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

Cevelop
Your C++ deserves it

HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL
FHO Fachhochschule Ostschweiz
IFS INSTITUTE FOR SOFTWARE

Recap Week 9

Solution with Tags & Traits

- **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;
};
```

Solution with Tags & Traits

- **Add a `speedCast` function**

```
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**

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

```
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)};
}

//...

}
```

Goals

- **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)

Cevelop

Your C++ deserves it

HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL

FHO Fachhochschule Ostschweiz

IFS INSTITUTE FOR SOFTWARE

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

- **TCP Sockets**
  - Reliable
  - Stream-oriented
  - Requires connection setup

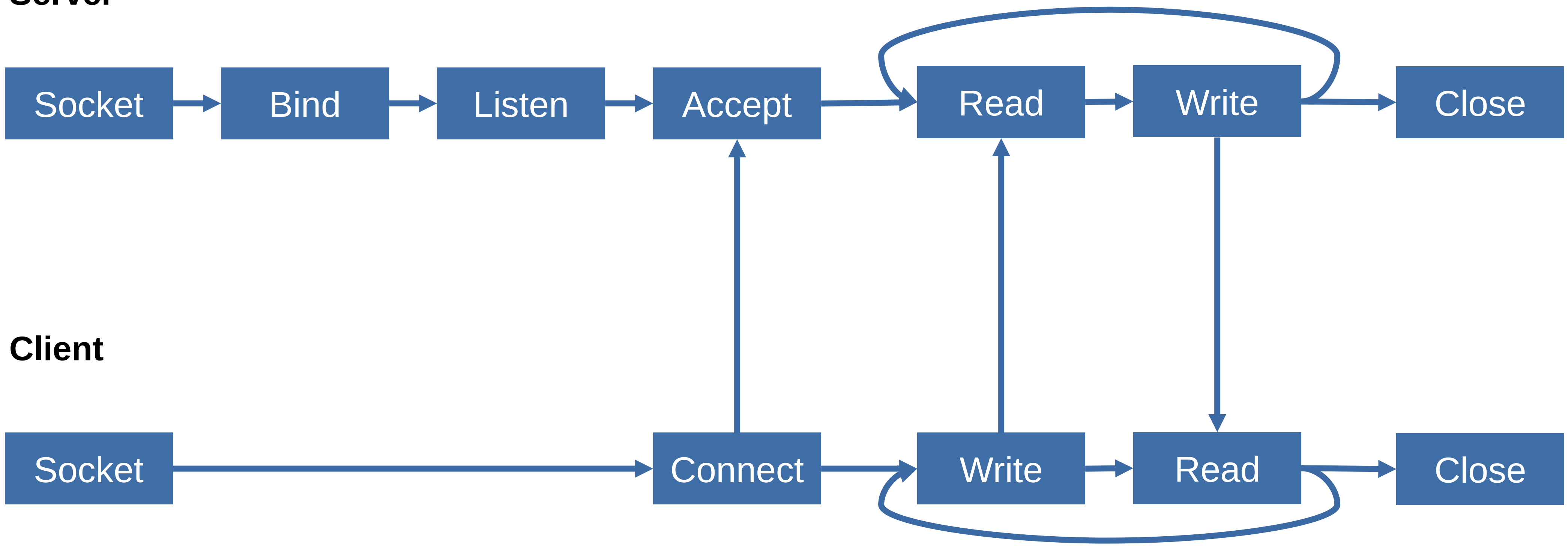
- **UDP Sockets**
  - Packets might get lost or arrive out of order
  - Datagram-oriented (max 65k)
  - Sending/receiving possible without a connection

