(asio::io_context & context, unsigned short port)
    : acceptor{context, tcp::endpoint{tcp::v4(), port}}{
  accept();
}
```

- **The constructor creates the server**

- **It initializes its acceptor with the given io_context and port**

- **It calls accept for registering the accept handler for the next incoming connection attempt**

  - This function does not block

```cpp
int main() {
    asio::io_context context{};
    Server server{context, 1234};
    context.run();
}
```

- **Create an `io_context`**

  - ◼ It has an associated executor that handles the asynchronous calls

- **Create the server on port `1234`**

- **Run the executor of the `io_context` until no async operation is left**

  - ◼ Since we already have an `async_accept` request pending this operation does not return immediately

  - ◼ We will keep the this `run()` call busy

- **It is important that the `server` object lives as long as async operations on it are processed**

- **Constructor**

  - Stores the socket with the client connection

- **start() initiates the first async read**

- **read() invokes async reading**

- **write() invokes async writing**

  - Called by the handler in read

- **The fields store the data of the session**

- **Why enable_shared_from_this?**

```cpp
struct Session
    : std::enable_shared_from_this<Session> {
  explicit Session(asio::ip::tcp::socket socket);
  void start() {
    read();
  }

private:
  void read();
  void write(std::string data);

  asio::streambuf buffer{};
  std::istream input{&buffer};
  asio::ip::tcp::socket socket;
};
```

- **The session object would die at the end of the accept handler**

  - Thus it needs to be allocated on the heap

```cpp
//In the accept handler
if (!ec) {
   auto session = std::make_shared<Session>(std::move(peer));
   session->start();
}
```

- **The handlers need to keep the object alive**

```cpp
//In the accept handler
void Session::read() {
   auto handler = [self = shared_from_this()](error_code ec, size_t length) {
   //...
```

```
auto session = std::make_shared<Session>(std::move(peer));
session->start();
```

```
void Session::read() {
  auto readCompletionHandler = [self = shared_from_this()]
```

```
void Session::write(std::string input) {
  auto data = std::make_shared<std::string>(input);
  auto writeCompletionHandler = [self = shared_from_this(), data]
```
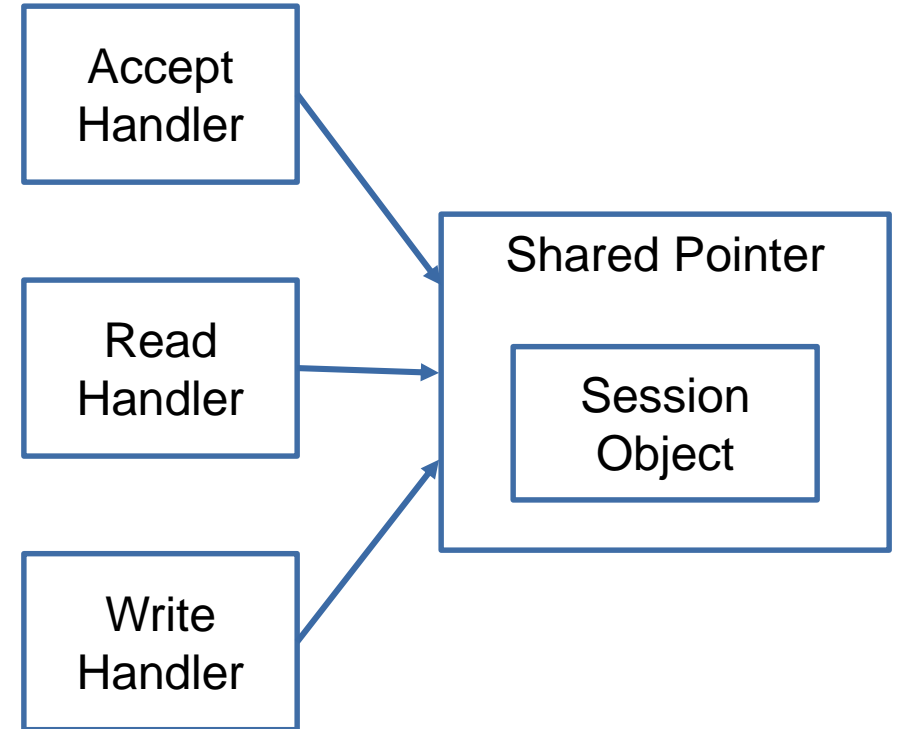
Accept Handler

Read Handler

Write Handler

Shared Pointer

Session Object

