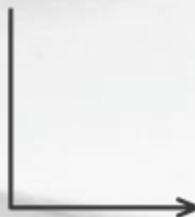




Углубленное программирование на языке C / C++

Лекция № 8



Алексей Халайджи

Лекция №8. Введение в Boost. Особенности новых стандартов. Функциональное и асинхронное программирование. Работа с внешним миром




1. Состав и назначение Boost.
2. Примеры использования Boost: проверки времени компиляции, обобщённые контейнеры, «умные» указатели.
3. Элементы функционального программирования.
4. Подходы к написанию асинхронных программ
5. Работа с сетью при помощи Boost
6. Сериализация данных. Работа с конфигурационными файлами
7. Работа с файловой системой

Отметьтесь на портале



- Посещение необязательное, но тем, кто пришёл, следует отмечаться на портале в начале каждого занятия
- Это позволяет нам анализировать, какие занятия были более или менее интересны студентам, и менять курс в лучшую сторону
- Также это даст возможность вам оставить обратную связь по занятию после его завершения

The screenshot shows a portal interface with a calendar of events. The calendar has tabs for dates from September 23 to 29. The selected date is September 24. A pop-up window for a lecture is displayed, showing the lecturer's name and photo, the lecture title, a description, the time, and the location. An arrow points to the 'Отметиться' (Mark) button at the bottom of the pop-up.

вс, 23 Сентября	пн, 24 Сентября	вт, 25 Сентября	ср, 26 Сентября	чт, 27 Сентября	пт, 28 Сентября	сб, 29 Сентября
Занятий нет.	 Алексей Халайдж Углубленное программирование на C/C++ Лекция: Цели и задачи курса. Организация и использование оперативной и сверхоперативной памяти в программах на языке C. 🕒 18:00-21:00 📍 433 🎓 АПО-11, АПО-12, АПО-13	Занятий нет.	Занятий нет.	Занятий нет.	Занятий нет.	13:00 395 - зал 1 2 3 Углубленное програ...

Библиотека Boost: общие сведения



Boost — набор из порядка 150 автономных библиотек на языке C++, задуманный в 1998 г. Б. Давесом (Beman Dawes), Д. Абрахамсом (David Abrahams) и др.



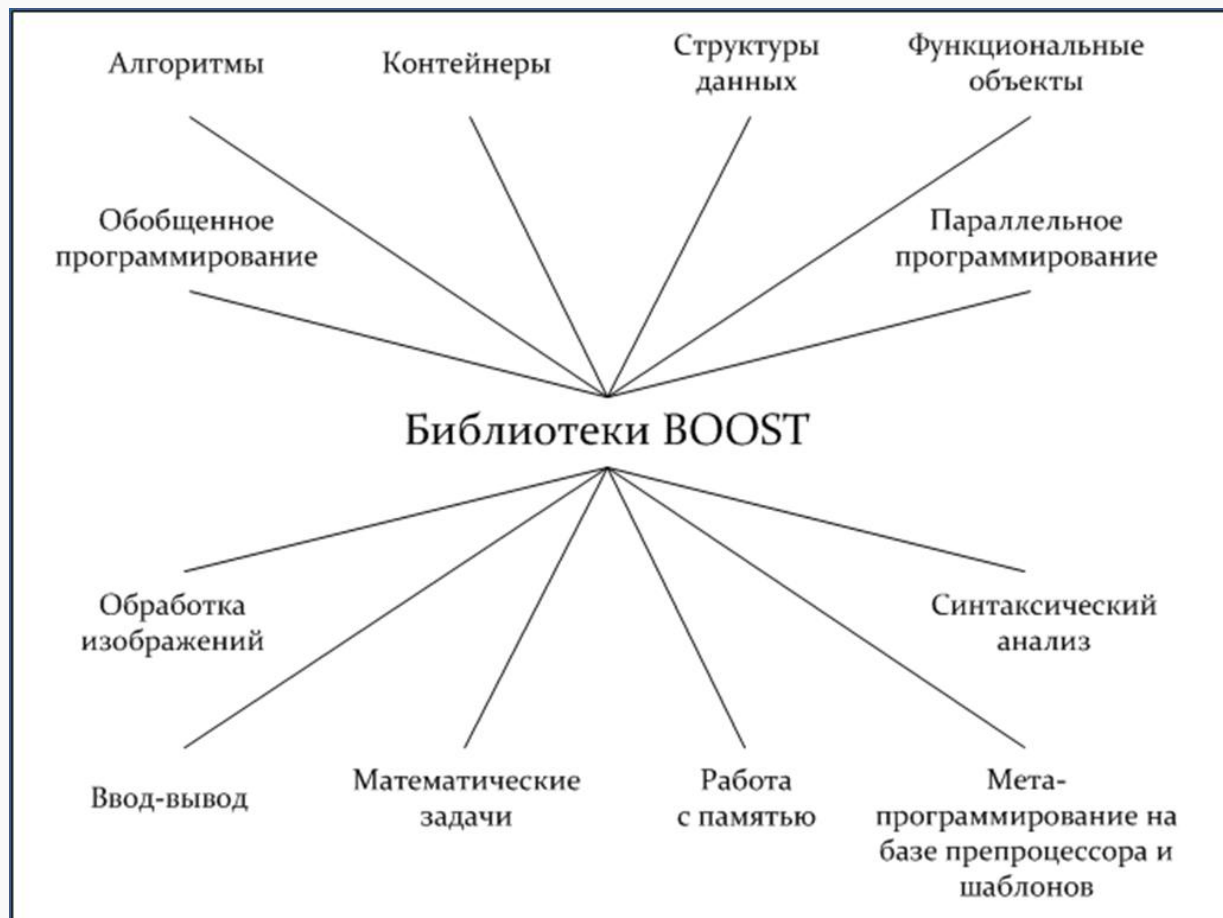
Последняя стабильная версия – 1.68.0 (от 9 августа 2018)

Основными **целями разработки** Boost выступают:

- решение задач, выходящих за пределы возможностей стандартной библиотеки C++ в целом и STL — в частности;
- тестирование новых библиотек-кандидатов на включение в принимаемые и перспективные стандарты языка C++.

Преимущества и недостатки Boost связаны с активным использованием в Boost шаблонов и техник обобщенного программирования, что открывает перед программистами новые возможности, но требует немалой предварительной подготовки.

Состав и назначение Boost



Проверки времени компиляции: общие сведения



Цель. Проверки времени компиляции ([англ.](#) static assertions) призваны предупредить случаи некорректного использования библиотек, ошибки при передаче параметров шаблонам и пр.

Библиотека.

```
#include <boost/static_assert.hpp>
```

Состав. Проверки времени компиляции представляют собой два макроопределения ([x](#) — целочисленная константа, [msg](#) — строка):

```
BOOST_STATIC_ASSERT(x)
```

```
BOOST_STATIC_ASSERT_MSG(x, msg)
```

являются **статическим аналогом** стандартного макроопределения [assert](#) и пригодны для применения **на уровне пространства имен, функции или класса**.

Примечание: появилось в стандарте C++11

Проверки времени компиляции: реализация



Реализация. На уровне программной реализации в Boost макроопределения `BOOST_STATIC_ASSERT*` задействуют общий и полностью специализированный шаблон структуры вида:

```
namespace boost {  
    template <bool>  
    struct STATIC_ASSERTION_FAILURE;  
  
    template <>  
    struct STATIC_ASSERTION_FAILURE<true> {};  
}
```



Проверки времени компиляции: использование



```
#include <climits>
#include <limits>
#include <boost/static_assert.hpp>

namespace my_conditions {
    // проверка: длина int - не менее 32 бит
    BOOST_STATIC_ASSERT(
        std::numeric_limits<int>::digits >= 32);
}
```


Вариантный контейнер: общие сведения



Цель. Предоставление безопасного обобщенного контейнера-объединения различных типов со следующими возможностями:

- поддержка семантики значений, в том числе стандартных правил разрешения типов при перегрузке;
- безопасное посещение значений с проверками времени компиляции посредством `boost::apply_visitor()`;
- явное извлечение значений с проверками времени выполнения посредством `boost::get()`;
- поддержка любых типов данных (POD и не-POD), отвечающих минимальным требованиям ([см. далее](#)).

Библиотека.

```
#include <boost/variant.hpp>
```

Состав. Шаблон класса `boost::variant` с переменным числом параметров, сопутствующие классы и макроопределения.

Примечание: появилось в стандарте C++17

Вариантный контейнер: требования к типам-параметрам



Обязательные характеристики типов-параметров шаблона `boost::variant`:

- наличие конструктора копирования;
- соблюдение безопасной по исключениям гарантии `throw()` для деструктора;
- полнота определения к точке инстанцирования шаблона `boost::variant`.

Желательные характеристики типов-параметров шаблона `boost::variant`:

- возможность присваивания (*отсутствует для константных объектов и ссылок!*);
- наличие конструктора по умолчанию;
- возможность сравнения по отношениям «равно» и «меньше»;
- поддержка работы с выходным потоком `std::ostream`.