Berkley Socket Primitives for TCP

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

Distributed Systems: Principles and Paradigms, 2nd Edition Andrew S. Tanenbaum

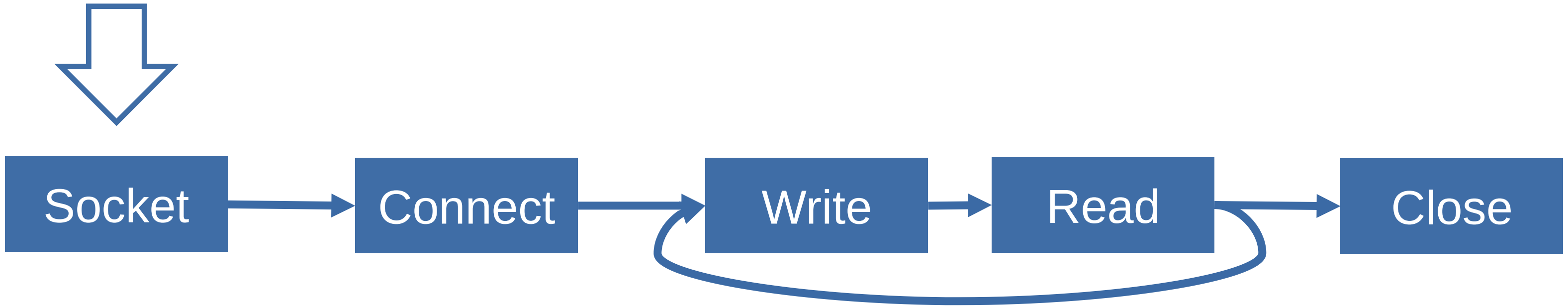
- **Server**
- **Client**



Example Synchronous TCP Client Connection with ASIO

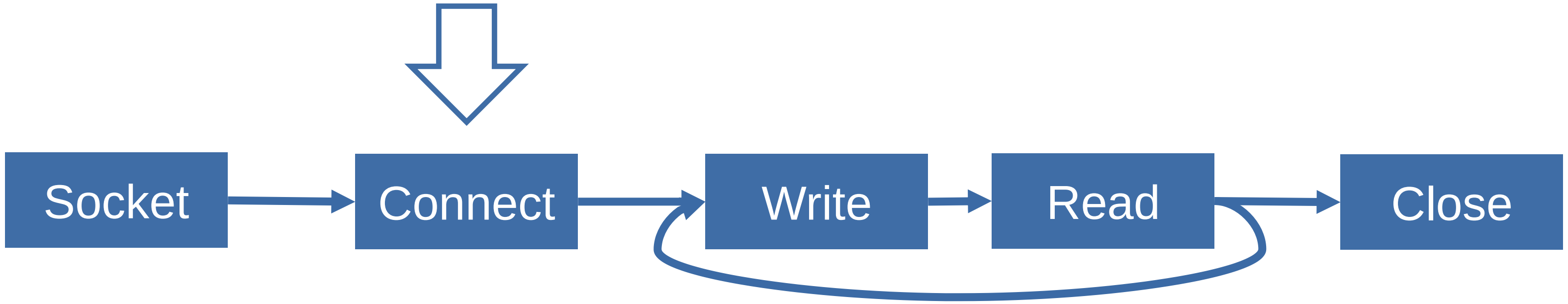
- **All ASIO operations require an I/O context (more on that later)**
- **Create a TCP socket using the context**

```
asio::io_context context{};

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



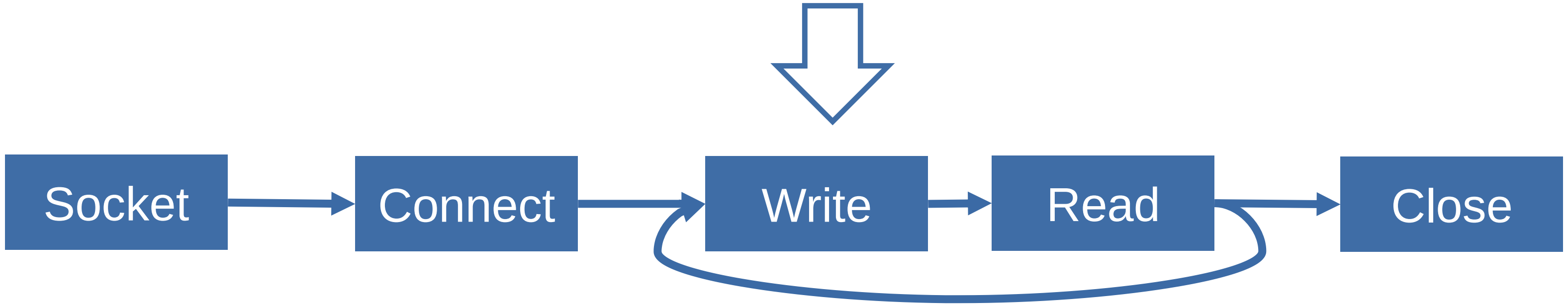
- **A resolver resolves host names to end points (for TCP IP address and port)**
- **`asio::connect()` tries to establish a connection to given end point(s)**

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

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



- **`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`)

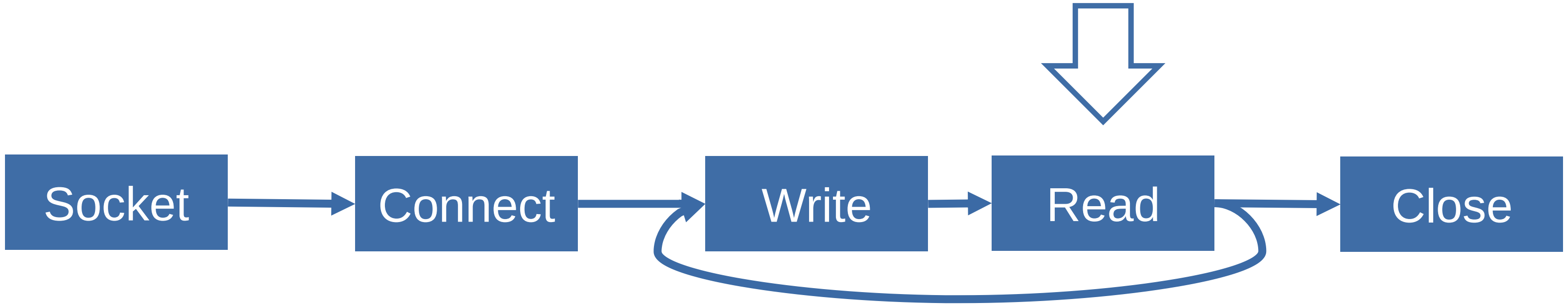
```
std::ostringstream request{};
request << "GET / HTTP/1.1\r\n";
request << "Host: " << domain << "\r\n";
request << "\r\n";

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



Example Synchronous TCP Client Connection with ASIO

- **`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`)**

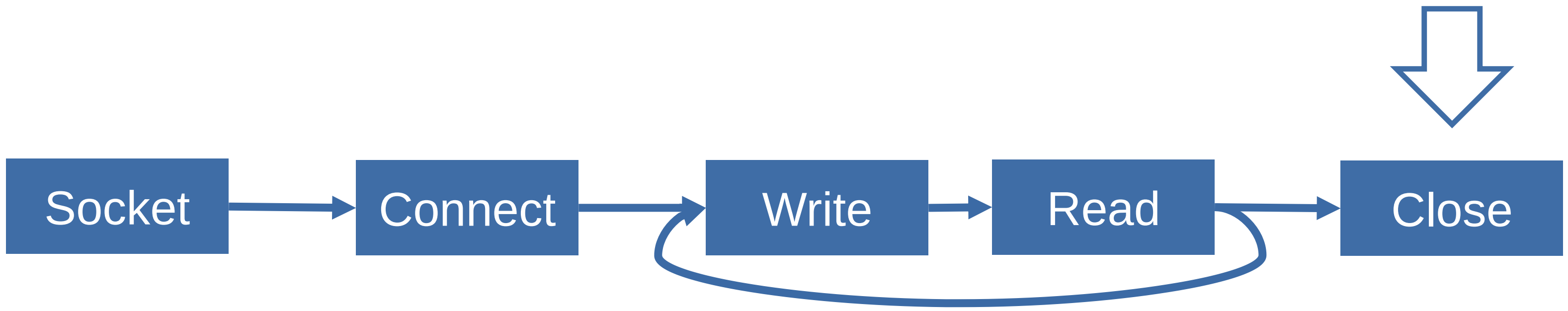
```
constexpr size_t bufferSize = 1024;
std::array<char, bufferSize> reply{};
asio::error_code errorCode{};
auto readLength = asio::read(socket, asio::buffer(reply.data(), bufferSize), errorCode);
```



