

- 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**
- Небольшие нововведения
  - C++11
  - Спецификаторы

- **'default' + 'deleted' specifiers**
  - **'override' + 'final' specifiers**
  - **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 elision/combining**
- C++17
  - **noexcept**
  - **Copy elision**
  - **Structure bindings**
  - **Последовательность операций вызова**
  - **'if' / 'switch' with initialization**
  - **inline variables**
  - **\_\_has\_include()**
  - **alignas (32)**
  - **static\_assert(true)**
  - **Nested namespaces**
- Атрибуты
  - C++11
  - C++14
  - C++17
- 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
    - **Гетрогенный 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`
  - Правила вывода для возвращаемого типа
  - Как найти\отладить выводимый тип
  - Вывод типов на runtime: RTTI
- Метапрограммирование
  - Не типовые шаблонные параметры
  - Типовые шаблонные параметры
  - Ключевое слово `typename`
  - Explicit (full) specialization (явная\полная специализация)
  - Partial specialization (частичная специализация)

- Variadic template (вариативные шаблоны)
- Вычисления на этапе компиляции
- Compile-time type manipulation (Преобразование с типами)
  - Primary type categories
  - Composite type categories
  - Type properties
  - Supported operations properties
  - Type relationships
  - Property queries
  - Type transformations
- Curiously recurring template pattern : CRTP
- SFINAE (Subsituation Failure Is Not An Error)
  - Tag dispatch
  - Практический пример основанный на SFINAE
- Special metafunctions
- void\_t
- Detectors
- STL
  - Базовая структура STL
  - Контейнеры
    - **std::allocator**
    - **Последовательные контейнеры**
      - **std::vector**
      - **std::array**
      - **std::forward\_list**
      - **std::list**
      - **std::deque**
    - **Упорядоченные ассоциативные контейнеры**
      - **std::set / std::multiset**
      - **std::map / std::multimap**
    - **Неупорядоченные ассоциативные контейнеры**
      - **\*\*std::unordered\_set \*\***
      - **\*\*std::unordered\_map \*\***
    - Адаптеры
      - **std::stack**
      - **std::queue**
      - **std::priority\_queue**
    - Итераторы
      - **InputIterator**
      - **ForwardIterator**
      - **BidirectionalIterator**
      - **RandomAccessIterator**
      - **OutputIterator**
      - **std::iterator\_trait**
  - Алгоритмы и функциональные объекты
    - Адаптеры итераторов

- `std::reverse_iterator`
- `std::back_insert_iterator`
- `std::move_iterator`
- Функциональные объекты (Function objects)
  - `std::hash`
  - Частичное применение функций
- Алгоритмы
  - Не модифицирующие последовательные алгоритмы
  - Модифицирующие последовательные алгоритмы
  - Разделяющие функции (Partitioning algorithms)
  - Алгоритмы с кучей (Heap algorithms)
  - Сортирующие алгоритмы
  - Бинарные алгоритмы поиска
  - Алгоритмы для множеств
  - min\max алгоритмы
  - Алгоритмы сравнения
  - Алгоритмы перестановок
  - Numeric algorithms
  - Алгоритмы с неинициализированной памятью
- `TODO`

## C++11, C++14, C++17

---

Ниже рассмотрены нововведения 3х стандартов языка C++.

## 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&& fcvi = cvi; // 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 осуществленно 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> cpl(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();
    }
}
```



# Небольшие нововведения

---

## C++11

---

### 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.

### 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;
```

## 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]] };
```

---

## 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(), ::cend(), 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 нижлежащим 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.

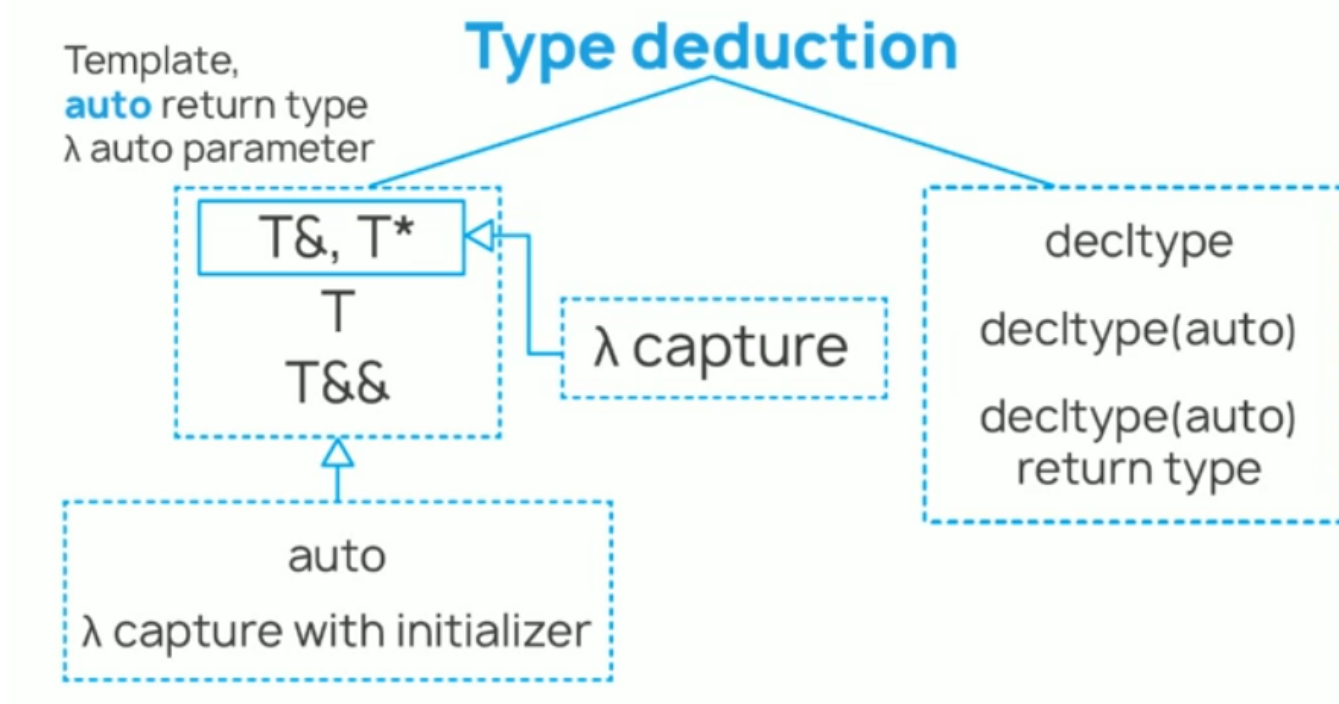
Потом приехали новые конструкции языка.

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

---

Список типов захвата:

```
[=]
[&]
[this]
[*this] // C++17
[identifier]
[&identifier]
[identifier initializer] // C++14
[&identifier initializer] // C++14
```

Захват по копии:

```
const int cx = 42;
auto lambda = [cx] { ... };

// При раскручивании в компиляторе:
class LambdaCompilerRepresentation
{
    // Сохраняется const\volatile:
    const int cx;
public:
    auto operator()() const { ... }
}
```

Влияние mutable спецификатора:

```

int x = 42;
// Compile error:
auto lambda = [x] { x = 0; }; // Нехватает mutable

class LambdaCompilerRepresentation
{
    int x;
public:
    // const модификатор причина проблемы выше
    auto operator()() const { ... }
    // требуется модификатор mutable в lambda
}

```

```

const int x = 42;
// Compile error:
auto lambda = [x] mutable { x = 0; };

class LambdaCompilerRepresentation
{
    // const модификатор причина проблемы выше:
    const int x;
public:
    auto operator()() { x = 0; }
}

```

Захват по ссылке:

```

int x = 42;
auto lambda = [&x] { x = 0; }; // ok

class LambdaCompilerRepresentation
{
    int& x;
public:
    auto operator()() const { x = 0; }
}

```

```

const int x = 42;
auto lambda = [&x] { x = 0; }; // compile error

class LambdaCompilerRepresentation
{
    const int& x;
public:
    auto operator()() const { x = 0; }
}

```

Список захвата с инициализацией:

```
auto p = std::make_unique<SomeClass>();

auto lambda = [p = std::move(p)] { ... }; // ok

class LambdaCompilerRepresentation
{
    //Если захват не по ссылке const\volatile отбросятся
    std::make_unique<SomeClass> p;
public:
    auto operator()() const { ... }
}
```

```
int x = 42;
auto lambda = [&rx = x] { rx = 0; }; // ok

class LambdaCompilerRepresentation
{
    //Если захват по ссылке const\volatile сохраняются
    int& rx;
public:
    auto operator()() const { rx = 0; }
}
```

## Правила вывода для decltype

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

int arr[10];

int v1 = 0.0; //int
const int& v2 = v1; //const int &
int&& v3 = 0; //int&&

decltype(auto) v4 = v1; //int
decltype(auto) v5 = (v1); //int&
decltype(auto) v6 = v2; //const int&

decltype(auto) v7 = foo(); //int
decltype(auto) v8 = bar(); //int &&

decltype(auto) v9 = arr[0]; //int &
```

```
//Если не использовать (auto) - compile errors:
decltype(foo) v10 = foo(); //int ()()
decltype(bar) v11 = bar(); //int && ()()
//Исправляется через decltype(foo()), decltype(bar())
```

## Правила вывода для возвращаемого типа

---

```
// Будет использоваться шаблонный вывод типов:
[capture-list](params) -> T
{
    return ...;
}

//В C++ не обязательно использовать ->
//Тогда будут применены правила вывода auto

// Будет использоваться шаблонный вывод типов:
auto foo() -> T
{
    return ...;
}

//Не обязательно использовать ->, как выше

// Будет использовать decltype вывод типов:
decltype(auto) bar()
{
    return ...;
}
```

Применение механизмов выше, создание обобщенного оператора суммации:

```
template <typename T1, typename T2>
auto operator+(T1&& lhs, T2&& rhs)
{
    return std::forward<T1>(lhs) + std::forward<T2>(rhs);
}
```

```
template <typename Callable, typename ...Args>
auto operator+(Callable&& op, Args&& args) // auto не может вернуть ссылку
{
    return std::forward<Callable>(op)(std::forward<Args>(args)...);
}

template <typename Callable, typename ...Args>
auto&& operator+(Callable&& op, Args&& args) // Могут быть проблемы!
```

```
{
    return std::forward<Callable>(op)(std::forward<Args>(args)...);
}
// Если Callable возвращает просто тип T, тогда вернется ссылка
// на локальный объект, который погибнет сразу же: undefined behavior

template <typename Callable, typename ...Args>
decltype(auto) operator+(Callable&& op, Args&& args) // Perfect returning
{
    return std::forward<Callable>(op)(std::forward<Args>(args)...);
}
```

Но с decltype нужно быть аккуратным:

```
template <typename T>
decltype(auto) lookup(T value)
{
    static const std::vector<SomeClass> values = {...};
    size_t idx = ...; // Найти индекс по value

    auto ret = values[idx];
    return ret; // Возвращаемый тип SomeClass
}

// НО:

template <typename T>
decltype(auto) lookup(T value)
{
    static const std::vector<SomeClass> values = {...};
    size_t idx = ...; // Найти индекс по value

    auto ret = values[idx];
    return (ret); // Возвращаемый тип SomeClass&
}
// Из-за лишних скобок вернётся ссылка на локальный объект
// А это выстрел в ногу
```

## Как найти\отладить выводимый тип

Следующий код выведет ошибку компиляции, из которой можно понять выводимый тип:

```
template <typename T, typename ...Types>
class TypePrinter;

template<typename T>
void foo(const T& t)
{
```

```
TypePrinter<T, decltype(t)> _;  
}  
  
class SomeClass { ... };  
  
SomeClass obj;  
foo(obj);
```

## Вывод типов на runtime: RTTI

---

```
template <typename T>  
void print_type(const T& arg)  
{  
    std::cout << "T = " << typeid(T).name() << "\n";  
    std::cout << "arg = " << typeid(arg).name() << "\n";  
}  
  
SomeClass { ... };  
  
void foo()  
{  
    std::vector<SomeClass> vec { ... };  
    print_type(vec.data());  
}  
  
//Ожидание:  
//T = SomeClass *  
//arg = SomeClass * const&  
  
//Реальность:  
//T = P9SomeClass, demangle - SomeClass*  
//arg = P9SomeClass, demangle - SomeClass*
```

Если необходимо - можно решить задачу через `boost::typeindex`.