Вариантный контейнер: определение и обход элементов



```
// создание и использование экземпляра
boost::variant<int, std::string> u("hello world");
std::cout << u << std::endl; // выдача: hello world

// для безопасного обхода элементов контейнера
// может использоваться объект класса-посетителя
// (см. далее):

boost::apply_visitor(
    times_two_visitor(), // объект-посетитель
    v                    // контейнер
);
```



Вариантный контейнер: класс-посетитель (1 / 2)



```
class times_two_visitor : public boost::static_visitor<> {
public:
    void operator()(int& i) const {
        i *= 2;
    }

    void operator()(std::string& str) const {
        str += str;
    }
};
```



Вариантный контейнер: класс-посетитель (2 / 2)



```
// реализация класса-посетителя может быть обобщенной
class times_two_generic : public boost::static_visitor<> {
public:
    template <typename T>
    void operator()(T& operand) const {
        operand += operand;
    }
};
```

Произвольный тип: общие сведения



Цель. Предоставление безопасного обобщенного класса-хранилища единичных значений любых различных типов, в отношении которых не предполагается выполнение произвольных преобразований. Основные возможности:

- копирование значений без ограничений по типам данных;
- безопасное проверяемое извлечение значения в соответствии с его типом.
- **Библиотека.**

```
#include <boost/any.hpp>
```

Состав. Шаблон класса `boost::any`, сопутствующие классы, в том числе производный от `std::bad_cast` класс `boost::bad_any_cast`, и другие программные элементы.

Примечание: появилось в стандарте C++17



Произвольный тип: класс `boost::any`



```
class any {
public:
    // конструкторы, присваивания, деструкторы
    any();
    any(const any&);
    template<typename ValueType> any(const ValueType&);
    any& operator=(const any&);
    template<typename ValueType>
    any& operator=(const ValueType&);
    ~any();
    any& swap(any&); // модификатор
    bool empty() const; // запрос #1
    const std::type_info& type() const; // запрос #2
};
```



Произвольный тип: работа со стандартными списками



```
// двусвязный список значений произвольных типов
// может формироваться и использоваться так:
typedef std::list<boost::any> many;

void append_string(many& values, const std::string& value) {
    values.push_back(value);
}

void append_any(many& values, const boost::any& value) {
    values.push_back(value);
}

void append_nothing(many& values) {
    values.push_back(boost::any());
}
```




Произвольный тип: проверка типов значений



```
bool is_int(const boost::any& operand) {  
    return operand.type() == typeid(int);  
}  
  
bool is_char_ptr(const boost::any& operand) {  
    try {  
        boost::any_cast<const char *>(operand);  
        return true;  
    }  
    catch(const boost::bad_any_cast&) {  
        return false;  
    }  
}
```

Циклический буфер: общие сведения



Цель. Снабдить программиста STL-совместимым контейнером типа «кольцо» или «циклический буфер», который служит для приема поступающих данных, поддерживает перезапись элементов при заполнении, а также:

- реализует хранилище фиксированного размера;
- предоставляет итератор произвольного доступа;
- константное время вставки и удаления в начале и конце буфера;

Библиотека.

```
#include <boost/circular_buffer.hpp>
```

Состав. Шаблон класса `boost::circular_buffer`, адаптер `boost::circular_buffer_space_optimized` и другие программные элементы.

Циклический буфер: техника применения



Использование. Циклический буфер и его адаптированный вариант на физическом уровне работают с непрерывным участком памяти, в силу чего не допускают неявных или непредсказуемых запросов на выделение памяти.

Возможные применения буфера включают, в том числе:

- хранение последних полученных (обработанных, использованных) значений;
- создание программной кэш-памяти;
- реализацию ограниченных буферов ([англ. bounded buffer](#));
- реализацию FIFO/LIFO-контейнеров фиксированного размера.



Циклический буфер: порядок использования



```
boost::circular_buffer<int> cb(3);

cb.push_back(1);
cb.push_back(2);
cb.push_back(3);
// буфер полон, дальнейшая запись приводит
// к перезаписи элементов
cb.push_back(4); // значение 4 вытесняет 1
cb.push_back(5); // значение 5 вытесняет 2

// буфер содержит значения 3, 4 и 5
cb.pop_back(); // 5 выталкивается из посл. позиции
cb.pop_front(); // 3 выталкивается из нач. позиции
```



Boost::lexical_cast



```
#include <boost/lexical_cast.hpp>

void log_fail(int i) {
    std::cerr << boost::lexical_cast<std::string>(i) << std::endl;
}

int main(int argc, char *argv[]) {
    using boost::lexical_cast;
    using boost::bad_lexical_cast;
    std::vector<short> args;
    for (int i = 1; i < argc; ++i) try {
        args.push_back(lexical_cast<short>(argv[i]));
    } catch (const bad_lexical_cast &) {
        log_fail(i);
        args.push_back(0);
    }
    return 0;
}
```



Регулярные выражения



```
// Появились в стандарте C++11
#include <iostream>
#include <regex>

int main() {
    const char *fnames[] = {"foo.txt", "bar.jpg"};
    std::regex txt_regex("([a-z]+)\\.txt");
    std::cmatch m; // cmatch == match_results<const char *>
    for (const auto &fname : fnames) {
        std::cout << fname << ": "
            << std::regex_match(fname, txt_regex) << "; ";
        if (std::regex_match(fname, m, txt_regex)) {
            std::cmatch mcopy(m);
            for (auto i = 0; i < mcopy.size(); ++i) {
                std::cout << "match " << i << ": "
                    << mcopy[i] << "; ";
            }
        }
    }
}

// Выведет: foo.txt: 1; match 0: foo.txt; match 1: foo; bar.jpg: 0
```

Boost Interval Container Library



Boost Interval Container Library (ICL) — библиотека интервальных контейнеров с поддержкой множеств и отображений интервалов:

```
// работа с интервальным множеством
boost::interval_set<int> my_set;
my_set.insert(42);
bool has_answer = boost::contains(my_set, 42);
typedef boost::icl::interval_set<unsigned int> set_t;
typedef set_t::interval_type ival;
set_t outages;
outages.insert(ival::closed(1,1)); outages.insert(ival::open(7,10));
outages.insert(ival::open(8,11)); outages.insert(ival::open(90,120));
for(set_t::iterator it = outages.begin(); it != outages.end(); it++){
    std::cout << it->lower() << ", " << it->upper() << "\n";
}
```

Boost – многозначная логика и опциональное значение



Boost.Tribool — поддержка тернарной логики «да, нет, ВОЗМОЖНО»::

```
boost::tribool b(true);  
b = false;  
b = boost::indeterminate;
```

Boost.Optional — поддержка отсутствия значения объекта
(Примечание: появилось в стандарте C++17)

```
boost::optional<int> getConfigParam(std::string name);  
/* .. */  
if (boost::optional<int> oi = getConfigParam("MaxValue"))  
    runWithMax(*oi);  
else  
    runWithNoMax();
```


Boost - размерности



Boost.Units — библиотека поддержки анализа размерностей (единиц измерения) операндов вычислительных операций. Задача анализа рассматривается как обобщенная задача метапрограммирования времени компиляции:

```
quantity<force>  F(2.0 * newton); // сила
quantity<length> dx(2.0 * meter); // расстояние
quantity<energy>  E(work(F, dx));  // энергия
```

Примечание: в C++14 появилась возможность задавать пользовательские суффиксы как операторы. Например, в `std::literals` есть такие:

```
std::cout << "Hello"s << std::endl;
usleep(500us);
```

Математическими библиотеками Boost, в частности, выступают:

- **Geometry** — решение геометрических задач (например, вычисление расстояния между точками в сферической системе координат);
- **Math Toolkit** — работа со статистическими распределениями, специальными математическими функциями (эллиптическими интегралами, гиперболическими функциями, полиномами Эрмита и пр.), бесконечными рядами и др.;
- **Quaternions** — поддержка алгебры кватернионов;
- **Ratio** — поддержка рациональных дробей (ср. `std::ratio` в C++11);
- **Meta State Machine** — работа с автоматными структурами;
- и др.

Пакеты параметров шаблонов.

Развертывание пакетов (C++11)



Пакеты параметров шаблонов, подобно пакетам параметров функций, являются инструментом определения шаблонов с переменным числом параметров ([англ. variadic templates](#)).

Каждый пакет параметров ([англ. parameter pack](#)) вводит один параметр шаблона, способный в ходе конкретизации принимать один и более аргумент шаблона ([типов, значений или шаблонов](#)) либо не принимать ничего.

Развертывание пакета параметров в теле шаблона с переменным числом параметров носит название **образца** ([англ. pattern](#)). В ходе развертывания образец заменяется нулем или более экземплярами, разделенными запятыми и следующими в порядке их указания в качестве аргументов шаблона.

Если названия двух пакетов встречаются в одном образце, они должны быть одной длины, чтобы развертываться одновременно.