```cpp
void WebServer::accept() {
  auto acceptHandler = [this] (asio::error_code ec, tcp::socket peer) {
    if (!ec) {
      auto session = std::make_shared<Session>(std::move(peer));
      session->start();
    }
  };
  acceptor.async_accept(acceptHandler);
}
```

```cpp
void WebServer::accept() {
  auto acceptHandler = [this] (asio::error_code ec, tcp::socket peer) {
    if (!ec) {
      auto session = std::make_shared<Session>(std::move(peer));
      session->start();
    }
  };
  acceptor.async_accept(acceptHandler);
}
```

- **accept() should very likely be called at the end of the handler**

- **Otherwise only a single connection from a client will be possible**

```cpp
void Session::read() {
  auto readCompletionHandler = [this] (asio::error_code ec, std::size_t length) {
    if (ec) {
      //error handling
    }
    int number{};
    if (input >> number) {
      input.ignore(std::numeric_limits<std::streamsize>::max(), '\n');
      write(createReply(number));
    }
  };
  asio::async_read_until(socket, buffer, '\n', readCompletionHandler);
}
```

```cpp
void Session::read() {
  auto readCompletionHandler = [self = shared_from_this()] (asio::error_code ec, std::size_t length) {
  if (ec) {
    //error handling
  }
  int number{};
  if (self->input >> number) {
    self->input.ignore(std::numeric_limits<std::streamsize>::max(), '\n');
    self->write(self->createReply(number));
  }
  };
  asio::async_read_until(socket, buffer, '\n', readCompletionHandler);
}
```

- **It is likely that the shared pointer to the this object needs to be captured unless the session is stored somewhere else**

- **It needs to live longer than the operations performed on it**

Multi-Threading

- **You might find a lot of legacy C++ code using "pthreads" (aka POSIX-Threads) API**

  - There is no portability guarantee

  - Your compiler must know about your use of "pthreads" to generate safe code

- **C++11 removed the need to rely on POSIX-Threads**

  - Not 100% functionality equivalent for all tasks, so some implementations still use POSIX (or even Microsoft) Threads

  - BUT, it is guaranteed to be portable across platforms and compilers

  - BECAUSE
    C++11 and later define a Memory Model for the execution with concurrent execution agents (aka threads)

- **The slides distinguish `std::thread` (C++ class) and thread (OS execution agent)**

API of `std::thread`

```
class std::thread
```
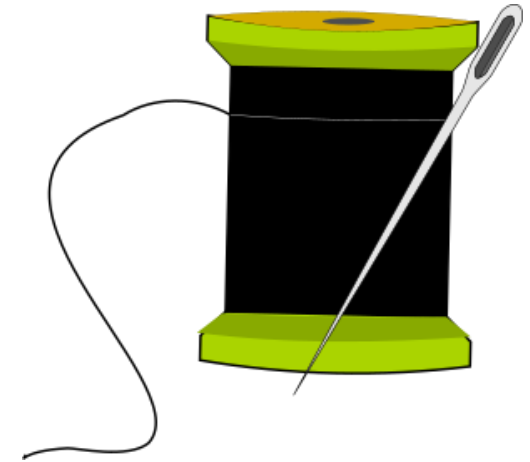
```cpp
int main() {
  std::thread greeter {
    [] { std::cout << "Hello, I'm thread!" << std::endl; }
  };
  greeter.join();
}
```

- **A new thread is created and started automatically**

- **Creates a new execution context (thread)**

- **`join()` waits for the thread to finish**

- **Besides lambdas also functions or functor objects can be executed in a thread**

- **Calls the given "function" in that thread**

- **Return value of the function is ignored**

- **Threads are default-constructible and moveable (construction and assignment)**

```cpp
struct Functor {
  void operator()() const {
    std::cout << "Functor" << std::endl;
  }
};

void function() {
  std::cout << "Function" << std::endl;
}

int main() {
  std::thread functionThread{function};
  std::thread functorThread{Functor{}};

  functorThread.join();
  functionThread.join();
}
```

```cpp
template<class Function, class... Args>
explicit thread(Function&& f, Args&&...args);
```

- **std::thread constructor takes a function/functor/lambda and arguments to forward**

- **You should pass all arguments by value to avoid data races and dangling references (if possible)**

- **Capturing by reference in lambdas creates shared data as well!**

```cpp
std::size_t fibonacci(std::size_t n) {
  if (n < 2) {
    return n;
  }
  return fibonacci(n - 1) + fibonacci(n - 2);
}

void printFib(std::size_t n) {
  auto fib = fibonacci(n);
  std::cout << "fib(" << n << ") is "
            << fib << '\n';

