Department I - C Plus Plus

Modern and Lucid C++ Advanced for Professional Programmers

Week 10 – Networking and Asynchronous Operations

Thomas Corbat / Felix Morgner
Rapperswil, 10.05.2022
FS2022

Agenda

- **Recap Week 9**
- **Networking in C++**
- **Signal Handling**
- **Shared Data with ASIO**

Goals

- **Participants should ...**
  - know the basics of communication with sockets
  - be able to implement applications that communicate over a network
  - be able to implement network IO with synchronous and asynchronous operation
  - describe how to access shared data safely within ASIO

Recap Week 9

Cevelop
Your C++ deserves it

OST
Ostschweizer
Fachhochschule

- **The C++ Memory Model defines**
  - When the effect of an operation is visible to other threads
  - How and when operations might be reordered
- **Visibility of effects:**
  - *sequenced-before*: within a single thread
  - *happens-before*: either sequenced-before or inter-thread happens-before
  - *synchronizes-with*: inter-thread sync.
- **Note: Reads/writes in a single statement are “unsequenced”!**

```
std::cout << ++i << ++i; //don't know output
```

- **Memory orderings define when effects become visible**
  - Sequentially-consistent
    - “Intuitive” and the default behavior
  - Acquire/Release
    - Weaker guarantees than sequentially-consistent
  - Consume (discouraged)
    - Slightly weaker than acquire-release
  - Relaxed
    - No guarantees besides atomicity!

- **Sequential Consistency**
  - Global execution order of operations
  - Every thread observes the same order
- **Memory order flag**
  - `std::memory_order_seq_cst`
- **Default behavior**
- **The latest modification (in the global execution order) will be available to a read**

```
std::atomic<bool> x, y;
std::atomic<int> z;
```

Every function runs on its own thread

```
void write_x() {
  x.store(true);
}
```

```
void read_x_then_y() {
  while (!x.load());
  if (y.load()) ++z;
}
```

```
void write_y() {
  y.store(true);
}
```

```
void read_y_then_x() {
  while (!y.load());
  if (x.load()) ++z;
}
```

What are possible values of z after the execution of all threads?

Example taken from Anthony Williams: C++ Concurrency in Action

Sequential Consistency (z == 2)

