

# C++

---

- C++
- Атрибуты
  - C++11
  - C++14
  - C++17
- Lambda
  - C++11
  - C++14
  - C++17
- POD-type
- auto/decltype
  - C++11
    - **Альтернативный синтаксис шаблонных функций**
  - C++14
- Literals
  - C++11
    - **Строковые литералы**
    - **Пользовательские литералы**
  - C++14
    - **Строковый литерал**
    - **Бинарные литералы**
    - **Разделители числовых литералов**
    - **STL литералы**
- Initialization
  - C++11
    - **Универсальная инициализация**
    - **std::initializer\_list**
  - C++14
    - **Aggregate initialization with default member initializer**
  - C++17
    - **auto + std::initializer\_list**
    - **Агрегатная инициализация базового класса**
- constexpr
  - C++11
  - C++14
  - C++17
- Шаблоны
  - C++11
    - **Вариативные шаблоны (Variadic template)**
    - **Extern templates**
  - C++14
    - **Шаблон переменной (Variable template)**
  - C++17

- **Выведение типов шаблонных аргументов**
  - **template auto**
  - **Fold expressions (свертка функций)**
  - **constexpr if**
- Спецификаторы
  - **'default' + 'deleted' specifiers**
  - **'override' + 'final' specifiers**
- Небольшие нововведения
  - C++11
    - **Move semantics**
    - **noexcept**
    - **Range based for cycle**
    - **Delegate constructors**
    - **Default values for non-static class members**
    - **nullptr**
    - **enum class**
    - **enum underlying type**
    - **Explicit cast operators**
    - **Relaxed rules for unions**
    - **static\_assert**
    - **alignof, alignas**
    - **'using' for types**
  - C++14
    - **Memory allocation ellision/combining**
  - C++17
    - **noexcept**
    - **Copy elision**
    - **Structure bindings**
    - **Последовательность операций вызова**
    - **'if' / 'switch' with initialization**
    - **inline variables**
    - **\_\_has\_include()**
    - **alignas (32)**
    - **static\_assert(true)**
    - **Nasted namespaces**
- STL
  - C++11
    - **Chrono**
    - **Random**
    - **Regex**
    - **Multithreading**
    - **Обновления вызванные новым стандартом**
    - **std::tuple**
    - **Accosicative unordered containers**
    - **Smart pointers**
    - **std::function**

- **std::reference\_wrapper**
- C++14
  - Гетерогенный поиск по ассоциативным контейнерам
  - Адресация элементов кортежа через тип
  - **std::make\_unique**
  - **std::exchange**
  - **rbegin, rend, cbegin, cend, rcbegin, rcend**
- C++17
  - **string\_view**
  - **std::to\_chars/std::from\_chars**
  - **std::optional**
  - **std::variant**
  - **std::any**
  - **std::filesystem**
  - **std::byte**
  - **std::apply**
  - **std::as\_const**
  - **std::clamp**
  - Ассоциативные контейнеры
  - **std::size, std::data, std::empty**
  - **non const std::string::data**
  - **std::not\_fn**
  - **emplace\_back**
  - **std::scoped\_lock**
  - **shared\_ptr для массивов**
  - Математические функции
  - **Parallel algorithms**
- Undefined behavior
  - Неуточненное поведение
  - Примеры undefined behavior
  - Более серьёзные, и менее очевидные случаи:
- Выведение типов лекция
  - Обзор
  - Правила вывода для шаблонов
    - **Правила вывода типов по значению**
    - **Правила вывода типов для указателей и ссылок**
    - **Правила вывода типов для forwarding reference**
  - Правила вывода для auto
  - Правила вывода для lambda capture-list
  - Правила вывода для decltype
  - + , + , +
  - RTTI
- TODO

## Атрибуты

---

## C++11

---

[[noreturn]]

Функция помеченная так не должна возвращать поток управления.

[[carries\_dependencies]]

Атрибут связан с моделями памяти.

## C++14

---

[[deprecated]]

Атрибут позволяет разметить устаревший код, вызывая warning'и при его использовании.

```
struct [[deprecated]] Name;
[[deprecated]] typedef S* pS;
using PS [[deprecated]] = S*;
[[deprecated]] int x;
union U { [[deprecated]] int n; }
[[deprecated]] void f();
namespace [[deprecated]] {NS { int x; }
enum [[deprecated]] E {};
enum E { a [[deprecated]], b [[deprecated]] = 1 };
template < > struct [[deprecated]] X<int> {};
```

## C++17

---

[[fallthrough]]

Используется для switch блоков, сообщая что оператор break не был пропущен по ошибке.

```
switch (x)
{
    case 1:
        [[fallthrough]] //No warning
    case 2:
        break;
    case 3: //Warning
    case 4:
        break;
}
```

[[nodiscard]]

Атрибут требует чтобы результат функции не был проигнорирован.

```
[[nodiscard]] bool isEmpty() { ... }

bool status = isEmpty(); //No warning

isEmpty(); //Warning - результат возвращаемый функцией проигнорирован
```

[[maybe\_unused]]

Атрибут убирает warning от неиспользуемых аргументов\переменных\функций итд.

```
struct [[maybe_unused]] S;
[[maybe_unused]] typedef S* PS;
using PS [[maybe_unused]] = S*;
[[maybe_unused]] int x;
union U { [[maybe_unused]] int n; };
[[maybe_unused]] void f();
enum [[maybe_unused]] E {};
enum { A [[maybe_unused]], B [[maybe_unused]] };
```

## Lambda

### C++11

Анонимные функции, вызываемого типа `std::function`, могут использоваться в STL.

Общий вид:

```
auto lamda = [capture-list](arguments) mutable -> ret_type
{
    ...
}; //Создание

//Если не указывать mutable - по дефолту он не включен

lambda(arguments); //Вызов
```

Списки захвата:

```
[] // ничего не захватывается
[=] // локальные переменные по значению
[&] // локальные переменные по ссылке
```

```
[this] // this по ссылке  
[a, &b] // захват отдельных переменных, по значению и ссылке
```

## C++14

---

Дополнены правила списка захвата:

```
[&r = x, x = x + 1]  
//в lambda можно захватить ссылку, и назвать её как удобно, и можно использовать  
выражение для инициализации переменной  
  
[x = factory(2)]  
[p = std::move(p)]  
  
//Пример генератора  
auto generator = [x = 0]() mutable { return x++; }  
int a = generator(); // == 0  
int b = generator(); // == 1
```

Так же перестал быть необходим trailing return type, для возвращаемого типа auto.