Пакеты параметров: пример (C++11)



```
// пакет параметров шаблона класса
template <class... Types> struct tuple {};

tuple<>          t0;
tuple<void*>     t1;
tuple<double, double> t2;

// пакет параметров шаблона функции
template <class... Types, int... N>
int foo(Types (&...arr)[N]) {}// параметр-троеточие в ()
int container[42];           // д.б. именован (CWG #1488)
int result = foo<const char, int>("42", container);
// Types (&...arr)[N] разворачивается в
// const char (&)[3], int(&)[42], см. также примеры далее
```



Развертывание пакетов параметров: пример (1 / 3, C++11)



```
template <typename...> struct tuple {};  
template <typename T, typename U> struct pair {};  
template <class... Args1> struct outer {  
    template <class... Args2> struct inner {  
        // pair<Args1, Args2>... - развертывание шаблона  
        // pair<Args1, Args2> - образец  
        typedef tuple<pair<Args1, Args2>...> type;  
    };  
};  
typedef outer<signed short, signed int>::  
    inner<unsigned short, unsigned int>::type T;  
// T есть tuple<pair<short, unsigned short>,  
// pair<int, unsigned int>>
```



Развертывание пакетов параметров: пример (2 / 3, C++11)



```
// в спецификаторах базовых классов и списках инициализации
template <class... Mixins>
class Provider : public Mixins... {
    Provider(const Mixins&... mixins) : Mixins(mixins)... {}
};

// в операторе sizeof...
template <class... Types> struct count {
    static const std::size_t value = sizeof...(Types);
};

// в спецификаторе динамических исключений
template <class... Exceptions>
void bad(int) throw(Exceptions...) { }
```



Развертывание пакетов параметров: пример (3 / 3, C++11)



```
// Perfect forwarding
template <class Type, class ...Args>
auto make_unique(Args &&... params) -> std::unique_ptr<Type>
{
    return std::unique_ptr<Type>(
        new Type(std::forward<Args>(params)...));
}

int a = 5;
auto p = make_unique<std::vector>(a, 42);
// p == std::unique_ptr<std::vector>(
//     new std::vector(std::forward<int> &(param1),
//                     std::forward<int>(param2)));
```

Выражения-свертки (C++17)



Выражения-свертки выполняют свертку (**редукцию**) пакетов параметров по одной из 32 разрешенных бинарных операций языка C++17. При этом различают:

- правую унарную свертку:

$$(\mathfrak{P} \circ \dots) \rightarrow p_1 \circ (\dots \circ (p_{N-1} \circ p_N))$$

- левую унарную свертку:

$$(\dots \circ \mathfrak{P}) \rightarrow ((p_1 \circ p_2) \circ \dots \circ) p_N$$

- правую бинарную свертку:

$$(\mathfrak{P} \circ \dots \circ I) \rightarrow p_1 \circ (\dots \circ (p_{N-1} \circ (p_N \circ I)))$$

- левую бинарную свертку:

$$(I \circ \dots \circ \mathfrak{P}) \rightarrow (((I \circ p_1) \circ p_2) \circ \dots \circ) p_N$$

Операция, свертка по которой осуществляется, имеет наивысший приоритет. Свертка пакетов нулевой длины разрешена только по *



Выражения-свертки: пример (C++17)



```
#include <iostream>

template <typename... Args> bool logProduct(Args... args) {
    return (... && args);    // левая унарная свертка
}

bool b = logProduct(true, true, true, false);
// ((true && true) && true) && false; b == false

template <typename... Args> void print(Args&&... args) {
    (std::cout << ... << args) << std::endl;
}

int main() {
    print(4, 2, " Forty-two");    // "42 Forty-two"
    return 0;
}
```



Пример: Кортеж – tuple (1 / 2)



```
// Вошли в стандарт C++11
#include <tuple>
#include <iostream>
typedef std::tuple<int, std::string, int> dbRecord;

dbRecord getRecordById(int id) {
    return std::make_tuple(id, "First", 42);
}

int main() {
    dbRecord first = getRecordById(1);
    print(dbRecord);
    int id; std::string name; int age;
    name = std::get<1>(dbRecord);
    std::cout << name << std::endl;
    std::tie(id, name, age) = first;
    id++;
    std::cout << id << name << age << std::endl;
    print(dbRecord);
    return 0;
}
```



Пример: Кортеж – tuple (2 / 2)



```
// C++17

template<class T, size_t... I>
void print(const T &t, std::index_sequence<I...>) {
    std::cout << std::tuple_size<T>::value << " <";
    (... , (std::cout << (I == 0 ? "" : ", ") << std::get<I>(t)));
    std::cout << ">" << std::endl;
}

template<class... T>
void print(const std::tuple<T...> &t) {
    print(t, std::make_index_sequence<sizeof...(T)>());
}
```

Соответствуют идиоме **RAII**.

Впервые появились в Boost в `<boost/smart_ptr/*.hpp>`

Некоторые из них перешли в стандарт C++11:

auto_ptr (deprecated – до C++17). **unique_ptr** – эксклюзивное владение объектом, move-семантика

shared_ptr – разделяемый объект, подсчёт ссылок

weak_ptr – слабая ссылка на разделяемый объект

```
template<class T> struct scoped_ptr {  
    explicit scoped_ptr(T *p = 0) : p(p) {}  
    ~scoped_ptr() { delete p; }  
    void reset(T *p = 0) { delete this->p; this->p = p; }  
    T *get() const { return p; }  
private:  
    scoped_ptr(scoped_ptr const &);  
    scoped_ptr operator=(scoped_ptr const &);  
    T *p;  
}
```



Пример: `unique_ptr`



```
#include <memory>

struct A {
    A(int a): a(a) {}
private:
    int a;
};

std::unique_ptr<A> foo() {
    return make_unique<A>(42);
}

// ...
auto p = foo();
auto r = p;    // ошибка компиляции, operator= и
               // конструктор копирования удалены
auto q = std::move(p);
```



Пример: `shared_ptr` (1 / 2)



```
#include <memory>

class A{}; class B{};
void f(std::shared_ptr<A> a, std::shared_ptr<B> b) {
    // счётчик ссылок на *a и *b увеличены на 1
    return 42;
}

void g(A *a, B *b) {
    std::shared_ptr<A> pa(a); // некорректно!
    std::shared_ptr<B> pb(b); // некорректно!
    f(pa, pb);
}

A *pa = new A; B *pb = new B;
g(pa, pb);
delete pa; delete pb;
```



Пример: `shared_ptr` (2 / 2)



```
#include <memory>

class A{}; class B{};
void f(std::shared_ptr<A> a, std::shared_ptr<B> b) {
    // счётчик ссылок на *a и *b увеличены на 1
    return 42;
}

// некорректно - возможна утечка ресурсов
f(std::shared_ptr<A>(new A), std::shared_ptr<B>(new B));

// корректно
f(std::make_shared<A>(), std::make_shared<B>());
```



Пример: retain cycle (1 / 2)



```
struct Child; // forward declaration
struct Parent {
    std::shared_ptr<Child> child;
    ~Parent() { std::cout << "~Parent()\n"; }
    void foo() { std::cout << "foo()\n"; }
};

struct Child {
    std::shared_ptr<Parent> parent;
    ~Child() { std::cout << "~Child()\n"; }
    void call_foo() {
        parent->foo();
    }
};

{
    auto p = std::make_shared<Parent>();
    auto c = std::make_shared<Child>();
    p->child = c; c->parent = p; c->call_foo();
}
```




Пример: retain cycle (2 / 2)



```
struct Child; // forward declaration
struct Parent {
    std::shared_ptr<Child> child;
    ~Parent() { std::cout << "~Parent()\n"; }
    void foo() { std::cout << "foo()\n"; }
};
struct Child {
    std::weak_ptr<Parent> parent;
    ~Child() { std::cout << "~Child()\n"; }
    void call_foo() {
        if (auto sp = parent.lock()) sp->foo();
    }
};
{
    auto p = std::make_shared<Parent>();
    auto c = std::make_shared<Child>();
    p->child = c; c->parent = p; c->call_foo();
}
```



Пример: enable_shared_from_this

(1 / 3)



```
struct A {
    std::shared_ptr<A> getPtr() {
        return std::shared_ptr<A>(this);
    }
};

int main() {
    std::shared_ptr<A> a(new A);
    std::shared_ptr<A> b = a->getPtr();
    std::cout << "b.use_count = " << b.use_count();
    // Выведет b.use_count = 1
    return 0;
}
```



Пример: `enable_shared_from_this` (2 / 3)



```
struct A : std::enable_shared_from_this<A> {
    std::shared_ptr<A> getPtr() {
        return shared_from_this();
    }
};

int main() {
    std::shared_ptr<A> a(new A);
    std::shared_ptr<A> b = a->getPtr();
    std::cout << "b.use_count = " << b.use_count();
    // Выведет b.use_count = 2
    return 0;
}
```



Пример: enable_shared_from_this (3 / 3). Идея реализации



```
template <typename T> class enable_shared_from_this {
protected:
    constexpr enable_shared_from_this() {}
    enable_shared_from_this(enable_shared_from_this const &) {}
    enable_shared_from_this &operator=(
        enable_shared_from_this const &) { return *this; }
public:
    std::shared_ptr<T> shared_from_this() { return self_.lock(); }
private:
    std::weak_ptr<T> self_;
    friend std::shared_ptr<T>;
};

template <typename T> shared_ptr<T>::shared_ptr(T *ptr) {
    /* ... */
    if (std::is_base_of<enable_shared_from_this<T>, T>::value) {
        enable_shared_from_this<T> &base = *ptr;
        base.self_ = *this;
    }
}
```

Элементы функционального программирования.

Указатели на функции



```
typedef int (*Comparator)(void const *lhs, void const *rhs);
int DoubleCmp(void const *lhs, void const *rhs) {
    double lhs_value = *static_cast<double const *>(lhs);
    double rhs_value = *static_cast<double const *>(rhs);
    if (lhs_value < rhs_value) return -1;
    if (lhs_value > rhs_value) return 1;
    return 0;
}
template <typename Iterator, typename CompareFunc>
void sort(Iterator begin, Iterator end, CompareFunc cmp) {
    qsort(begin, std::distance(begin, end), sizeof(*begin),
          cmp);
}

Comparator cmp_func = &DoubleCmp;
double a[]{5.0, 2.0, 4, 1.0f, 2};
sort(std::begin(a), std::end(a), cmp_func);
```

Элементы функционального программирования.