---

## Метапрограммирование

---

Вид программирования, связанный с созданием программ, которые порождают другие программы, как результат своей работы.

В C++ реализуется при помощи шаблонов: инстанцируемые функции и классы.

## Не типовые шаблонные параметры

---

Существует 4 вариации:

```

template <size_t> // или <size_t N>
struct int_array { ... };

template <size_t = 42> // или <size_t N = 42>
struct array { ... };

// Начиная с C++11:
template <size_t ...> // или <size_t ...ints>
class sizeT_sequence { ... };

// Начиная с C++17:
template<auto V> // или <decltype(auto) V>
struct B { .... };

```

Параметром могут выступать:

- l-value reference
- `std::nullptr_t`
- integral type (bool, char, signed char, unsigned char, short, ...)
- pointer
- pointer to member
- enumeration

## Типовые шаблонные параметры

---

Три наиболее часто используемых варианта:

```

template <class> // или <typename T>
class FalseVector { ... };

template <class T, class Alloc = std::allocator<T>>
class TrueVector { ... };

// Начиная с C++11:
template <class ...> // или <typename ...Types>
class tuple { .... };

```

Начиная с C++17 доступны три более экзотических варианта, шаблон в шаблоне:

```

template <class K, class T, template <class> class Container>
class MyMap
{
    Container<K> keys;
    Container<T> values;
};

template<class T> class my_array { ... };

```

```
template<class K, class T, template <class> class Container = my_array>
class MyMap { ... };

template <class K, class T, template <class, class> class ...Map>
class MyMap : Map<K, T>... { ... };
```

## Ключевое слово typename

---

Может быть использованно несколькими разными способами:

```
template <typename T>
struct X : B<T> // B<T> is dependent T
{
    //Если не написать typename T::A может интерпретироваться не верно
    typename T::A* pa; // T::A is dependent name from T

    void f(B<T>* pb)
    {
        static int i = B<T>::i; // B<T>::i is dependent variable on T
        pb->j++; // pb->j is dependent variable from T ??? B ???
    }
}
```

## Explicit (full) specialization (явная\полная специализация)

---

Пример для классов:

```
template <class T>
class vector // class template
{
    ...
};

// full specialization for vector<bool>:
template<>
class vector<bool>
{
    ....
};
```

Пример для функций:



```
template <class T>
void print(const T& obj) // function template
{
    std::cout << obj;
};

class SomeClass {...};

// full specialization for print:
template<>
void print<SomeClass>(const SomeClass& obj)
{
    std::cout << obj;
};
```

## Partial specialization (частичная специализация)

---

```
// Шаблонный класс
template <class T, class Deleter>
class unique_ptr
{
public:
    T* operator->() const noexcept;
}

// Частичная специализация для шаблонного класса
// Реализация unique_ptr для массивов
template <class T, class Deleter>
class unique_ptr<T[], Deleter>
{
public:
    T& operator[](size_t idx) noexcept;
    const T& operator[](size_t idx) const noexcept;
}
```

Для функций частичная специализация не доступна.

## Variadic template (вариативные шаблоны)

---

```
template <class T1, class T2, class T3>
bool equalsAnyOf(const T1& t1, const T2& t2, const T3& t3)
{
    return t1 == t2 || t1 == t3;
```

```
}
// 4,5,6 и больше аргументов - стали уже огромными
```

Решение:

```
// C++11:
template <class T1>
bool equalsAnyOf(const T1& t1) noexcept
{
    return false;
}

template <class T1, class T2, class ...TN>
bool equalsAnyOf(const T1& t1, const T2& t2, const TN&... tN) noexcept
{
    // Вызывается первый аргумент и оставшиеся tN
    // каждый раз на 1 меньше, за счёт вышедшего T2
    return t1 == t2 || EqualsAnyOf(t1, tN...); // рекурсия
}

// Вызов:
std::cout << equalsAnyOf(0, 'a', 0.0, 42);
```

Свертка позволяет избежать рекурсии и переполнение стека, в отличие от решения стандарта 11 года.

```
// C++17:

template <class T1, class T2, class ...TN>
bool equalsAnyOf(const T1& t1, const T2& t2, const TN&... tN) noexcept
{
    // Лаконичное решение через свертку функций
    return ((t1 == t2) || ... || (t1 == tN));
}

std::cout << equalsAnyOf(0, 'a', 0.0, 42);
```

Пример использования в std::vector:

```
template <class T, class Alloc = std::allocator<T>>
class vector
{
public:
    template <class ...Args>
    T& emplace_back(Args&&... args)
    {
        T* ptr = ...; // указатель на новый объект
```

```

        new (ptr) T { std::forward<Args>(args)...};
    // new (ptr) T { std::forward<Arg1>(arg1), std::forward<Arg2>(arg2), ...};
    return *ptr;
}
};

```

## Вычисления на этапе компиляции

---

```

template <size_t N>
struct Facrotial;

template <>
struct Facrotial<0>
{
    // enum использовались в старых компиляторах
    // так как должны вычисляться на этапе компиляции
    // До C++11
    enum { value = 1 };
};

template <>
struct Facrotial<1>
{
    enum { value = 1 };
};

template <>
struct Facrotial<2>
{
    enum { value = 2 };
};

```

Более разумное решение, без определения каждой частичной спецификации:

```

// Если не реализовать <0> и <1>
// Тогда рекурсия будет бесконечной

template <size_t N>
struct Facrotial
{
    // До C++11
    enum { value = N * Factorial<N - 1>::value };
};

template <>
struct Facrotial<0>
{

```

```

    enum { value = 1 };
};

template <>
struct Facrotial<1>
{
    enum { value = 1 };
};

const auto fac5 = Factorial<5>::value;

```

Однако в данном случае присутствует рекурсия, но её глубина может достигать 1024 вызовов.

Начиная с C++11:

```

template <size_t N>
struct Facrotial
{
    static constexpr size_t value = N * Facrotial<N - 1>::value;
};

template <>
struct Facrotial<0>
{
    static constexpr size_t value = 1;
};

template <>
struct Facrotial<1>
{
    static constexpr size_t value = 1;
};

const auto fac5 = Factorial<5>::value;

```

Такие структуры называются метафункции. Стандартное название для переменной метафункций ::value - негласное соглашение программистов.

Вариант с использованием функций constexpr:

```

constexpr size_t Factorial(size_t n) noexcept
{
    return n > 1 ? n * Facrotial(n - 1) : 1; // Начиная с C++11
}

// Не рекурсивный вариант:
constexpr size_t Facrotial(size_t n) noexcept
{
    size_t acc = 1;

```

```
    for (size_t i = 2; i <= n; ++i)
        acc *= i;

    return acc; // Начиная с C++14
}
```

Возведение некоторого числа, в степень:

```
template<size_t Exp>
struct pow1
{
    double operator()(double base) const noexcept
    {
        return base * pow1<Exp - 1>{}(base);
    }
}

template <>
struct pow1<0>
{
    double operator()(double base) const noexcept
    {
        return 1.0;
    }
}
```

Компилятор зная степень, подготовит вычисления на этапе компиляции.

Возможные оптимизации:

```
template<size_t Exp>
struct pow2
{
    double operator()(double base) const noexcept
    {
        return (Exp % 2 != 0)
            ? base * pow2<(Exp - 1) / 2>{}(base) * pow2<(Exp - 1) / 2>{}(base)
            : pow2<Exp / 2>{}(base) * pow2<Exp / 2>{}(base);
    }
}

template <>
struct pow2<0>
{
    double operator()(double base) const noexcept
    {
        return 1.0;
    }
}
```

Так же на C++17 можно записать прошлый вариант короче:

```
template <size_t Exp>
struct pow1
{
    double operator()(double base) const noexcept
    {
        if constexpr(Exp == 0)
        {
            return 1.0;
        }
        else
        {
            return base * pow1<Exp - 1>{}(base);
        }
    }
};
```

И для оптимизированной версии:

```
template <size_t Exp>
struct pow2
{
    double operator()(double base) const noexcept
    {
        if constexpr(Exp == 0)
        {
            return 1.0;
        }
        else if constexpr (Exp & 1 != 0)
        {
            return base * pow2<(Exp - 1) / 2>{}(base)
                * pow2<(Exp - 1) / 2>{}(base);
        }
        else
        {
            return pow2<Exp / 2>{}(base)
                * pow2<Exp / 2>{}(base);
        }
    }
};
```

В итоге pow2 почти всегда лучше встроенной функции в компилятор.

