Филипп Хандельянц

Лекция 4/12

Нововведения стандарта C++17

Докладчик

Хандельянц

Филипп Александрович

- Ведущий разработчик в команде PVS-Studio (C++/C#)
- 3 года участвую в разработке ядра C++ анализатора
- Автор статей о проверке open source-проектов

First X3J16 meeting

Somerset, NJ, USA (1990)

Completed C++11

Madrid, Spain (2011)

Completed C++14

Issaquah, WA, USA (2014)

Photo: Chandler Carruth and Olivier Giroux. License: tinyurl.com/9wn439f

Completed C++17

Kona, HI, USA (2017)

First X3J16 meeting

Somerset, NJ, USA (1990)

Completed C++11

Madrid, Spain (2011)

Completed C++14

Issaquah, WA, USA (2014)

Photo: Chandler Carruth and Olivier Giroux. License: tinyurl.com/9wn439f

Completed C++17

Kona, HI, USA (2017)

Внесенные изменения в C++17

Расширения ядра языка:

- Attributes
- noexcept
- Class template argument deduction
- template <auto>
- Structured bindings
- Stricter expression evaluation order
- 'auto' deduction from brace-init-list
- Copy elision
- Fold expressions
- Aggregates with public base classes
- '*this' in lambda capture list
- 'if' / 'switch' with initialization
- constexpr 'if'
- inline variables
- ___has_include
- Dynamic allocation of over-aligned types
- static_assert without message
- Removed things
- Other features

Расширения стандартной библиотеки:

- std::string_view
- std::optional
- std::variant
- std::any
- std::filesystem
- Parallel algorithms

Attributes

- [[fallthrough]]
- [[nodiscard]]
- [[maybe_unused]]

```
switch (i)
{
case 1:
  ....
  break;
case 2:
  ....
  break;
case 3:
  ....
  [[fallthrough]]; // No warning
case 50:
  ....
  break;
}
```

```
void foo(std::vector &vec)
{
  // some manipulations with vector

  vec.empty(); // doesn't erase vector!
}
```

```
template <typename T, typename Alloc>
class vector
{
  ....
  [[nodiscard]] bool empty() const noexcept { .... }
};


void foo(std::vector &vec)
{
  // some manipulations with vector

  vec.empty(); // warning: function result hasn't been used
  (void)vec.empty(); // return value has been ignored
}
```

```
struct [[nodiscard]] Foo
{
  ....
};

enum [[nodiscard]] Bar
{
  ....
};
```

```
void f(bool flag1, bool flag2)
{
  bool b = thing1 && thing2;
  assert(b); // disappears in release mode
}
```

```
[[maybe_unused]] void f([[maybe_unused]] bool flag1,
                        [[maybe_unused]] bool flag2)
{
  [[maybe_unused]] bool b = thing1 && thing2; // No warning
  assert(b); // disappears in release mode
}
```

- `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]] = 42 };`

noexcept

```
void foo() throw(A, B);
```

```
void foo() throw(A, B);

void (*pf1)() throw(A, B); // ok
pf1 = foo;
```

```
void foo() throw(A, B);

void (*pf1)() throw(A, B); // ok
pf1 = foo;

class C
{
  virtual void f() throw(A, B);
};

class D : C
{
  void f(); // compile-time error
};
```

```
void foo() throw(A, B);

void (*pf1)() throw(A, B); // ok
pf1 = foo;

class C
{
  virtual void f() throw(A, B);
};

class D : C
{
  void f(); // compile-time error
};

typedef void (*Pf2)() throw(A, B); // compile-time error
Pf2 pf = f;
```

```
throw(A, B, ...) // removed
throw() -> deprecated
throw() ≡ noexcept

typedef void (*nef)() noexcept; // ok
typedef void (*ef)();           // ok

void foo() noexcept;
void bar();

 ef pf1 = foo; // ok
nef pf2 = foo; // ok

 ef pf3 = bar; // ok
nef pf4 = bar; // compile-time error
```