Указатели на поля и методы



```
struct A {  
    bool EqualsTo(int number) const {  
        return b_ == number;  
    }  
    int b_;  
};  
  
typedef A::*pAField; // using int pAField = A::*;  
typedef bool (A::*pAMethod)(int) const;  
// using pAMethod = bool (A::*)(int) const;  
  
pAMethod equals_method = &A::EqualsTo;  
A a{42};  
pAField field = &A::b_;  
auto pA = new A{57};  
std::cout << (a.*equals_method)(a.*stored_field) <<  
    (pA->*equals_method)(pA->*stored_field) << "\n";
```

Лямбда-функции и замыкания (C++11, 1/2)



Лямбда-функция – один из вариантов реализации функциональных объектов в языке C++11, обязанный своим названием λ -исчислению – математической системе определения и применения функций, в котором аргументом одной функции (оператора) может быть другая функция (оператор).

Как правило, лямбда-функции являются **анонимными** и определяются **в точке их применения**.

Возможность присваивать такие функции переменным позволяет именовать их **лямбда-выражениями**.

Лямбда-функции и замыкания (C++11, 2/2)



Лямбда-функции (лямбда-выражения) могут использоваться **всюду**, где требуется **передача вызываемому объекту функции** соответствующего типа, в том числе – как фактические параметры обобщённых алгоритмов STL.

В лямбда-функции могут использоваться внешние по отношению к ней переменные, образующие **замыкание** такой функции.

Многие лямбда-функции весьма просты. Например, функция

```
[ ] (int x) { return x % m == 0; }
```

эквивалентна функции вида

```
bool foo(int x) { return x % m == 0; }
```


Основные правила оформления лямбда-функций (C++11)



При преобразовании функции языка C++ в лямбда-функцию необходимо учитывать, что имя функции заменяется в лямбда-функции квадратными скобками `[]`, а возвращаемый тип лямбда-функции:

- не определяется явно (**слева**);
- при анализе лямбда-функции с телом вида
return expr;
тип автоматически выводится компилятором как
decltype(expr);
- при отсутствии в теле однооператорной лямбда-функции оператора **return** автоматически принимается равным **void**;
- в остальных случаях должен быть задан программистом при помощи «хвостового» способа записи:

```
[](int x) -> int { int y = x; return x - y; }
```

Ключевые преимущества лямбда-функций (C++11)



- **Близость** к точке использования – анонимные лямбда-функции всегда определяются в месте их дальнейшего применения и являются единственным функциональным объектом, определяемым внутри вызова другой функции.
- **Краткость** – в отличие от классов-функторов немногословны, а при наличии имени могут использоваться повторно.
- **Эффективность** – как и классы-функторы, могут встраиваться компилятором в точку определения на уровне объектного кода
- **Дополнительные возможности** – работа с внешними переменными, входящими в замыкание лямбда-функции



Именованные и анонимные лямбда-функции: пример (C++11, 1/2)



```
// для анонимных лямбда-функций:
std::vector<int> v1;

std::vector<int> v2;

std::transform(std::begin(v1), std::end(v1),
               std::begin(v2), [](auto x) { return ++x; });

// для именованных лямбда-функций:
// тип lt10 зависит от реализации компилятора
auto lt10 = [](int x) { return x < 10; };

int cnt = std::count_if(std::begin(v1), std::end(v1), lt10);

bool b = lt10(300); // b == false
```



Именованные и анонимные лямбда-функции: пример (C++11, 2/2)



```
// тип lt10 зависит от реализации компилятора
auto lt10 = [](int x) { return x < 10; };

auto lt10_copy = lt10;

auto another_lt10_copy = [](int x) { return x < 10; };

const std::type_info &t1 = typeid(lt10);

const std::type_info &t2 = typeid(lt10_copy);

const std::type_info &t3 = typeid(another_lt10_copy);

std::cout << t1.name() << " " << t2.name() << " " << t3.name();
// Z4mainEUliE_ Z4mainEUliE_ Z4mainEUliE0_
```

Внешние переменные и замыкание лямбда-функций (C++11)



• Внешние по отношению к лямбда-функции **автоматические переменные**, определённые в одной с ней области видимости, могут захватываться лямбда-функцией и входить в её **замыкание**. При этом в отношении доступа к переменным действуют следующие соглашения:

- **[z]** – доступ по значению к одной переменной (z);
- **[Gz]** – доступ по ссылке к одной переменной (z);
- **[=]** – доступ по значению ко всем автоматическим переменным;
- **[G]** – доступ по ссылке ко всем автоматическим переменным;
- **[G, z], [=, Gz], [z, Gzz]** – смешанный вариант доступа



Лямбда-функции: замыкания: пример (C++17)



```
int found_index = -1;
int treshold = 10;
auto find_first = [found = false, index = 0,
                  &found_index] (auto input)
                        mutable -> void {
    while (!found && index < std::size(input)) {
        if (input[index] < 10) {
            found_index = index;
            found = true;
        } else {
            ++index;
        }
    }
};
treshold = 30;
first_first(std::vector<int>{20, 30, 5, 7, 15});
std::cout << found_index << "\n";
```



Шаблоны и автоматический вывод типов



```
// описание укороченного шаблона (abbreviated template)
// с использованием неограниченного заместителя типа (auto)
void foo(auto a, auto *b);
// эквивалентно:
// template <typename T, typename U> foo(T a, U *b);
// каждый неограниченный заместитель вводит собственный
// параметр-тип
void bar(std::vector<auto*>...);
// эквивалентно:
// template <typename... T> void bar(std::vector<T*>...);
void foobar(auto (auto::*)(auto));
// эквивалентно:
// template <typename T, typename U, typename V>
// void foobar(T (U::*)(V));
```



Лямбда-функции: как это устроено внутри (C++14)



```
struct LambdaSumOrMax {
    LambdaSumOrMax(double &result) : result_(result) {}
    template <typename FirstType, typename SecondType>
    auto operator()(FirstType lhs, SecondType rhs)
        -> decltype(lhs + rhs) const {
        result_ = std::max<decltype(lhs + rhs)>(lhs + rhs, max_);
        return result_;
    }
    int max_ = 5;
    int &result_;
};

int result = 0;
auto sum_or_max = [max = 5, &result](auto lhs, auto rhs) {
    result = std::max<decltype(lhs + rhs)>(lhs + rhs, max);
    return result;
};

auto my_sum = LambdaSumOrMax(result);
std::cout << sum_or_max(1, 5.3) << " " << my_sum(5.3, 1);
```




Пример: функторы и лямбда-выражения (1 / 2)



```
int foo(std::string &a, int b) { /* ... */ }

struct Foo {
    int operator()(std::string &a, int b) { /* ... */ }
};

auto l = [](std::string &a, int b) -> int { /* ... */ };

template<class T, class U, class V>
V tfoo(T &a, U b) { /* ... */ }

struct TFoo {
    template<class T, class U, class V>
    V operator()(T &a, U b) { /* ... */ }
};

auto tl = [](auto &a, auto b) { /* ... */ };
```



Пример: функторы и лямбда-выражения (2 / 2)



```
int main() {  
    std::string s("qwerty");  
    foo(s, 42);  
    Foo()(s, 42);  
    l(s, 42);  
    tfoo<std::string, int, int>(s, 42);  
    TFoo().operator<std::string, int, int>(s, 42);  
    tl(s, 42);  
    return 0;  
}  
  
// Везде выведет qwerty42
```



Пример: function (1 / 2)



```
// Начиная с C++11, также входит в стандарт!  
#include <functional>
```

```
std::function<int(std::string, int)> f;
```

```
f = foo; f(s, 42); /* f = &foo; */
```

```
f = Foo(); f(s, 42);
```

```
f = l; f(s, 42);
```

```
f = tfoo<std::string, int, int>; f(s, 42);
```

```
// НО: ошибки компиляции
```

```
f = &TFoo().operator()<std::string, int, int>;
```

```
f = tl;
```



Пример: function (2 / 2)



```
#include <functional>

struct A {
    double f;
};

void mySort(A *first, A *last, asc bool) {
    std::function<bool(A, A)> compare;
    if (asc) {
        compare = [](A a, A b) { return a.f < b.f; }
    } else {
        compare = [](A a, A b) { return a.f > b.f; }
    }
    std::sort(first, last, compare);
}
```



Пример: bind (1 / 4)



```
// Начиная с C++11, также входит в стандарт!  
#include <functional>  
#include <iostream>  
#include <vector>  
#include <algorithm>  
  
void get_sum(int &a, int b) { a += b; }  
  
int main() {  
    using namespace std::placeholders;  
    std::vector<int> elems {1, 2, 3, 4, 5};  
    int sum = 0;  
    std::for_each(elems.begin(), elems.end(),  
        std::bind(&::get_sum, std::ref(sum), _1));  
    std::cout << sum;  
    return 0;  
}
```



Пример: bind (2 / 4)



```
#include <functional>
#include <iostream>
#include <algorithm>
struct Logger {
    Logger(std::string prefix) : prefix(prefix) {}
    void log(std::string message) {
        std::cout << prefix << message << std::endl;
    }
private:
    std::string prefix;
};
// ...
using namespace std::placeholders;
using namespace std::literals;
Logger loggers[2] = {"DEBUG"s, "WARN"s};
std::string message = "Hello!";
std::for_each(loggers, loggers + 2,
    std::bind(&Logger::log, _1, message));
```