Подсчёт бит в числе:

```
constexpr uint8_t popcount(uint64_t value) noexcept
{
    uint8_t res = 0;
    while (value != 0)
    {
        res += value & 1;
        value >>= 1;
    }

    return res;
}

constexpr uint8_t pop_count = popcount(0b010010010); // == 3
```

Наибольший общий делитель:

```
constexpr uint64_t gcd(uint64_t a, uint64_t b) noexcept
{
    while (a != b)
    {
        if (a > b)
        {
            a -= b;
        }
        else
        {
            b -= a;
        }
    }
    return a;
}

constexpr auto gcd_a_b = gcd(15, 125); // == 5
```

Решето Эратосфена:

```
template <uint64_t N, size_t ...Idx>
constexpr std::array<bool, N + 1> sieve_impl(std::index_sequence<Idx...>) noexcept
{
    //Нужно просто чтобы заполнить массив true:
    std::array<bool, N + 1> primes { (Idx, true)... };
    //В C++20 можно использовать вектора, и это не понадобится.

    primes[0] = primes[1] = false;

    for (size_t i = 2; i * i <= N; ++i)
    {
        if (primes[i])
        {

```

```

        for (size_t j = i * i; j <= N; j += i)
            primes[j] = false;
    }
}
return primes;
}

template <uint64_t N, typename Idx = std::make_index_sequence<N + 1>>
constexpr std::array<bool, N + 1> sieve() noexcept
{
    return sieve_impl<N>(Idx {});
}

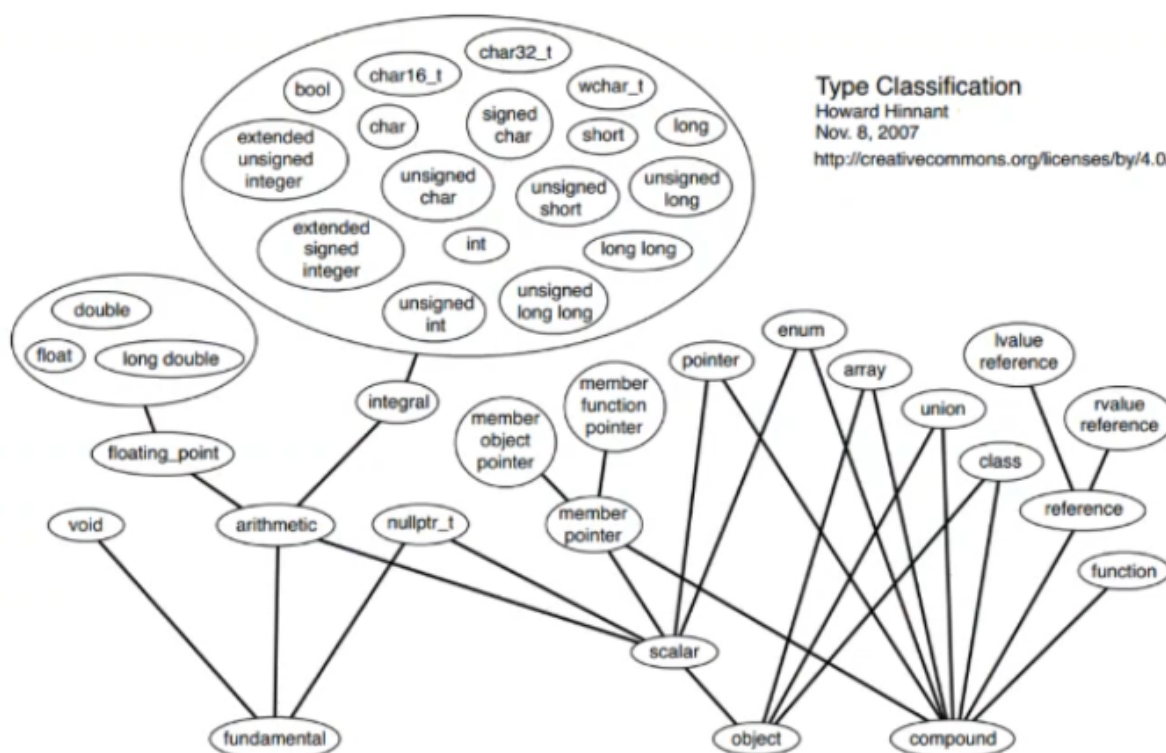
constexpr auto primes = seive<5>();
// 0, 1, 4 == false
// 2, 3, 5 == true

//Вывод в поток:
std::copy(primes.begin(), primes.end(),
          std::ostream_iterator<bool> {std::cout, " " });
//0 0 1 1 0 1

```

## Compile-time type manipulation (Преобразование с типами)

C++ типы:



Чтобы проверить является ли тип ссылкой - определим метафункцию:



```

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;
};

```

Может потребоваться вытянуть это значение иначе. Например через вызов функции и cast is\_reference в bool.

Чтобы не писать вручную такой код - применяют шаблонную магию integral\_constant:

```

template<class T, T v>
struct integral_constant
{
    using value_type = T;
    using type = std::integral_constant<T, v>;

    static constexpr value_type value = v;

    constexpr operator value_type() { return v; } const noexcept
    constexpr value_type operator()() { return v; } const noexcept
};

using true_type = integral_constant<bool, true>;
using false_type = integral_constant<bool, false>;

//Теперь перепишем прошлый код:
template <class T>
struct is_reference : false_type {};

template <class T>
struct is_reference<T&> : true_type {};

template <class T>
struct is_reference<T&&> : true_type {};
//Теперь у нас будет не только ::value
//Но и все необходимые перегрузки в каждом варианте

```

В C++17 ввели шаблонные переменные, в данной ситуации они могут помочь, чтобы каждый раз не писать `is_reference::value`.

```
template<class T>
inline constexpr bool is_reference_v = is_reference<T>::value;
```

Рассмотрим функции, которые есть в стандартной библиотеке:

## Primary type categories

---

- `is_void`
- `is_null_pointer`
- `is_integral`
- `is_floating_point`
- `is_array`
- `is_enum`
- `is_union`
- `is_class`
- `is_function`
- `is_pointer`
- `is_lvalue_reference`
- `is_rvalue_reference`
- `is_member_object_pointer`
- `is_member_function_pointer`

И так же шаблонные переменные с постфиксом `_v`.

## Composite type categories

---

- `is_fundamental`
- `is_arithmetic`
- `is_scalar`
- `is_object`
- `is_compound`
- `is_reference`
- `is_member_pointer`

И так же шаблонные переменные с постфиксом `_v`.

## Type properties

---

- `is_const`
- `is_volatile`
- `is_trivial`
- `is_trivial_copyable`

- `is_standard_layout`
- `is_pod`
- `is_literal_type`
- `has_unique_object_representations`
- `is_empty`
- `is_polymorphic`
- `is_abstract`
- `is_final`
- `is_aggregate`
- `is_signed`
- `is_unsigned`
- `is_bounded_array`
- `is_unbounded_array`

И так же шаблонные переменные с постфиксом `_v`.

## Supported operations properties

---

- `is_constructible`
- `is_trivially_constructible`
- `is_nothrow_constructible`
- `is_default_constructible`
- `is_trivially_default_constructible`
- `is_nothrow_default_constructible`
- `is_copy_constructible`
- `is_trivially_copy_constructible`
- `is_nothrow_copy_constructible`
- `is_move_constructible`
- `is_trivially_move_constructible`
- `is_nothrow_move_constructible`
- `is_assignable`
- `is_trivially_assignable`
- `is_nothrow_assignable`
- `is_copy_assignable`
- `is_trivially_copy_assignable`
- `is_nothrow_copy_assignable`
- `is_move_assignable`
- `is_trivially_move_assignable`
- `is_nothrow_move_assignable`
- `is_destructible`
- `is_trivially_destructible`
- `is_nothrow_destructible`
- `is_default_destructible`
- `has_virtual_destructor`
- `is_swappable_with`
- `is_swappable`

- `is_nothrow_swappable_with`
- `is_nothrow_swappable`

И так же шаблонные переменные с постфиксом `_v`.

## Type relationships

---

- `is_same`
- `is_base_of`
- `is_convertible`
- `is_nothrow_convertible`
- `is_invocable`
- `is_invocable_r`
- `is_nothrow_invocable`

И так же шаблонные переменные с постфиксом `_v`.

## Property queries

---

- `alignment_of` - работает как `alignof`
- `rank` - число элементов массива
- `extent` - сколько размерностей внутри массива

И так же шаблонные переменные с постфиксом `_v`.

## Type transformations

---

Другой вариант использования - это преобразование типа:

```
template <class T>
struct remove_reference
{
    using type = T;
};

template <class T>
struct remove_reference<T&>
{
    using type = T;
};

template <class T>
struct remove_reference<T&&>
{
    using type = T;
};

//Начиная с C++14
```

```
template <class T>
using remove_reference_t = typename remove_reference<T>::type;
```

Наподобии с неявным правилом constexpr value, здесь так же есть неявное правило using type.

Список метафункций из стандартной библиотеки:

- remove\_const
- remove\_volatile
- remove\_cv
- add\_const
- add\_volatile
- add\_cv
- remove\_reference
- add\_lvalue\_reference
- add\_rvalue\_reference
- remove\_cvref
- remove\_pointer
- add\_pointer
- make\_signed
- make\_unsigned
- remove\_extent
- remove\_all\_extents
- decay
- conditional
- underlying\_type
- common\_type
- result\_of
- invoke\_result

И так же есть специальные перегрузки с постфиксом \_t.

Для чего это может быть нужно:

```
template <class T, class Alloc = std::allocator<T>>
class vector
{
public:
    template <class ...Args>
    T& emplace_back(Args ...args);
};

template <class ...Args>
T& vector::emplace_back(Args ...args)
{
    if (size() == capacity())
    {
        const auto oldCap = capacity();
        const auto newCap = computeGrowth(oldCap + 1);
```

```

    auto *newVec = allocator_traits<Alloc>::allocate(al, newCap);
    allocator_traits<Alloc>::construct(al, newVec, oldCap + 1,
                                       forward<Args>(args)...);

    if constexpr (is_nothrow_move_constructible_v<T>
                  || !is_copy_constructible_v<T>)
    {
        uninitialized_move(begin(), end(), ptr);
    }
    else
    {
        uninitialized_copy(begin(), end(), ptr);
    }

    //Освобождение старых объектов и региона памяти
    change_array(newVec, size() + 1, newCap);
}
}

```

## Curiously recurring template pattern : CRTP

---

```

template <class T>
struct Base
{
};

struct Derived : Base<Derived>
{
};

```

Проблемная ситуация, для которой нужна такая странная композиция:

```

template <class T = intmax_t>
class Rational
{
    T m_num = T(0), m_denom = T(1);

public:
    Rational() = default;

    explicit Rational(T num, T denom = T(1))
        : m_num { num }
        , m_denom { denom }
    { ... }

    friend bool operator<(const Rational<T>& l, const Rational<T>& r) noexcept
    {
        const auto lcm = std::lcm(l.m_denom, r.m_denom);
    }
}

```

```

        return l.m_num * (lcm / l.m_denom)
            < r.m_num * (lcm / r.m_denom);
    }
}
//Однако возникает сложность, если требуется оператор >

```

Пример из boost:

```

template <class T>
struct less_than_comparable
{
    friend bool operator>(const T& l, const T& r) noexcept
    {
        return r < l;
    }
    friend bool operator<=(const T& l, const T& r) noexcept
    {
        return !(r < l);
    }
    friend bool operator>=(const T& l, const T& r) noexcept
    {
        return !(l < r);
    }

    template <class T = uint64_t>
    class Rational : less_than_comparable<Rational<T>>
    { //Класс объявлен выше с операцией <
    };

    //Обобщенная задача решена!
}

```

Так же эта идеома применима при статическом полиморфизме, но это не даёт большого прироста производительности.

## SFINAE (Subsituation Failure Is Not An Error)

---

```

template <class Container>
void printContainer(const Container& cont)
{
    if (!cont.empty())
    {
        std::cout << "(";

        for (const auto& value: cont)
        {
            std::cout << " " << value;
        }
    }
}

```

```

        std::cout << ")";
    }
}

```

Примитивные типы элементов из листа\вектора\дека - работают из коробки.

Для своего класса потребуется только определить оператор <<.