Class template argument deduction

```
// until C++17
std::pair<int, int> pair { 0, 0 };
auto pair = std::pair<int, int> { 0, 0 };

std::mutex mutex;
std::lock_guard<std::mutex> lock { mutex };
```

```
// until C++17
std::pair<int, int> pair { 0, 0 };
auto pair = std::pair<int, int> { 0, 0 };

std::mutex mutex;
std::lock_guard<std::mutex> lock { mutex };


// since C++17
std::pair pair { 0, 0 };
auto pair = std::pair { 0, 0 };

std::lock_guard lock { mutex };
```

```
namespace std
{
  template <class Mutex>
  class lock_guard
  {
    explicit lock_guard(Mutex &mtx) { mtx.lock(); .... }
    ....
  };
}
```

```
namespace std
{
  template <class Mutex>
  class lock_guard
  {
    explicit lock_guard(Mutex &mtx) { mtx.lock(); .... }
    ....
  };

  // template <class Mutex>
  // explicit lock_guard(Mutex &mtx) -> lock_guard<Mutex>;
}
```

```
std::mutex mutex;

std::lock_guard lock { mutex };
```

```
std::mutex mutex;

std::lock_guard lock { mutex };

// auto lock = std::lock_guard(mutex);
```

```cpp
std::mutex mutex;

std::lock_guard lock { mutex };

// auto lock = std::lock_guard(mutex);

// template <class Mutex>
// explicit lock_guard(Mutex &mtx) -> lock_guard<Mutex>;

// explicit lock_guard<std::mutex>(std::mutex &mtx);

// Mutex ≡ std::mutex

std::lock_guard<std::mutex> lock { mutex }; // PROFIT!!!
```

```
// <array> header

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

std::array<int, 5> arr { 0, 1, 2, 3, 4 };
```

```
// <array> header

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>;
}

std::array<int, 5> arr { 0, 1, 2, 3, 4 };
```

```cpp
std::array arr { 0, 1, 2, 3, 4 };

// auto arr = std::array(0, 1, 2, 3, 4);

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

// array<int, int, int, int, int>(int, int, int, int, int);

// T ≡ int
// U ≡ (int, int, int ,int)
// sizeof...(U) == 4

std::array<int, 5> arr { 0, 1, 2, 3, 4 }; // PROFIT!!!
```

template <auto>

```
template <typename Type, Type n>
struct integral_constant
{
  ....
  using value_type = Type;
  static constexpr value_type value = n;
};

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

```
template <auto Val> // template <decltype(auto) Val>
struct integral_constant
{
  ....
  using value_type = decltype(Val);
  static constexpr value_type value = Val;
};

using  true_type = integral_constant<true>;
using false_type = integral_constant<false>;
```

Structured bindings

```
using Person = std::tuple<std::string, std::string, uint8_t>;

Person foo(size_t idx)
{
  ....
}

void bar()
{
  size_t idx = ....;

  std::string name, surname;
  uint8_t age;

  std::tie(name, surname, age) = foo(idx);
  ....
}
```

```
using Person = std::tuple<std::string, std::string, uint8_t>;

Person foo(size_t idx)
{
  ....
}

void bar()
{
  size_t idx = ....;

  auto [name, surname, age] = foo(idx); // since C++17
  ....
}
```

```
void bar()
{
  size_t idx = ....;

  // auto [name, surname, age] = foo(idx);
  auto &&__sym_1_2 = foo(idx);

  auto    name = std::get<0>(__sym_1_2); // std::string
  auto surname = std::get<1>(__sym_1_2); // std::string
  auto     age = std::get<2>(__sym_1_2); // uint8_t
  ....
}
```

```
void bar()
{
  size_t idx = ....;

  // auto &[name, surname, age] = foo(idx);
  auto &&__sym_1_2 = foo(idx);

  auto    &name = std::get<0>(__sym_1_2); // std::string &
  auto &surname = std::get<1>(__sym_1_2); // std::string &
  auto     &age = std::get<2>(__sym_1_2); // uint8_t &
  ....
}
```

```
void bar()
{
  size_t idx = ....;