Example Synchronous TCP Client Connection with ASIO

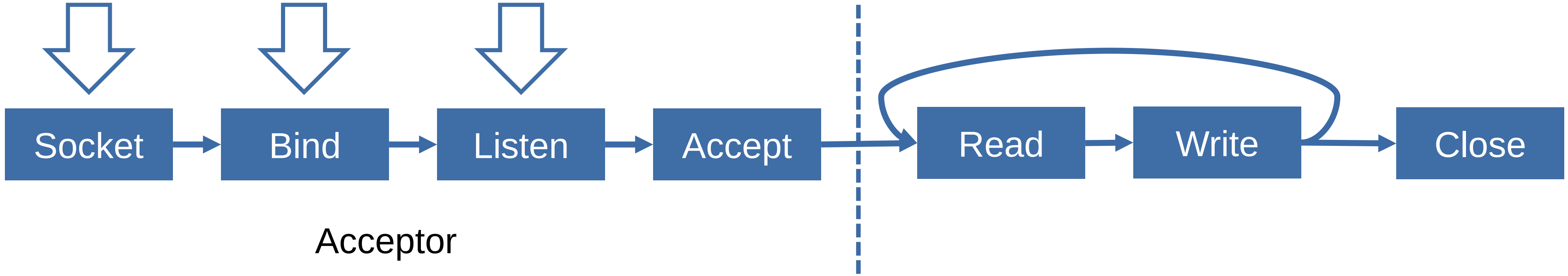
- **`shutdown()` closes the read/write stream associated with the socket, indicating to the peer that no more data will be received/sent**
- **The destructor of the socket cancels all pending operations and destroys the object**