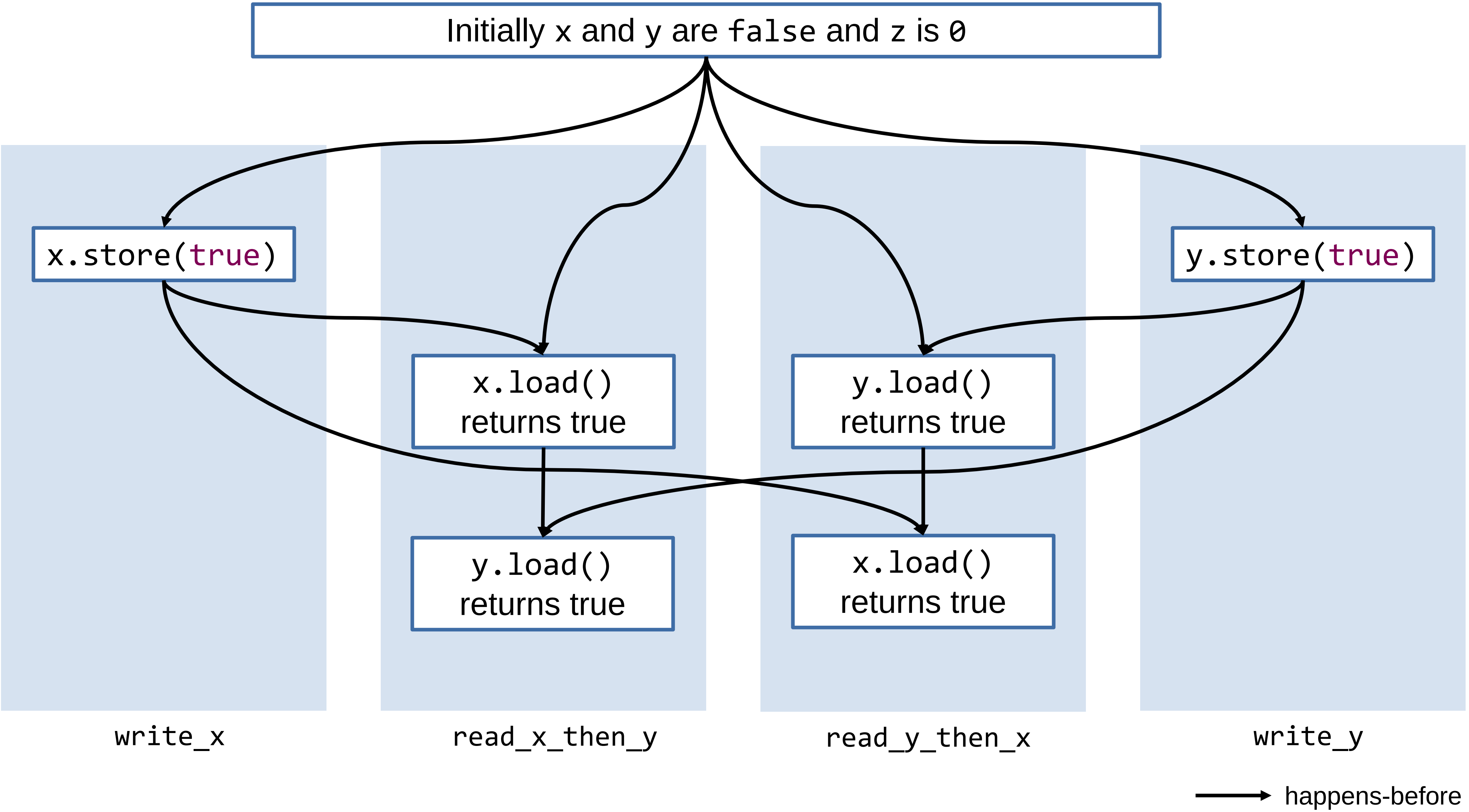


Illustration adapted from Anthony Williams: C++ Concurrency in Action

Custom Types with `std::atomic`

- **Custom types need to be trivially copyable**
  - You can not have a custom copy ctor
  - You can not have a custom move ctor
  - You can not have a custom copy assignment operator
  - You can not have a custom move assignment operator
- **Object can only be accessed as a whole**
  - No member access operator in `std::atomic`

```
struct SimpleType {
  int first;
  float second;
};
```

```
struct NonTrivialCCtor {
  NonTrivialCCtor(NonTrivialCCtor const&) {
    std::cout << "copied!\n";
  }
};
```

```
struct NonTrivialMember {
  int first;
  std::string second;
};
```

Networking in C++ (With ASIO)

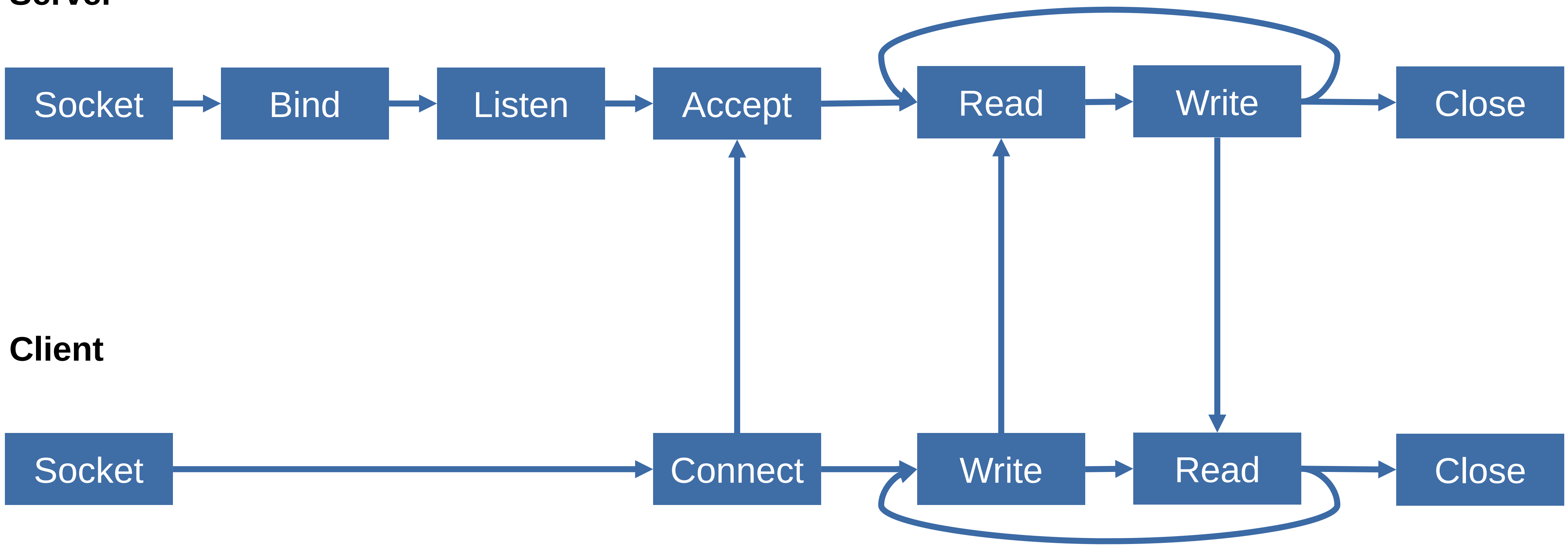
- **Sockets are an abstraction of endpoints for communication**

- **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

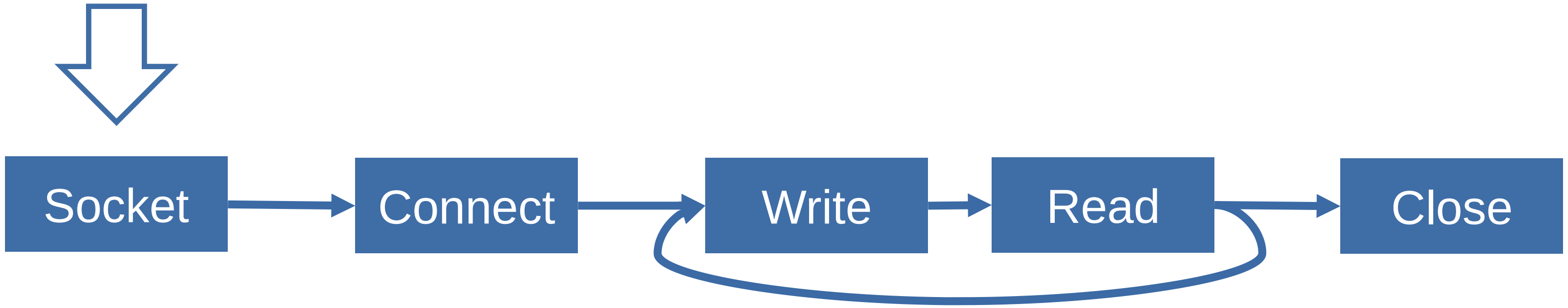
Primitive	Meaning
Socket	Create a new communication end point
Bind	Attach a local address to a socket
Listen	Announce willingness to accept connections
Accept	Block caller until a connection request arrives
Connect	Actively attempt to establish a connection
Send	Send some data over the connection
Receive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