  // auto &&[name, surname, age] = foo(idx);
  auto &&__sym_1_2 = foo(idx);

  auto    &&name = std::get<0>(__sym_1_2); // std::string &&
  auto &&surname = std::get<1>(__sym_1_2); // std::string &&
  auto     &&age = std::get<2>(__sym_1_2); // uint8_t &&
  ....
}
```

```
void bar()
{
  size_t idx = ....;

  // const auto &[name, surname, age] = foo(idx);
  auto &&__sym_1_2 = foo(idx);

  // const std::string &
  const auto &name = std::get<0>(__sym_1_2);

  // const std::string &
  const auto &surname = std::get<1>(__sym_1_2);

  // const uint8_t &
  const auto &age = std::get<2>(__sym_1_2);
  ....
}
```

```
void bar()
{
  int arr[3] = { 0, 1, 2 };
  auto [elem1, elem2, elem3] = arr;

  std::tuple<std::string, std::string, uint8_t> person { .... };
  auto [name, surname, age] = person;

  template <typename T1, typename T2>
  struct Pair { T1 first; T2 second; };

  auto [first, second] = Pair { .... };
}
```

```
class Person
{
  std::string m_name, m_surname;

public:
  Person(std::string name, std::string surname) : .... { .... }

  const std::string& GetName() const noexcept { .... }
  const std::string& GetSurname() const noexcept { .... }
};
```

```
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();
}
```

```
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 &;
  };
}
```

```
void bar()
{
  auto &&[name, surname] = Person { "Phillip", "Khandeliants" };

  // name's type ≡ const std::string &
  // name == "Phillip"

  // surname's type ≡ const std::string &
  // surname == "Khandeliants"
}
```

Stricter expression evaluation order

```
void bar()
{
  using std::literals;

  auto str = "I heard it even works if you don't believe"s;

  str.replace(0, 8, "")
     .replace(s.find("even"), 4, "sometimes")
     .replace(s.find(" don't"), 6, "");

  assert(s == "it sometimes works if you believe");
}
```

```
void bar()
{
  using std::literals;

  auto str = "I heard it even works if you don't believe"s;

  str.replace(0, 8, "")
     .replace(s.find("even"), 4, "sometimes")
     .replace(s.find(" don't"), 6, "");

  assert(s == "it sometimes works if you believe");
}

// s == "it sometimes works if you believe"

// s == "itsometimesn works if you dolieve"
// s == "itsometimesn works idon't believe"
```

- `a.b`
- `a->b`
- `a->*b`
- `a(b1, b2, b3)`
- `b @= a`
- `a[b]`
- `a << b << c`
- `a >> b >> c`

'auto' deduction from brace-init-list

```
// before 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>
```

```
// before 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>

// since C++17
auto v1 { 1, 2, 3 };   // compile-time error
auto v2 = { 1, 2, 3 }; // std::initializer_list<int>
auto v3 { 42 };        // int
auto v4 = { 42 };      // std::initializer_list<int>
```

Copy elision

```
std::string helloWorld()
{
  return "Hello, World"; // construct temporary std::string
}

void bar()
{
  std::string str = helloWorld(); // move temporary object
}
```

```
std::string helloWorld()
{
  return "Hello, World"; // Return value optimization
}

void bar()
{
  std::string str = helloWorld(); // construct object in-place
}
```

```
template <typename T>
T foo()
{
  return T();
}

template <typename T>
void bar()
{
  T tmp = T(T(foo<T>())); // no copy/move constructors called
}
```

```
template <typename T>
T foo()
{
  return T();
}

template <typename T>
void bar(T param)
{
  ....
}

template <typename T>
void foobar()
{
  bar(foo<T>()); // construct parameter in-place
}
```

Fold expressions

```
template <typename T>
constexpr auto sum(T t)
{
  return t;
}

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

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

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

// sum(0, 1, 2, 3) ≡ return 0 + sum(1, 2, 3);
// sum(1, 2, 3)    ≡ return 1 + sum(2, 3);
// sum(2, 3)       ≡ return 2 + sum(3);
// sum(3)          ≡ return 3;

