boost::asio

Asynchronous network programming in C++

Why `boost::asio`?

- Because it's “standard”
- Cross-platform
- Asynchronous!
- Because low-level sockets are %&#!

Once, the world was blocking...

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <stdio.h>

int sockfd = socket(AF_INET,SOCK_STREAM,0);
struct sockaddr_in servaddr;
bzero(&servaddr,sizeof(servaddr));
servaddr.sin_family = AF_INET;
servaddr.sin_addr.s_addr = inet_addr("127.0.0.1");
servaddr.sin_port = htons(4711);

connect(sockfd, (struct sockaddr*)&servaddr, sizeof(servaddr));
[...]
```

What about OO? What about timeouts?
Can I print a `sockaddr`?
How many threads do I need for every connection?
How about other OSes?

This is madness!

This is ASIO!

- Object oriented: it's C++!
- It uses namespaces and templates instead of cryptic constants

A blocking world with `asio`...

```
#include "boost/asio.hpp"
namespace ip = boost::asio::ip;
using boost::asio::tcp;

boost::asio::io_service io_service;
tcp::socket s(io_service);
tcp::endpoint endpoint(ip::address("127.0.0.1"), 4711);

boost::asio::connect(s, endpoint);
boost::asio::write(s, /* my data here */);
size_t n = boost::asio::read(s, /* my buffer here */);
```

What about timeouts / threads / ...?

This is ASIO!

- Code with the Hollywood Principle:

  Don't call us, we'll call you!

- Just let me know when anything happens on my sockets...

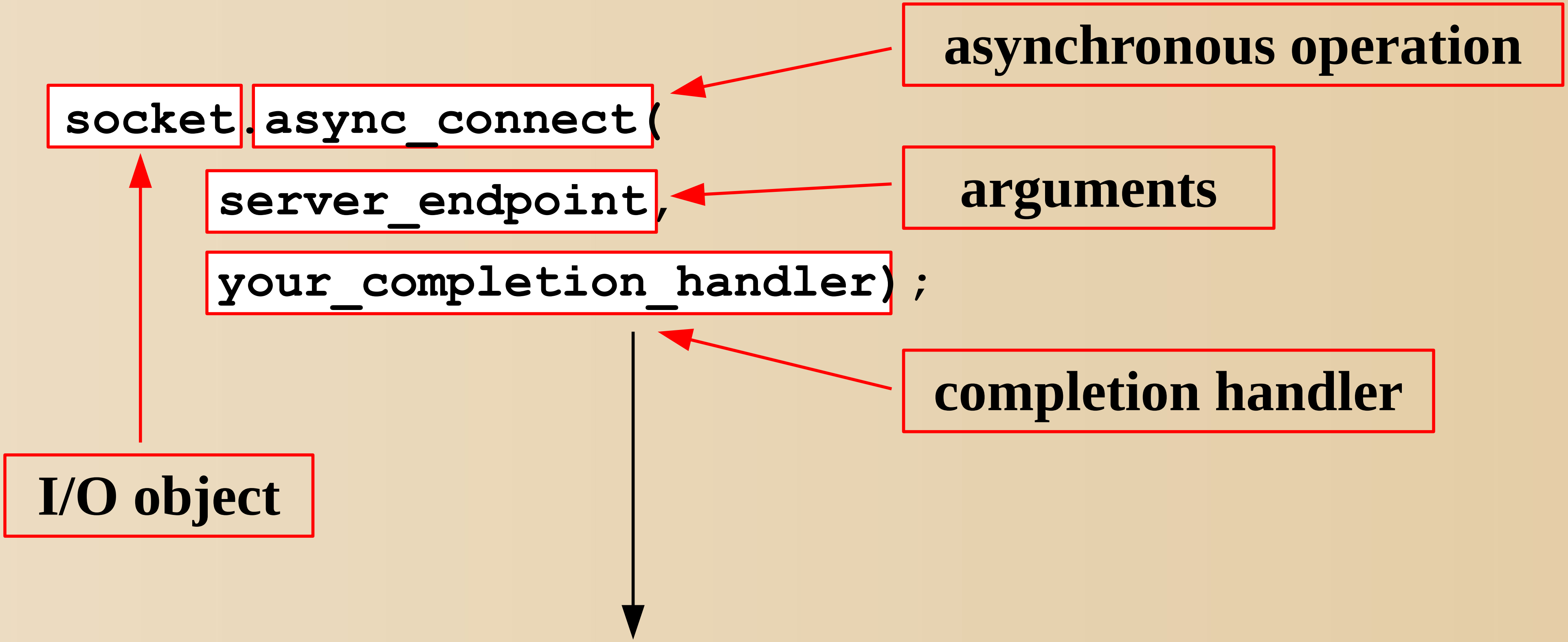
Asynchronous TCP server

```
#include "boost/asio.hpp"
[…]
using boost::system::error_code;

boost::asio::io_service io_service;
tcp::endpoint ep(ip::v4(), 4711)
tcp::acceptor acceptor(io_service, ep);

startAccept(acceptor, io_service);

io_service.run();
```

Find the bug!