}

int main() {
  std::thread function { printFib, 46 };
  std::cout << "waiting..." << std::endl;
  function.join();
}
```

● **Guess what happens**

```cpp
int main() {
  std::thread lambda {
    [] {std::cout << "Lambda" << std::endl; }
  };
  std::cout << "Main" << std::endl;
}
```
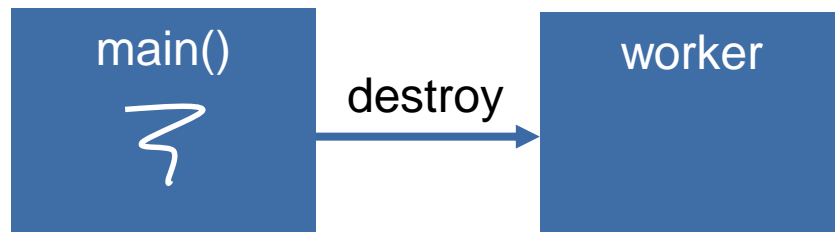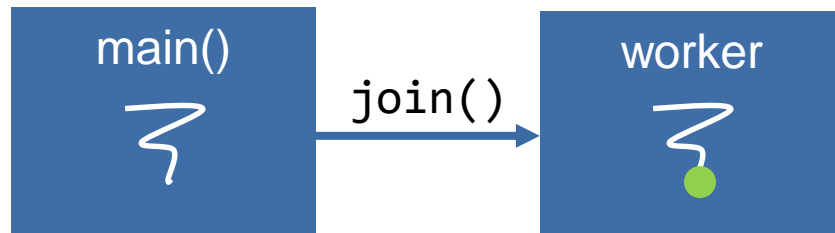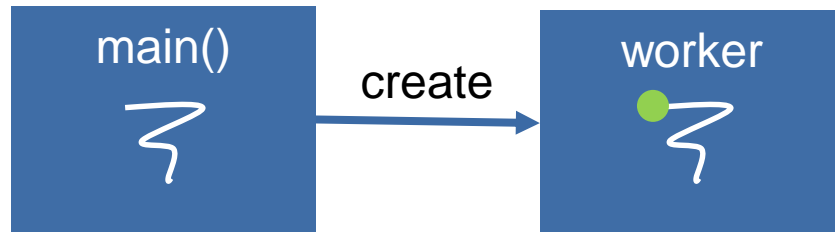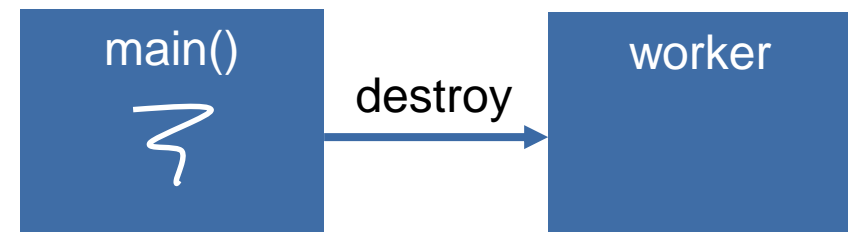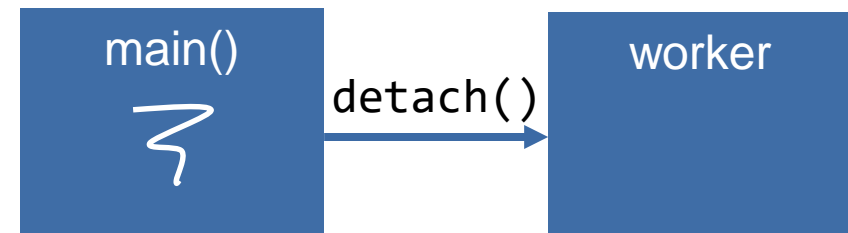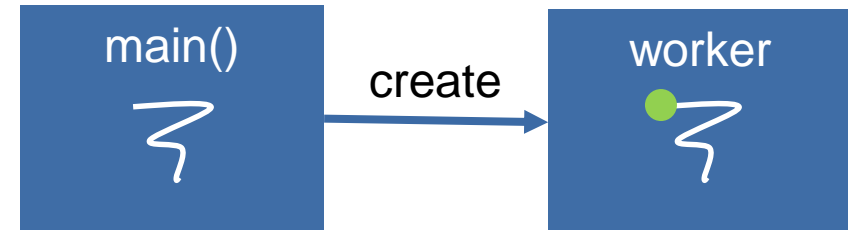
```
Main
Lambda
terminate called without an active exception
```

● **Before the `std::thread` object is destroyed you must `join()` or `detach()` the thread**