Пример: bind (3 / 4)



```
#include <functional>

struct OnClickListener {
    std::function<void()> onClick;
};
struct Button : OnClickListener {};

struct Game {
    void start();
    void exit();
};

Button startButton;
Button exitButton;
void createGame() {
    Game game;
    startButton.onClick = std::bind(&Game::start, &game);
    exitButton.onClick = std::bind(&Game::exit, &game);
}
```



Пример: bind (4 / 4)



```
// std::bind в отличие от boost::bind не умеет работать с  
// перегруженными методами!
```

```
void foo() {}
```

```
void foo(int i) {}
```

```
auto badStd = std::bind(foo); // ошибка компиляции
```

```
auto badStd2 = std::bind(foo, 1); // ошибка компиляции
```

```
auto std1 = std::bind(static_cast<void(*)()>(foo)); // OK
```

```
auto std2 = std::bind(static_cast<void(*)(int)>(foo)); // OK
```

```
auto boost1 = boost::bind(foo); // OK
```

```
auto boost2 = boost::bind(foo, 1); // OK
```


Отметьтесь на портале



- Посещение необязательное, но тем, кто пришёл, следует отмечаться на портале в начале каждого занятия
- Это позволяет нам анализировать, какие занятия были более или менее интересны студентам, и менять курс в лучшую сторону
- Также это даст возможность вам оставить обратную связь по занятию после его завершения

The screenshot shows a portal interface with a calendar of events. The calendar has tabs for dates from September 23 to 29. The selected date is September 24, showing a lecture by Алексей Халайджи titled 'Углубленное программирование на C/C++'. The lecture details include the topic 'Лекция: Цели и задачи курса. Организация и использование оперативной и сверхоперативной памяти в программах на языке C.', the time '18:00-21:00', the location '433', and the groups 'АПО-11, АПО-12, АПО-13'. A button labeled 'Отметиться' (Mark) is visible at the bottom of the pop-up window, with an arrow pointing to it from the right.

Асинхронное программирование в современном C++



В стандарте C++11 появились примитивы для работы с несколькими потоками, а также их синхронизации.

Ниже представлен пример простейшей программы, запускающей отдельный поток:

```
#include <iostream>
#include <thread>
#include <string>

int main() {
    auto func = [](const std::string &message) {
        std::cout << message << std::endl;
    };
    std::thread thread(func, "Hello, world!");
    thread.join();
}
```

При создании потока **все** значения передаются по значению. Если принимаемый аргумент является ссылкой, то это будет ссылка на **копию** оригинального значения. Для передачи ссылки на **оригинальное** значение необходимо передавать **std::ref**.

Контроль завершения дочернего потока



Функция начинает своё исполнение **сразу** после завершения работы конструктора. **Окончание** потока может осуществляться по одному из следующих сценариев:

- `thread.join()` – главный поток, который вызвал `join`, будет ожидать завершения дочернего потока, блокируя своё выполнение
- `thread.detach()` – главный поток «освобождает» дочерний. При этом главный поток не блокируется и сразу заканчивает своё выполнение
- в остальных случаях в деструкторе `std::thread` будет вызван `std::termination`, что приведёт к его аварийному завершению

Полезные функции при работе с потоками



Общее число потоков, которое может быть создано, исходя из аппаратной конфигурации устройства, может быть получено с помощью функции **std::hardware_concurrency**.

Применительно к текущему выполняемому потоку, есть несколько полезных функций:

- **get_id** – позволяет получить **идентификатор текущего** исполняемого потока
- **yield** – сигнал для ОС, что поток может приостановить своё выполнение и передать управление другому потоку
- **sleep_until** – приостановление работы потока до наступления определённого аргументом времени
- **sleep_for** – приостановление работы потока на определённое аргументов время

```
#include <chrono>
auto days = std::chrono::system_clock::now() + std::chrono::hours(72);
std::this_thread::sleep_until(days);
std::this_thread::sleep_for(std::chrono::hours(72));
```

Стандарт предлагает на выбор несколько механизмов синхронизации:

- `std::mutex` – `lock`, `try_lock`, `unlock`
 - `std::recursive_mutex` – может «войти сам в себя» неопределённое количество раз, после чего будет брошено исключение `std::system_error`
 - `std::timed_mutex` – `try_lock_for`, `try_lock_until`
 - `std::recursive_timed_mutex` – комбинация `recursive_mutex` и `timed_mutex`
- `std::lock_guard` – RAII обёртка над `std::mutex`. В конструкторе осуществляется захват, в деструкторе – освобождение. Может быть скопирован от объекта с уже захваченным мьютексом. В C++17 появился `std::scoped_lock`, принимающий произвольное количество мьютексов.
- `std::unique_lock` – позволяет:
 - принимать не захваченный мьютекс в конструкторе
 - захватывать и освобождать мьютекс с `lock/unlock`
 - выполнять временной захват
 - может быть **перемещён** в другой объект `unique_lock`
- `std::shared_lock` (C++14) – move-based обёртка для предоставления совместного доступа (может быть использован с `conditional_variable_any`)

Также есть три основные стратегии захвата мьютекса: `defer_lock` (не захватывать), `try_to_lock` (попытаться захватить без блокировки) и `adopt_lock` (предполагается, что текущий поток уже захватил мьютекс).

Пример использования мьютексов



```
{ // мьютексы не захватываются в связи со стратегией
    std::unique_lock<std::mutex> lock1(from.m, std::defer_lock);
    std::unique_lock<std::mutex> lock2(to.m, std::defer_lock);
    std::lock(lock1, lock2);
    exchange();
} // освобождение мьютексов при вызове деструктора

{
    std::scoped_lock lock(from.m, to.m);
}

#include <shared_mutex>

std::shared_timed_mutex _protect;
int field;
void write(int newValue) {
    std::unique_lock<std::shared_timed_mutex> w(_protect);
    field = newValue;
}
void read() {
    std::shared_lock<std::shared_timed_mutex> r(_protect);
    return field;
}
```

Стандарт также предоставляет возможность работы с условными переменными (`std::condition_variable`). Перед использованием необходимо захватить `unique_lock`. Основными методами при работе с ними являются:

- `wait` – ставит поток в ожидание сигнала без привязки ко времени. В качестве аргумента может быть передан предикат, проверяющий необходимость повторно встать в очередь после получения сигнала.
- `wait_for` – передаётся время ожидания
- `wait_until` – передаётся время, до которого продлится ожидание
- `notify_one` – посылка сигнала **одному (любому!)** из ожидающих потоков
- `notify_all` – посылка сигнала всем ожидающим потокам

```
std::mutex mtx;
std::condition_variable cv;
bool done = false;
void worker() {
    std::unique_lock<std::mutex> locker(mtx);
    done = true;
    cv.notify_one();
}
void checker() {
    std::unique_lock<std::mutex> locker(mtx);
    while (!done) { cv.wait(locker); } // cv.wait(locker, [&]() {return done;});
}
```

Асинхронный запуск задачи



Помимо явного создания потоков стандарт даёт возможность создания отложенной задачи `std::future`, выполнение которой может осуществляться даже на том же физическом ядре, но позже и асинхронно по отношению к основной программе.

Результат может быть получен синхронным вызовом метода `get()`. Он вернёт значение, если оно уже было вычислено, и будет ожидать его получения синхронно, в противном случае.

Также важно, что при вызове `get` может быть выброшено исключение, которое могло возникнуть при выполнении другого потока (до вызова `get` оно выбрасываться не будет).

```
std::future<bool> do_task(std::string);

auto result = do_task("test"); // управление сразу вернётся
if (result.get()) // синхронный вызов, возможно ожидание результата
    std::cout << "Success";
else
    std::cout << "Fail";
```


std::future и std::promise



- Способ передачи результатов между потоками
- Для каждого promise есть соответствующий ему future
- promise ответственен за установку, future – за получение значения

```
std::promise<int> p;  
std::future<int> f = p.get_future();  
std::thread([](promise<int> p) {  
    int val = calc();  
    p.set_value(val);  
}, std::move(p)).detach();  
  
// ...  
  
f.get();
```

Обработка исключений в дочернем потоке



- `std::exception_ptr` – `shared_ptr` для хранения информации об исключении
- `std::current_exception()` – получение `exception_ptr` на пойманное исключение в текущем контексте
- `std::make_exception_ptr()`

```
// приведёт к вызову std::terminate
std::thread([]() { throw std::runtime_exception("Oops"); }).join();
std::promise<void> p;
std::future<void> f = p.get_future();
std::thread([](std::promise<void> p) {
    try {
        throw std::runtime_exception("Oops");
    } catch (...) {
        p.set_exception(std::current_exception());
    }
}, std::move(p)).detach();
try {
    future.get();
} catch (...) { /* Обработка пойманного исключения */ }
```

std::make_exception_ptr



```
template <class E>
std::exception_ptr make_exception_ptr(E e) {
    try {
        throw e;
    } catch (...) {
        return std::current_exception();
    }
}
```

Одним из способов отправки задачу на асинхронную работу и получения `std::future` является использование `std::packaged_task`.

```
std::future<bool> do_task(std::string) {  
    Auto handle = [](const std::string &form) -> bool {  
        /* Some logic */  
        return true;  
    }  
    std::packaged_task<bool(const std::string &)> task(handle);  
    auto future = task.get_future();  
    std::thread thread(std::move(task), message);  
    thread.detach();  
    return std::move(future);  
}
```

std::packaged_task – идея реализации



```
template<...>
class packaged_task {
    std::function<...> f;
    std::promise<...> p;
    std::future<...> get_future() { return p.get_future(); }
    void operator()(Args... args) {
        try {
            p.set_value(f(args...));
        }
        catch (...) {
            p.set_exception(std::current_exception());
        }
    }
};
```

Последним из способов запуска асинхронной задачи является использование вызова `std::async`.