Для set\unordered всё работает из коробки.

Для map\unordered потребуется определить оператор << для пары.

Однако если попробовать использовать std::stack в качестве контейнера - возникнет ошибка компиляции. Это произойдёт потому что у него нет функций begin\end.

Нужно сделать защиту, чтобы нельзя был использовать не итерируемый тип Container:

```

template <class Container,
          class It = typename Container::const_iterator>
void printContainer(const Container& cont)
{
    if (!cont.empty())
    {
        std::cout << "(";

        for (const auto& value: cont)
        {
            std::cout << " " << value;
        }

        std::cout << ")";
    }
}

//Использование It не даст собраться нежелательным вызовам:
void printContainer(1);
void printContainer(std::stack<int>{1});
//И будут выведены понятные сообщения об ошибке

```

Альтернативный метод реализации SFINAE через std::enable\_if:

```

template <bool Cond, class T = void>
struct enable_if {};

template <class T>
struct enable_if<true, T>
{
    using type = T;
};

```



```
template <bool Cond, class T = void>
using enable_if_t = typename enable_if<Cond, T>::type;
```

Где это необходимо:

```
template <typename It>
using ItTag = typename std::iterator_traits<It>::iterator_category;

// Не будет работать с итераторами не random access
template <class It, class Diff,
         std::enable_if_t<std::is_base_of_v<
             std::random_access_iterator_tag, ItTag<It>>,
             int> = 0>

It next(It it, Diff n) noexcept
{
    return it + n;
}

// Данный экземпляр будет вызван например для list
template <class It, class Diff,
         std::enable_if_t<std::is_same_v<
             std::bidirectional_iterator_tag, ItTag<It>>,
             int> = 0>

It next(It it, Diff n) noexcept
{
    for (; n > Diff(0); --n)
        ++it;

    for (; n < Diff(0); ++n)
        --it;

    return it;
}

// input или forward итераторы
template <class It, class Diff,
         std::enable_if_t<std::is_same_v<std::input_iterator_tag, ItTag<It>>
             || std::is_same_v<std::forward_iterator_tag,
             ItTag<It>>>,
         int> = 0>

It next(It it, Diff n) noexcept
{
    assert(n >= 0);
    for (; n != Diff(0); --n)
        ++it;
```

```
    return it;
}
```

## Tag dispatch

---

Был придуман, чтобы выглядеть удобнее.

Подготовка функций:

```
template <typename It, class Diff>
It next_impl(It it, Diff n, std::input_iterator_tag)
{
}

template <typename It, class Diff>
It next_impl(It it, Diff n, std::bidirectional_iterator_tag)
{
}

template <typename It, class Diff>
It next_impl(It it, Diff n, std::random_access_iterator_tag)
{
}
```

Основная функция:

```
template<typename It>
using ItTag = typename std::iterator_traits<It>::iterator_category;

template <typename It, class Diff>
It next(It it, Diff n)
{
    return (it, n, ItTag<It>{});
}
```

Вариант реализации, с использованием if constexpr:

```
template<typename It>
using ItTag = typename std::iterator_traits<It>::iterator_category;

template <typename It, class Diff>
It next(It it, Diff n)
{
    if constexpr(std::is_base_of_v<std::random_access_iterator_tag, ItTag<It>>)
    {} //Для random access итераторов
}
```

```

else if constexpr(std::is_same_v<std::bidirectional_iterator_tag, ItTag<It>>)
{} //Для bidirectional итераторов
else
{} //Для input and forward итераторов
}

```

## Практический пример основанный на SFINAE

Ленивые вычисления. У нас функтор инициализатор, и есть optional - в котором значение либо уже проинициализированно, либо ещё нет. Есть потребность проверить, является возвращаемый результат void или ссылкой, т.е. optional не умеет работать со ссылками, для этого используем reference\_wrapper. Всё это реализуется кодом ниже:

```

#include <optional>
#include <functional>

template <class T>
using OptRefType =
std::optional<std::reference_wrapper<std::remove_reference_t<T>>>>;

template <class Func>
class Lazy
{
    using ResultType = std::invoke_result_t<Func>;
    using OptionalType = std::conditional_t<std::is_void_v<ResultType>, bool,
std::conditional_t<std::is_reference_v<ResultType>,
OptRefType<ResultType>, std::optional<ResultType>>>;

    Func m_initializer;
    OptionalType m_value;

public:

    Lazy(Func&& init) : m_initializer { std::forward<Func>(init) }
    {}

};

```

Теперь, как возвращать значение: оно может быть результирующим типом или void. Для этого используем SFINAE

```

template <class Func>
class Lazy
{
public:

    template <class T = ResultType, std::enable_if_t<!std::is_void_v<T>, int> = 0>

```

```

    ResultType operator()()
    {
        if (m_value)
            return *m_value;

        return m_value.emplace(m_initializer());
    }

    template <class T = ResultType, std::enable_if_t<std::is_void_v<T>, int> = 0>
    void operator()()
    {
        if (m_value)
            return;

        m_value = true;
        m_initializer();
    }
};

```

Пример использования:

```

// C++ 14: Не удастся автоматически deduce типа Func
// Потому необходим CreateLazy + auto idx = CreateLazy([]{});
template <class Func>
Lazy<Func> CreateLaxy(Func&& f)
{
    return { std::forward<Func>(f); };
}

auto idx = CreateLazy([] {auto i = 0; return i;});

// C++ 17: может вывести всё без template CreateLazy
Lazy idx { []{...} };

// Использование: (оба стандарта языка)
if (.... & idx())
{
    ....
}

```

## Special metafunctions

В примерах выше была реализованна функция equalsAny, которая сравнивала произвольное количество, произвольных типов.

Теперь реализуем её вариант, когда необходимо чтобы все типы были идентичными:

```
template <class T1, class T2, class ...TN,
         class = std::enable_if_t< (std::is_same_v<T1, T2>
                                     && ... && std::is_same<T1, TN>)>

bool equalsAny(const T1& t1, const T2& t2, const TN&... tN) noexcept
{
}
```

Проблема такого метода в раскрутке свертки, она может быть дорого стоит компилятору.

Решаются подобные задачи с помощью функций из C++17:

- `std::conjunction_v` - конъюнкция (соединение через логическое И)
- `std::disjunction` - дизъюнкция (соединение через логическое ИЛИ)
- `std::negation` - отрицание

Метод исправить этот недостаток:

```
template <class T1, class T2, class ...TN,
         class = std::enable_if_t< std::conjunction_v<std::is_same<T1,T2>,
                                                     std::is_same<T1, TN>...>>

bool equalsAny(const T1& t1, const T2& t2, const TN&... tN) noexcept
{
}
```

## void\_t

---

```
//Не именованный шаблон pack
template <class ...>
using void_t = void;

//Если в него протолкнуть что угодно, он вернёт void:
void_t<int, unsigned long long, float,
      double, SomeClass, std::vector<int>,
      std::stack<bool>>; //void
```

Проблема:

```
int* pi = nullptr;
float* pf = nullptr;

std::vector<int*> vi;
std::vector<float*> vf;
```

```
// Допустим нам нужна перегрузка, чтобы вектор мог сравниваться
// с нижележащим типом: Некоторая реализация std::all_of
bool result = equalsAny(nullptr, pi, pf, vi, vf);
//Из коробки, это не будет работать
```

Чтобы защититься от проблем, нужно применить SFINAE, который позволит получать лаконичные сообщения об ошибке:

```
template <class T1, class T2, class... TN,
          class = void_t<decltype(std::declval<T1&>() == std::declval<T2&>()),
                        decltype(std::declval<T1&>() == std::declval<T2&>())...>

bool equalsAny(const T1& t1, const T2& t2, const TN&... tN) noexcept
{
    //Начиная с C++17 можно обрмить выражение в скобки ( )
    return ( (t1 == t2) || ... || (t1 == tN) );
}
```

declval это шаблонная функция без тела. Рассмотрим применение declval. Допустим у нас есть две структуры Default и NonDefault:

```
template <class T>
std::add_rvalue_reference<T>_t declval();

struct Default { int foo() const { return 1; } };

struct NonDefault {
    NonDefault(const NonDefault&) {}
    int foo() const { return 1; }
}

decltype(Default.foo()) i1 = 1; // type = int
decltype(NonDefault().foo()) i2 = i1; // Compile error

decltype(std::declval<NonDefault>().foo()) i2 = i1; // type = int
```

Важно! declval работает только внутри:

- decltype
- sizeof
- typeid
- noexcept

Т.е. в тех операторах, где выражение не вычисляется.

## Detectors

Проверка на то, что контейнер может итерироваться:

```
template <class T, typename = void>
struct is_iterable: std::false_value
{
};

template <class T>
struct is_iterable<T, std::void_t<
    decltype(std::begin(std::declval<T>())),
    decltype(std::end(std::declval<T>()))>>
    : std::true_type
{
};

template <class T>
inline constexpr auto is_iterable = is_iterable<T>::value;

static_assert(is_iterable<std::vector<int>>); // ok
static_assert(is_iterable<std::list<int>>); // ok
static_assert(is_iterable<std::map<int, int>>); // ok
static_assert(is_iterable<std::stack<int>>); // compile time error
```

Если требуется так же проверять есть ли возможность rbegin\rend:

```
template <class T, typename = void>
struct is_reverse_iterable: std::false_value
{
};

template <class T>
struct is_reverse_iterable<T, std::void_t<
    decltype(std::begin(std::declval<T>())),
    decltype(std::end(std::declval<T>())),
    decltype(std::rbegin(std::declval<T>())),
    decltype(std::rend(std::declval<T>()))>>
    : std::true_type
{
};

template <class T>
inline constexpr auto is_reverse_iterable_v = is_reverse_iterable<T>::value;

static_assert(is_iterable<std::vector<int>>); // ok
static_assert(is_iterable<std::list<int>>); // ok
static_assert(is_iterable<std::map<int, int>>); // ok
static_assert(is_iterable<std::forward_list<int>>); // compile time error
```

Для того чтобы сделать обобщенный вариант, когда не нужно будет определять 2 структуры каждый раз, придумали детекторы:

```
template <class Default, class AlwaysVoid, template <class...> class Op, class
...Args>
struct detector
{
    using value_t = std::false_type;
    using type = Default;
}

template <class Default, template<class ...>, class Op, class... Args>
struct detector<Default, std::void_t<Op<Args...>>, Op, Args>
{
    using value_t = std::true_type;
    using type = Op<Args...>;
}

struct nonesuch
{
    nonesuch() = delete;
    nonsuch(const nonesuch&) = delete;
    nonsuch(nonesuch&&) = delete;
    nonesuch& operator=(const nonesuch&) = delete;
    nonesuch& operator=(nonesuch&&) = delete;
};
```

Где Op - это метафункция, например std::is\_same, а Args - это наши типы, например T1\T2.