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

socket.close();
```



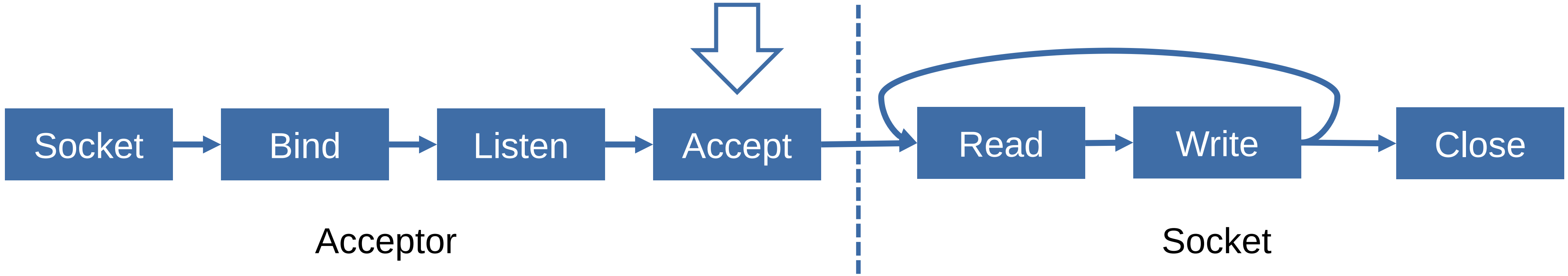
- **An acceptor is a special socket responsible for establishing incoming connections**
- **In ASIO the acceptor is bound to a given local end point and starts listening automatically**

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



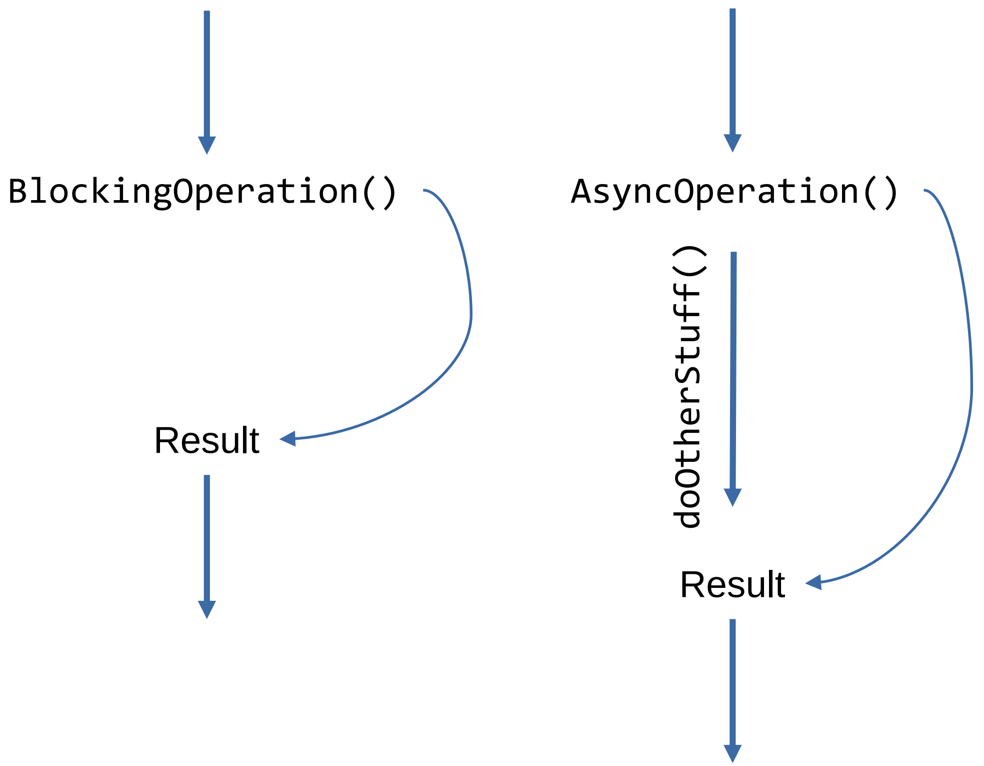
- **The `accept()` member function blocks until a client tries to establish a connection (with connect)**
- **It returns a new socket through which the connected client can be reached**

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



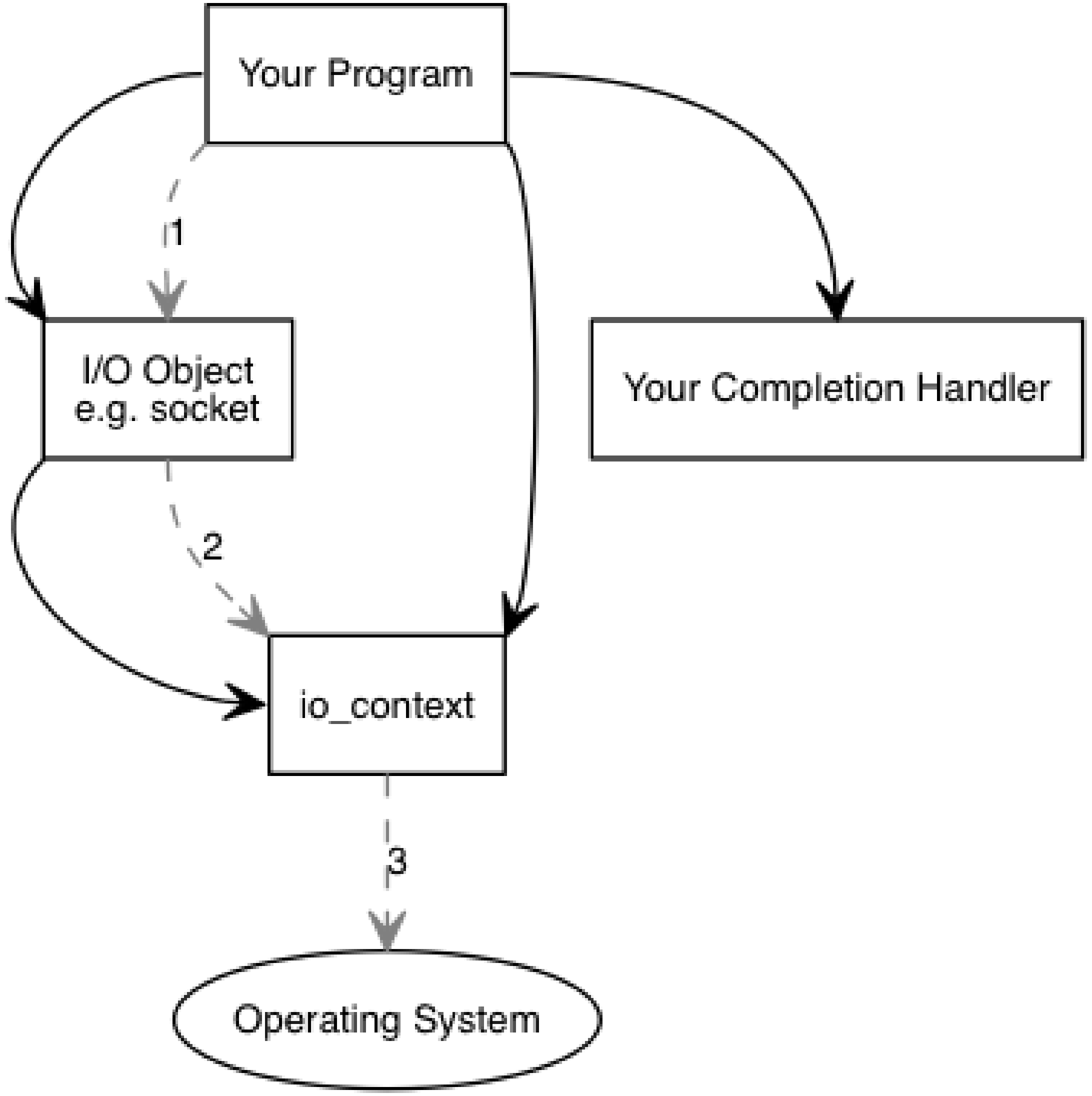
Handling Multiple Requests Simultaneously

- **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**



ASIO Basics of Asynchronous Operations (1/2)

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**



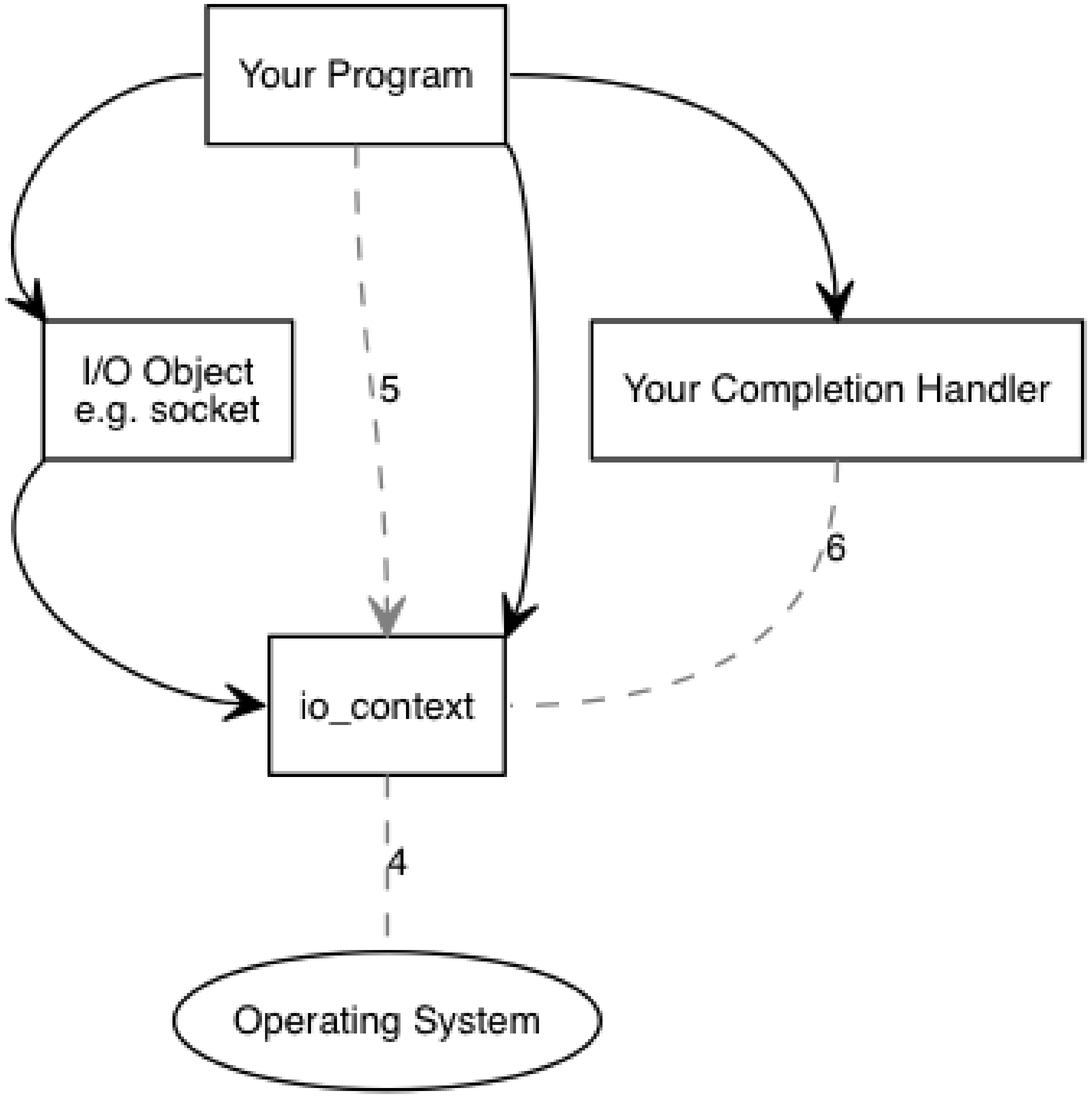
https://think-async.com/Asio/asio-1.12.2/doc/asio/overview/core/basics.html

ASIO Basics of Asynchronous Operations (2/2)

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)**

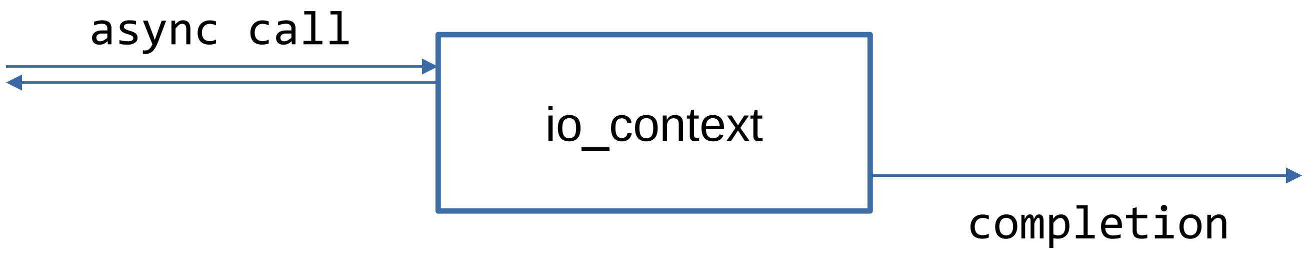
6. **Still inside io_context::run() the completion handler is called to handle the result (or error) of the asynchronous operation**



https://think-async.com/Asio/asio-1.12.2/doc/asio/overview/core/basics.html

Asynchronous Read/Write on Sockets

- **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**



Example Asynchronous Read from Socket

- **`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**

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

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

Example Asynchronous Write to Socket

- **`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**