Были введены генерализированные lambdas, когда аргументы указаны типа auto.

## C++17

---

Добавлена возможность захвата текущего объекта по копии, а не по ссылке.

```
[*this]
```

Необходим спецификатор mutable, для того чтобы иметь возможность вызывать неконстантные версии функций класса.

## POD-type

---

Plain old data - структура размещающаяся в памяти таким образом, как её описал программист, исключая оптимизации. Это может быть необходимо для передачи данных в другие языки программирования.

POD = Тривиальный класс + Класс со стандартным размещением

Тривиальный класс:

- T() = default;
- T(const T&) = default;

- T& operator=(const T&) = default;
- T(T&&) = default;
- T& operator=(T&&) = default;
- ~T() = default;
- Нет виртуальных методов и виртуального наследования
- Все нестатические поля тривиальны
- Все базовые классы тривиальны(при наличии)

Класс со стандартным размещением:

- Все нестатические поля имеют одинаковый доступ private\public\protected
- Нет виртуальных методов и вирт. наследования
- Нет нестатических полей-ссылок
- Все нестатические поля и базовые классы со стандартным размещением
- Все нестатические поля объявлены в одном классе в иерархии наследования
- Нет базовых классов того же типа, что и первое нестатическое поле

## auto/decltype

---

auto - возможность замена типа на auto.

Примеры типов: переменной, возвращаемого значения функции, и шаблоных аргументов.

decltype() - позволяет выводить тип переменной или выражения.

## C++11

---

Особенности работы auto:

```
int bar();

auto i = 0; //int
auto ui = 0u; //unsigned int
volatile auto ci = i; //volatile int
const volatile auto cvi = i; // const volatile int
auto j = cvi; //int

auto& ri = i; //int &
const auto& cri = i; //const int&

auto&& fri = i; // int &
auto&& fcri = cri; // const int &

auto &&frv = 0; // int &&
auto &&frvf = bar(); // int &&
```

## Альтернативный синтаксис шаблонных функций

Позволяет выводить возвращаемый тип шаблонной функции.

```
template <typename T1, typename T2>
auto sum(const T1& lhs, const T2& rhs) -> decltype(lhs + rhs)
{
    return lhs + rhs;
}
```

## C++14

---

Не нужен trailing return type, достаточно auto, Можно реализовать функцию факториал с возвращаемым типом auto, но факториал от нуля должен быть определён до рекурсивного использования этой функции.

Так же было осуществлено послабление, теперь внутри decltype() можно указывать не выражение\переменную, а auto.

Примеры:

```
double foo();
double&& bar();

double v1 = 0.0; //double
const double& v2 = v1; //const double &

decltype(auto) v3 = v1; //double
decltype(auto) v4 = (v1); //double&
decltype(auto) v5 = v2; //const double&

decltype(auto) v6 = foo(); //double
decltype(auto) v7=bar(); //double &&
```

## Literals

---

### C++11

---

#### Строковые литералы

```
//Было до C++11

"Text" //char
L"Text" //wchar_t

//Появилось в C++11 - utf
```



```

u8"Text" //char - utf8
u"Text" //char16_t
U"Text"//char32_t

//Сырые строки обрамляются в ( ) в "" и могут иметь произвольны delemiter
R"delimiter( raw string )delimiter"
LR"delimiter( raw string )delimiter"
u8R"delimiter( raw string )delimiter"
uR"delimiter( raw string )delimiter"
UR"delimiter( raw string )delimiter"

```

## Пользовательские литералы

Пример пользовательского литерала преобразования радиан в градусы.

```

long double operator""_degrees(long double value)
{
    return value * M_PI / 180.0;
}

double degrees = 0.38__degrees

```

Список возможных аргументов, при определении пользовательского литерала:

```

( const char * )
( unsigned long long int )
( long double )
( char )
( wchar_t )
( char16_t )
( char32_t )
( const char * , std::size_t )
( const wchar_t * , std::size_t )
( const char16_t * , std::size_t )
( const char32_t * , std::size_t )

```

## C++14

### Строковый литерал

```

std::string from_literal = "some string"s;

```

## Бинарные литералы

```
int a = 0b111; // == 7
int b = 0B11; // == 3
```

## Разделители числовых литералов

```
int a = 1'000'000;
int b = 3.14'15'92'65;
```

## STL литералы

```
auto half_minute = 30s; // std::chrono::duration
auto day = 24h; // std::chrono::duration

auto complex = 1 + 1i; //std::complex
```

# Initialization

---

## C++11

---

### Универсальная инициализация

Везде можно использовать {}:

```
// До C++11

int a;           //(1) default init
int b(2);        //(2) direct init
int c = 2;       //(3) copy init
int d = int();   //(4) value init
int arr[] = {1, 2, 3}; //(5) aggregate init

struct Point { double x, y; } point (0.0, 0.0); //(5)
std::complex<double> cml(0.0, 0.0); //(2)
std::complex<double> c2 = std::complex<double>(0.0, 0.0); //(3)

// Начиная с C++11 можно везде {}

int a;
int b{2};
int c = {2};
int d{};
```

```
int arr[] = {1, 2, 3};

struct Point { double x, y; } point {0.0, 0.0}; //(5)
std::complex<double> cml{0.0, 0.0}; //(2)
std::complex<double> c2 = std::complex<double>{0.0, 0.0}; //(3)
```

## std::initializer\_list

Возможность использовать список инициализации для создания конструкторов или операторов присвоения.