Далее:

```
template <template <class ...> class Op, class... Args>
using is_detected = typename detector<nonesuch, void, Op, Args...>::value_t;

template <template<class...> class Op, class... Args>
inline constexpr bool is_detected_v = is_detected<Op, Args...>::value;

template<template<class...> class Op, class.. Args>
using detected_t = typename detector<nonesuch, void, Op, Args...>::type;

template <class Default, template<class...> class Op, class... Args>
using detected_or = detector<Default, void, Op, Args...>;

template <class Default, template<class ...> class Op, class... Args>
using detected_or_t = typename detected_or<Default, Op, Args...>::type;
```

Как это использовать:



```

template <class T>
using free_begin = decltype(std::begin(std::declval<T>()));

template <class T>
using free_end = decltype(std::end(std::declval<T>()));

template <class T>
using free_rbegin = decltype(std::rbegin(std::declval<T>()));

template <class T>
using free_rend = decltype(std::rend(std::declval<T>()));

template <class T>
inline constexpr auto is_iterable_v = is_detected_v<free_begin, T>
&& is_detected_v<free_end, T>;

template <class T>
inline constexpr auto is_reverse_iterable_v = is_detected_v<free_rbegin, T>
&& is_detected_v<free_rend, T>;

```

Для того чтобы проверить, что в нашем классе есть функция foo:

```

template <class T>
using v_foo_v = decltype(std::declval<T&>().foo());

template <class T>
using v_foo_i = decltype(std::declval<T&>().foo(std::declval<int>()));

template <class T>
inline constexpr bool has_foo = is_detected_v<v_foo_v, T>
&& is_detected_v<v_foo_i, T>;

```

Детекторы позволяют делать проверку, что наши объекты соответствуют нашим ожиданиям. В C++20 появятся концепты, которые так же позволяют решать данные задачи.

---

## STL

---

### Базовая структура STL

---

- Контейнеры
  - Итераторы
  - Алгоритмы
  - Адаптеры
  - Функциональные объекты
-

## Контейнеры

- Последовательные
- Упорядоченные ассоциативные
- Неупорядоченные ассоциативные
- Адаптеры

### std::allocator

Изначально алокатеры появились для того чтобы проще было работать со старой моделью памяти, в которой существовали ближние и дальние указатели. Но на текущий момент у них другие задачи. Рассмотрим пример реализации:

```
template <class T>
struct allocator { ... };

template <class Alloc>
struct allocator_traits
{
    using allocator_type = Alloc;
    using value_type = Alloc::value_type;
    using pointer = Alloc::pointer;
    using const_pointer = Alloc::const_pointer;
    using difference_type = Alloc::difference_type;
    using size_type = Alloc::size_type;

    [[nodiscard]] static pointer allocate(Alloc& a, size_type n);

    static void deallocate(Alloc& a, pointer p, size_type n);

    template <class T, class... Args>
    static void construct(Alloc& a, T* p, Args&&... args);

    template<class T>
    static void destroy(Alloc& a, T* p);
}
```

Allocator умеет:

- Аллоцировать кусок памяти через функцию allocate
- Очищать аллоцированную память через deallocate
- Через функцию construct формируется объект (placement new)
- Разрушать объект, через функцию destroy

Начиная с C++11 эти функции были перенесены в allocator\_traits, а так же были добавлены using'и для шаблонной магии.

Пример использования:

```
#include <memory>

std::allocator<int> a1; //Стандартный алокатор для int
int* p = a1.allocate(1); //алокация памяти для 1 элемента
a1.construct(p, 7); //Конструирование и инициализация

std::cout << *p; // 7

using prev_alloc = decltype(a1);
std::allocator<std::allocator_traits<prev_alloc>::value_type> a2;

a2.deallocate(p, 1); //Этот код корректен, алокаторы не хранят состояния
```

Контейнеры используют алокаторы как второй шаблонный параметер.

## Последовательные контейнеры

### std::vector

Вектор хранит объекты типа T в динамически выделенной памяти.

```
template <class T, class Alloc = std::allocator<T>>
class vector
{
    using value_type = T;
    using allocator_type = Alloc;
    using size_type = std::size_t;
    using difference_type = std::ptrdiff_t;
    using reference = value_type&;
    using const_reference = const value_type&;
    using pointer = std::allocator_traits<Alloc>::pointer;
    using const_pointer = std::allocator_traits<Alloc>::const_pointer;

    //+ итераторы (о них позже)
};
```

Функции класса вектор можно разбить на 4 типа.

Доступ к элементам:

- at, operator[] - асимптотическая сложность O(1)
- data - асимптотическая сложность O(1)
- front, back - асимптотическая сложность O(1)

Функция at может бросить исключение, если вышли за границы, operator[] - нет.

Размеры (capacity):

- empty - асимптотическая сложность O(1)

- `size`, `max_size`, `capacity` - асимптотическая сложность  $O(1)$
- `resize`, `reserve`, `shrink_to_fit` - асимптотическая сложность  $O(n)$

При инициализации вектора при помощи `std::initializer_list` или `{}` - `capacity` равно его размеру.

При `resize` происходит изменение размера вектора, новые элементы (выше прошлого `size()`) заполняются дефолтным значением.

При `reserve` происходит увеличение `capacity`, но не `size`.

Функция `shrink_to_fit` делает `capacity = size`, т.е. обрезает лишнюю память.

Модификаторы (modifiers):

- `clear`, `erase` - асимптотическая сложность  $O(n)$
- `insert`, `emplace`, `push_back` - асимптотическая сложность  $O(1)$  или  $O(n)$
- `emplace_back`, `pop_back` - асимптотическая сложность  $O(1)$
- `swap` - асимптотическая сложность  $O(1)$

Разница в асимптотической сложности `push_back` итд связана с тем нужна ли реаллокация, или нет. Если требуется сделать вставку не в конец, `insert`/`emplace` тоже работают за  $O(n)$ , иначе  $O(1)$ .

`Emplace` и `emplace_back` формируют объект при помощи perfect forwarding.

Аллокатор:

- `get_allocator` - асимптотическая сложность  $O(1)$

## **`std::array`**

Отличается от вектора тем, что хранит элементы на стеке, а не в динамически выделенной памяти.

```
template <class T, size_t N>
struct array
{
    using value_type = T;
    using size_type = std::size_t;
    using difference_type = std::ptrdiff_t;
    using reference = value_type&
    using const_reference = const value_type&
    using pointer = T*;
    using const_pointer = const T*;

    //+ итераторы (о них позже)
};
```

В отличие от сырых массивов его можно передавать как по значению, так и по ссылке, и в первом случае он будет копироваться.

`std::array` содержит 3 группы функций.

Доступ к элементам:

- `at`, `operator[]` - асимптотическая сложность  $O(1)$
- `data` - асимптотическая сложность  $O(1)$
- `front`, `back` - асимптотическая сложность  $O(1)$

Размер (capacity):

- `empty` - асимптотическая сложность  $O(1)$
- `size`, `max_size` - асимптотическая сложность  $O(1)$

Модификаторы:

- `swap` - асимптотическая сложность  $O(n)$

### **`std::forward_list`**

Представляет однонаправленный список. Так же использует аллокатор как вектор и содержит все те же `using`'и, для метапрограммирования.

У него есть 5 групп функций.

Доступ к элементам:

- `front` - асимптотическая сложность  $O(1)$

Размер:

- `empty` - асимптотическая сложность  $O(1)$
- `max_size` - асимптотическая сложность  $O(1)$
- `resize` - асимптотическая сложность  $O(n)$

Модификаторы:

- `clear` - асимптотическая сложность  $O(n)$
- `erase_after` - асимптотическая сложность  $O(1)-O(n)$
- `insert_after` - асимптотическая сложность  $O(1)-O(n)$
- `push_front`, `emplace_after`, `emplace_front` - асимптотическая сложность  $O(1)$
- `pop_front` - асимптотическая сложность  $O(1)$
- `swap` - асимптотическая сложность  $O(1)$
- `merge` - асимптотическая сложность  $O(n)$

`Merge` переносит все элементы из листа переданного в качестве аргумента функции, второй лист становится пустым.

Встроенные алгоритмы:

- `splice_after` - асимптотическая сложность  $O(1)-O(n)$
- `remove`, `remove_if` - асимптотическая сложность  $O(n)$
- `reverse` - асимптотическая сложность  $O(n)$
- `sort` - асимптотическая сложность  $O(n \log n)$
- `unique` - асимптотическая сложность  $O(n)$

Функция `splice_after` работает схоже с `merge`, но можно задать позицию для вставки.

Функции `remove`\`remove_if` - удаление диапазона.

Функция `unique` - оставляет только уникальные значения, однако перед тем как её вызвать нужно обязательно отсортировать элементы, функцией `sort`.

Аллокатор:

- `get_allocator` - асимптотическая сложность  $O(1)$

### **`std::list`**

Похож на `std::forward_list` - но умеет двигаться в обоих направлениях. Это приводит к тому что он занимает больше места в памяти, но у него появляются новые функции.

У него тоже есть 5 групп функций.

Доступ к элементам:

- `front`, `back` - асимптотическая сложность  $O(1)$

Размер:

- `empty` - асимптотическая сложность  $O(1)$
- `max_size` - асимптотическая сложность  $O(1)$
- `resize` - асимптотическая сложность  $O(n)$

Модификаторы:

- `clear` - асимптотическая сложность  $O(n)$
- `erase` - асимптотическая сложность  $O(n)$
- `insert` - асимптотическая сложность  $O(1)$ - $O(n)$
- `push_front`, `push_back`, `emplace`, `emplace_front`, `emplace_back` - асимптотическая сложность  $O(1)$
- `pop_front`, `pop_back` - асимптотическая сложность  $O(1)$
- `swap` - асимптотическая сложность  $O(1)$
- `merge` - асимптотическая сложность  $O(n)$

Встроенные алгоритмы:

- `splice_after` - асимптотическая сложность  $O(1)$ - $O(n)$
- `remove`, `remove_if` - асимптотическая сложность  $O(n)$
- `reverse` - асимптотическая сложность  $O(n)$
- `sort` - асимптотическая сложность  $O(n \log n)$
- `unique` - асимптотическая сложность  $O(n)$

Аллокатор:

- `get_allocator` - асимптотическая сложность  $O(1)$

### **`std::deque`**

Дек - двусторонняя очередь (стек + очередь). Саму структуру возможно реализовать через `std::list`, однако последний не дружелюбен к кэшированию, т.к. элементы не хранятся блоком памяти.

Другой возможный способ - это реализация через массив указателей на chunk'и.

Содержит 4 группы функций:

Доступ:

- `at`, `operator[]` - асимптотическая сложность  $O(1)$
- `front`, `back` - асимптотическая сложность  $O(1)$

Размер:

- `empty` - асимптотическая сложность  $O(1)$
- `size`, `max_size` - асимптотическая сложность  $O(1)$
- `resize`, `shrink_to_fit` - асимптотическая сложность  $O(n)$

Модификаторы:

- `clear`, `erase` - асимптотическая сложность  $O(n)$
- `insert`, `emplace`, `push_front`, `push_back`,  $O(1)$ ,  $O(n)$
- `emplace_front`, `emplace_back`, `pop_front`, `pop_back` - асимптотическая сложность  $O(n)$
- `swap` - асимптотическая сложность  $O(1)$

Аллокатор:

- `get_allocator`

## Упорядоченные ассоциативные контейнеры

Поиск в последовательных контейнерах достаточно тяжелая операция, более эффективная структура для поиска элементов - дерево.