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



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

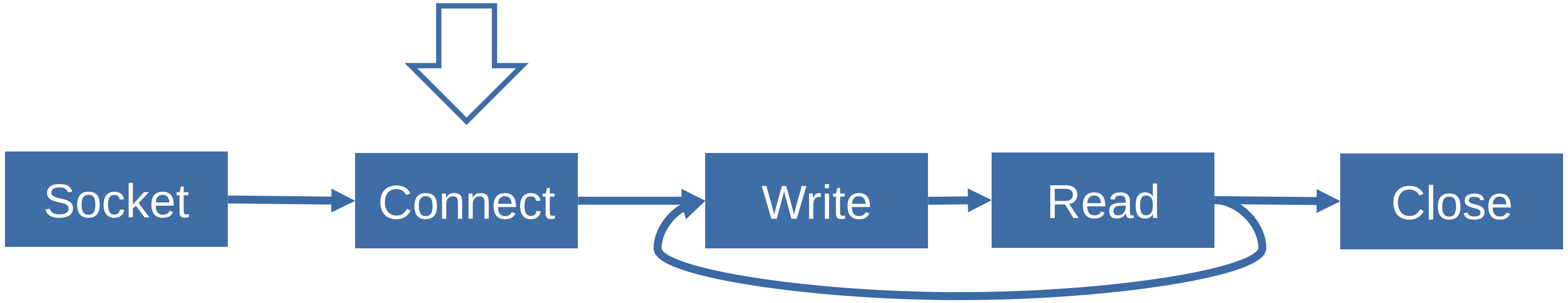
```
asio::io_context context{};

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



Example Synchronous TCP Client Connection with ASIO (With IP Addresses)

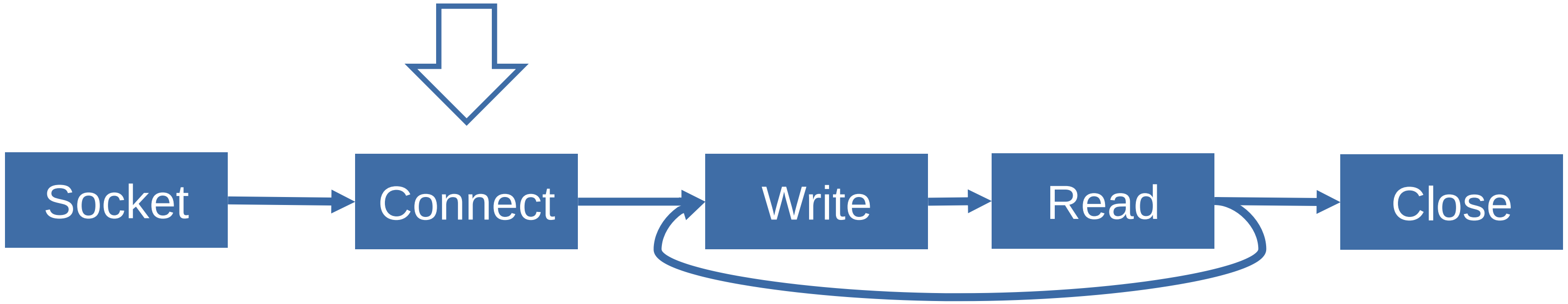
- **If the IP address is known, an endpoint can be constructed easily**
- **`socket.connect()` tries to establish a connection to the given endpoint**

```
auto address = asio::ip::make_address("127.0.0.1");
auto endpoint = asio::ip::tcp::endpoint(address, 80);

socket.connect(endpoint);
```



- **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

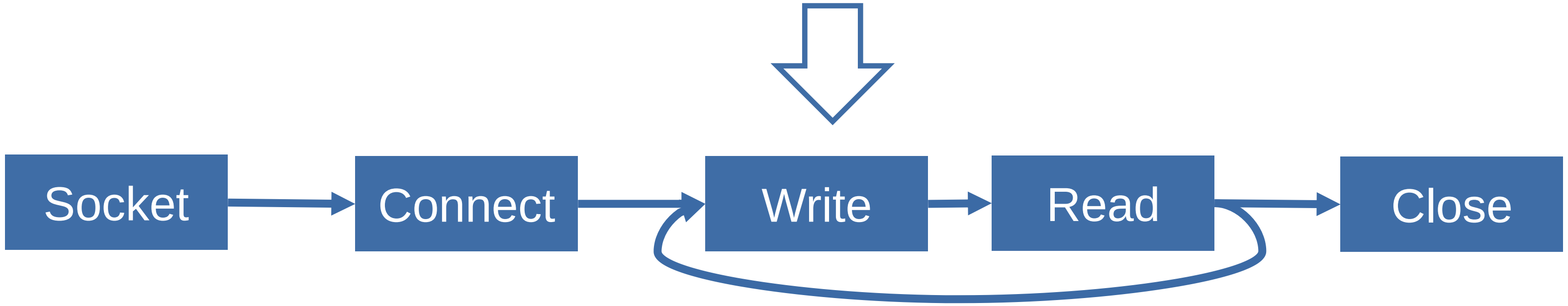
```
asio::ip::tcp::resolver resolver{context};
auto endpoints = resolver.resolve(domain, "80");