При вызове есть возможность передачи дополнительного флага, который будет определять характер выполнения:

- `launch::async` – в отдельном потоке
- `launch::deferred` – отложено.

```
auto future = std::async(/* launch policy, */  
                        []() { return 42; }  
                        /* ,args */);  
future.get();
```

boost::asio: общие сведения



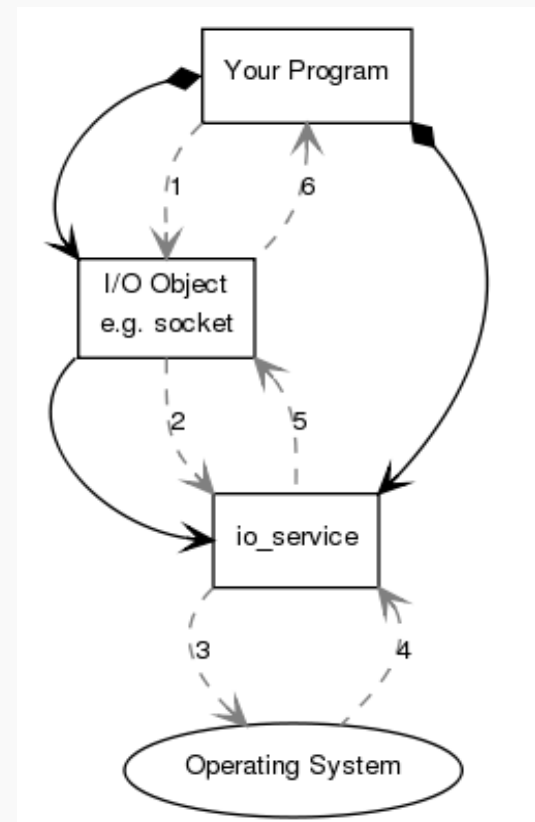
- хорошая реализация асинхронной модели;
- открытый исходный код;
- кроссплатформенность;
- максимальная производительность для каждой платформы;
- позволяет писать синхронный и асинхронный код;
- обработка ошибок;
- легко интегрируется с ssl;
- хорошо документирован.

<http://www.boost.org/> → Documentation → asio

boost::asio: анатомия синхронных операций



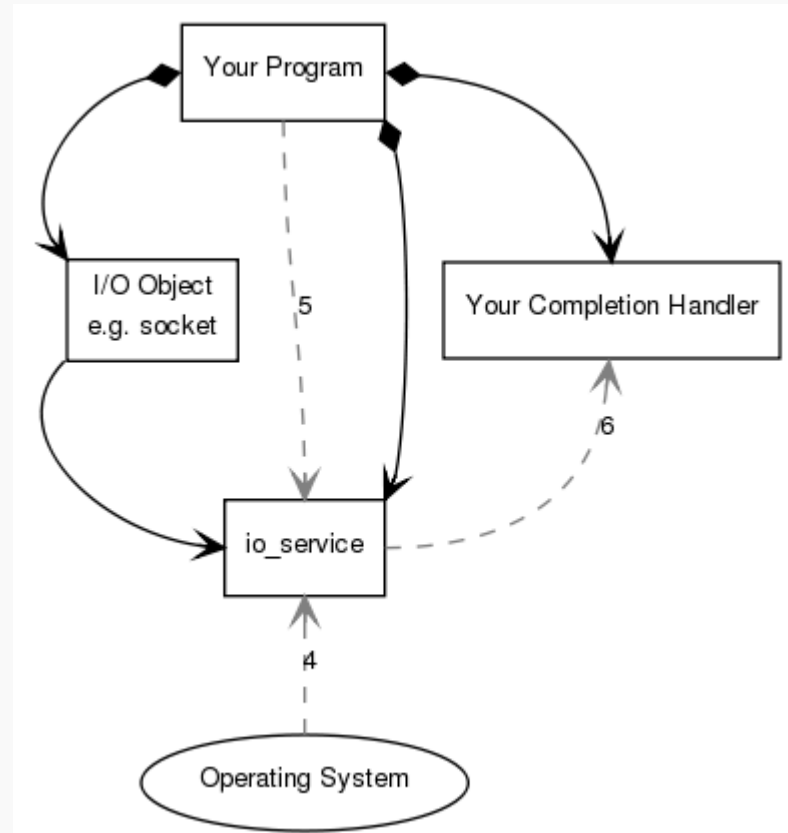
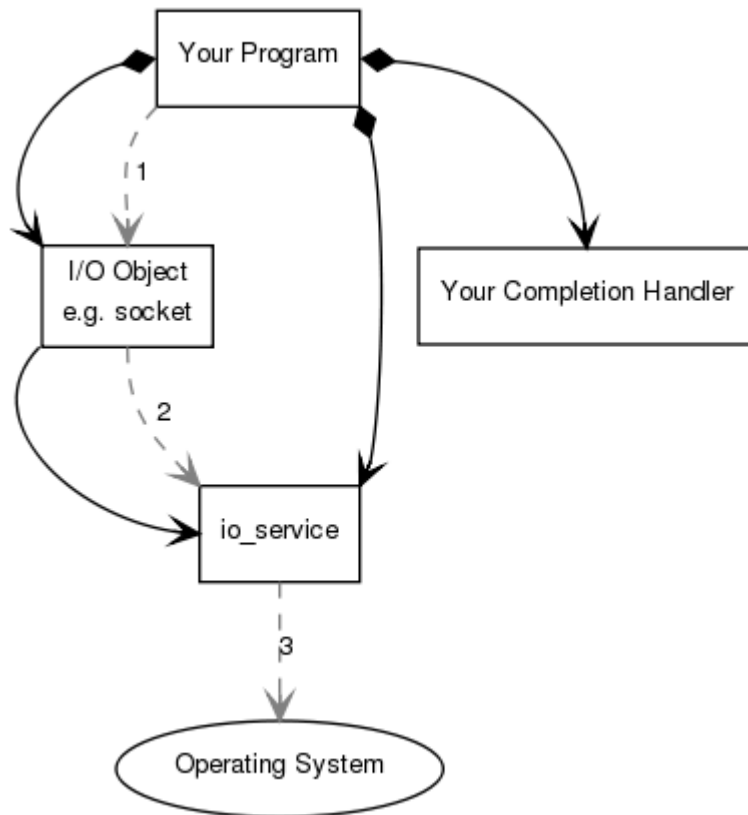
```
boost::asio::io_service io_service;           // как минимум один в программе
boost::asio::ip::tcp::socket socket(io_service);
boost::system::error_code ec;
socket.connect(server_endpoint, ec);
if(ec)
    throw std::runtime_error(ec.message());
```



boost::asio: анатомия асинхронных операций



- `socket.async connect(server endpoint, your completion handler);`
- `void your_completion_handler(const boost::system::error_code& ec);`
- `io_service::run()`



boost::asio: общие сведения



boost::asio:: - это namespace, содержащий все классы, функции и т.п.
boost::asio::io_service io_service; - должен быть в каждой программе

Есть аналоги для всех “нативных” функций:

connect() => boost::asio::ip::tcp::socket::connect()
read() => boost::asio::ip::tcp::socket::read_some()
и т.д.

Есть асинхронные аналоги для всех “нативных” функций:

async_connect()/async_read() => отправить в очередь + callback

Есть таймеры, ICMP, RS232 и т.д.

“Не умеет” работать с std::string, в целях эффективности использует:

- boost::asio::streambuf (async_read_until(m_socket, stream, “\r\n”))
- boost::asio::buffer (async_read_some(asio::buffer(data, length)))

boost::asio: зависимости



Пример: <https://github.com/o2gy84/misc/tree/master/technopark/asio>

В коде:

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

При сборке:

```
g++ -I/usr/boost156 -L/usr/boost156 -o app -std=c++14 main.cpp -
lboost_system -lpthread
```

boost::asio: пример синхронной работы, клиент



```
boost::asio::io_service io_service;
boost::asio::ip::tcp::socket s(io_service);
boost::asio::ip::tcp::resolver resolver(io_service);
boost::asio::ip::tcp::resolver::query query("localhost", "7789");

boost::system::error_code ec;
auto endpoint = resolver.resolve(query, ec);
if (ec) throw std::runtime_error(ec.message());

boost::asio::connect(s, endpoint, ec);
if (ec) throw std::runtime_error(ec.message());

boost::asio::write(s, boost::asio::buffer("LOGIN\r\n", 7), ec);
if (ec) throw std::runtime_error(ec.message());

char reply[max_len];
s.read_some(boost::asio::buffer(reply, max_len), ec);
```

boost::asio: пример синхронной работы, нужно больше C++!



```
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();
```

boost::asio: можно считать очередью



```
void func() { std::cerr << "func" << std::endl; }

struct Struct { void operator() () {cerr << "operator()" << std::endl; }};

boost::asio::io_service io;
io.post(func);
io.post([]()>void { std::cerr << "lambda" << std::endl; });
io.post(Struct());

std::shared_ptr<Class> c = std::make_shared<Class>();
io.post(std::bind(&Class::funcC, c->shared_from_this()));

io.run_one(); // func
io.run_one(); // lambda
io.run_one(); // Struct::operator()
io.run_one(); // Class...

io.run(); // ИЛИ ТАК: func + lambda + Struct::operator() + Class + ...
```