```
void startAccept(...)
{
    tcp::socket sock(io_service);
    acceptor.async_accept(sock,
    [&](const error_code& ec)
    {
        handleAccept(sock, ec);
        if (!ec) startAccept(sock);
    });
}
```

The basics

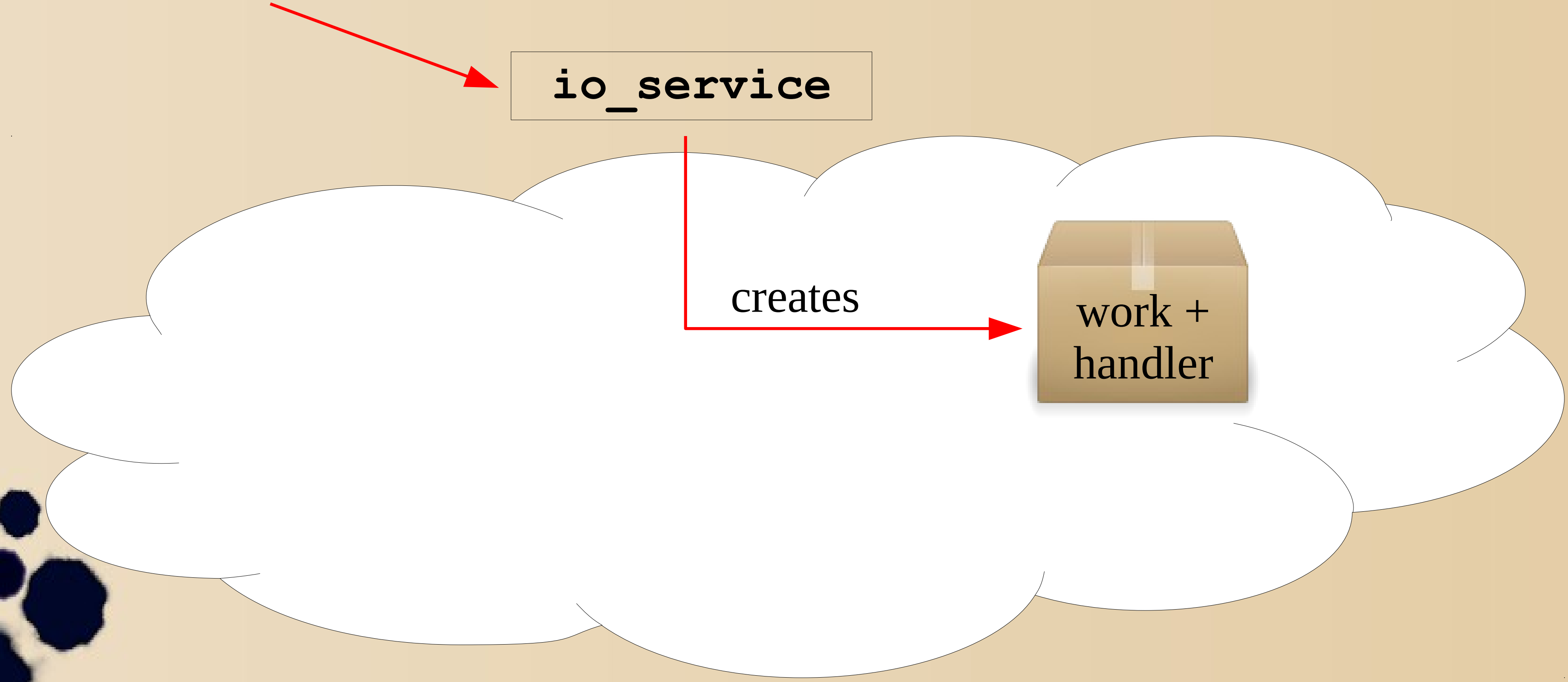


