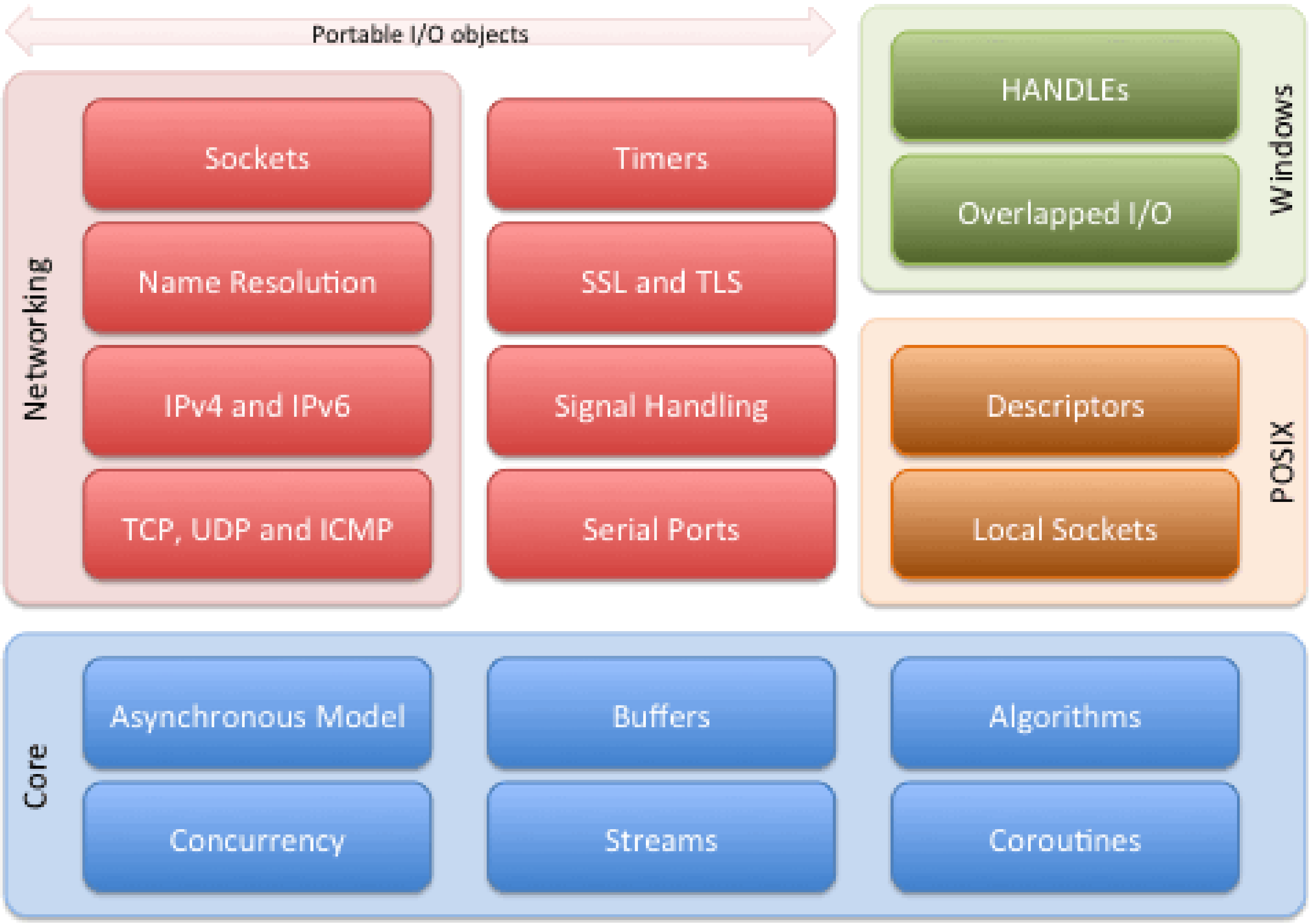
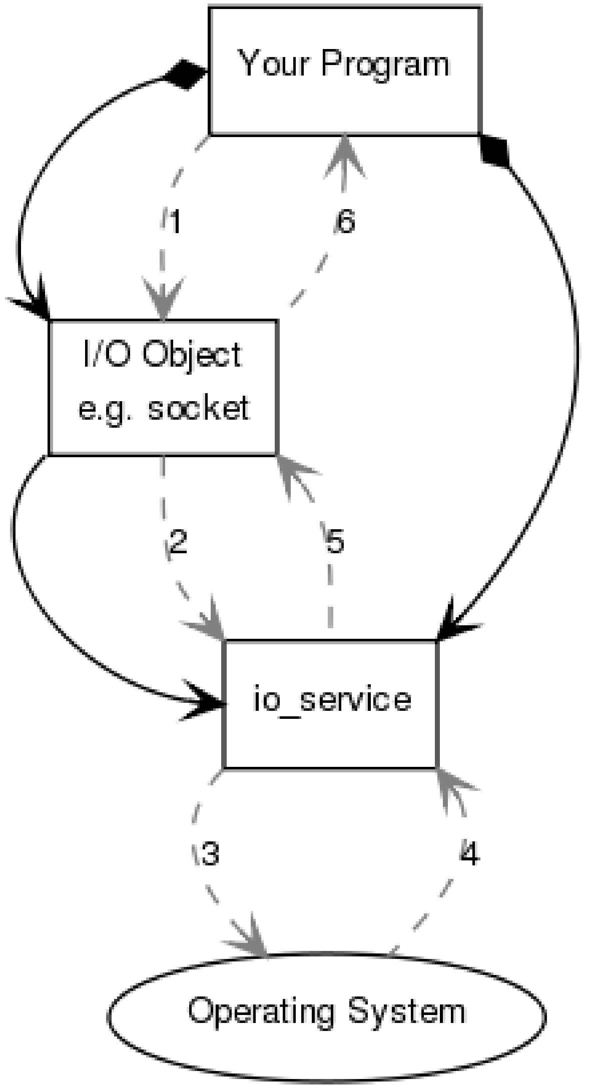
boost::asio