Значения задаются между {} и через запятую.

initializer\_list содержит следующие функции:

```
auto init_list = initializer_list<int> { 1, 2, 3};
init_list.size();
init_list.begin();
init_list.end();

init_list.r/c/begin/end(); // Начиная с C++14

init_list.empty(); // Начиная с C++17
init_list.data(); // Начиная с C++17
```

## C++14

---

### Aggregate initialization with default member initializer

```
struct x
{
    int a,b;
    char c = '0';
};

x v { 1, 2 }; // До C++14 нельзя было опустить третье поле "c"
```

## C++17

---

### auto + std::initializer\_list

```
// До C++17

auto v1 { 1, 2, 3}; // std::initializer_list<int>
```

```

auto v2 = { 1, 2, 3, }; // std::initializer_list<int>
auto v3 {42};           // std::initializer_list<int>
auto v4 = { 42 };       // std::initializer_list<int>

// Начиная с C++17

auto v1 { 1, 2, 3};      // compile error
auto v2 = { 1, 2, 3, }; // std::initializer_list<int>
auto v3 {42};           // int
auto v4 = { 42 };       // std::initializer_list<int>

```

## Агрегатная инициализация базового класса

Возможность вложенной инициализации:

```

struct Base
{
    std::string name;
    std::string sur_name;
};

struct Child : public Base
{
    int age;
}

Child ch1; //name, sur_name - empty, age undefined
Child ch2{}; //all fields empty

Child ch3 {"name", "sur", 99};
Child ch4 {"name", "sur", 99};

```

## constexpr

### C++11

Функции помеченные constexpr могут вычислять на этапе компиляции. Изначально такие функции имели большое количество ограничений, например должны были состоять из только 1 блока return.

### C++14

Ограничения были существенно ослаблены. Запрещенным остались:

```
__asm__  
goto  
метки, кроме case\default в switch,  
блок try,  
переменные нелитерального типа,  
static \ thread_local переменные,  
переменные без инициализации
```

Так же они удобны для применения в шаблонной магии, например в вариативных шаблонах, о них ниже.

## C++17

---

Лямбда может быть помечена как constexpr:

```
constexpr auto add = [](int a, int b) { return a + b; }
```

Если она может быть вызвана на этапе компиляции - это будет осуществлено, иначе она будет работать в run-time.

## Шаблоны

---

### C++11

---

#### Вариативные шаблоны (Variadic template)

Используются для создания функций с переменным числом аргументов:

```
template <typename... Args>  
void printf(const char* const format, const Args&... args);  
  
//При вызове  
printf("test", 1, 0.1);  
  
// Произойдёт инстанцирование  
printf<int, double>("test", 1, 0.1);
```

Помимо этого, используются в кортежах (tuple).

#### Extern templates

Используются с целью осуществить единичное инстанцирование при компиляции, для её ускорения.

```
extern template void foo<int>(int);  
extern template class SomeClass<int>;
```

## C++14

---

### Шаблон переменной (Variable template)

```
template <class T>  
struct is_reference  
{  
    static constexpr bool value = false;  
};  
  
template <class T>  
struct is_reference<T&>  
{  
    static constexpr bool value = true;  
};  
  
template <class T>  
struct is_reference<T&&>  
{  
    static constexpr bool value = true;  
};  
  
template <typename T>  
constexpr bool is_reference_v = is_reference<T>::value;  
  
static_assert(!is_reference_v<SomeType>, " SomeType is reference");
```

## C++17

---

### Выведение типов шаблонных аргументов

Возможность не использовать указание типа шаблонного параметра в <>:

```
std::pair m {0, 0}; //Вместо std::pair<int, int> { 0, 0};  
std::vector v { 0.0 }; // Вместо std::vector<double> { 0.0; }  
std::lock_guard lock(mutex); // Вместо std::lock_guard<std::mutex>
```

Так же deduction guide может быть определен вручную. Пример для std::array:

```

namespace std
{
template <class T, size_t N>
struct array
{
    T arr[N];
};

template <class T, class... U>
array(T, U...) -> array<T, sizeof...(U) + 1>

};

//Тогда возможно использование
std::array arr {0, 1, 2, 3}; //Вместо std::array<int, 4>;

```

## template auto

Полезно для template not-type параметров.

```

template <auto Val> // Эквивалент template <decltype(auto) Val>
struct integral_const
{
    using value_type = decltype(Val);
    static constexpr value_type value = Val;
};
using true_type = integral_const<true>; //Не требуется задавать тип вручную
using false_type = integral_const<false>; //integral_const<bool, false>

//Схожий пример:
template <auto.. seq>
struct my_sequence
{
    ...
};

auto seq = std::integer_sequence<int, 0, 1, 2>(); //int задан явно
auto seq2 = my_sequence<1, 2, 3>(); //int будет выведен из значений

```

## Fold expressions (свертка функций)

Позволяет записывать операции для вариативного числа шаблонных аргументов:

```

template <typename T, typename ..Types>
constexpr auto sum(T t1, Types ..tN)
{
    return (t1 + ... + tN);
}

```

```
constexpr size_t res = sum(0, 1, 2, 3);
```

Четыре вида свёрток функций:

```
(pack op ...) = (E_1 op (... op (E_N-1 op E_N)))
(... op pack) = (((E_1 op E_2) op ...) op E_N)
(pack op ... op init) = (E_1 op (... op (E_N-1 op (E_N op I))))
(init op ... op pack) = (((I op E1) op E2) op ...) op E_N
```

Операции:

```
op:
+, -, *, /, %, ^, &, |, =, <, >, <<, >>,
+=, -=, *=, /=, %=, ^=, &= |=,
<=<=, >>=, ==, !=, <=, >=, &&, ||, .*, ->*
и оператор ,
```

Начиная с C++17 возможна запись:

```
template <typename ...Types>
void print(const Types& ...tN)
{
    std::cout << ... << tN;
}
```

## constexpr if

Метод разметить ветки для шаблонов:

```
template <size_t N>
decltype(auto) get(const Person& )
{
    if constexpr (N == 0)
    {
        return p.Name();
    }
    else if constexpr (N == 1)
    {
        return p.GetSurname();
    }
}
```



# Спецификаторы

---

## 'default' + 'deleted' specifiers

---

Возможность либо пометить удалённой и недопустимой функцию (deleted). Либо реализовать стандартное поведение для конструкторов\операторов присваивания итд.