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



- **Transmit / Receive functions need sources or destinations buffers**
  - ASIO generally does not manage memory for you!
  - Fixed size buffers using `asio::buffer()`
    - Must provide at least as much memory as you would like to read
    - Can use several standard containers as a backend
    - Pointer + Size combinations are also available
  - Dynamically sized buffers using `asio::dynamic_buffer()`
    - For use with `std::string` and `std::vector`
  - Streambuf buffers using `asio::streambuf`
    - Works with `std::istream` and `std::ostream`

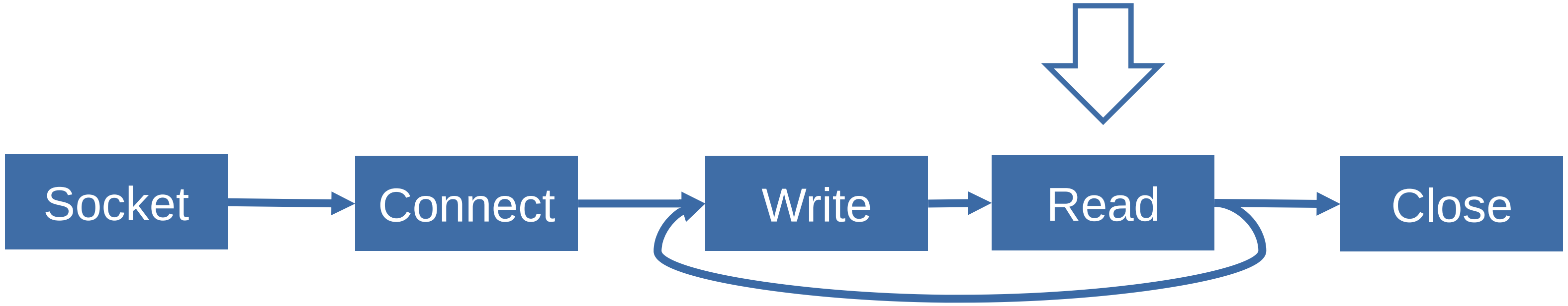
- **`asio::write()` sends the data to the peer the socket is connected to**
  - It returns when all data is sent or an error occurred (exception: `asio::system_error`)

```
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()));
```



- **`asio::read()` receives data sent by the peer the socket is connected to**
  - It returns when the read-buffer is full, or an error occurred
- **The error code is set if a problem occurs, or the stream has been closed (`asio::error::eof`)**
  - If you don't pass an "out-parameter" for the error, the error will be thrown as `std::system_error`

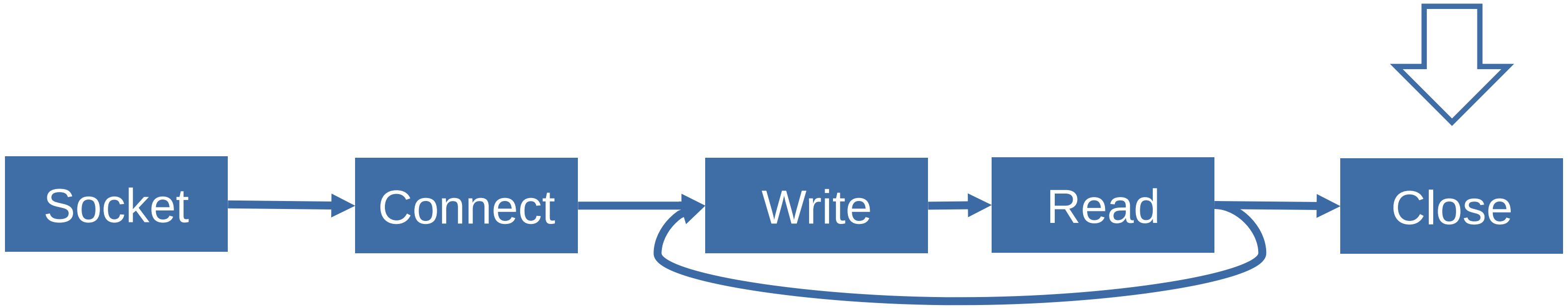
```
constexpr size_t bufferSize = 1024;
std::array<char, bufferSize> reply{};
asio::error_code errorCode{};
auto readLength = asio::read(socket, asio::buffer(reply.data(), bufferSize), errorCode);
```



- **`asio::read` also allows you to specify completion conditions**
  - `asio::transfer_all()` – Default behavior, transfer all available data or until the buffer is full
  - `asio::transfer_at_least(std::size_t bytes)` – Read at least bytes number of bytes (may transfer more)
  - `asio::transfer_exactly(std::size_t bytes)` – Read exactly bytes number of bytes
- **`asio::read_until` allows you to specify conditions on the data being read**
  - Simple matching of characters or strings
  - More complex matching using `std::regex`
  - Also allows you to specify a callable object
    - Expects `std::pair<iterator, bool> operator()(iterator begin, iterator end)`
  - May read more! You need to work with the number of bytes returned by the call