Лекция

boost::asio


Portable I/O objects
Networking
Sockets
Name Resolution
IPv4 and IPv6
TCP, UDP and ICMP
Timers
SSL and TLS
Signal Handling
Serial Ports
HANDLEs
Overlapped I/O
Windows
Descriptors
Local Sockets
POSIX
Core
Asynchronous Model
Concurrency
Buffers
Streams
Algorithms
Coroutines

boost::asio : синхронный режим



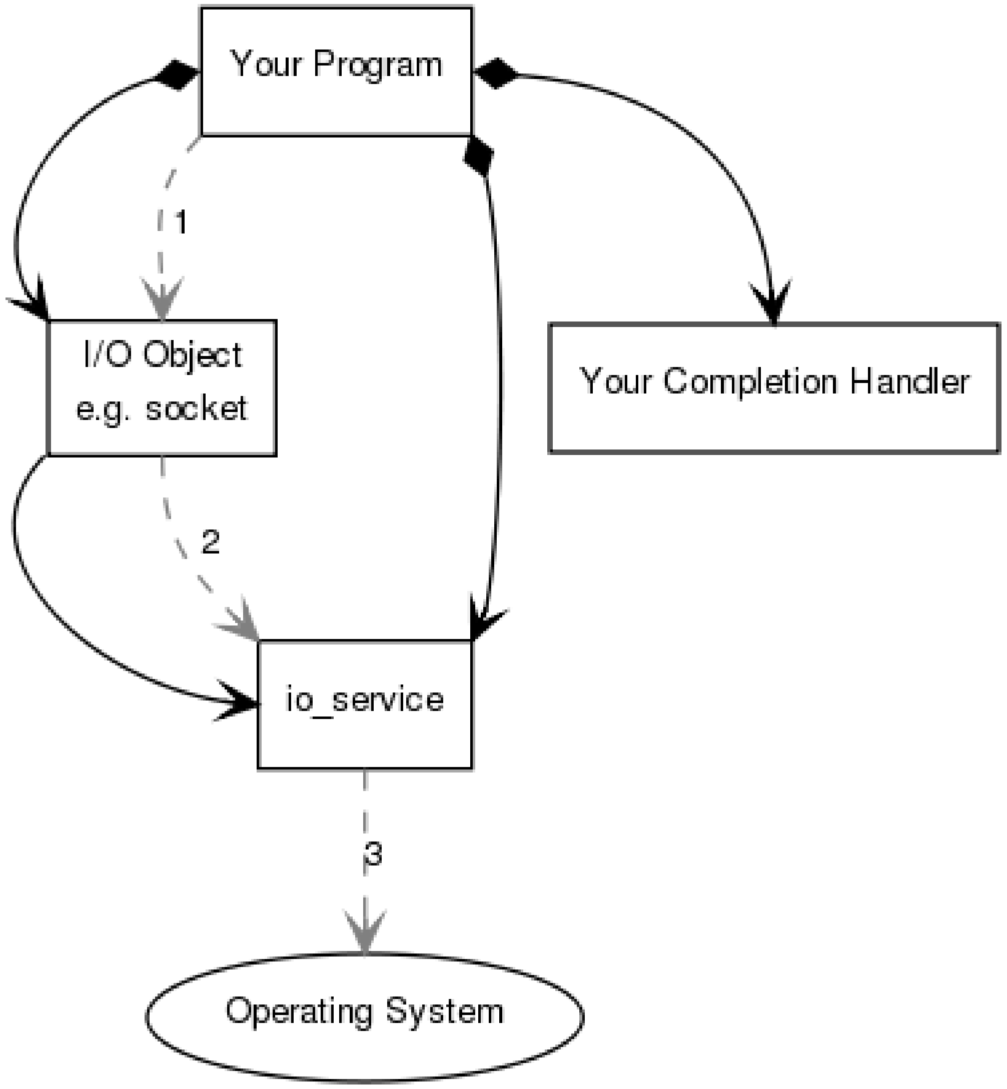
boost::asio

Синхронная работа.

```
boost::asio::io_service io_service;