```cpp
int main() {
    std::thread worker { doWork };
    worker.join();
}
```

```cpp
int main() {
    std::thread worker { doWork };
    worker.detach();
}
```

- **The destructor doesn't call `join()` nor `detach()`**

  - ▪ If it called `join()` the program might hang when leaving the scope (possibly unexpected due to an exception)
  - ▪ If it called `detach()` the thread continues with possibly deallocated resources (references to local variables)

- **If an unjoined and undetached thread object is destroyed `std::terminate()` will be called**

- **When using `.detach()` beware of the lifetime of objects referred from the detached thread's function, global or passed, or even local ones.**

  - ▪ When the `main()` thread ends globals like `std::cout` will be destroyed

```cpp
void startThread() {
  using namespace std::chrono_literals;
  std::string local{"local"};
  std::thread t{[&] {
    std::this_thread::sleep_for(1s);
    std::cout << local << std::endl;
  }};
  t.detach();
}

int main() {
  using namespace std::chrono_literals;
  startThread();
  std::this_thread::sleep_for(2s);
}
```

DANGER

Dangling References

- **Suggested by Anthony Williams in "C++ Concurrency in Action"**

  - Adapted version from the book

- **RAII wrapper that automatically calls join()**

  - Probably won't be standardized

```cpp
struct ScopedThread {
  explicit ScopedThread(std::thread && t)
      : the_thread{std::move(t)} {
    if (!the_thread.joinable())
      throw std::logic_error { "no thread" };
  }
  ~ScopedThread() {
    the_thread.join();
  }
private:
  std::thread the_thread;
};

int main() {
  ScopedThread t { std::thread {
    [] {std::cout << "Hello Thread"<< std::endl;}
  }};
  std::cout << "Hello Main" << std::endl;
}
```

- **Using global streams does not create data races, but sequencing of characters could be mixed**

?

```cpp
#include <thread>
#include <iostream>

int main() {
  using std::cout;
  using std::endl;

  std::thread t {[] {cout << "Hello Thread" << endl;}};
  cout << "Hello Main" << endl;
  t.join();
}
```

```
Hello Main
Hello Thread
```

```
HHeelllloo MTahirnead
```

```
<Arbitrary Combination>
```

- **namespace `std::this_thread` provides some helper functions**

- **`get_id()`, also available as member function**

  - An id of the underlying OS thread

  - Distinguishes one thread from all others and can be used as a key in a map of threads

- **`sleep_for(duration)`**

  - Suspends threat for a duration

- **`sleep_until(time_point)`**

- **`yield()`**

  - Allows OS to schedule another thread

- **NB: Timing doesn't guarantee sequence!**

```cpp
int main() {
  using std::cout;
  using std::endl;
  using namespace std::chrono_literals;

  std::thread t { [] {
    std::this_thread::yield();
    cout << "Hello ID: "
         << std::this_thread::get_id()
         << endl;
    std::this_thread::sleep_for(10ms);
  }};
  cout << "main() ID: "
       << std::this_thread::get_id()
       << endl;
  cout << "t.get_id(): "
       << t.get_id()
       << endl;
  t.join();
}
```

```cpp
void calcAsync() {
  std::thread t{longRunningAction};
  doSomethingElse();
  t.join();
}
```

```cpp
void countAsync(std::string_view input) {
  std::thread t{[&] {
    countAs(input);
  }};
  t.detach();
}
```

```cpp
void calcAsync() {
  std::thread t{longRunningAction};
  doSomethingElse();
  t.join();
}
```

Depends

If doSomethingElse() does not throw an exception the code is correct.
If an exception is thrown you are doomed!

```cpp
void countAsync(std::string_view input) {
  std::thread t{[&] {
    countAs(input);
  }};
  t.detach();
}
```

Incorrect

The value parameter is captured by reference. It runs out of scope after the function execution. Furthermore, string_view is a reference wrapper itself. The string it is referring to might be destroyed as well.

- **Until the C++ standard specifies the networking TS (>C++20) you have to rely on external implementations (like ASIO)**

- **ASIO is best used to implement asynchronous operations (available in boost or standalone)**

- **The asynchronous programming model has special pitfalls in non-garbage collected environments**

- **The C++ standard features API for using threads**