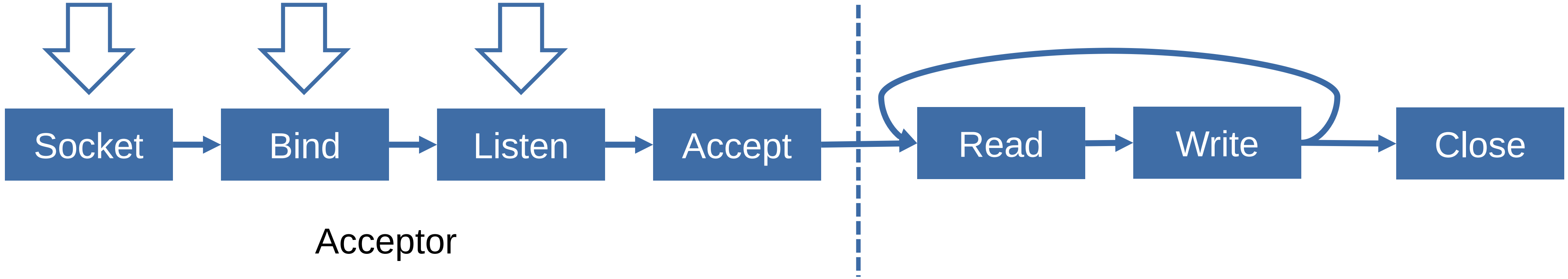
- `shutdown()` **closes the read/write stream associated with the socket, indicating to the peer that no more data will be received/sent**
- **The destructor of the socket cancels all pending operations and destroys the object**