boost::asio::ip::tcp::socket socket(io_service);

boost::system::error_code ec;
socket.connect(server_endpoint, ec);
```

boost::asio : асинхронный режим



boost::asio

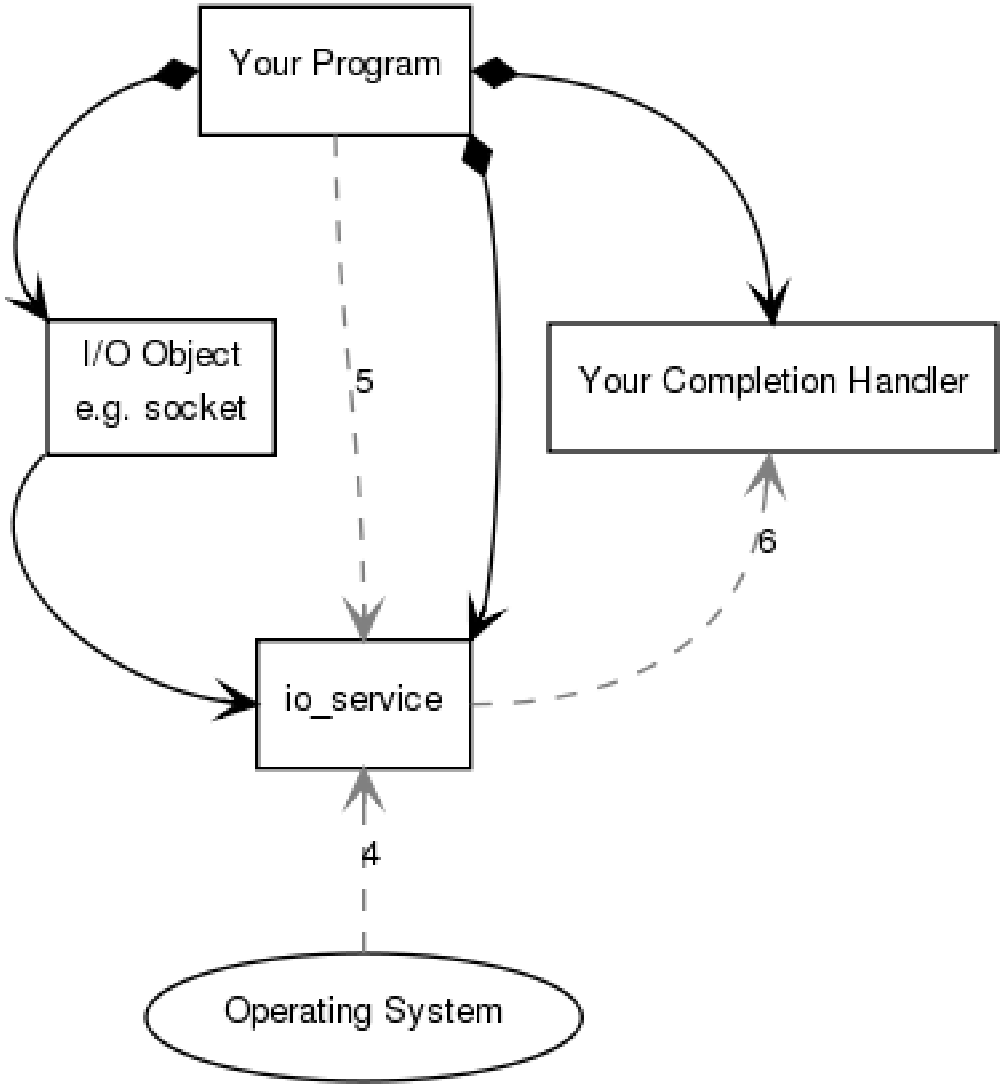
Асинхронная работа.

```
void your_completion_handler(const boost::system::error_code &ec)
{
    /* ... */
}

boost::asio::io_service io_service;
boost::asio::ip::tcp::socket socket(io_service);
socket.async_connect(server_endpoint, your_completion_handler);

io_service.run();
```