```
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;
};
```

Asynchronous Acceptor (`accept()`)

```
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**

Asynchronous Acceptor (Constructor)

```
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

Asynchronous Acceptor (Use)

```
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`?**

```
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

```
//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**

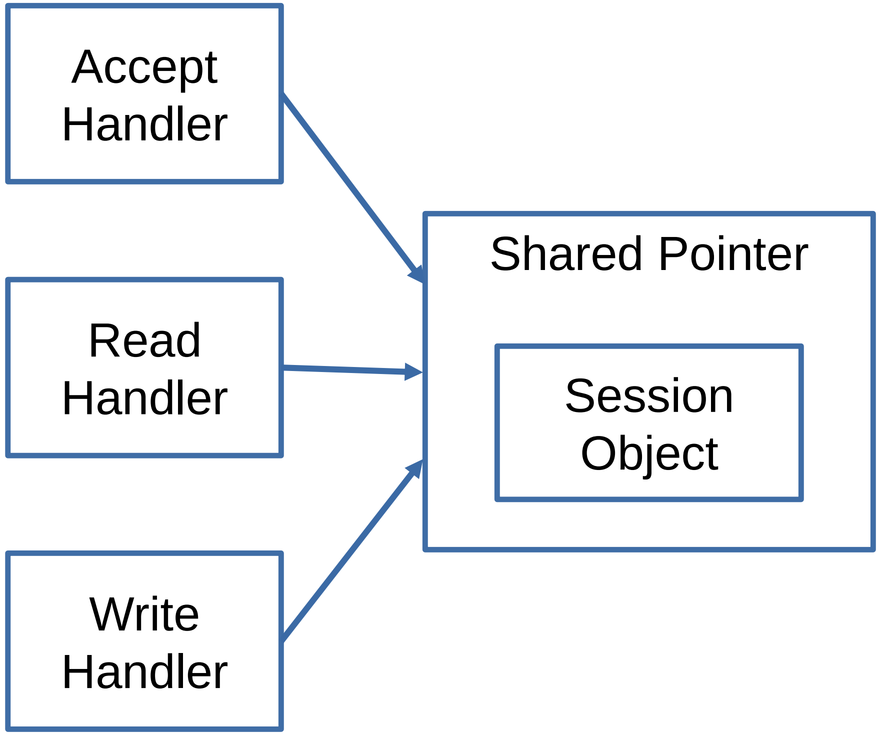
```
//In the accept handler
void Session::read() {
  auto handler = [self = shared_from_this()](error_code ec, size_t length) {
  //...
```