- 1. Дефотный конструктор
- 2. Констуктор копирования
- 3. Конструктор перемещения
- 4. Оператор копирования
- 5. Оператор перемещения

Если компилятор может - он постарается вывести поехсепт версии функций

## 'override' + 'final' sepcifiers

---

override - указывает на то, что функция переопределяет виртуальную функцию из наследуемого класса.

final - не даст переопределять функции дальше, т.е. означает что это финальная версия перезагруженной функции.

```
virtual void foo(int) const override {}

virtual void foo(int) const final {}
```

Так же final может запретить дальнейшее наследование, если мы хотим создать класс\структуру, от которой нельзя наследоваться дальше.

# Небольшие нововведения

---

## C++11

---

### Move semantics

Добавлен новый тип r-value ссылка T&&, который представляет собой временное значение, например результат вычисления выражений или результат вызова функций.

Для того чтобы перенести такое значение без копирования введена специальная функция std::move().

Move семантика полезна когда объект тяжелый для копирования, но легкий для перемещения. Или же когда объект запрещено копировать, например unique\_ptr.

### noexcept

Метод пометить функцию, что она не должна вызывать исключения.

Необходимо для создания move-конструктора и оператора присвоения, если они не помечены как noexcept будут вызываться конструктор копирования и оператор копирования (например при создании векторов нашего произвольного класса).

## Range based for cycle

Вызов цикла в конструкции вида:

```
for (const auto& element: container)
{
    ...
}
```

Где container это класс с функциями begin/end, возвращающих итератороподобный объект, который должен уметь инкрементироваться и разыменовываться как указатель.

## Delegate constructors

Возможность вызова одного из конструкторов из тела другого.

## Default values for non-static class members

Возможность проинициализировать переменную класса в месте её определения

## nullptr

Общий тип для обозначения пустых указателей. Можно перегружать функции, используя std::nullptr\_t как аргумент.

## enum class

Не позволяет сравнивать поля разных enum'ов.

## enum underlying type

Позволяет задать тип, в котором хранится перечисление, например:

```
enum X : int
{
    A,
    B
};
```

Тем самым можно задать размер переменной типа X.

## Explicit cast operators

Операторы явного каста:

```
class P
{
    explicit operator bool() { return ...; }
};

P ptr;
int flag = ptr; // Преобразования не будет,
//т.к. помечено explicit: ошибка компиляции
```

## Relaxed rules for unions

До 11 стандарта можно было использовать только POD внутри union.

Теперь почти любой, но важно для юнона так же объявить конструктор, если он есть у вложенной структуры. Но в 17 стандарте это стало не обязательным для реализации.

## static\_assert

Возможность использования ассертов на этапе компиляции, условие + строка сообщения, например:

```
static_assert(std::is_pod(variable), "ERROR: !!");
```

В 17 стандарте строка стала не обязательной.

## alignof, alignas

Позволяет использовать нужное выравнивание или узнать его

## 'using' for types

Более современная замена typedef, способная принимать шаблонные аргументы:

```
typedef std::vector<int>::iterator vec_iter;

template <typename T>
typedef std::vector<T>::iterator vec_t_iter;
//Ошибка при компиляции

Альтернативная запись:
using vec_iter = std::vector<int>::iterator;

template <typename T>
using vec_t_iter = std::vector<T>::iterator;

vec_t_iter<int> it; //ok!
```

## C++14

---

### Memory allocation ellision/combining

Вызовы new\delete могут оптимизироваться.

## C++17

---

### noexcept

Спецификатор того, что функция не выбрасывает исключения - теперь часть системы типов функции.

```
typedef void (*nef)() noexcept;
typedef void (*ya)();

void foo() noexcept;
void bar();

ef pf1 = foo; // +
nef pf2 = foo; // +
ef = bar; // +
nef = bar; //Compile error
```

### Copy elision

Создание объекта не при выходе из функции, а в месте его последующего применения, там где эта функция вызывалась.

### Structure bindings

Возможность раскрутить группу значений в серию переменных. Можно раскрыть:

- array
- tuple
- pair/structure

```
const auto& [field1, field2, field2] = structure/tupple/..
```

Можно реализовать для произвольного класса:

```
template <size_t N>
decltype(auto) get(const Person&);

template <>
decltype(auto) get<0>(const Person& p)
```

```

{
    return p.GetName();
}

template <>
decltype(auto) get<1>(const Person& p)
{
    return p.GetSurname();
}

// Далее нужно определить tuple_size в std::

namespace std
{
    template <>
    struct tuple_size<Person> : std::integral_constant<size_t, 2>
    {};

    template <>
    struct tuple_element<0, Person>
    {
        using type = const std::string &;
    };

    template <>
    struct tuple_element<1, Person>
    {
        using type = const std::string &;
    };
}

```

## Последовательность операций вызова

```

a.b
a->b
a->*b
a(b1, b2, b3)
// b1, b2, b3 не последовательны
// их порядок не определен
b @= a
a[b]
a << b << c
a >> b >> c

```

## 'if' / 'switch' with initialization

Возможность задать значение в теле условия:

```
if (int a = f(5); a > 2)
{
    //а существует здесь
}
//а не существует здесь
```

Можно использовать structure bindings на этапе if initialization. Тем самым подготовить сразу несколько переменных для условий и вычислений.

## inline variables

Необходимы чтобы быть разделяемыми между файлами, будучи определенными в хэдере. Или для функций - чтобы писать определение прямо в хэдере.