boost::asio


Your Program
I/O Object
e.g. socket
5
Your Completion Handler
6
io_service
4
Operating System

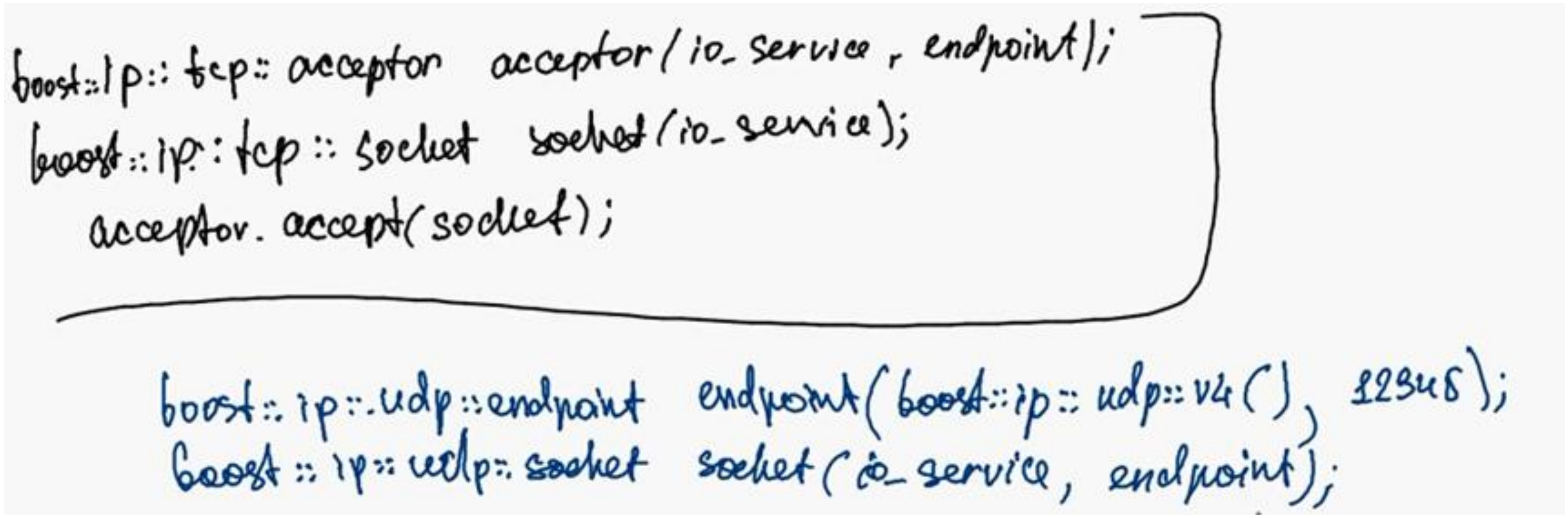
boost::asio

std::bind (boost::bind).

```
double my_divide (double x, double y) {
    return x/y;
}

int main ()
{
    using namespace std::placeholders;
    auto fn_five = std::bind (my_divide,10,2); // returns 10/2
    auto fn_half = std::bind (my_divide,_1,2); // returns x/2
    auto fn_invert = std::bind (my_divide,_2,_1); // returns y/x
    auto fn_rounding = std::bind<int> (my_divide,_1,_2); // returns int(x/y)
}
```