### `std::set` / `std::multiset`

```
template <class Key, class Compare = std::less<Key>,  
         class Alloc = std::allocator<T>>  
  
class set  
{  
    using key_type = Key;  
    using value_type = Key;  
    using key_compare = Compare;  
    using value_compare = Compare;  
    //И все остальные using'и, которые были в std::vector  
}
```

Реализация сета это красно-черное дерево даёт сбалансированное бинарное дерево поиска, благодаря этом средняя скорость поиска имеет логарифмическую сложность.

Содержит 5 групп функций:

Размер:

- `empty` - асимптотическая сложность  $O(1)$
- `size`, `max_size` - асимптотическая сложность  $O(1)$

Модификаторы:

- `clear` - асимптотическая сложность  $O(n)$
- `erase` - асимптотическая сложность  $O(1)$ ,  $O(\log n)$ ,  $O(n)$
- `insert`, `emplace` - асимптотическая сложность  $O(\log n)$
- `emplace_hint` - асимптотическая сложность  $O(1)$ ,  $O(\log n)$
- `swap` - асимптотическая сложность  $O(1)$
- `merge` - асимптотическая сложность  $O(n \log n)$
- `extract` - асимптотическая сложность  $O(1)$ ,  $O(\log n)$

Удаление при помощи `erase` по итератору работает за константное время, если удаление по ключу - то логарифмическое, и если последний случай с `multiset` тогда за линейное время.

Так же определяется асимптотическая сложность `extract`.

Функция `emplace_hint` может быть использована, чтобы "подсказать" куда вставить элемент - если позиция будет верная - тогда вставка будет за константное время, иначе - логарифмическое.

Важное замечание: `extract\insert` можно использовать только между контейнерами с одинаковыми алокаторами, иначе это `undefined behavior`.

Поиск \ нахождение элементов по условиям (Lookup):

- `count`, `find`, `contains`, `equal_range` - асимптотическая сложность  $O(\log n)$
- `lower_bound`, `upper_bound` - асимптотическая сложность  $O(\log n)$

`Contains` появилась в C++20 и возвращает `bool`, в остальном работая как `count`.

Функция `lower_bound` для ключа возвращает итератор на элемент, который не меньше по компаратору с переданным.

Функция `upper_bound` для ключа возвращает итератор на элемент, который больше чем заданный.

Наблюдатели (Observers):

- `key_comp`, `value_comp` - асимптотическая сложность  $O(1)$

Аллокатор:

- `get_allocator` - асимптотическая сложность  $O(1)$

## **`std::map` / `std::multimap`**

```
template <class Key, class T, class Compare = std::less<Key>,  
          class Alloc = std::allocator<std::pair<const Key, T>>>
```



```
class map
{
    using key_type = Key;
    using mapped_type = T;
    using value_type = std::pair<const Key, T>;
    using key_compare = Compare;
    using value_compare = Compare;
    //И все остальные using'и, которые были в std::vector
}
```

Есть 6 групп функций std::map.

Доступ к элементам:

- at, operator[] - асимптотическая сложность  $O(\log n)$

operator[] может использоваться для добавления нового элемента, не только для получения существующего.

Размер\длина:

- empty - асимптотическая сложность  $O(1)$
- size, max\_size - асимптотическая сложность  $O(1)$

Модификаторы:

- clear - асимптотическая сложность  $O(n)$
- erase - асимптотическая сложность  $O(1)$ ,  $O(\log n)$ ,  $O(n)$
- insert, insert\_or\_assign, emplace, try\_emplace -  $O(\log n)$
- emplace\_hint - асимптотическая сложность  $O(1)$ ,  $O(\log n)$
- swap - асимптотическая сложность  $O(1)$
- merge - асимптотическая сложность  $O(n \log n)$
- extract - асимптотическая сложность  $O(1)$ ,  $O(\log n)$

Почти всё действует как в std::set.

Функция insert\_or\_assign работает как оператор [], отличие в том что возвращается пара - итератор на вставленный элемент и bool определяющий был ли вставлен элемент.

Функция try\_emplace - пытается создать объект на месте, но если он уже создан - не будет делать ничего

Поиск:

- count, find, contains, equal\_range - асимптотическая сложность  $O(\log n)$
- lower\_bound, upper\_bound - асимптотическая сложность  $O(\log n)$

Наблюдатели (observers):

- key\_comp, value\_comp - асимптотическая сложность  $O(1)$

Аллокатор:

- `get_allocator` - асимптотическая сложность  $O(1)$

## Неупорядоченные ассоциативные контейнеры

Особенность неупорядоченных ассоциативных контейнеров заключается в том, что они используют хэш-функцию, для хранения ключей. Это позволяет добиться более быстрого доступа к элементам, в среднем за константное время.

### **`**std::unordered_set**`**

```
template <class Key, class Hash = std::hash<Key>,  
         class KeyEqual = std::equal_to<Key>,  
         class Allocator = std::allocator<Key>>  
class unordered_set  
{  
    //Те же самые using что и в std::set  
}
```

Содержит те же 5 групп функций, что и `std::set`:

Размер:

- `empty` - асимптотическая сложность  $O(1)$
- `size`, `max_size` - асимптотическая сложность  $O(1)$

Модификаторы:

- `clear` - асимптотическая сложность  $O(1)$
- `erase` - асимптотическая сложность  $O(1)$ ,  $O(\log n)$ ,  $O(n)$
- `insert`, `emplace` - асимптотическая сложность  $O(1)$
- `emplace_hint` - асимптотическая сложность  $O(1)$ ,  $O(\log n)$
- `swap` - асимптотическая сложность  $O(1)$
- `merge` - асимптотическая сложность  $O(n)$
- `extract` - асимптотическая сложность  $O(1)$ ,  $O(n)$

Поиск \ нахождение элементов по условиям (Lookup):

- `count`, `find`, `contains`, `equal_range` - асимптотическая сложность  $O(\log n)$
- `lower_bound`, `upper_bound` - асимптотическая сложность  $O(\log n)$

Наблюдатели (Observers):

- `key_eq`, `hash_function` - асимптотическая сложность  $O(1)$

Аллокатор:

- `get_allocator` - асимптотическая сложность  $O(1)$

### **`**std::unordered_map**`**

```
template <class Key, class Hash = std::hash<Key>,
          class KeyEqual = std::equal_to<Key>,
          class Allocator = std::allocator<Key>>
class unordered_map
{
    //Те же самые using что и в std::map

    using hasher = Hash;
    using key_equal = KeyEqual;
}
```

Присутствуют те же 6 групп функций как и в std::map.

Доступ к элементам:

- at, operator[] - асимптотическая сложность  $O(1)$ ,  $O(n)$

Размер\длина:

- empty - асимптотическая сложность  $O(1)$
- size, max\_size - асимптотическая сложность  $O(1)$
- reserve - асимптотическая сложность  $O(n)$ ,  $O(n^2)$

Модификаторы:

- clear - асимптотическая сложность  $O(n)$
- erase - асимптотическая сложность  $O(1)$ ,  $O(n)$
- insert, insert\_or\_assign, emplace, try\_emplace -  $O(\log n)$
- emplace\_hint - асимптотическая сложность  $O(1)$ ,  $O(\log n)$
- swap - асимптотическая сложность  $O(1)$
- merge - асимптотическая сложность  $O(n)$
- extract - асимптотическая сложность  $O(1)$ ,  $O(\log n)$

Поиск:

- count, find, contains, equal\_range - асимптотическая сложность  $O(\log n)$
- lower\_bound, upper\_bound - асимптотическая сложность  $O(\log n)$

Наблюдатели (observers):

- key\_eq, hash\_function - асимптотическая сложность  $O(1)$

Аллокатор:

- get\_allocator - асимптотическая сложность  $O(1)$

Функции load factor, max load factor позволяют регулировать количество элементов на bucket'е. По умолчанию это 1.0. Значение больше - создаст больше коллизий, но потребует меньше памяти, значение меньше - наоборот.

Адаптеры

Это разновидность контейнеров, которые могут использовать для своей реализации другой класс контейнера.

### **std::stack**

```
template <class T,
          class Container = std::deque<T>>

class stack
{
    using container_type = Container;
    //value_type, size_type, reference, const_reference

    Container c;
}
```

Функции:

- top - O(1)

Элемент на вершине стека.

- empty - O(1)
- size - O(1)
- swap - как в контейнере ниже
- push, emplace - как в контейнере ниже push\_back, emplace\_back
- pop - как в контейнере pop\_back

Условиям функций соответствуют помимо std::deque, std::vector и std::list.

### **std::queue**

Функции:

- front, back - O(1)
- empty - O(1)
- size - O(1)
- swap - как в контейнере ниже
- push, emplace - сложность как в контейнере ниже у функций push\_back, emplace\_back
- pop - сложность как в контейнере ниже у pop\_front

Типы доступных контейнеров: deque, list. std::vector нельзя, у него нет pop\_front.

### **std::priority\_queue**

Очередь с приоритетом. По умолчанию нижележащий контейнер это std::vector.

```
template <....  
    class Compare = std::less<typename Container::value_type>>  
class priority_queue  
{  
}; // Остальные using схожие
```

Функции:

- top - O(1)
- empty - O(1)
- size - O(1)
- swap - как в контейнере ниже
- push, emplace - как в контейнере ниже push\_back, emplace\_back
- pop - как в контейнере pop\_back

Помимо std::vector могут быть использованы std::list, std::deque. Очередь с приоритетом реализуется через кучу, она похожа на дерево. На вершине находится максимальный элемент, слева максимальный наследник, справа минимальный наследник.

## Итераторы

Итераторы это обобщенные указатели, которые используются для унифицированного доступа к контейнерам (и не только).

InputIterator:

- std::istream

ForwardIterator:

- std::forward\_list
- std::unordered\_set
- std::unordered\_map
- std::unordered\_multiset
- std::unordered\_multimap

BidirectionalIterator:

- std::list
- std::set
- std::multiset
- std::map
- std::multimap

RandomAccess \ ContinuesIterator (C++17):

- std::array
- std::vector
- std::string
- std::string\_view

- `std::valarray`

OutputIterator:

- `std::ostream`

Последовательность первых 5 итераторов иерархична, и каждая следующая группа имеет больше возможностей, чем предшествующая.

## InputIterator

```
//std::istream& is;

std::istreambuf_iterator it {is};
std::istreambuf_iterator<char> end;

//size_t count = std::distance(it, end); //O(n)
//Если строчку выше раскомментировать, то buf окажется пустой, т.к. it смещён!

std::string buf {it, end};
```

```
std::vector<int> v { std::istreambuf_iterator<int> { is },
                  std::istreambuf_iterator<int> {}};
```

## ForwardIterator

У него есть перегруженная функция `++`, но отсутствует `--`.

Оператор `+=` не перегружен, но его эффекта можно достичь при помощи `std::next` \ `std::advance`.