## \_\_has\_include()

Директива препроцессора, проверяет наличие хэдеров

## alignas (32)

Теперь выравнивание структуры по границе заданной, при динамическом размещении

## static\_assert(true)

Теперь можно использовать без строки, просто 1 условие

## Nasted namespaces

```
namespace A::B::C {
    int i;
}

//Эквивалентно:
namespace n1 {
    namespace n2 {
        int n;
    };
};

//Вызов
n1::n2::n;
```

---

# STL

---

# C++11

---

## Chrono

Используется для измерения времени:

```
#include <chrono>

template <class Clock, class Duration = typename Clock::duration>
std::chrono::time_point; //Тип для хранения момента времени

std::chrono::system_clock; //Возможные типы отсчётов
std::chrono::high_resolution_clock;
std::chrono::steady_clock; //Наиболее приоритетный

auto start = std::chrono::steady_clock::now();
auto end = std::chrono::steady_clock::now();

std::chrono::duration<double> elapsed_seconds = end - start;
auto durMs = duration_cast<std::chrono::milliseconds>(end - start);

//Другие варианты для std::chrono::duration_cast:
std::chrono::nanoseconds;
std::chrono::microseconds;
std::chrono::milliseconds;
std::chrono::seconds;
std::chrono::minutes;
std::chrono::hours;
```

## Random

Используется для генерации случайных чисел.

```
Random number engines
{
    linear_congruential_engine,
    mersenne_twister_engine,
    subtract_with_carry_engine
};

Random number engine adaptors
{
    discard_block_engine,
    independent_bits_engine,
    shuffle_order_engine
};

Predefined generators
{
    minstd_rand0,
    minstd_rand,
    mt19937,
```

```

    mt19937_64,
    ranlux24_base,
    ranlux48_base,
    ranlux24,
    ranlux48,
    knuth_b,
    default_random_engine
};

Non-deterministic random numbers : random_device;

//Внутри каждого из них есть несколько вариаций
Distributions
{
    Uniform distributions,
    Bernoulli distributions,
    Poisson distributions,
    Normal distributions,
    Sampling distributions
};

```

Пример:

```

#include <random>

std::mt19937_64 engine { std::random_device{}() };
std::uniform_int_distribution<> distr { 0, 100 };
std::cout << distr(engine);
auto generator = std::bind(distr, engine);
std::cout << generator();

```

## Regex

Регулярные выражения:

```

#include <regex>

std::regex pattern { R"((\d{2}).(\d{2}).(\d{2,4}))"};
std::string str{"I was born 01.02.1993"};

for (auto it = std::sregex_iterator {str.begin(), str.end(), pattern},
      end = std::sregex_iterator {}; it != end; ++it)
{
    auto&& match = *it;
    std::string day = match[1]; //01
    std::string day = match[2]; //02
    std::string day = match[3]; //1993
    std::string day = match[4]; // ""
}

```



```
//Другой вариант использования - замена:  
  
auto replaced = std::regex_replace(str, pattern, "xx.xx.xxxx");
```

## Multithreading

Используется для реализации многопоточных или асинхронных приложений.

```
#include <thread>  
  
// Потоки и синхронизация:  
std::thread  
std::mutex  
std::recursive_mutex  
std::timed_mutex  
std::recursive_timed_mutex  
std::conditional_variable  
  
// Модели и барьеры памяти:  
std::memory_order  
std::atomic_thread_fence  
  
// Атомарные переменные:  
std::atomic  
  
// Асинхронные вычисления:  
std::future  
std::packaged_task  
std::promise
```

## Обновления вызванные новым стандартом

```
// конструирование на месте, на подобии как make_pair: только 1 вызов move  
конструктора  
std::container<T>::emplace();  
std::container<T> ::cbegin(), ::end(), std::begin, std::end;  
  
// если unordered контейнер, когда есть ясность куда вставить значение - это может  
улучшить скорость  
std::associative_container<T>::emplace_hint();  
  
// обрезать по границе использования  
std::seq_container<T>::shrink_to_fit();  
std::vector<T>::data();  
std::list<T>; // complexity constraints
```

## std::tuple

Можно использовать функцию `make_tuple()`.

Доставать значения можно `std::get(v)`;

Функция `tie` - которая может сформировать tuple от левых ссылок, `std::tie(name, surname) = get_person(1)`;

В C++17 он перестаёт быть нужен, но можно им сравнивать группы значений:

```
std::tie(year, month, day) > std::tie(year2, month2, day2);
```

## Accosicative unordered containers

`unordered_set`, `_multiset`, `_map`, `_multimap`,

Поиск за  $O(1)$ , как и вставка\удаление. Но зависит от количества элементов на bucket'e.

## Smart pointers

```
//Можно настроить делитер - который закроет файл
std::unique_ptr<FILE, decltype(deleter)>;

std::unique_ptr<T>
std::shared_ptr<T>
std::weak_ptr<T> //решение для перекрестных ссылок
```

## std::function

Обертка для callable объекта, которым может выступать лямбда.

Или результат `std::bind`.

## std::reference\_wrapper

Модулирование поведения ссылки. Нужны для thread'ов - чтобы протолкнуть объект по ссылке

`std::ref` + `std::cref` - функции помогающие сгенерировать объект типа `reference_wrapper`.

## C++14

---

## Гетрогенный поиск по ассоциативным контейнерам

```
//Гетрогенный компаратор less
std::set<std::string, std::less<>> elements { ... };
//При вызове не будет формироваться новые std::string для сравнения:
elements.find("const char*");
```

## Адресация элементов кортежа через тип

Стала доступна адресация по типу `::get()`.

Если будет указан несуществующий тип - ошибка будет на этапе компиляции. Но элементов с одинаковым типом не должно быть, для корректной работы функции.

## **std::make\_unique**

Подобие `make_shared`, `make_pair`, `make_tuple`.

## **std::exchange**

`std::exchange( объект, следующее его значение )`. Результат вызова это изначальный объект.

Можно использовать чтобы пробежать по массиву и обнулить его:

```
for (const auto x: std::exchange(vec, {}))
    std::cout << x << std::endl;
```

Другая область использования это реализация своего `move` конструктора, или `move` оператора присваивания.

## **rbegin, rend, cbegin, cend, rbegin, rcend**

Константные и реверсивные интераторы для контейнеров.

## **C++17**

---

## **string\_view**

Обобщенный и легковесный вариант для хранения строчек `std::string`/`c_string` `std::string_view` // `std::wstring_view`

## **std::to\_chars/std::from\_chars**

Функции преобразования цифр. Может содержать ошибку парсинга.