Responsibility of Keeping Session Alive

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

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

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



What is Missing? (Preview)


```
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);
}
```

What is Missing?


```
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**

What is Missing? (Preview)


```
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);
}
```

What is Missing?


```
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

Cevelop
Your C++ deserves it

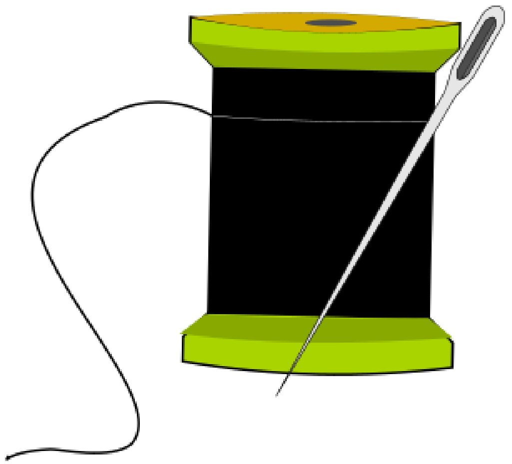
HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL
FHO Fachhochschule Ostschweiz

IFS
INSTITUTE FOR
SOFTWARE

- **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`

Cevelop
Your C++ deserves it



HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL
FHO Fachhochschule Ostschweiz

IFS INSTITUTE FOR SOFTWARE

```
class std::thread
```

```
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)**

```
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();
}
```

Passing Arguments to a `std::thread`

`#include <thread>`

```
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!**

```
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();
}
```

What If We Don't Join?

- **Guess what happens**

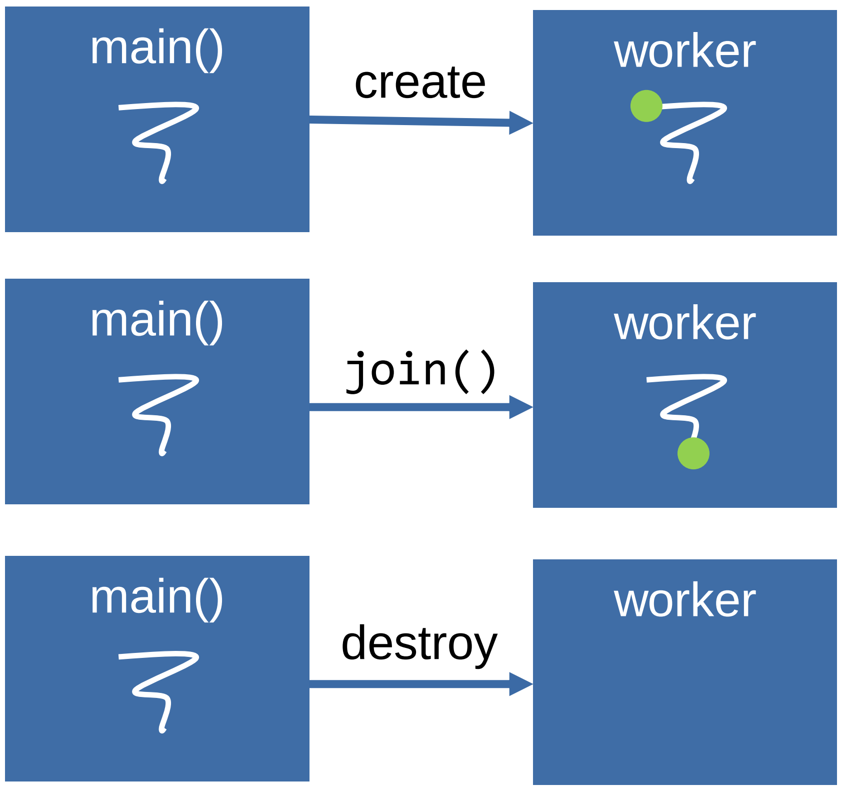
```
int main() {
  std::thread lambda {
    [] {std::cout << "Lambda" << std::endl; }
  };
  std::cout << "Main" << std::endl;
}
```