Любой контейнер можно сконструировать используя итераторы, таким образом можно из `std::forward_list` или `std::unordered_map` создать вектор, с соответствующей ниже лежащей структурой.

## BidirectionalIterator

В дополнение имеет перегрузку `--`. Для смещения на несколько элементов можно использовать `std::prev` \ `std::advance`.

Контейнеры имеют реверсированные итераторы `rbegin` \ `rend`.

## RandomAccessIterator

В дополнение перегружены операторы `+=`, `-=`. Но `prev`\`next`\`advance` - работают так же.

## OutputIterator

```
std::string buf { "text" };

std::ostream& os;

std::copy(buf.begin(), buf.end(), os); //Копирует строку в поток
```

## std::iterator\_trait

Структуры для метапрограммирования, позволяющие узнать тип итератора.

```
template <class Iter>
struct iterator_traits
{
    using difference_type = typename Iter::difference_type;
    using value_type = typename Iter::value_type;
    using pointer = typename Iter::pointer;
    using reference = typename Iter::reference;
    using iterator_category = typename Iter::iterator_category;
};

template <class T>
struct iterator_traits<T*>
{
    using difference_type = ptrdiff_t;
    using value_type = T;
    using pointer = T*;
    using reference = T&;
    using iterator_category = random_access_iterator_tag;
}
```

Существует 6 типов итератор тэгов:

- input\_iterator\_tag
- forward\_iterator\_tag - наследует предшествующий
- bidirectional\_iterator\_tag - наследует предшествующий
- random\_access\_iterator\_tag - наследует предшествующий
- continues\_iterator\_tag - наследует предшествующий
- output\_iterator\_tag

```
template <typename It>
using it_cat = typename std::iterator_trait<It>::iterator_category;

template <typename It>
using it_diff = typename std::iterator_trait<It>::difference_type;
```

```

template <class It>
auto distance(It first, It last, std::input_iterator_tag)
{
    it_diff<It> dist = 0;
    for (; first != last; ++first, ++dist);

    return dist;
}

template <class It>
auto distance(It first, It last, std::random_access_iterator_tag)
{
    return first < last ? (last - first) : -(first - last);
}

template <class It>
auto distance(It first, It last)
{
    return distance(first, last, it_cat<It>{});
}

```

## Алгоритмы и функциональные объекты

### Адаптеры итераторов

#### **std::reverse\_iterator**

```

std::string str {"text"};

std::reverse_iterator<std::string::iterator> rit {str.end()}; // C++11
auto rit = std::make_reverse_iterator(str.end()); // C++14
std::reverse_iterator rit{str.end()}; // C++17
auto rit = str.rbegin();

std::reverse_iterator<std::string::iterator> rend {str.begin()}; // C++11
auto rend = std::make_reverse_iterator(str.begin()); // C++14
std::reverse_iterator rend{str.begin()}; // C++17
auto rend = str.rend();

std::string reversed {rit, rend}; // == "txet"

```

#### **std::back\_insert\_iterator**

Данный адаптер при резименовании приводит к вставке элемента в конец, в данном случае вызовется `std::vector::push_back`:



```
std::vector<uint64_t> generate_vector(size_t n, std::functional<uint64_t()> g)
{
    std::vector<uint64_t> res;
    res.reserve(n);
    //Функция заполнен n элементов, функтором g
    std::generate_n(std::back_inserter<std::vector<uint64_t>> {res}, n, g);
    //C++11
    std::generate_n(std::back_inserter(res), n, g); //альтернатива
    std::generate_n(std::back_inserter{res}, n, g); //C++17
}
```

## std::move\_iterator

При разименовывании адаптера std::move\_iterator нижележащий тип кастится к r-value&, т.е. будут извлекаться ресурсы.

```
std::vector<std::string> v1 { "Hello", "my", "beautiful", "world" };
std::vector<std::string> v2;
v2.reserve(v1.size());

using iter_t = decltype(v1)::iterator;
std::copy(std::move_iterator<iter_t> {v.begin() }, //<iter_t> не нужно для C++17
          std::move_iterator<iter_t> {v1.end()}, //<iter_t> не нужно для C++17
          std::back_inserter(v2));

//v1 { "", "", "", "" }
//v2 { "Hello", "my", "beautiful", "world" }
```

## Функциональные объекты (Function objects)

- Функции
- Функторы - структуры с перегруженным operator()
- Lambda

Type	Name	Function object type	Return type	Equivalent
Comparisons	equal_to	Binary	bool	lhs == rhs
	not_equal_to	Binary	bool	lhs != rhs
	less	Binary	bool	lhs < rhs
	less_equal	Binary	bool	lhs <= rhs
	greater	Binary	bool	lhs > rhs
	greater_equal	Binary	bool	lhs >= rhs
Logical	logical_and	Binary	bool	lhs && rhs
	logical_or	Binary	bool	lhs    rhs
	logical_not	Unary	bool	!op
Arithmetic	plus	Binary	T	lhs + rhs
	minus	Binary	T	lhs - rhs
	multiplies	Binary	T	lhs * rhs
	divides	Binary	T	lhs / rhs
	modulus	Binary	T	lhs % rhs
	negate	Unary	T	-op
Bitwise	bit_and	Binary	T	lhs & rhs
	bit_or	Binary	T	lhs   rhs
	bit_xor	Binary	T	lhs ^ rhs
	bit_not	Unary	T	~op

## std::hash

Функтор вычисляющий значения хэш функции для встроенных типов, включая std::string и многие другие. Используется в неупорядоченных ассоциативных контейнерах, а так же может быть удобна при создании хэш функции из структур, содержащих несколько типов, которые умеет обрабатывать std::hash.

```

template <class Key>
struct hash;

template<> struct hash<bool>;
template<> struct hash<char>;
template<> struct hash<int8_t>;
template<> struct hash<uint8_t>;
template<> struct hash<char16_t>;
template<> struct hash<char32_t>;
template<> struct hash<wchar_t>;
template<> struct hash<int16_t>;
template<> struct hash<uint16_t>;
template<> struct hash<int32_t>;
template<> struct hash<uint32_t>;
template<> struct hash<long>;
template<> struct hash<unsigned long>;
template<> struct hash<int64_t>;
template<> struct hash<uint64_t>;
template<> struct hash<float>;
template<> struct hash<double>;
template<> struct hash<long double>;

template <class T> struct hash<T*>;

template<> std::hash<std::string>
template<> std::hash<std::u8string>
template<> std::hash<std::u16string>
template<> std::hash<std::u32string>
template<> std::hash<std::wstring>
template<> std::hash<std::error_code>
template<> std::hash<std::bitset>
template<> std::hash<std::unique_ptr>
template<> std::hash<std::shared_ptr>
template<> std::hash<std::type_index>
template<> std::hash<std::vector<bool>>
template<> std::hash<std::thread::id>
template<> std::hash<std::optional>
template<> std::hash<std::variant>
template<> std::hash<std::string_view>
template<> std::hash<std::wstring_view>
template<> std::hash<std::u8string_view>
template<> std::hash<std::u16string_view>
template<> std::hash<std::u32string_view>

```

```

struct Person
{
    std::string name, surname;
    uint8_t age;
}

template <typename T, typename... Rest>
size_t hash_combine(const T& obj1, const Rest&... objN)
{
    size_t res = 0; //Шаблонный сатанизм из boost:
    res ^= std::hash<T>{}(obj1) + 0x9e3779b9 + (res << 6) + (res >> 2);
    (... , (res ^= std::hash<Rest>{}(objN) + 0x9e3779b9 + (res << 6) + (res >> 2))
);

    return res;
}

namespace std
{
    template <>
    struct hash<Person>
    {
        size_t operator()(const SomeClass& obj) const noexcept
        {
            return hash_combine(obj.name, obj.surname, obj.age);
        }
    };
}

Person p { "A", "B", 7};
const auto hash_value = std::hash<decltype(obj)>{}(obj);

```

## Частичное применение функций

Если у нас есть функция с большим количеством аргументов, и мы хотим сделать вторую, которая будет вызывать изначальную, но принимать только часть её аргументов, а остальные проталкивать в соответствии с требуемыми условиями.

Для этого используется `std::bind`.

Она принимает функциональный объект и список аргументов.

```

void f(int a1, int a2, int a3, int a4, double a5)
{...}

using namespace std::placeholders;

int n = 7;
auto f2 = std::bind(f, _1, 42, _2, std::cref(n), 24);
//без cref мы передадим 7 и она всегда будет такой (копия)

```

```
f(1, 2, 3); //1 связано _1, 2 связано _2, 3 не используется
//вызовется f1(1, 42, 2, n, 24);
```

Это может быть полезно, например, для функции возвращающей вектор случайных чисел.

```
std::vector<int64_t> foo(int64_t a, int64_t b, size_t n)
{
    std::mt19937_64 engine { std::random_device{}() };
    std::uniform_int_distribution<int64_t> {a, b};

    std::vector<int64_t> res;
    res.reserve(n);

    std::generate_n(std::back_inserter(res), n,
                    [distr, engine]() mutable { return distr(engine); })
    //Если сделать захват по ссылкам &distr, &engine и тогда не mutable

    //Вариант с bind:
    std::generate_n(std::back_inserter(res), n,
                    std::bind(std::ref(distr), std::ref(engine)));

    return res;
}
```

## Алгоритмы

Причины использования алгоритмов:

- Делает код чище и читабельней
- Нет ошибок связанных с границами
- Высокий уровень оптимизации
- Исчерпывающая документация

### Не модифицирующие последовательные алгоритмы

- `std::all_of`
- `std::any_of`
- `std::none_of`

Принимают 2 итератора и применяют к каждому элементу предикат.

`std::all_of` вернёт true, если все элементы удовлетворяют предикату. `std::any_of` - если хотя бы один удовлетворяет предикату. `std::none_of` - если нет ни одного элемента, удовлетворяющего предикату.

- `std::for_each`
- `std::for_each_n`

Проходит по каждому элементу и выполняет функтор, если он модифицирует элемент - он будет изменен.

- `std::count`
- `std::count_if`

Подсчёт количества значений на диапазоне итераторов. `std::count_if` вместо значения принимает функциональный объект возвращающий `true`, если подсчёт необходим.

- `std::find`
- `std::find_if`
- `std::find_if_not`

Поиск элемента по значению, или по функциональному объекту возвращающему `bool`.

- `std::find_first_of`

Ищет группу элементов, заданных диапазоном итераторов, и возвращает итератор на первое вхождение одного из этих элементов.

- `std::find_end`

Ищет целую цепочку элементов, возвращается итератор на последнее вхождение этой цепочки.

- `std::mismatch`

Ищет точку в двух последовательностях, когда элементы не совпадают. Возвращает пару итераторов.

- `std::adjacent_find`

??? Ищет пару элементов смежных ??? Возвращает итератор на первый из этих двух элементов.

- `std::search`

Находит первое вхождение группы элементов. Т.е. почти как `std::find_end`.

В C++17 подвезли версию алгоритма с `searcher`. Например для строки появились `searcher`'ы, которые работают с алгоритмами Бойера-Мура-Хорспула и Бойера-Мура.