## **std::optional**

Хранит либо значение, либо `nullopt`

```
#include <optional>

std::optional<int> opt = 3;

opt.has_value(); // == if (optional)
opt.value(); // == *optional

//Возвращает значение, если оно есть, или переданный объект:
opt.value_or({});
```

```
//Операции сравнения в условиях с нижележащим классом
if (optional > 2) {}
```

## std::variant

Метод хранения множества разнотипных значений вместе:

```
#include <variant>

std::get<0>();
std::get<std::string>();

//Возвращает const type* ptr, или nullptr если не удалось преобразовать к типу
std::get_if<type>(variant);

//возможность установки базового состояния variant
//на случай если другие объекты не имеют конструктора по умолчанию
std::monostate;

//Можно всё обработать единственной лямбдой с auto аргументом
std::visit( [](auto arg) { std::cout << arg << ' '; }, v);
```

## std::any

Принимает произвольный тип, но почти всегда происходит динамическая локация. Если возможно, лучше использовать variant.

Пример:

```
std::any x{5};
x.has_value(); // == true
std::any_cast<int>(x); // == 5
```

## std::filesystem

Позволяет использовать функции доступа к файловой системе:

```
if (std::filesystem::exists(my_path))
{
    const auto fileSize { std::filesystem::file_size(my_path)};
    std::filesystem::path tmpPath { "/tmp"};
    if (std::filesystem::space(tmpPath).available > fileSize )
    {
        std::filesystem::create_directory(tmpPath.append("example"))
        std::filesystem::copy_file(my_path, tmpPath.append("newFile"))
    }
}
```

```
}
}
```

## std::byte

```
//Новый тип для хранения "сырых" байтов, перегружен
std::byte a { 0 };
int x = std::to_integer<int>(a);
```

## std::apply

Применение функции к tuple\pair:

```
auto add = [](int x, int y)
{
    return x + y;
};
std::apply(add, std::make_tuple(2, 3)); // == 5
std::apply(add, std::make_pair(1, 2)); // == 3
```

## std::as\_const

Обертка для получение const-ref.

## std::clamp

Клипует значение по 2м границам - верхней и нижней.

## Ассоциативные контейнеры

Добавлены функции: try\_emplace, insert\_or\_assign.

Добавлены функции: extract, insert, merge.

```
// merge:
std::set<int> src { 1, 3, 5};
std::set<int> dst { 2, 4, 5};
dst.merge(src);
// dst == {1, 2, 3, 4, 5}
// src == {5} !!!

// extract\insert - позволяют move'нуть объект из одного контейнера, в другой
// Или изменить ключ у поля
std::map m;
auto e = m.extract(2); // key == 2
e.key() = 4;
m.insert(std::move(e));
```

## **std::size, std::data, std::empty**

Свободные обобщенные функции для всех контейнеров.

## **non const std::string::data**

Доступ к сырой памяти строки.

## **std::not\_fn**

Wrapper возвращающий отрицательное\обратное значение функции.

## **emplace\_back**

Функции теперь возвращают ссылку на объект.

## **std::scoped\_lock**

Возможность использовать несколько мьютексов в одном локе.

## **shared\_ptr для массивов**

TODO дополнить.

## **Математические функции**

TODO дополнить + (std::gcd, std::lcm).

## **Parallel algorithms**

Возможность использовать параллельные вычисления в стандартных алгоритмах.

TODO дополнить с примерами.

---

# Undefined behavior

---

Стандарт языка допускает **неопределенное поведение**, в некоторых ситуациях. Это сделано с целью сделать код наиболее эффективным и быстрым, и не платить за дорогие проверки.

## Неуточненное поведение

**Неуточненное поведение** или **поведение определяемое реализацией** - поведение, которое может различаться на разных платформах и компиляторах, т.к. спецификация языка предлагает несколько доступных вариантов реализации конструкции.

В отличие от **неопределённого поведения**, программа с неуточненным поведением с точки зрения соответствия спецификации языка не считается ошибочной. Но писать такой код - плохая идея.

```
int a = 0;
// Неуточненное поведение:
foo(a = 2, a);
// Последовательность вычисления аргументов не гарантирована стандартом
```

```
// -1 знаковое целое, вычисление b будет неуточненным поведением:
int b = (-1) >> 5;
```

## Примеры undefined behavior

```
void foo()
{
    int a[10];
    //Выход за границу массива:
    a[22] = 10;
}
```

```
struct Base
{
    //virtual ~Base() = default;
    virtual void f();
}

struct Derived : Base {};

void foo()
{
    Base* b = new Derived();
    delete b; // UB т.к. нет виртуального деструктора в Base
}
```

```
auto p1 = new int[10];
delete p1; //Должно быть delete[]

auto p2 = new int;
delete[] p2; //Должно быть delete

auto p3 = new int[10];
free(p3); //Должно быть delete[]

auto p4 = new int;
free(p4); //Должно быть delete
```

## Более серьёзные, и менее очевидные случаи:

```
int try_init(struct usb_line6_podhd* podhd)
{
    //Отсутствует проверка что podhd != nullptr
    struct usb_line* line6 = &podhd->line6;

    //Тут у нас уже возможно UB:
    if (podhd == nullptr) //Проверять надо раньше
        return -ENODEV;
    //Компилятор может оптимизировать условие!
}
```

### Пример из JPEG:

```
//Схожая ситуация, как с >>
((-1) << 2) + 1;
//Правильный unsigned вариант
((~0u) << 2) | 1;
```

### Целочисленное переполнение:

```
size_t count = (size_t)(5) * 1024 * 1024 * 1024; // 5 Gb
//... выделим array размера count

// count не поместится в int, если он 32
for (int i = 0; i != count; ++i)
    //Произойдёт переполнение i
    array[i] = (char)(i) | 1;

//Если вдруг count == 0, тут тоже UB
if (array[count - 1] == 0)
    std::cout << "Issue";
```

```
int foo(const unsigned char* s)
{
    int r = 0; //Fix: unsigned
    while (*s)
    {
        //Возможно переполнение r
        //Но это не рассматривается, т.к. запрещено переполнять знаковые числа
        r += ((r * 20891 + *s * 200) | *s ^ 4 | *s ^ 3) ^ (r >> 1);
        s++;
    }
    //Компилятор может оптимизировать и убрать операцию ниже
    //Т.к. суммация положительного числа с положительным
```



```
    return r & 0x7fffffff  
    // Станет: return r;  
    // И мы вернём отрицательное число, после оптимизации  
}
```

## Выведение типов лекция

---

До C++11 вывод типов применялся только в шаблонах.