```
socket.async_connect(
    server_endpoint,
    your_completion_handler);
```

```
void your_completion_handler(
      const boost::system::error_code& ec);
```

The basics

```
socket.async_connect(
        server_endpoint,
        your_completion_handler);
```

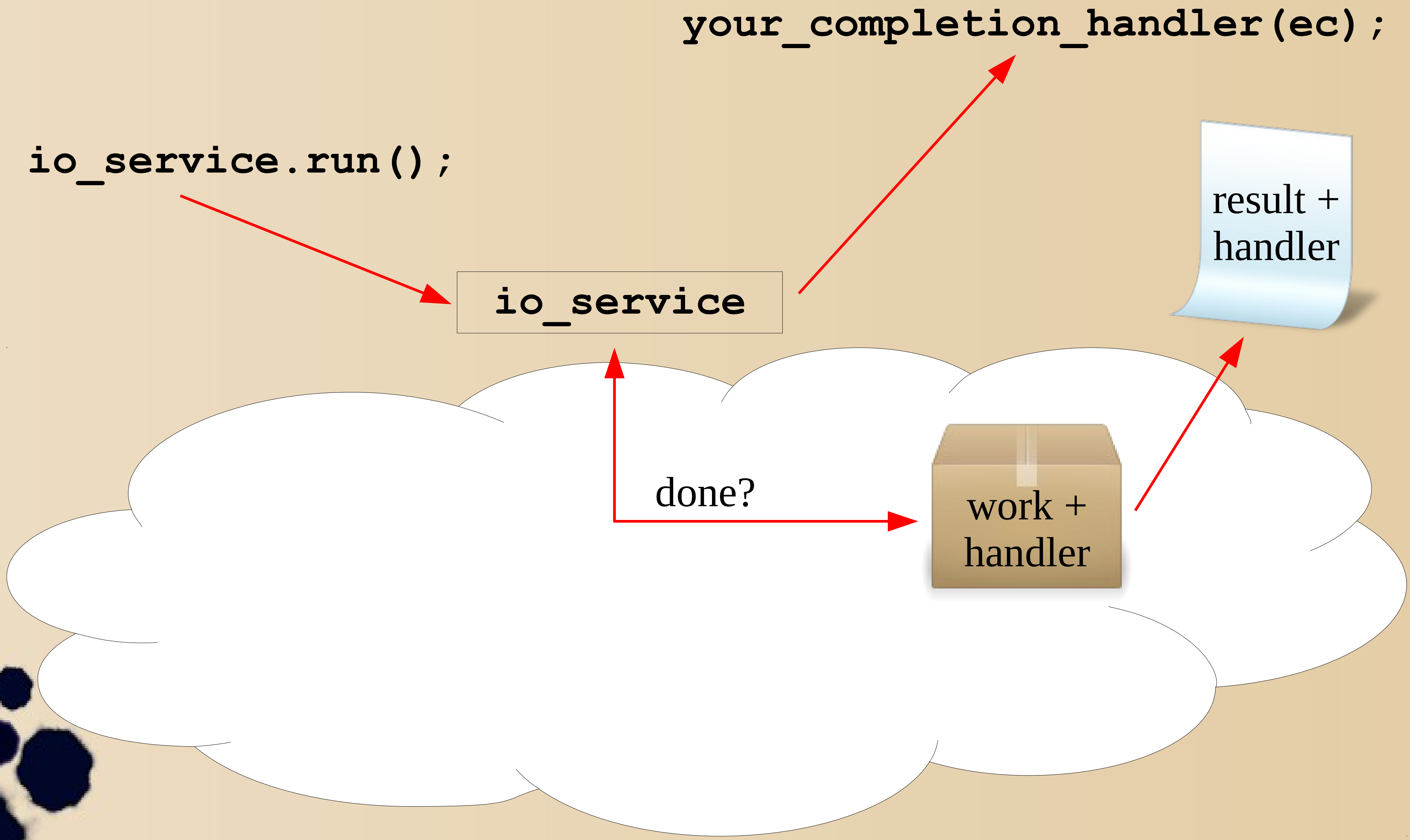


io_service

creates

work + handler

The basics



your_completion_handler(ec);

io_service.run();

result + handler

io_service

done?

work + handler

`asio` classes

• `socket`	I/O for TCP/UDP/...
• `acceptor`	Server-side listener
• `endpoint`	Connection end
• `resolver`	DNS resolver
• `deadline_timer`	Timeout handling
• `io_service`	Work management

Sockets

- Object functions:
  - `async_connect(endpoint, handler);`
  - `async_read_some(buffer, handler);`
  - `async_write_some(buffer, handler);`
  - `close();`
  - `...`

Free functions

- `boost::asio::async_read`
- `boost::asio::async_read_until`
- `boost::asio::async_write`
- `boost::asio::async_connect`

Acceptor

- Object functions:
  - bind(endpoint);
  - listen();
  - async_accept(socket, endpoint,
    handler);
  - close();
  - ...

Endpoint

- Object functions:
  - `address();`
  - `port();`
  - `protocol();`
  - `...`

Resolver

- Object functions:
  - `async_resolve( endpoint_or_query, handler);`
  - `...`
- Types:
  - `query`
  - `...`

Deadline timer

- Alternatives: `high_resolution_timer`, `steady_timer`, `system_timer`
- Object functions:
  - `expires_at(absolute_time);`
  - `expires_from_now(delta);`
  - `async_wait(handler);`
  - `cancel();`

Timeout example