- `std::search_n`

Алгоритм ищет `n` вхождений одного элемента, т.е. когда подряд идет `n` одинаковых элементов.

## Модифицирующие последовательные алгоритмы

- `std::copy`
- `std::copy_if`
- `std::copy_n`

Копируют элементы из диапазона. Есть вариации когда используется предикат для проверки требуется ли копирование, и функция когда элемент копируется `n` раз.

- `std::copy_backward`

Копирует элементы начиная с последнего копируемого. Может оказаться полезной, при копировании внутри одного контейнера.

- `std::move`

Принимает диапазон итераторов, и переносит их по указанному итератору.

- `std::move_backward`

Как и прошлый алгоритм, но перемещение начинается с конца диапазона.

- `std::fill`
- `std::fill_n`

Заполнения нашей последовательности определенным значением. `fill_n` вставит только `n` элементов в последовательность.

- `std::generate`
- `std::generate_n`

Выполняет функциональный объект (без аргументов) для диапазона или `n` элементов.

- `std::transform`

Преобразует каждый элемент из последовательности при помощи унарного функционального объекта, и вставляет их по указанному адресу. Вторая версия алгоритма принимает 2 контейнера и осуществляет преобразование с использованием бинарного функционального объекта.

- `std::remove`
- `std::remove_if`

Принимает последовательность со значением для удаление, либо же функтор вместо значения.

Важно!Функция не удаляет сами элементы, а только смещает их так, словно произошло удаление, после этого необходимо вызвать функцию `erase`, в которую передать итератор возвращенный функцией `std::remove`.

- `std::remove_copy`
- `std::remove_copy_if`

Тоже осуществляет удаление, но записывает новую последовательность в другой контейнер. Т.е. изначальная последовательность остаётся без изменений.

- `std::replace`
- `std::replace_if`

Ищет элемент и меняет его на выбранное значение. Либо же выполняется предикат и меняется на выбранное значение.

- `std::replace_copy`
- `std::replace_copy_if`

Как и выше, но записывает в другую последовательность (контейнер).

- `std::unique`

Удаляет повторы элементов. Действует схоже как и `std::remove` - после требуется вызов `erase`.

Важно! Последовательность должна быть отсортирована.

- `std::unique_copy`

Как и функция выше, однако копирует в новую последовательность, т.е. не нужен вызов `erase` - старая последовательность неизменна.

- `std::swap_ranges`

Меняет местами объекты в двух последовательностях.

- `std::reverse`
- `std::reverse_copy`

Разворачивает последовательность задом наперёд, `_copy` версия копирует развернутую последовательность внутрь другой последовательности.

- `std::rotate`
- `std::rotate_copy`

Задается последовательность и точка в ней, точка становится первым элементом, остальное вращается по кругу 1,2,3,4 -> 3,4,1,2.

- `std::sample`

Принимает на вход совокупность элементов, и случайным образом генерируется `n` элементов.

### Разделяющие функции (Partitioning algorithms)

Данная группа алгоритмов осуществляет разделения на части.

- `std::partition`
- `std::stable_partition`
- `std::is_partitioned`
- `std::partition_point`

По предикату, позволяет разграничить последовательности на левую и правую. Например {x1, y1, x2, y2, x3} -> {x1, x2, x3, y1, y2}.

`std::partition` не гарантирует исходную последовательность перемещённых объектов, `stable_partition` гарантирует изначальный порядок исходной последовательности.

`std::partition_point` находит точку где последовательности разделяются на две под группы.

`std::is_partitioned` проверяет что элементы разделены на 2 последовательности.

- `std::partition_copy`

Работает схожим образом, но не изменяет исходную последовательность, а создаёт две новых.

## Алгоритмы с кучей (Heap algorithms)

Куча это сортирующее дерево, у которого корневой элемент максимальный (если куча отсортирована по максимуму), правый дочерний элемент больше левого дочернего элемента.

- `std::make_heap`

`{2, 3, 4, 5, 6, 7, 8, 9} -> {9, 4, 8, 2, 3, 6, 5, 2}`

Возможно передать компаратор, так чтобы сделать минимальную кучу, вместо максимальной.

Чтобы для ноды  $n$  найти наследников нужно  $n * 2 + 1$  или  $+ 2$ .

- `std::is_heap`
- `std::is_heap_until`

Проверяют является ли последовательность кучей, или есть ли подмножество кучи в последовательности. `std::is_heap_until` возвращают указатель на первый элемент нарушающий структуру кучи.

Так же есть версии с компараторами.

- `std::push_heap`

Алгоритм вставки в кучу, есть версия с компаратором.

- `std::pop_heap`

Сортирует кучу, так словно из неё удалён первый элемент.

Если каждый раз применять к последовательности на 1 элемент короче (убирая элемент с начала) - это будет алгоритм пирамидальной сортировки.

- `std::heap_sort`

Как и в других случаях с кучей, есть версия принимающая компаратор.

## Сортирующие алгоритмы

- `std::sort`
- `std::stable_sort`

Сортирует последовательность, есть возможность передать компаратор.

Если требуется чтобы элементы, которые одинаковые по значению - не переставлялись, есть вариант `std::stable_sort`, но он медленней.

- `std::is_sorted`
- `std::is_sorted_until`

Проверяет упорядоченна ли последовательность. `std::is_sorted_until` возвращает итератор на первый элемент, который выбивается из отсортированной последовательности. Есть версии с компараторами.

- `std::partial_sort`



Сортирует только часть последовательности, но другая хранит элементы строго больше или меньше (в зависимости от компаратора) чем отсортированная последовательность.

Алгоритм нужен например когда надо найти 10 лучших значений, но дальше сортировать 1000 элементов массива не нужно.

- `std::partial_sort_copy`

Как и алгоритм выше, но копирует всё в отдельную последовательность.

- `std::nth_element`

Сортирует только 1 элемент, при этом правая и левые части последовательности от него соблюдают условия что каждый их элемент меньше или больше (или равны) отсортированному элементу.

Может быть полезно чтобы найти кто находится на 11 месте, без того чтобы сортировать ни весь массив, ни 11 элементов.

- `std::merge`
- `std::inplace_merge`

Соединение отсортированных последовательностей - слияние двух массивов. Результат записывается в отдельную последовательность.

`std::inplace_merge` - отличается тем, что части находятся в одной последовательности, и результат остаётся в ней же.

## Бинарные алгоритмы поиска

Когда у нас отсортирована последовательность - появляется возможность осуществлять ускоренные методы поиска.

- `std::lower_bound`

Возвращает первый элемент, который не меньше, чем переданный нами как аргумент.

Т.е. {0, 1, 2, 3, 4, 5, 6}; и элемент 3, дадут указатель на 3, т.к.  $3 \geq 3$ .

- `std::upper_bound`

Возвращает первый элемент, который больше, в примере выше это будет 4.

- `std::binary_search`

Алгоритм поиска работающий с использованием `upper\lower_bound`. Работает за логарифмическое время.

- `std::equal_range`

Работает как комбинация `std::lower_bound` и `std::upper_bound`, возвращая пару итераторов.

## Алгоритмы для множеств

- `std::includes`

Проверка является ли одно множество подмножеством другого.

- `std::set_difference`

Разница множеств, из A вычитается B.

- `std::set_intersection`

Пересечение множеств.

- `std::set_symmetric_difference`

Подсчёт симметричной разницы (всё кроме пересечения).

- `std::set_union`

Совокупность двух множеств.

### **min\max алгоритмы**

- `std::min_element`

Поиск минимального в последовательности.

- `std::max_element`

Поиск максимального в последовательности.

- `std::minmax_element`

Сочитание двух предшествующих функций.

### **Алгоритмы сравнения**

- `std::equal`

Проверяет равны ли две наших последовательности. Но чаще используют `std::vector::operator==`.

- `std::lexicographical_compare`

Лексикографическое сравнение двух последовательностей. Используется в следующем алгоритме.

### **Алгоритмы перестановок**

Функции комбинаторики:

- `std::next_permutation`

Ищет следующую комбинацию, например {1, 1, 2} -> { 1, 2, 1 }.

Возвращает true если ещё возможны перестановки, или false - если они все уже осуществлены.

- `std::prev_permutation`

Обратный алгоритм.

- `std::is_permutation`

Проверка являются ли 2 последовательности перестановками.

## **Numeric algorithms**

- `std::iota`

Заполняет последовательность значениями, начиная с переданного значения, и инкрементируя его каждый шаг.

- `std::accumulate`

Считает сумму по последовательности, которая может начинаться с инициализированного значения.

- `std::reduce`

Работает схоже, но порядок не определен. Необходимо для возможности использовать параллельный алгоритм.

- `std::inner_product`

Скалярное произведение двух векторов.

- `std::partial_sum`

Суммирует элементы и в каждую ячейку записывает текущую сумму. Например {1, 2, 3, 4} -> {1, 3, 6, 10};

- `std::inclusive_scan`

Схоже как выше, учитывает *i* элемент в *i*-ой сумме. Может быть параллелен.

- `std::exclusive_scan`

Схоже как выше, но не учитывает *i* элемент в *i*-ой сумме. Может быть параллелен.

- `std::transform_reduce`
- `std::transform_inclusive_scan`
- `std::transform_exclusive_scan`

Как выше, но до суммирования ещё применяется функциональный объект.

## **Алгоритмы с неинициализированной памятью**

- `std::uninitialized_copy`
- `std::uninitialized_copy_n`

Вызывается конструктор от элемента на последовательности. (???)

- `std::uninitialized_fill`
- `std::uninitialized_fill_n`

Действует схоже, но конструирует объекты по передаваемому значению в функцию.

- `std::uninitialized_move`
- `std::uninitialized_move_n`

Как и выше, но перемещение.

- `std::uninitialized_default_construct`
- `std::uninitialized_default_construct_n`

Вызывает дефолтные конструкторы на последовательности.

- `std::uninitialized_value_construct`
- `std::uninitialized_value_construct_n`

Делает value initialization (???).

- `std::destroy`
- `std::destroy_n`

Разрушает объекты (вызывает деструктор) на последовательности.

---

## TODO

---

Проработать оглавление, поставить в нужном порядке темы, добиться того чтобы до любой подтемы можно было добраться через оглавление( например как сейчас не очень с алгоритмами, т.к. уже 4 уровня вложенности)

++ Forwarding reference дополнить и изучить внимательней

++ inline namespaces тоже в 11 фитчи

++std::invoke

++ Searcher function objects

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

+++ Идеомы

+++ шпоры filesystem +?

++ advanced constexpr?

++ TODO скользкие места C++ в UB

// TODO placement new

++ `std::true_type` пометить о существовании этих функций, где есть пример их реализации

++ Изучить мелкие фитчи, например

`int* pArr = new int[N]{};` - создаёт массив с дефолтными значениями (+ массивы на умных указателях)

++ узнать больше о `multiset` \ `multimap`

TODO при описании контейнеров выводить только разницу между ними (не дублировать море лишнего кода и названий полей) А может лучше всего даже сделать сводную таблицу, чтобы видеть сразу разницу между всеми контейнерами