boost::asio: пример многопоточного асинхронного echo-сервера



```
// Класс, представляющий клиентское соединение
class Client: public std::enable_shared_from_this<Client> {
public:
    Client(boost::asio::io_service &io) : m_Sock(io) {}
    boost::asio::ip::tcp::socket& sock() { return m_Sock; }
    void read();
    void handleRead(const boost::system::error_code& e, size_t bytes);

private:
    boost::asio::ip::tcp::socket m_Sock;
    char m_Buf[1024];
};
```

boost::asio: пример многопоточного асинхронного echo-сервера



```
void Client::read() {
    m_Sock.async_read_some(boost::asio::buffer(m_Buf),
                           bind(&Client::handleRead, shared_from_this(),
                               boost::asio::placeholders::error,
                               ::placeholders::bytes_transferred));
}

void Client::handleRead(const boost::system::error_code& e, size_t bytes) {
    if (e) return;
    m_Sock.async_write_some(boost::asio::buffer(m_Buf),
                            [self = shared_from_this()](const error_code& e, size_t bytes) {
                                // После того, как запишем ответ, можно снова читать
                                self->read();
                            });
}

};
```


boost::asio: пример многопоточного асинхронного echo-сервера



```
class Server {
    boost::asio::io_service m_Service;
    boost::asio::ip::tcp::acceptor m_Acceptor;
    void onAccept(std::shared_ptr<Client> c, const error_code& e) {
        if (e) return;
        c->read();
        startAccept();
    }
    void startAccept() {
        std::shared_ptr<Client> c(new Client(m_Service));
        m_Acceptor.async_accept(c->sock(),
                                bind(&Server::onAccept, this, c, asio::placeholders::error));
    }
public:
    Server() : m_Acceptor(m_Service) {}
    void startServer();
};
```

boost::asio: пример многопоточного асинхронного echo-сервера



```
void Server::startServer() {
    boost::asio::ip::tcp::endpoint endpoint(boost::asio::ip::tcp::v4(), 5001);
    m_Acceptor.open(endpoint.protocol());
    m_Acceptor.bind(endpoint);
    m_Acceptor.listen(1024);
    startAccept();

    std::vector<std::thread> threads;
    for (int i = 0; i < 4; ++i)
        threads.push_back(thread(bind(&io_service::run, &m_Service)));
    for (auto &thread: threads)
        thread.join();
}

int main(int argc, char *argv[]) {
    Server().startServer();
    return 0;
}
```

Итоги: какая архитектура лучше?



- 1) fork/thread per connection - “домашние” проекты, прототипы
- 1) prefork - если не критично количество ресурсов, или заранее известно, что будет небольшое количество клиентов
- 1) multiplexing - мало бизнес логики (прокси, балансеры и т.п.);
- 1) prefork + multiplexing - годится для “большого” прода, но может быть дорого (один httpd может “кушать” и гигабайт), а делать prefork на тредах - уменьшается надежность, уж лучше тогда asio;
- 1) asio - годится для “большого” прода, требует меньше памяти, но сильно сложнее в реализации.

Примеры кода



<https://github.com/o2gy84/misc/tree/master/technopark>

- `common` - общий класс для работы с сетью
 - `simple-server` - работа только с одним клиентом
 - `client_connect_timeout` - клиент, умеющий таймауты на соединение
 - `client_read_timeout` - клиент, умеющий таймауты на чтение
 - `http_client` - пример выполнения http-запроса
 - `multiplexing` - “примитивное” мультиплексирование
 - `prefork` - форкающийся сервер
-
- `libevent` - echo-сервер на libevent
 - `asio` - echo-сервер на boost::asio
-
- `libtpevent` - сервер, в котором можно выбирать несколько вариантов `event_loop`’ов, в том числе реализован асинхронный движок на poll

- Библиотека boost::asio предоставляет возможность **быстрой** и **асинхронной** работы с сетью через общую очередь событий и операциями чтения/записи.
- Однако часто этого бывает недостаточно, поскольку в ряде случаев необходимо **удерживать** соединение для общения между участниками сети по определённом протоколу (например, в онлайн-играх или чатах).
- Обычно для этого используются **вебсокеты**. Однако в Boost::asio их прямой поддержки нет.
- Такая поддержка есть в библиотеке **Boost::Beast**, которая основана на **той же асинхронной модели** boost::asio, реализует низкоуровневые операции по поддержанию протоколов HTTP и WebSocket и предоставляет высокоуровневый интерфейс.



boost::Beast (2 / 8)



В целях упрощения рассмотрим работу по HTTP и вебсокету при синхронном взаимодействии!

Основные подключаемые заголовки:

```
#include <boost/beast/core.hpp>
// для работы по HTTP
#include <boost/beast/http.hpp>
#include <boost/beast/version.hpp>
// для работы по Websocket
#include <boost/beast/websocket.hpp>

#include <boost/asio/connect.hpp>
#include <boost/asio/ip/tcp.hpp>

#include <cstdlib>
#include <iostream>
#include <string>

using tcp = boost::asio::ip::tcp; // <boost/asio/ip/tcp.hpp>
namespace http = boost::beast::http; // <boost/beast/http.hpp>
namespace websocket = boost::beast::websocket; //
    <boost/beast/websocket.hpp>
```

boost::Beast (3 / 8) – реализация синхронного HTTP клиента (1 / 2) – отправка запроса



```
// создание io_context (он же io_service)
boost::asio::io_context ioc;

// создание вспомогательных объектов для выполнения IO
tcp::resolver resolver{ioc};
tcp::socket socket{ioc};

// разрешение доменного имени
auto const results = resolver.resolve(host, port);

// создание подключения по полученному IP адресу
boost::asio::connect(socket, results.begin(), results.end());

// создание HTTP GET-запроса
http::request<http::string_body> req{http::verb::get, target,
    version};
req.set(http::field::host, host);
req.set(http::field::user_agent, BOOST_BEAST_VERSION_STRING);

// отправка HTTP-запроса на сервер
http::write(socket, req);
```

boost::Beast (4 / 8) – реализация синхронного HTTP клиента (2 / 2) – получение ответа



```
// создание персистентного буфера для чтения
boost::beast::flat_buffer buffer;

// создание контейнера для хранения ответа
http::response<http::dynamic_body> res;

// получение ответа от сервера
http::read(socket, buffer, res);

// вывод сообщения на экран
std::cout << res << std::endl;

// закрытие сокета
boost::system::error_code ec;
socket.shutdown(tcp::socket::shutdown_both, ec);

// обработка ошибок закрытия соединения.
// по документации not_connected – это не существенная ошибка
if (ec && ec != boost::system::errc::not_connected)
    throw boost::system::system_error{ec};
```


boost::Beast (5 / 8) – реализация синхронного Websocket-клиента (1 / 2) – отправка запроса



```
// создание io_context
boost::asio::io_context ioc;

// создание дополнительных объектов для выполнения I/O
tcp::resolver resolver{ioc};
websocket::stream<tcp::socket> ws{ios};

// разрешение доменного имени
auto const results = resolver.resolve(host, port);

// установка соединения по полученному IP-адресу
boost::asio::connect(ws.next_layer(), results.begin(), result.end());

// создание websocket-соединения (handshake)
ws.handshake(host, "/");

// отправка сообщения на сервер
ws.write(boost::asio::buffer(std::string(text)));
```

boost::Beast (6 / 8) – реализация синхронного Websocket-клиента (2 / 2) – получение ответа



```
// создание буфера для хранения входящего сообщения
boost::beast::multi_buffer buffer;

// чтение сообщения в буфер
ws.read(buffer);

// закрытие websocket-соединения – выбрасывает исключение при неудаче
ws.close(websocket::close_code::normal);

// вывод ответа на экран
std::cout << boost::beast::buffers(buffer.data()) << std::endl;
```

boost::Beast (7 / 8) – реализация синхронного Websocket-сервера (1 / 2) – получение запросов



```
int main() { try {
    auto const address = boost::asio::ip::make_address("127.0.0.1");
    auto const port = static_cast<unsigned short>(8080);

    // создание io_context
    boost::asio::io_context ioc{1}; // 1 поток

    // создание агрегатора, который будет принимать входящие соединения
    tcp::acceptor acceptor{ioc, {address, port}};
    for (;;) {
        // создание сокета, в котором будет ожидатьсся соединение
        tcp::socket socket{ioc};

        // синхронное ожидание соединения
        acceptor.accept(socket);

        // запуск websocket-соединения и обработчика клиента
        std::thread{std::bind(&do_session, std::move(socket))}.detach();
    }
} catch (const std::exception &e) { std::cerr << e.what(); return
EXIT_FAILURE; } }
```

boost::Beast (8 / 8) – реализация синхронного WebSocket-сервера (2 / 2) – обработка соединения и отправка ответа



```
void do_session(tcp::socket &socket) { try {
    // апгрейд соединения до протокола WebSocket
    websocket::stream<tcp::socket> ws{std::move(socket)};
    // принятие websocket-соединения от клиента
    ws.accept();
    for (;;) {
        // создание буфера для чтения сообщения от клиента
        boost::beast::multi_buffer buffer;
        // чтение сообщения
        ws.read(buffer);
        // отправка ответа клиенту
        ws.text(ws.got_text());
        ws.write(buffer.data());
    }
} catch (boost::system::system_error const &se) {
    // это означает, что сессия была закрыта
    if (se.code() != websocket::error::closed) {
        std::cerr << "Error: " << se.code().message() << "\n";
    }
} catch (std::exception const &e) {
    std::cerr << "Error: " << e.what() << std::endl;
}
}
```

Boost::PropertyTree предоставляет **структуру данных**, которая хранит вложенное **дерево** значений **произвольной** длины, проиндексированное на каждом уровне определённым **ключом**.