boost::asio : UDP



```
boost::ip::tcp::acceptor acceptor(io_service, endpoint);
 boost::ip::tcp::socket socket(io_service);
   acceptor.accept(socket);

      boost::ip::udp::endpoint endpoint(boost::ip::udp::v4(), 12345);
       boost::ip::udp::socket socket(io_service, endpoint);
```

boost::asio

accept

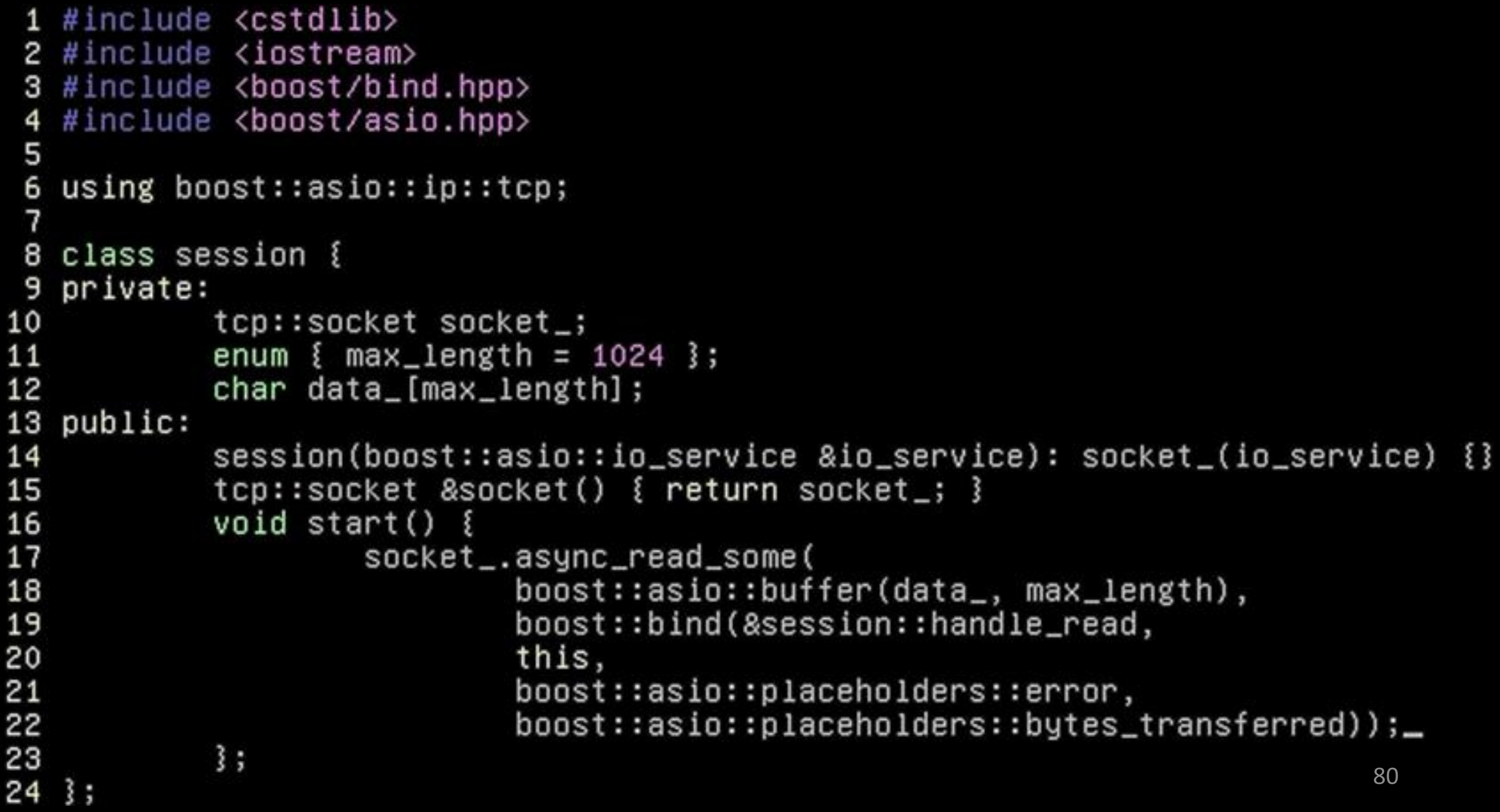
```
ip::tcp::acceptor acceptor(my_io_service, my_endpoint);
ip::tcp::socket socket(my_io_service);
acceptor.accept(socket);
```

boost::asio

TCP-Сокет.

```
ip::udp::endpoint endpoint(ip::udp::v4(), 12345);
ip::udp::socket socket(my_io_service, endpoint);
```