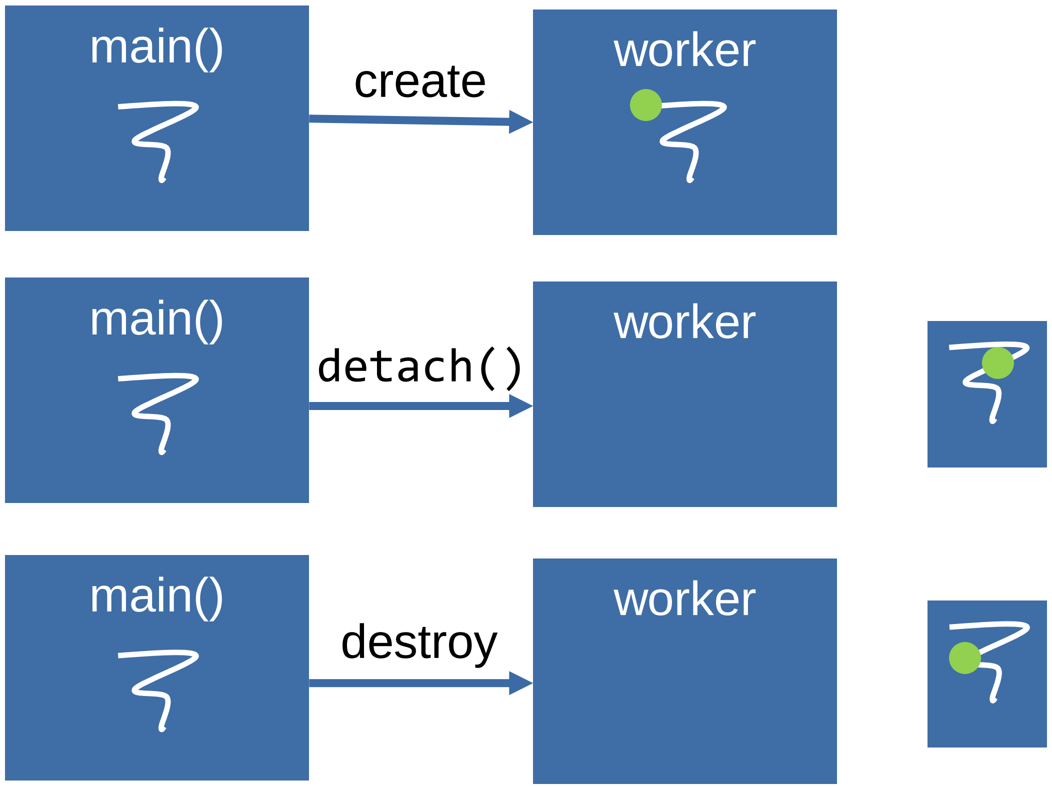
```
Main
Lambda
terminate called without an active exception
```

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

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

```
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);
}
```

Dangling References

RAII Wrapper for Threads

- **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

```
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!**

```
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();
}
```

Right or Wrong? (Preview)


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

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

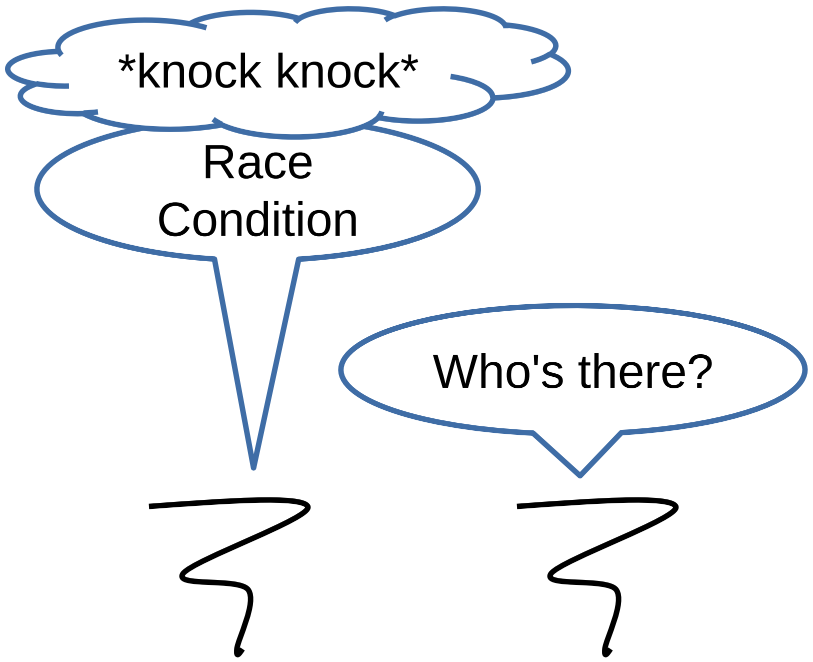
Right or Wrong?

`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!

`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.

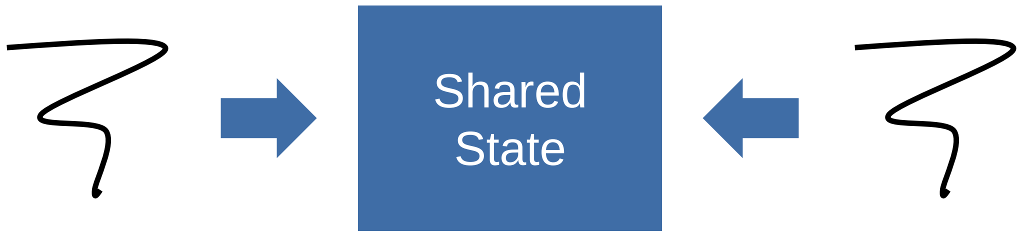
Communication Between Threads

Cevelop
Your C++ deserves it



How Can Threads Communicate With Eachother?

- **Mutable Shared State**
- **Problem: Data Race**
  - Two operations on the same memory location
  - At least one is not atomic (at the same time)
  - At least one is a modifying operation
- **Solution**
  - Locking the shared access
  - Make access atomic

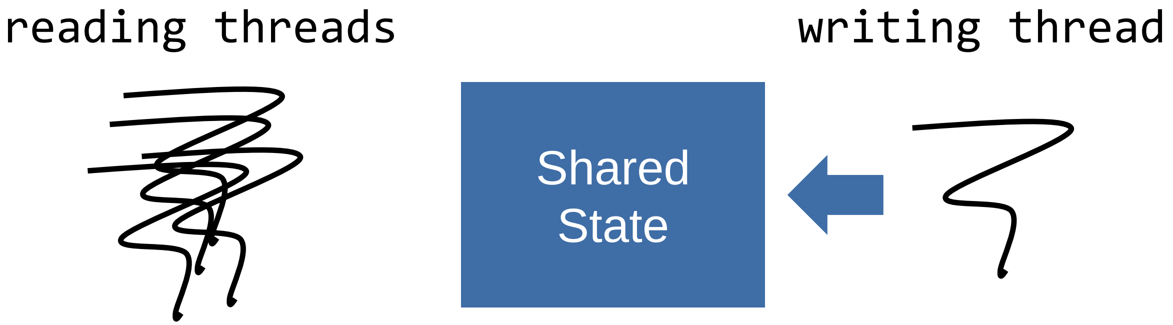


- **All mutexes provide the following operations**
  - Acquire:
    - `lock()` – blocking
    - `try_lock()` – non-blocking
  - Release:
    - `unlock()` – non-blocking

- **Two properties specify the capabilities**
  - Recursive – Allow multiple nested acquire operations of the same thread
    - Prevents self-deadlock
  - Timed - Also provide timed acquire operations:
    - `try_lock_for(<duration>)`
    - `try_lock_until(<time>)`

		Recursive: No	Recursive: Yes
Timed	No	`std::mutex`	`std::recursive_mutex`
Timed	Yes	`std::timed_mutex`	`std::recursive_timed_mutex`

- **Reading operations don't need exclusive access**
  - Only concurrent writes need exclusive locking
- **std::shared_mutex (C++17) und std::shared_timed_mutex (C++14) provide exclusive and shared locking**
- **Additional functions for read-locking:**
  - `lock_shared()`
  - `try_lock_shared()`
  - `try_lock_shared_for(<duration>)`
  - `try_lock_shared_until(<time>)`
  - `unlock_shared()`



- **Usually you will not acquire and release mutexes directly through the supplied member functions**
- **Instead you use a lock that manages the mutex**

`std::lock_guard`	RAII wrapper for a single mutex: • Locks immediately when constructed • Unlocks when destructed
`std::scoped_lock`	RAII wrapper for mutliple mutexes • Locks immediately when constructed • Unlocks when destructed
`std::unique_lock`	Mutex wrapper that allows defered and timed locking: • Similar interface to timed mutex • Allows explicit locking/unlocking • Unlocks when destructed (if still locked)
`std::shared_lock`	Wrapper for shared mutexes • Allows explicit locking/unlocking • Unlocks when destructed (if still locked)

Mutex / Guard Example

- **Threadsafe queue**
  - Delegate functionality to `std::queue`
  - Make every member function **mut**ually **ex**clusive
- **Scoped Locking Pattern**
  - Create a lock guard that locks and unlocks the mutex automatically
- **Strategized Locking Pattern**
  - Template parameter for mutex type
  - Could also be `null_mutex` (boost)