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

socket.close();
```



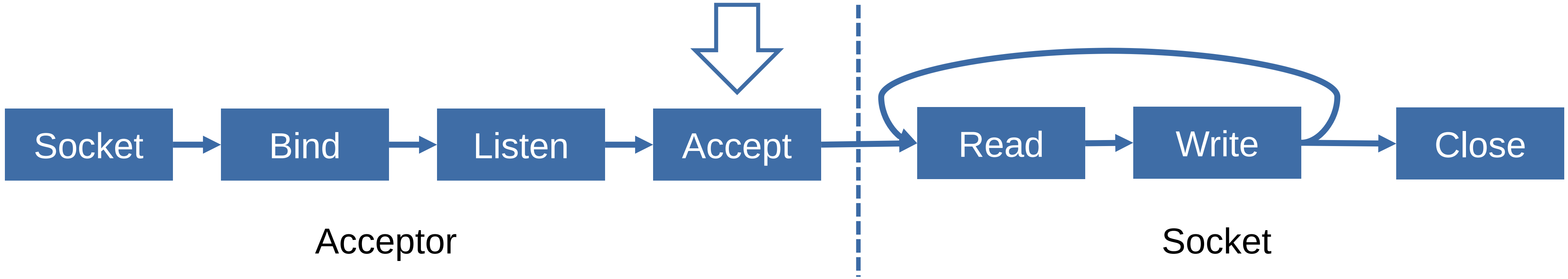
- **An acceptor is a special socket responsible for establishing incoming connections**
- **In ASIO the acceptor is bound to a given local end point and starts listening automatically**

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

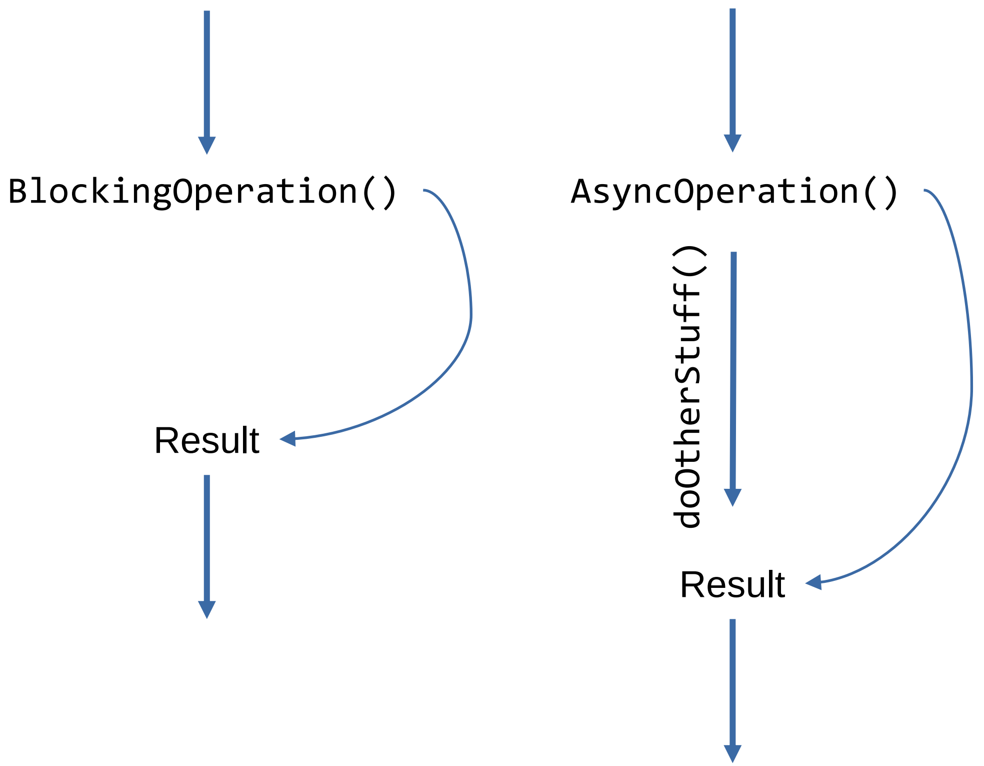


- **The `accept()` member function blocks until a client tries to establish a connection (with connect)**
- **It returns a new socket through which the connected client can be reached**

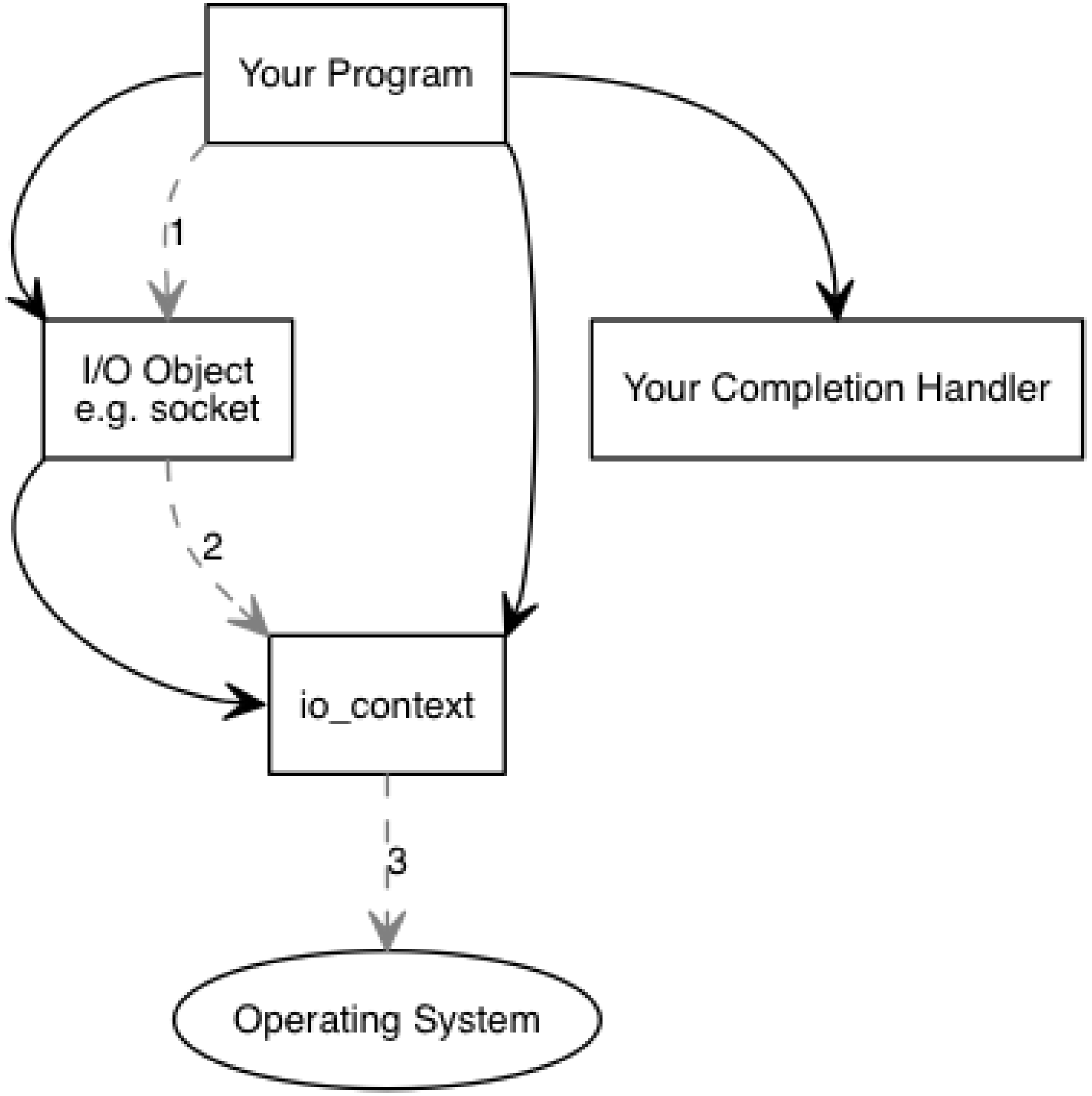
```
asio::ip::tcp::endpoint peerEndpoint{};
asio::ip::tcp::socket peerSocket = acceptor.accept(peerEndpoint);
```



- **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**



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**

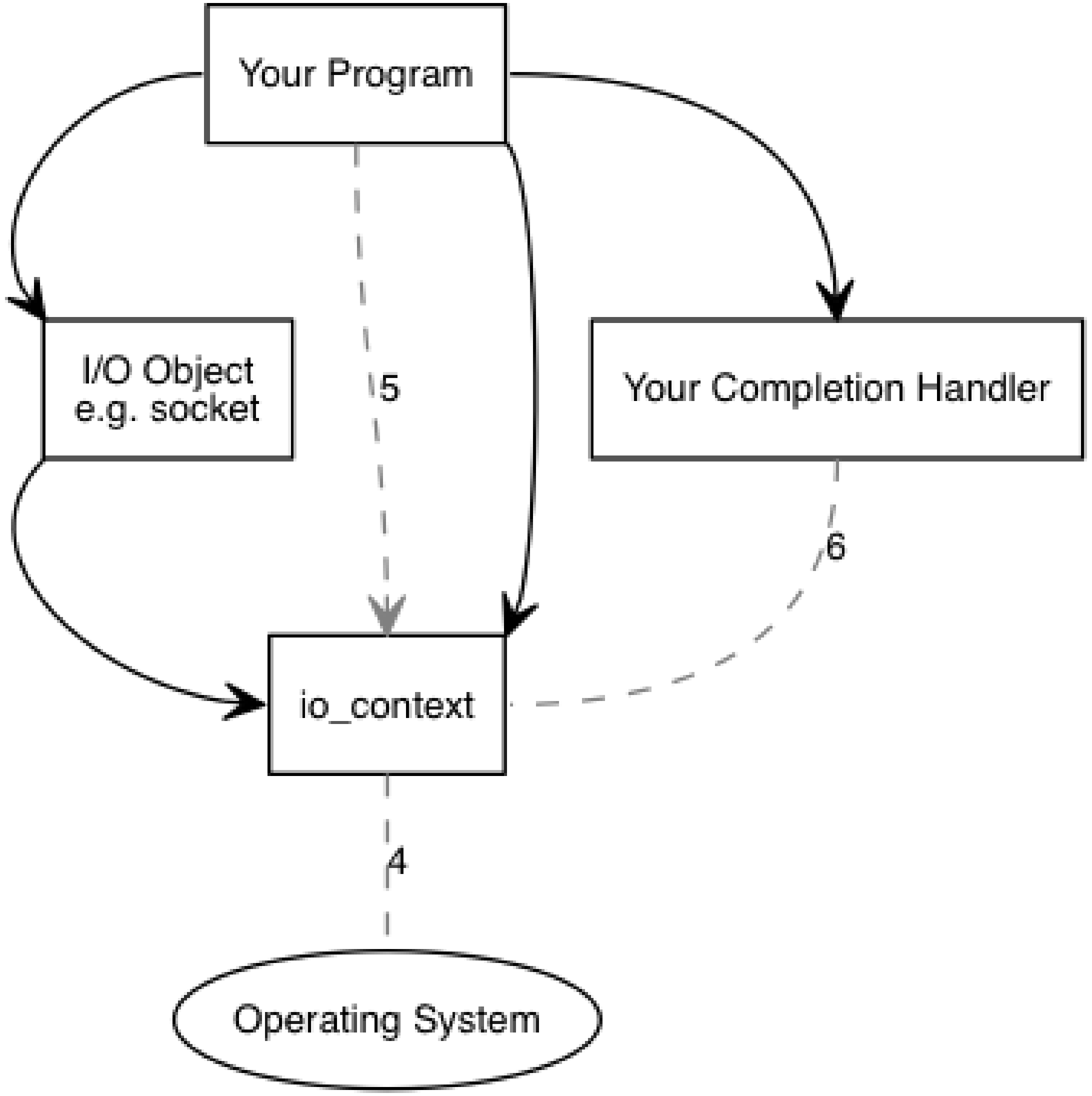


https://think-async.com/Asio/asio-1.12.2/doc/asio/overview/core/basics.html

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)**

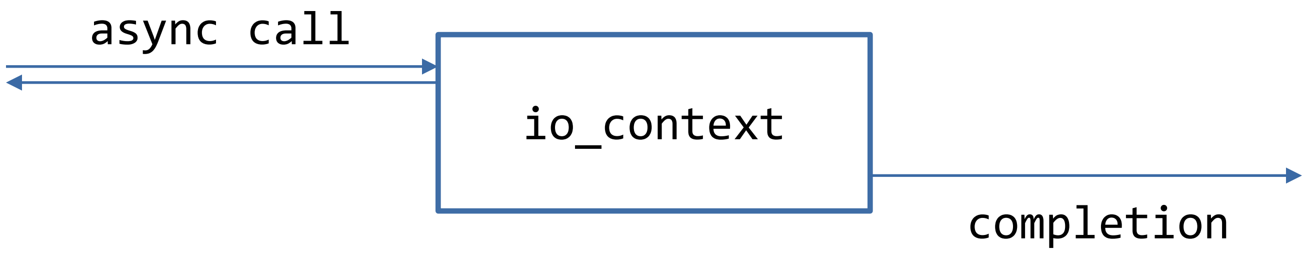
6. **Still inside `io_context::run()` the completion handler is called to handle the result (or error) of the asynchronous operation**



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 operation 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);
```