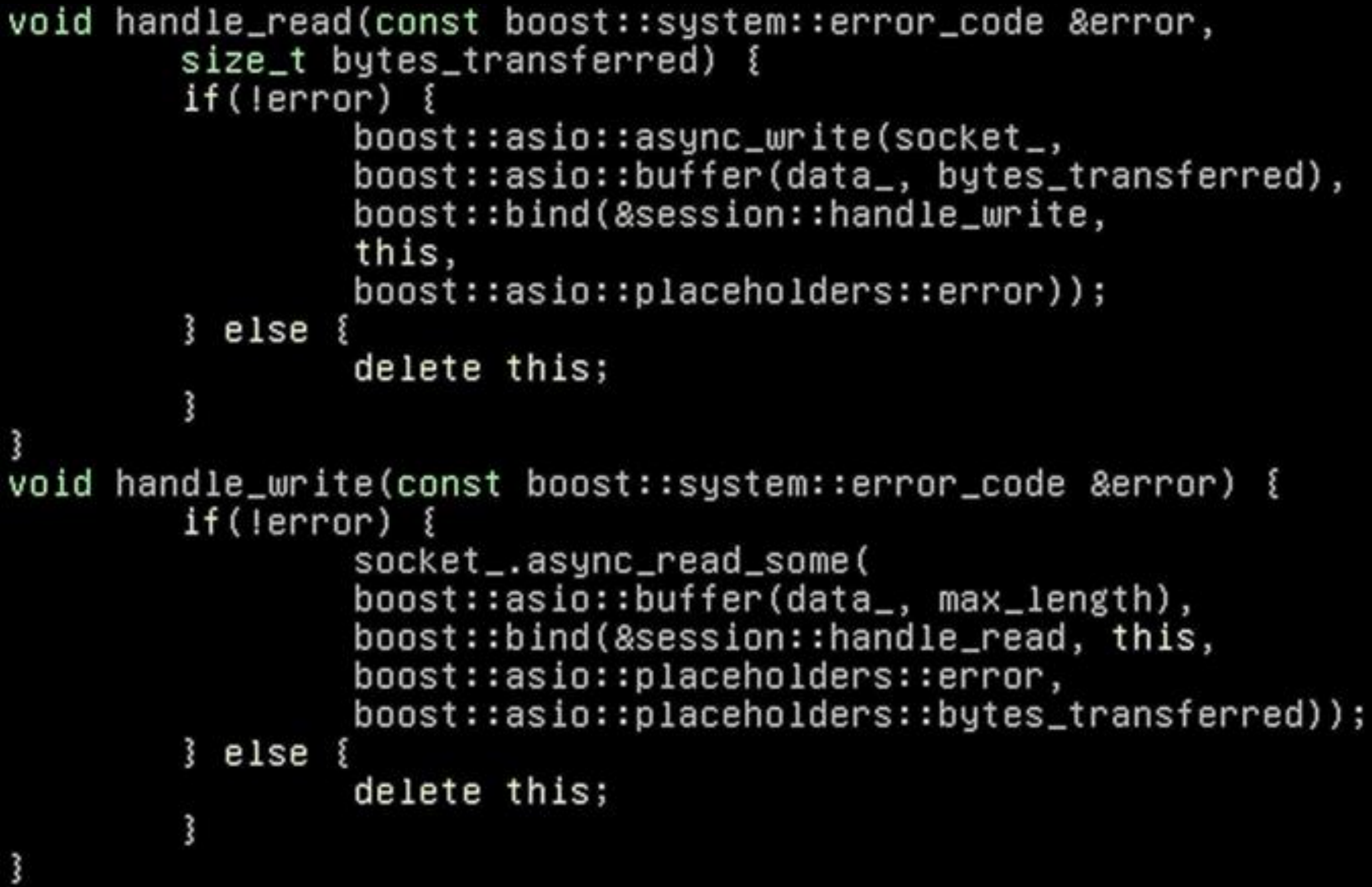
boost::asio : tcp session



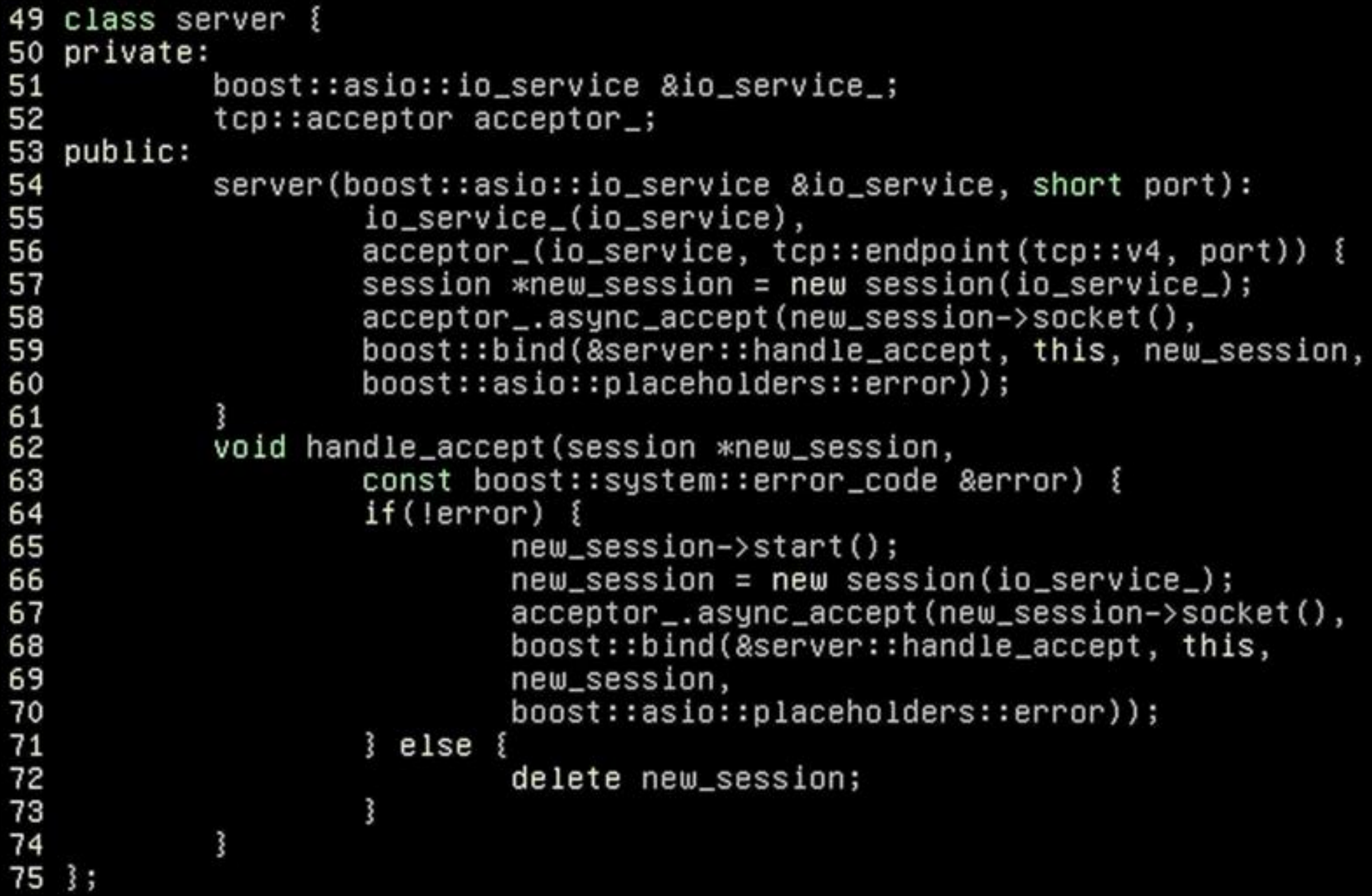
```
#include <cstdlib>
#include <iostream>
#include <boost/bind.hpp>
#include <boost/asio.hpp>

using boost::asio::ip::tcp;

class session {
private:
        tcp::socket socket_;
        enum { max_length = 1024 };
        char data_[max_length];
public:
        session(boost::asio::io_service &io_service): socket_(io_service) {}
        tcp::socket &socket() { return socket_; }
        void start() {
                socket_.async_read_some(
                        boost::asio::buffer(data_, max_length),
                        boost::bind(&session::handle_read,
                        this,
                        boost::asio::placeholders::error,
                        boost::asio::placeholders::bytes_transferred));
        };
};
```