Помимо этого, библиотека предоставляет **парсеры** и **генераторы** для работы с **популярными** форматами данных, включая INI, XML, JSON.

Деревья свойств – универсальные структуры данных, но особенно подходят для хранения **конфигурационных** данных. Дерево предоставляет свой собственный, свойственный деревьям интерфейс, и каждый элемент также является STL-совместимой последовательностью для дочерних элементов.

Концептуально, дерево представляет собой следующую структуру:

```
struct ptree {  
    data_type data; // данные элемента  
    list< pair<key_type, ptree> > children;  
    // упорядоченный список именованных дочерних элементов  
};
```

Пример: boost::property_tree на примере работы с XML (1 / 4)



Представим, что мы храним файл конфигурации логирования в формате XML, и он имеет следующий вид:

```
<debug>
  <filename>debug.log</filename>
  <modules>
    <module>Finance</module>
    <module>Admin</module>
    <module>HR</module>
  </modules>
  <level>2</level>
</debug>
```

В частности, он содержит имя файла для логирования, список модулей, для которых будет осуществляться логирование, а также уровень важности логов.

Для начала работы с boost::property_tree, необходимо подключить ряд заголовочных файлов:

```
#include <boost/property_tree/ptree.hpp>
#include <boost/property_tree/xml_parser.hpp>
#include <boost/foreach.hpp>
#include <string>
#include <set>
#include <exception>
#include <iostream>
namespace pt = boost::property_tree;
```

Пример: boost::property_tree на примере работы с XML (2 / 4)



Выберем подходящую структуру данных для хранения конфигурации логирования:

```
struct debug_setting {  
    std::string m_file; // имя файла для логирования  
    int m_level;  
    std::set<std::string> m_modules;  
    void load(const std::string &filename);  
    void save(const std::string &filename);  
};
```



Пример: boost::property_tree на примере работы с XML (3 / 4)



```
// Метод считывания конфигурации с диска
void debug_settings::load(const std::string &filename) {
    // Создание пустого объекта дерева свойств
    pt::ptree tree;

    // Разбор XML в дерево
    pt::read_xml(filename, tree);

    // Эта реализация может выбросить исключение, если XPath не будет найден
    m_file = tree.get<std::string>("debug.filename");

    // Эта функция возвращает значение по умолчанию – оно же используется
    // для автоматического вывода типа шаблона
    m_level = tree.get("debug.level", 0);

    // Использование get_child для нахождения элемента, содержащего модули,
    // и итерация по его дочерним элементам. Если такого XPath не будет
    // найдено, то выбросится исключение. Можно использовать for_each из STL.
    BOOST_FOREACH(pt::ptree::value_type &v, tree.get_child("debug.modules")) {
        // Доступ к данным осуществляется через метод data()
        m_modules.insert(v.second.data());
    }
}
```




Пример: boost::property_tree на примере работы с XML (4 / 4)



```
// Метод сохранения конфигурации на диск
void debug_settings::save(const std::string &filename) {
    // Создание пустого объекта дерева свойств
    pt::ptree tree;

    // Сохранение простых значений в дерево. Целые числа автоматически
    // преобразуются в строки. Отсутствующие элементы создаются автоматически,
    // если не указываются явно - как тут, например, элемент "debug"
    tree.put("debug.filename", m_file);
    tree.put("debug.level", m_level);

    // Добавление всех модулей. В отличие от обычного сохранения, которое
    // перезаписывает существующие элементы, здесь используется добавление
    // на нижнем уровне, чтобы элемент "modules" имел несколько дочерних
    // элементов "module"
    BOOST_FOREACH(const std::string &name, m_modules)
        tree.add("debug.modules.module", name);

    // Сохранение дерева в XML файл
    pt::write_xml(filename, tree);
}
```



Пример: Разбор JSON-строки с использованием boost::property_tree



```
#include <boost/property_tree/ptree.hpp>
#include <boost/property_tree/json_parser.hpp>
#include <boost/foreach.hpp>
#include <cassert>
#include <exception>
#include <iostream>
#include <string>
namespace bpt = boost::property_tree;
int main() { try {
    std::stringstream ss;
    ss << "{ \"root\": { \"values\": [1, 2, 3, 4, 5 ] } }";
    bpt::ptree pt;
    bpt::read_json(ss, pt);
    BOOST_FOREACH(bpt::ptree::value_type &v, pt.get_child("root.values")) {
        assert(v.first.empty()); // массив не имеет имени
        std::cout << v.second.data() << std::endl;
    }
    return EXIT_SUCCESS;
} catch (std::exception const &e) {
    std::cerr << e.what() << std::endl;
}
return EXIT_FAILURE;
}
```



Пример: filesystem (1 / 5)



// Как было раньше

```
auto removeOne(const std::string &path) {  
#if defined(_WIN32)  
    std::wstring wstr;  
    LPCWSTR wname = utf8to16(path, wstr);  
    WIN32_FILE_ATTRIBUTE_DATA fad;  
    if (::GetFileAttributesExW(wname, GetFileExInfoStandard, &fad)) {  
        if (fad.dwFileAttributes & FILE_ATTRIBUTE_DIRECTORY)  
            return ::RemoveDirectoryW(wname) != 0;  
        return ::DeleteFileW(wname) != 0;  
    }  
    return false;  
#else  
    return ::remove(path.data()) == 0;  
#endif  
}
```



Пример: filesystem (2 / 5)



```
// Появилось в стандарте C++17

auto removeOne(const std::filesystem::path &path) {
    return std::filesystem::remove(path);
}
```



Пример: filesystem (3 / 5)



```
class path {  
public:  
    template<class Source> path(const Source &);  
  
    path root_path() const;  
    path relative_path() const;  
    path parent_path() const;  
    path filename() const;  
    path extension() const;  
  
    bool is_absolute() const;  
    bool is_relative() const;  
  
    void append(path);  
  
    const_iterator begin() const;  
    const_iterator end() const;  
};  
  
template<class Source>  
path u8path(const Source &source);
```



Пример: filesystem (4 / 5)



```
namespace fs = std::filesystem;

int main() {
    fs::create_directories("test/a/b/");
    std::ofstream("test/file1.txt");
    std::ofstream("test/file2.txt");
    for (auto &p : fs::directory_iterator("test"))
        std::cout << p << std::endl;
}

// Будет выведено:
// test/a
// test/file1.txt
// test/file2.txt

// для fs::directory_iterator переопределены begin() и end(),
// чтобы его можно было использовать в подобном контексте
```



Пример: filesystem (5 / 5)



Также в составе этой библиотеки присутствуют такие функции, как:

- rename
- remove
- copy
- exists
- file_size
- ...

Домашнее задание



В качестве домашнего задания рекомендуется:

- Зайти на официальный сайт boost.org и ознакомиться с существующими библиотеками
- Произвести рефакторинг кода своего проекта, внедрив в него возможности новых стандартов и библиотеки Boost

Отметьтесь на портале



- Посещение необязательное, но тем, кто пришёл, следует отмечаться на портале в начале каждого занятия
- Это позволяет нам анализировать, какие занятия были более или менее интересны студентам, и менять курс в лучшую сторону
- Также это даст возможность вам оставить обратную связь по занятию после его завершения

The screenshot shows a portal interface with a calendar of events. The calendar has tabs for dates from September 23 to 29. The selected date is September 24. A pop-up window for a lecture by Alexey Khalaidzhi is displayed, showing the lecture title, description, time, and location. An arrow points to the 'Отметиться' (Mark) button at the bottom of the pop-up.

вс, 23 Сентября	пн, 24 Сентября	вт, 25 Сентября	ср, 26 Сентября	чт, 27 Сентября	пт, 28 Сентября	сб, 29 Сентября
Занятий нет.	Алексей Халайджиди Углубленное программирование на C/C++ Лекция: Цели и задачи курса. Организация и использование оперативной и сверхоперативной памяти в программах на языке C. ⌚ 18:00-21:00 📍 433 🎓 АПО-11, АПО-12, АПО-13 Отметиться	Занятий нет.	Занятий нет.	Занятий нет.	Занятий нет.	13:00 395 - зал 1 2 3 Углубленное програ...

Оставьте обратную связь



Обратная связь позволяет нам улучшать курс **сразу**, не дожидаясь его завершения.

Умение оставлять конструктивную обратную связь является одним из **важнейших** при работе в **команде**, поэтому **воспользуйтесь** такой возможностью с целью **дополнительного обучения**.

Это занимает не более **5 минут** и абсолютно **анонимно**!

Была ли понятна цель занятия, основная мысль?

☐ 1
☐ 2
☐ 3
☐ 4
☐ 5

Оцените скорость подачи материала

☐ очень медленно
☐ медленно
☐ нормально
☐ быстро
☐ очень быстро

Здесь вы можете оставить комментарий к занятию. Преподаватель не увидит автора записи, только текст комментария.

Дополнительный комментарий:



Алексей Халайджи

Спасибо за внимание!