Asynchronous Acceptor (Overview)

```
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**

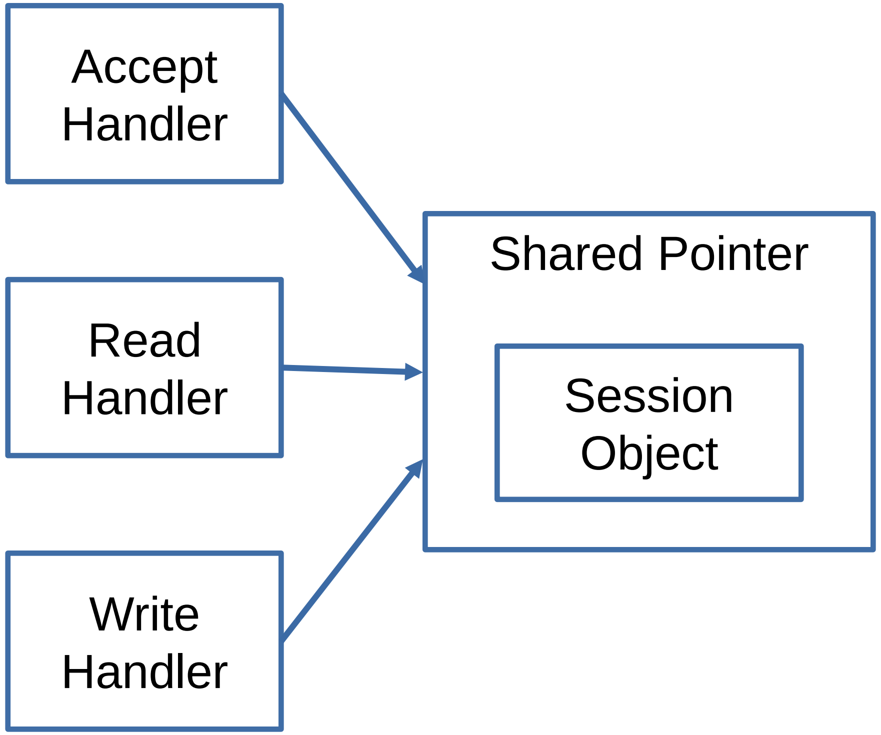
```
//In the accept handler
void Session::read() {
  auto handler = [self = shared_from_this()](error_code ec, size_t length) {
  //...
```

Responsibility of Keeping Session Alive