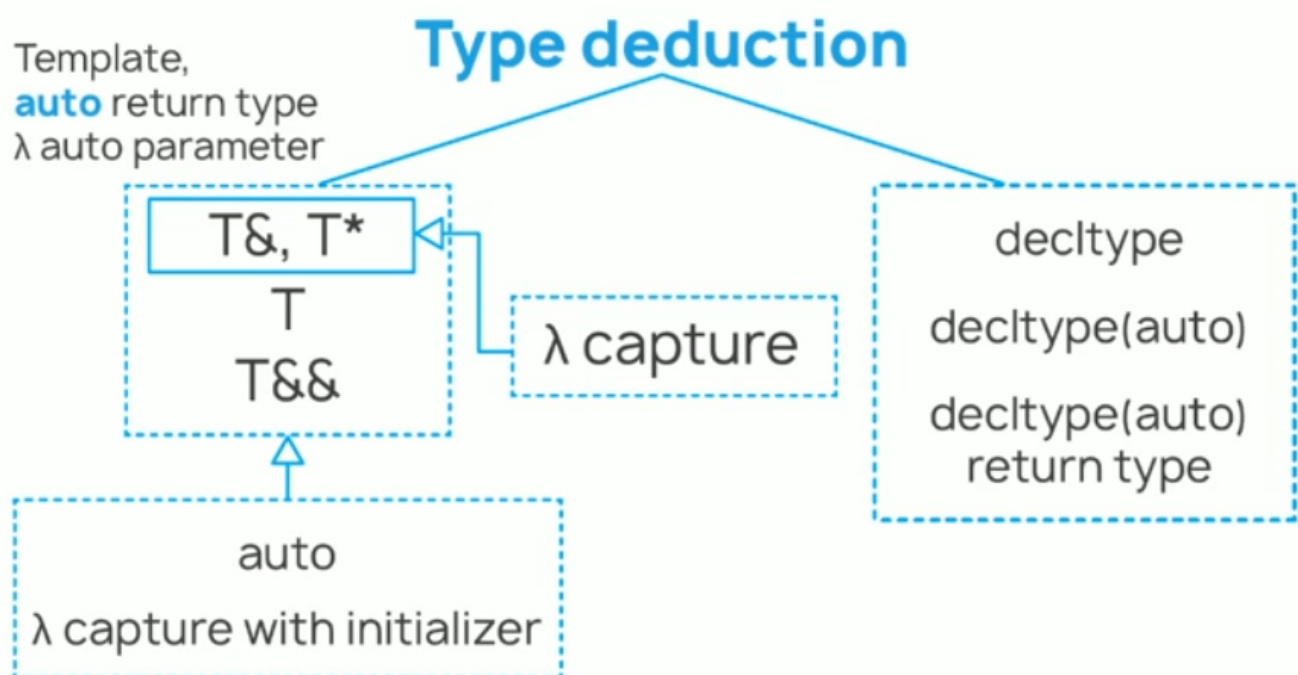
Потом приехали новые конструкции языка.

C++11: r-value/forwarding reference, auto, decltype, lambda capture, return type deduction for lambda.

C++14: function return type deduction, lambda caption with initialization.

### Обзор

---



Изначально было 2 типа правил, для вывода шаблонных типов:

- для указателей и ссылок
- для обычных типов

В C++11 появились r-value ссылки, которые в шаблонах работают не совсем как r-value, а как forwarding reference и в зависимости от того чем инициализируется становится либо r-value либо l-value ссылкой.

Появилось ключевое слово auto, которое наследует правила вывода всех шаблонных аргументов.

Далее появилось ключевое слово decltype.

Появились списки захвата lambda, которые наследуют правила вывода типов для ссылок и указателей.

Появился вывод типов lambda, который как auto наследует правила вывода шаблонных аргументов.

## Правила вывода для шаблонов

---

### Правила вывода типов по значению

Отбрасываются ссылки, const, volatile:

```
template <typename T>
void foo(T param); //param типа T

int i = 0;           // int
int &ri = i;         // int&
const int &rci = i;   // const int&
volatile int &rvi = i; // volatile int&
const volatile int &rcvi = i; // const volatile int&

foo(ri);   //T = int, param тип = int
foo(rci);  //T = int, param тип = int
foo(rvi);  //T = int, param тип = int
foo(rcvi); //T = int, param тип = int
```

```
//Если заменить на const T:
template <typename T>
void foo(const T param); //param типа T

//Тогда:
foo(ri);   //T = int, param тип = const int
foo(rci);  //T = int, param тип = const int
foo(rvi);  //T = int, param тип = const int
foo(rcvi); //T = int, param тип = const int

//Тоже самое для void foo(volatile T param);
//T = int, param тип = volatile int
```

Отбрасывается модификатор для указателя (const\volatile):

```
template <typename T>
void foo(T param); //param типа T

int i = 0;           //int
const int* pci = &i; //const int*
volatile int* pvi = &i; //volatile int*

//const int * const
```

```

const int* const cpci = &i;
//volatile int * volatile
volatile int* volatile vpvi = &i;

//cv int * cv
const volatile int* const volatile cvpcvi = &i;

foo(pci);    //T = const int*, param тип = const int*
foo(pvi);    //T = volatile int*, param тип = volatile int*
foo(cpci);   //T = const int*, param тип = const int*
foo(vpvi);   //T = volatile int*, param тип = volatile int*
foo(cvpcvi); //T = cv int*, param тип = cv int*

```

```

template <typename T>
void foo(T param);

void bar();
int arr[10]; //int[10]

foo(arr); //T = int*, param тип = int*
foo(bar); //T = void(*)(), param тип = void(*)()

foo({1, 2, 3}); //ERROR: fails to deduce type

```

## Правила вывода типов для указателей и ссылок

Если передаётся значение, у которого есть референс - он отбрасывается, остальные модификаторы сохраняются:

```

template <typename T>
void foo(T& param);

int i = 0;
const int ci = i;
volatile int vi = i;
const volatile int cvi = i;

foo(i); // T = int, param тип = int&
foo(ci); // T = const int, param тип = const int&
foo(vi); // T = volatile int, param тип = volatile int&
foo(cvi); // T = cv int, param тип = cv int&

//Если добавить ссылки перед ci, vi, cvi
//То результат не изменится