```
deadline_timer timer(io_service);
timer.expires_from_now(boost::posix_time::seconds(3));
timer.async_wait(handle_timer);

socket.async_read_some(mybuffer, handle_read);

io_service.run();
```

```
void handle_timer(const error_code& ec)
{
   if (!ec)
   {
      std::cout << "Ooops, timeout!\n";
      socket.close();
   }
}
```

```
void handle_read(const error_code& ec,
      std::size_t bytes_transferred)
{
   timer.cancel();
   // process data ...
}
```

Io_service

- Object functions:
  - `run();`
  - `stop();`
  - `post(handler);`
  - `dispatch(handler);`
  - `...`

Challenge: object lifetimes

- Handlers are taken by value
- Sockets, endpoints, buffers etc. are taken by (const) reference

Find the bug:

```
void send()
{
   std::string message = "Hello\n";
   async_write(socket,
                buffer(message),
                completion_handler);
}
```

Object lifetimes

Find the bug:

```
class Connection
{
   tcp::socket mSocket;
   std::vector<char> mData;
   // ...

   ~Connection()
   {
      mSocket.close();
   }
};
```

Solution: shared pointers

```
class Connection:
        enable_shared_from_this<Connection>
{
   tcp::socket mSocket;
   std::vector<char> mData;
   // ...
   void do_write()
   {
      async_write(mSocket, asio::buffer(mData),
         bind(&Connection::handle_write,
              shared_from_this(), _1, _2));
   }

};
```

Solution: shared pointers

```
void Connection::stop()
{
  mSocket.close();
}
void Connection::start()
{
  auto self = shared_from_this();
  mSocket.async_connect(mEndpoint,
     [this, self](const error_code& ec)
        { handle_connect(ec); }
  );
}

make_shared<Connection>(...)->start();
```

Threads

- 2 basic approaches:
  - Single-threaded
  - One `io_service`, multiple threads
- Extensions:
  - Additional background thread
  - Multiple `io_service` objects, one thread each

Single-threaded approach

- Easiest solution
- Preferred starting point when learning `boost::asio`
- Remember to keep handler functions short and non-blocking
- Just call `io_service::run()` in a thread

Caveats

- `io_service::run()` terminates when:
  - it runs out of work
  - `stop()` is called
- Avoid this by adding “work”:

```
io_service io_service;
io_service::work work(io_service);
io_service.run();
```

Multi-threaded `io_service`

- Handlers can be called from any thread
- Synchronize logic in “strands”
  - `io_service::strand`: wraps handler functions to serialize their execution
  - Avoids explicit locking with mutexes

Using background threads

- Run long-running jobs in another thread
- Pass the result back to the main thread when done
- Make sure the `io_service` doesn't run out of work

Background thread #1

```
class Connection:
         enable_shared_from_this<Connection>
{
   io_service& mIoService;
   // ...
   void start_job()
   {
      auto self = shared_from_this();
      io_service::work work(mIoService);
      mThread = new std::thread(
         [this, self, work]() {
            run_job(work);
         }
      );
   }
};
```

Background thread #2

```
class Connection:
       enable_shared_from_this<Connection>
{
  void run_job(const io_service::work&)
  {
    // … long running task …
    auto self = shared_from_this();
    mIoService.post(
      [this, self]() {
        work_done(/*result*/);
      }
    );
  }

};
```

Multiple `io_services`

- Communicate via “message passing”
- Keep logic in the “home” thread
  - via `post()` or `dispatch()`

Thank you!

Questions?