```cpp
template <typename T,
          typename MUTEX = std::mutex>
struct threadsafe_queue {
  using guard = std::lock_guard<MUTEX>;
  void push(T const &t) {
    guard lk{mx};
    q.push(t);
  }
  T pop() { /* later */ return T{};}
  bool try_pop(T & t){
    guard lk{mx};
    if (q.empty()) return false;
    t = q.front();
    q.pop();
    return true;
  }
  bool empty() const{
    guard lk{mx};
    return q.empty();
  }
private:
  mutable MUTEX mx{};
  std::queue<T> q{};
};
```

Why not `this->empty()`?

Why `mutable`?

- **`std::scoped_lock`**
  - Acquires multiple locks in the constructor
  - Avoids deadlocks, by relying on internal sequence
  - Blocks until all locks could be acquired
  - Class template argument deduction avoids the need for specifying the template arguments

```
// can't be noexcept, because locks might throw
void swap(threadsafe_queue<T> & other) {
  if (this == &other) return;

  std::scoped_lock both{mx, other.mx};

  std::swap(q, other.q);
  // no need to swap mutex or condition variable
}
```

- **`std::lock`**
  - Acquires multiple locks in a single call
  - Avoids deadlocks
  - Blocks until all locks could be acquired
- **`std::try_lock`**
  - Tries to acquire multiple locks in a single call
  - Does not block
  - When it returns...
    - `true`, all locks have been acquired
    - `false`, no lock has been acquired

```
// can't be noexcept, because locks might throw
void swap(threadsafe_queue<T> & other) {
  if (this == &other) return;

  // std::defer_lock prevents immediate locking
  lock my_lock{mx, std::defer_lock};
  lock other_lock{other.mx, std::defer_lock};

  // blocks until all locks are acquired
  std::lock(my_lock, other_lock);

  std::swap(q, other.q);
  // no need to swap mutex or condition variable
}
```

`std::condition_variable`

- **Similar to Java Condition**
  - But is not bound to a lock at construction
- **Waiting for the condition**
  - `wait(<mutex>)` – requires surrounding loop
  - `wait(<mutex>, <predicate>)` – loops internally
  - Timed waits `wait_for` and `wait_until`
- **Notifying a (potential) change**
  - `notify_one()`
  - `notify_all()`
- **`std::unique_lock` as condition releases lock (wait)**

```cpp
template <typename T,
          typename MUTEX = std::mutex>
struct threadsafe_queue {
  using guard = std::lock_guard<MUTEX>;
  using lock = std::unique_lock<MUTEX>;
  void push(T const & t) {
    guard lk{mx};
    q.push(t);
    notEmpty.notify_one();
  }
  T pop() {
    lock lk{mx};
    notEmpty.wait(lk, [this] {
      return !q.empty();
    });
    T t = q.front();
    q.pop();
    return t;
  }
private:
  mutable MUTEX mx{};
  std::condition_variable notEmpty{};
  std::queue<T> q{};
};
```

- **There is no thread-safety wrapper for standard containers! (yet)**
- **Access to different individual elements from different threads is not a data race**
  - Container must not change during the concurrent access to elements!
  - Using different elements of a `std::vector` from different threads is OK!
- **Almost all other concurrent uses of containers is dangerous**
- **`shared_ptr` copies to the same object can be used from different threads, but accessing the object itself can race if non-const**
  - Reference counter is an atomic

Summary

- **Create your own types if you want to represent values with dimension**
- **User defined literals help giving meaning to simple values**
- **They can be used to compute numbers at compile-time**