



Лекция № 7

# Углубленное программирование на языке C / C++

Алексей Петров

# Рекомендуемая литература:

## модуль №3



Лишнер Р. STL. Карманный справочник. — Питер, 2005. — 188 с.

Мюссер Д., Дердж Ж., Сейни А. C++ и STL. Справочное руководство. — Вильямс, 2010. — 432 с.

Саммерфилд М. Qt. Профессиональное программирование. Разработка кроссплатформенных приложений на C++. — Символ-Плюс, 2011. — 560 с.

Шлее М. Qt 4.8. Профессиональное программирование на C++. — БХВ-Петербург, 2012. — 894 с.

Demming, R., Duffy, D.J. *Introduction to the Boost C++ Libraries; Volume I – Foundations* (Datasim Education BV, 2010).

Demming, R., Duffy, D.J. *Introduction to the Boost C++ Libraries; Volume II – Advanced Libraries* (Datasim Education BV, 2012).

Musser, D.R., Saini, A. *STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library* (Addison-Wesley, 1995).

# Лекция №7. Функциональное программирование. Введение в Boost.

## Требования к типам

---



1. Параметризация алгоритмов STL лямбда-функциями и применение замыканий.
2. Состав и назначение Boost.
3. Примеры использования Boost: проверки времени компиляции, контейнеры, «умные» указатели.
4. Использование средств Boost для повышения производительности и безопасности кода.
5. Стандартные требования к типам.

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



**Лямбда-функция** — третий (наряду с указателями на функции и классами-функторами) вариант реализации функциональных объектов в языке C++11, обязанный своим названием  $\lambda$ -исчислению — математической системе определения и применения функций, в которой 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 обобщенных 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;
```

- 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)



```
// для анонимных лямбда-функций:  
std::vector<int> v1;  
std::vector<int> v2; // ...  
std::transform(v1.begin(), v1.end(),  
               v2.begin(), [](int 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(v1.begin(), v1.end(), lt10);  
bool b = lt10(300);    // b == false
```



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



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

- `[z]` — доступ по значению к одной переменной (`z`);
- `[&z]` — доступ по ссылке к одной переменной (`z`);
- `[=]` — доступ по значению ко всем автоматическим переменным;
- `[&]` — доступ по ссылке ко всем автоматическим переменным;
- `[&, z]` , `[=, &z]` , `[z, &zz]` — смешанный вариант доступа.



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



```
int countN;  
std::vector<double> vd; // ...  
countN = std::count_if(vd.begin(), vd.end(),  
                        [](double x) { return x >= N; });  
  
// эквивалентно  
int countN = 0;  
std::vector<double> vd; // ...  
std::for_each(vd.begin(), vd.end(),  
              [&countN](double x) { countN += x >= N; });
```

# Библиотека Boost: общие сведения



Boost — набор из более 80 автономных библиотек на языке C++, задуманный в 1998 г. Б. Давесом (Beman Dawes), Д. Абрахамсом (David Abrahams) и др.



Основными **целями разработки** Boost вы

- 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





Продemonстрируем работу Boost на следующих примерах:

- **внедрение в исходный код проверок времени компиляции** — позволяет не допустить компиляции логически или семантически неверного кода;
- **применение вариантных контейнеров и произвольных типов** — открывает возможность создания обобщенных и универсальных структур хранения данных;
- **применение циклических контейнеров** — дает возможность поддержки программной кэш-памяти и эффективных FIFO/LIFO-структур фиксированного размера.

# Пример 1. 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.

# Пример 1. 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> {};  
}
```



# Пример 1. 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);
}
```



# Пример 2. 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.

# Пример 2. 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`.



## Пример 2. 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  
);
```



## Пример 2. 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;
    }
};
```



## Пример 2. 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;
    }
};
```

# Пример 3. 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.



## Пример 3. 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
};
```



## Пример 3. 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());
}
```





## Пример 3. 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;  
    }  
}
```

# Пример 4. 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.

# Пример 4. 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.



## Пример 4. 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: что еще? (1 / 2)



**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);
```

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

```
boost::tribool b(true);
```

```
b = false;
```

```
b = boost::indeterminate;
```

## Boost: что еще? (2 / 2)



**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
```

# Математические библиотеки



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.

# Система типов языка C++: концепты (Concepts TS)



Исторически C++ всегда являлся **типизированным** языком со **слабой статической** типизацией, при этом абстрактные представления об универсальных (**повторяющихся**) свойствах типов как программных категорий в нем долгое время отсутствовали.

Согласно сегодняшним представлениям, **концепт** (**англ.** concept) — это **именованное множество требований** к типу в языке C++.

Формальное определение концептов средствами C++ будет закреплено технической спецификацией ISO/IEC PDTS 19217 Information Technology — Programming Languages, Their Environments and Systems Software — C++ Extensions for Concepts (**май 2015 г.** — в **разработке**).

Рассматриваемые далее концепты (**именованные требования к типам**) используются в тексте стандарта языка C++ и описывают **непроверяемые (сейчас) ожидания** стандартной библиотеки в отношении параметров функций и аргументов шаблонов.



# Именованные требования стандартной библиотеки C++ (1 / 2)



Широкое применение типовых сочетаний требований к характеристикам стандартных и пользовательских типов привело к появлению понятия **именованных требований** (стандартной библиотеки).

Среди них **базовыми** являются:

- `DefaultConstructible` / `Destructible` — объект типа может быть создан конструктором по умолчанию / тип имеет деструктор;
- `CopyAssignable` / `CopyConstructible` — объект типа может быть изменен присваиванием / создан из леводопустимого значения;
- `MoveAssignable` / `MoveConstructible` (оба — C++11) — объект типа может быть изменен / создан переносом из праводопустимого значения.

# Именованные требования стандартной библиотеки C++ (2 / 2)



Другими именованными требованиями являются:

- **требования к размещению:** `TriviallyCopyable`, `TrivialType`, `StandardLayoutType` (все — C++11), `PODType`;
- **требования уровня библиотеки:** `Swappable`, `ValueSwappable`, `NullablePointer`, `Hash` (все — C++11), `EqualityComparable`, `LessThanComparable`, `Allocator`, `FunctionObject`, `Callable`, `Predicate`, `BinaryPredicate`, `Compare`;
- **требования к контейнерам:** `Container` — тип является структурой данных с доступом к элементам по итераторам; `ReversibleContainer`, `AllocatorAwareContainer` (C++11), `SequenceContainer`, `ContiguousContainer` (C++17), `AssociativeContainer`, `UnorderedAssociativeContainer` (C++11);
- **требования к элементам контейнеров, функциям потокового В/В, генераторам случайных чисел (C++11), асинхронным функциям (C++11) и др.**

# Стандартные функции проверки соответствия требованиям



В стандартную библиотеку языка C++11 введены **шаблоны структур**, устанавливающие соответствие типов-параметров предъявляемым требованиям.

Например, для требования *CopyConstructible*:

```
template<class T> struct is_copy_constructible;  
template<class T> struct is_trivially_copy_constructible;  
template<class T> struct is_nothrow_copy_constructible;
```

Для интроспекции типов-параметров служит открытый статический константный атрибут шаблона структуры *value*.



# Стандартные функции проверки соответствия требованиям: пример



```
struct foo {  
    string _s;                // атрибут с нетривиальным  
                              // string::string(const string&)  
};  
  
struct bar {  
    int _n;                   // тривиальный конструктор,  
    bar(const bar&) = default; // безопасный по исключениям  
};  
  
// is_copy_constructible<foo>::value == true  
// is_trivially_copy_constructible<foo>::value == false  
// is_trivially_copy_constructible<bar>::value == true  
// is_nothrow_copy_constructible<bar>::value == true
```

# Требование CopyConstructible



Базовое именованное требование к типу — тип, отвечающий требованию `CopyConstructible`, реализует одну или несколько следующих функций:

```
Type::Type(Type& other);
```

```
Type::Type(const Type& other);
```

```
Type::Type(volatile Type& other);
```

```
Type::Type(const volatile Type& other);
```

и гарантирует работоспособность следующих выражений, вычисление которых должно давать правильный, с языковой точки зрения, результат:

```
Type a = v;
```

```
Type(v);
```

# Требования TrivialType, TriviallyCopyable



Требование к размещению объектов — тип **T**, удовлетворяющий требованию **TriviallyCopyable**, отвечает следующим критериям:

- обладает тривиальными конструкторами копирования и переноса;
- обладает тривиальными операциями присваивания путем копирования и переноса;
- обладает тривиальным деструктором;
- не имеет виртуальных методов и (или) виртуальных базовых классов;
- все (нестатические) члены данных и базовые классы **T** удовлетворяют требованию **TriviallyCopyable**.

Тривиально копируемыми являются **скалярные типы** и **массивы тривиально копируемых объектов**, а также квалифицированные **const** (но не **volatile**!) версии соответствующих типов.

Тривиально копируемый тип с тривиальным конструктором

# Требование PODType



Требование к размещению объектов — тип `T`, удовлетворяющий требованию `PODType` является **скалярным типом** либо **классом**, являющимся тривиальным типом (`TrivialType`) и типом со стандартным размещением (`StandardLayoutType`), не имеющим нестатических членов данных, которые не удовлетворяют требованию `PODType`, либо **массивом** таких классов или скалярных типов.

Соответствие типа требованию `PODType` указывает на то, что:

- тип совместим с типами, используемыми в языке C;
- объекты типа могут обрабатываться функциями из библиотеки языка C, размещаться в памяти при помощи `std::malloc()`, копироваться при помощи `std::memmove()` и т.д.;
- объекты типа могут передаваться в библиотеки языка C в двоичной (машинной) форме.

# Требование Container



Объекты типа, отвечающего требованию `Container`, содержат другие объекты и отвечают за управление памятью, выделенной для хранения содержащихся в них объектов.

Пусть `C` — тип `Container`, `T` — тип элемента. Тогда:

- тип `C` реализует поддержку встроенных типов `value_type`, `reference`, `const_reference`, `iterator`, `const_iterator`, `difference_type`, `size_type`;
- тип `C` реализует операции создания пустого и непустого контейнера, присваивания, сравнения, возврата итераторов на начало (конец) и пр.;
- тип `C` отвечает требованиям `DefaultConstructible`, `CopyConstructible`, `EqualityComparable`, `Swappable`;
- тип `T` отвечает требованиям `CopyInsertable`, `EqualityComparable`, `Destructible`.



# Требования к типам и характеристики типов



Именованные требования к типам не следует смешивать с рассмотренными ранее характеристиками типов ([англ. type traits](#)), посредством которых для соответствующих нужд выделяются простые и составные категории типов ([англ. type categories](#)):

- **порядковые** типы (проверочный шаблон — `std::is_integral<>`, C++11);
- **вещественные** типы (`std::is_floating_point<>`, C++11);
- массивы, перечисления, классы, объединения (все — C++11);
- **арифметические** типы (порядковые или вещественные; `std::is_arithmetic<>`, C++11);
- **фундаментальные** типы (арифметические, `void` или `std::nullptr_t`; `std::is_fundamental<>`, C++11);
- **скалярные** типы (арифметические, указатели, указатели на члены классов или `std::nullptr_t`; `std::is_scalar<>`; C++11);
- **объекты** (скаляры, массивы, классы или объединения; `std::is_object<>`, C++11) и пр.