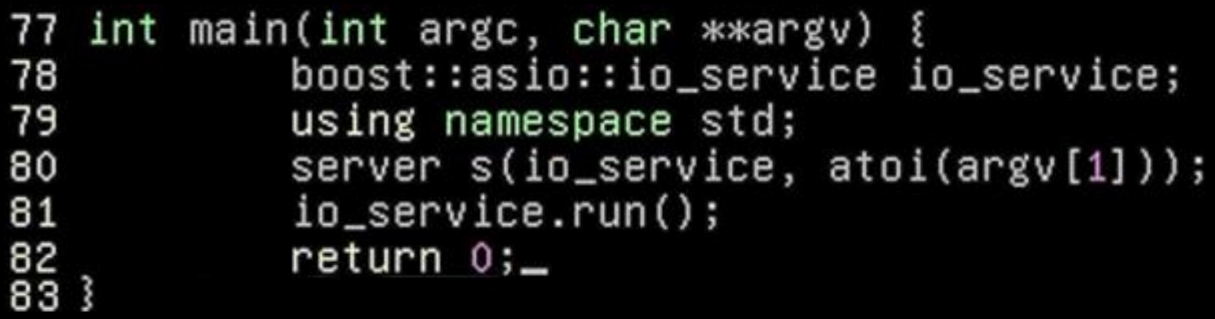


```
void handle_read(const boost::system::error_code &error,
        size_t bytes_transferred) {
        if(!error) {
                boost::asio::async_write(socket_,
                boost::asio::buffer(data_, bytes_transferred),
                boost::bind(&session::handle_write,
                this,
                boost::asio::placeholders::error));
        } else {
                delete this;
        }
}
void handle_write(const boost::system::error_code &error) {
        if(!error) {
                socket_.async_read_some(
                boost::asio::buffer(data_, max_length),
                boost::bind(&session::handle_read, this,
                boost::asio::placeholders::error,
                boost::asio::placeholders::bytes_transferred));
        } else {
                delete this;
        }
}
```



```
class server {
private:
        boost::asio::io_service &io_service_;
        tcp::acceptor acceptor_;
public:
        server(boost::asio::io_service &io_service, short port):
                io_service_(io_service),
                acceptor_(io_service, tcp::endpoint(tcp::v4, port)) {
                session *new_session = new session(io_service_);
                acceptor_.async_accept(new_session->socket(),
                boost::bind(&server::handle_accept, this, new_session,
                boost::asio::placeholders::error));
        }
        void handle_accept(session *new_session,
                const boost::system::error_code &error) {
                if(!error) {
                        new_session->start();
                        new_session = new session(io_service_);
                        acceptor_.async_accept(new_session->socket(),
                        boost::bind(&server::handle_accept, this,
                        new_session,
                        boost::asio::placeholders::error));
                } else {
                        delete new_session;
                }
        }
};
```

boost::asio : main



```
int main(int argc, char **argv) {
        boost::asio::io_service io_service;
        using namespace std;
        server s(io_service, atoi(argv[1]));
        io_service.run();
        return 0;
}
```

boost::asio

TCP-Клиент.

```
// Синхронно
ip::tcp::socket socket(my_io_service);
boost::asio::connect(socket, resolver.resolve(query));

// Асинхронно
boost::asio::async_connect(socket_, iter,
    boost::bind(&client::handle_connect, this,
        boost::asio::placeholders::error));
void handle_connect(const error_code& error)
{
    if (!error) {
        // Start read or write operations.
    } else {
        // Handle error.
    }
}
```

boost::asio

DNS.

```
ip::tcp::resolver resolver(my_io_service);
ip::tcp::resolver::query query("www.boost.org", "http");
ip::tcp::resolver::iterator iter = resolver.resolve(query);
ip::tcp::resolver::iterator end; // End marker
while (iter != end)
{
    ip::tcp::endpoint endpoint = *iter;
    iter++;
    std::cout << endpoint << std::endl;
}
```

boost::asio

Поток.

```
ip::tcp::iostream stream;
stream.expires_from_now(boost::posix_time::seconds(60));
stream.connect("www.boost.org", "http");
stream << "GET /LICENSE_1_0.txt HTTP/1.0\r\n”;
stream << "Host: www.boost.org\r\n”;
stream << "Accept: */*\r\n”;
stream << "Connection: close\r\n\r\n”;
stream.flush();
std::cout << stream.rdbuf();
```