```
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]
```



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**

Signal Handling

- **Most operating systems support signals**
  - Signals provide asynchronous notifications
  - Signals are used to
    - Gracefully terminate a program
    - Communicate errors
    - Notify about traps
- **The C++ standard defines a portable subset**
  - `#include <csignal>`

Signal	Description
SIGTERM	Termination requested
SIGSEGV	Invalid memory access
SIGINT	User interrupt
SIGILL	Illegal Instruction
SIGABRT	Abnormal termination (e.g. std::abort)
SIGFPE	Floating-point Exception

- **Operating systems may provide additional signals**
  - However: They are not portable!

- **ASIO provides signal handling**
  - asio::signal_set defines a set of signals to wait for
  - signal_set.wait(handler)
  - signal_set.async_wait(handler)
- **The signal handler receives**
  - The signal that occurred
  - An error if the wait was aborted
- **Useful to cleanly stop server applications**

```
#include <asio.hpp>

#include <csignal>
#include <iostream>

int main() {
  auto context = asio::io_context{};

  auto signals = asio::signal_set{context, SIGINT, SIGTERM};
  signals.async_wait([&](auto error, auto sig) {
    if (!error) {
      std::cout << "received signal: " << sig << '\n’;
    } else {
      std::cout << "signal handling aborted\n";
    }
  });

  context.run();
}
```

Accessing Shared Data

Your C++ deserves it

OST
Ostschweizer
Fachhochschule

Quick Review

- **Multiple async operations can be in flight**
  - E.g. reading from multiple sockets
- **All completion handlers are dispatched through `asio::io_context`**
  - Handlers run on a thread executing `io_context.run()`
  - Multiple threads can call `run()` on the same `asio::io_context`!
- **What could go wrong?**

```
int main() {
  auto context = asio::io_context{};

  // start some async operations

  auto runners = std::vector<std::thread>{};

  for(int i{}; i < 4; ++i) {
    runners.push_back(std::thread{[&]{
      context.run();
    }});
  }

  for_each(runners.begin(), runners.end(),
           [](auto & runner){
             runner.join();
  });
}
```

```cpp
// globally accessible

auto results = std::vector<int> { };

// in connection class

asio::async_read(socket, asio::buffer(buffer), [&](auto err, auto bytes) {
  auto result = parse(buffer);
  results.push_back(result);
});
```

```
// globally accessible

auto results = std::vector<int> { };

// in connection class

asio::async_read(socket, asio::buffer(buffer), [&](auto err, auto bytes) {
  auto result = parse(buffer);
  results.push_back(result); // <<< POSSIBLE DATA RACE!!!
});
```

- **Strands are a mechanism to ensure sequential execution of handlers**
  - Implicit Strands
    - if only one thread calls `io_context.run()`
    - or program logic ensures only one operation is in progress at a time
  - Explicit Strands
    - Objects of type `asio::strand<…>`
    - Created using `asio::make_strand(executor)`
    - Or `asio::make_strand(execution_context)`
    - Applied to handlers using `asio::bind_executor(strand, handler)`

```
// globally accessible

auto results = std::vector<int> { };
auto strand = asio::make_strand(context);

// in connection class

asio::async_read(socket, asio::buffer(buffer),
    asio::bind_executor(strand, [&](auto err, auto bytes) {
      auto result = parse(buffer);
      results.push_back(result); // <<< No more data race
}));
```

Summary

Cevelop
Your C++ deserves it

OST
Ostschweizer
Fachhochschule

Summary

- **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**