```

Если наш параметр должен быть ссылкой на константный объект:

```

template <typename T>
void foo(const T& param);

int i = 0;           // int
int &ri = i;         // int&
const int &rci = i;   // const int&
volatile int &rvi = i; // volatile int&
const volatile int &rcvi = i; // const volatile int&

foo(ri); //T = int, param тип = const int&
foo(rci); //T = int, param тип = const int&
foo(rvi); //T = volatile int, param тип = cv int&
foo(rcvi); //T = volatile int, param тип = int cv int&

```

Для указателей действуют схожие правила:

```

template <typename T>
void foo(T* param);

int i = 0;
int* pi = &i;
const int* pci = &i;
volatile int* pvi = &i;
const volatile int* pcvi = &i;

foo(pi); // T = int, param тип = int*
foo(pci); // T = const int, param тип = const int*
foo(pvi); // T = volatile int, param тип = volatile int*
foo(pcvi); // T = const volatile, param тип = const volatile int*

```

При добавлении константности для указателей:

```

template <typename T>
void foo(const T* param);

int i = 0;
int* pi = &i;
const int* pci = &i;
volatile int* pvi = &i;
const volatile int* pcvi = &i;

foo(pi); // T = int, param тип = const int*
foo(pci); // T = int, param тип = const int*
foo(pvi); // T = volatile int, param тип = volatile int*
foo(pcvi); // T = volatile int, param тип = const volatile int*

```

```

template <typename T>
void foo(T& param);

void bar();
int arr[10]; //int[10]

foo(arr); //T = int [10], param тип = int(&)[10]
foo(bar); //T = void(), param тип = void(&)()

foo({1, 2, 3}); //ERROR: fails to deduce type

```

## Правила вывода типов для forwarding reference

Если передается ссылка на объект l-value, т.е. объект у которого есть имя и адрес, тогда аргумент ссылка на l-value.

Если передаётся временный объект, то раскрывается аргумент на r-value ссылка.

```

template <typename T>
void foo(const T&& param);

int i = 0;           // int
int &ri = i;         // int&
const int &rci = i;   // const int&
volatile int &rvi = i; // volatile int&
const volatile int &rcvi = i; // const volatile int&

foo(ri);  //T = int&, param тип = int&
foo(rci); //T = const int&, param тип = const int&
foo(rvi); //T = volatile int&, param тип = volatile int&
foo(rcvi); //T = cv int&, param тип = int cv int&
foo(42);  //T = int, param тип = int&&

```

Подобное поведение было необходимо для реализации `emplace_back`.

```

//Плохо: копирование
template <class... Args>
void emplace_back(Args... args);

//Лучше - ссылки
template <class... Args>
void emplace_back(Args&... args);

//Идеально
template <class... Args>
void emplace_back(Args&... args)
{
    T* ptr = ....; //Memory region from allocator

```

```
new (ptr) T { std::forward<Args>(args)...}; //TODO placement new в конспект
}
```

TODO более детально про std::forward.

## Правила вывода для auto

---

```
int i = 0;           // int
int &ri = i;         // int&
const int &rci = i;   // const int&
volatile int &rvi = i; // volatile int&
const volatile int &rcvi = i; // const volatile int&
```

//Все auto = int, все типы переменных = int:

```
auto a_i = i;
auto a_ri = ri;
auto a_rci = rci;
auto a_rvi = rvi;
auto a_rcvi = rcvi;
```

//Для задания переменной со спецификатором:

```
const auto ca_i = i;
volatile auto va_i = ri;
volatile auto va_i = rvi;
const volatile auto cva_i = rcvi;
//Полный тип переменной = specifiers + int
```

При указании ссылки, работают правила вывода ссылки в шаблонах:

```
auto& a_i = i;           //auto == int, var type = int&
auto& a_ri = ri;         //auto == int, var type = int&
auto& a_rci = rci;       //auto == const int, var type = const int&
auto& a_rvi = rvi;       //auto == volatile int, var type = volatile int&
auto& a_rcvi = rcvi;     //auto == cv int, var type = cv int&
```

При добавлении спецификаторов немного меняется поведение:

```
int i = 0;           // int
int &ri = i;         // int&
const int &rci = i;   // const int&
volatile int &rvi = i; // volatile int&
const volatile int &rcvi = i; // const volatile int&

auto& a_i = i; //auto = int, var type = int&
```

```
const auto& ca_rci = rci; //auto = int, var type = const int&
volatile auto& va_rvi = rvi; //auto = int, var type = volatile int&
const volatile auto& cva_rcvi = rcvi; //auto = int, var type = cv int&
```

При применении двойного амперсанда:

```
int foo();
int&& bar();

int i = 0;      // int
int &ri = i;    // int&
int &rri = 42;  // int&&

auto&& a_i = i; //auto = int, var type = int&
auto&& a_ri = ri; //auto = int, var type = int&

auto&& a_foo = foo(); //auto = int, var type = int&&
auto&& a_bar = bar(); //auto = int, var type = int&&
```

Пример с массивом и функцией:

```
void bar();
int arr[10];

auto& rarr = arr; // auto = int[10], var type = int(&)[10]
auto& abar = bar; // auto = void(), var type = void(&)()

auto parr = arr; // auto = int*, var type = int*
auto pbar = bar; // auto = void(*)(), var type = void(*)()

auto init_list1 {1, 2, 3}; // auto = std::initializer_list<int>
auto init_list2 = {1, 2, 3}; // auto = std::initializer_list<int>

auto err_list = {1, 0.2}; // не удастся вывести тип
```

## Правила вывода для lambda capture-list

---

33-46

## Правила вывода для decltype

---

+ , + , +

---

## RTTI

---

---

---

# TODO

---

++ Forwarding reference

```
template <class T>
class A
{
    template <class U>
    void foo(T&& t, U&& u);
};
```

дополнить и изучить внимательней

++ inline namespaces тоже в 11 фитчи

++std::invoke

++ Searcher function objects

++ общие фитчи языка вроде const\volatile итд - из конспектов курсеры

+++ Идеомы

+++ шпоры filesystem +?

++ advanced constexpr?

++ TODO скользкие места C++ в UB