// sum(0, 1, 2, 3) ≡ return 0 + 1 + 2 + 3;

// sum(0, 1, 2, 3) == 6
```

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

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

// sum(0, 1, 2, 3) ≡ return (((0 + 1) + 2) + 3);
```

(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 E_1) op E_2) op ...) op E_N)

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

```
// before C++17

template <typename ...Args>
void pass(Args &&...) { }

template <typename ...Types>
void print(const Types &...tN)
{
  pass( ( [&]{ std::cout << tN; }(), 0 )... );
}

print("Hello, ", "World");
```

```
// since C++17

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

Aggregates with public base classes

```
struct Base
{
  std::string name, surname;
};

struct Derived : Base
{
  uint8_t age;
};

Derived d1; // name and surname are empty
            // age contains undefined value;

Derived d2 {}; // age == 0

Derived d3 { { "Phillip", "Khandeliants" }, 24 }; // ok since C++17

Derived d4 { "Phillip", "Khandeliants", 24 }; // ok since C++17
```

```
struct Base
{
  std::string name, surname;
};

struct Derived : Base
{
  Derived() = delete;
  uint8_t age;
};

Derived d1; // compile-time error, default constructor is deleted

Derived d2 {}; // ok!!!!
```

'*this' in lambda capture list

```
class SomeClass
{
  int m_x = 0;

  void f() const noexcept { std::cout << m_x << '\n'; }
  void g() noexcept { ++m_x; }

  void foo()
  {
    // before C++17
    auto lambda1 = [self = *this]() { self.f(); };
    auto lambda2 = [self = *this]() mutable { self.g(); }

    lambda1();
    lambda2();
  }
};
```

```
class SomeClass
{
  int m_x = 0;

  void f() const noexcept { std::cout << m_x << '\n'; }
  void g() noexcept { ++m_x; }

  void foo()
  {
    // since C++17
    auto lambda1 = [*this]() { f(); };
    auto lambda2 = [*this]() mutable { g(); }

    lambda1();
    lambda2();
  }
};
```

'if' / 'switch' with initialization

```
std::map<std::string, uint8_t> persons;

void find_person(const std::string &name)
{
  const auto it = m.find(name);
  if (it != persons.end())
  {
    ....
  }

  // 'it' is visible here
}
```

```
std::map<std::string, uint8_t> persons;

void find_person(const std::string &name)
{
  if (const auto it = m.find(name); it != persons.end())
  {
    ....
  }

  // 'it' isn't visible here
}
```

```
std::map<std::string, uint8_t> persons;

void find_person(const std::string &name)
{
  {
    const auto it = m.find(name);
    if (it != persons.end())
    {
      ....
    }
  }

  // 'it' isn't visible here
}
```

```
std::map<std::string, uint8_t> persons;

void find_person(const std::string &name)
{
  if (const auto it = m.find(name); it != persons.end())
  {
    ....
  }
  else if (const auto it = ....; ....)
  {
    ....
  }

  // 'it' isn't visible here
}
```

```
std::map<std::string, uint8_t> persons;

void find_person(const std::string &name)
{
  {
    const auto it = m.find(name);
    if (it != persons.end())
    {
      ....
    }
    else
    {
      const auto it = ....; // hides previous 'it' declaration
      if (....) { .... }
    }
  }

  // 'it' isn't visible here
}
```

```
std::map<std::string, uint8_t> persons;

void insert_person(std::string name, uint8_t age)
{
  if (auto [it, ok] = persons.emplace(std::move(name), age); ok)
  {
    ....
  }
}
```

constexpr 'if'

```
class Person
{
  std::string m_name, m_surname;

public:
  Person(std::string name, std::string surname) : .... { .... }

  const std::string& GetName() const noexcept { .... }
  const std::string& GetSurname() const noexcept { .... }
};
```

```
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();
}
```

```
// since C++17

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

inline variables

```
// MyLibrary.h

#pragma once

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

```
// MyLibrary.h

#pragma once

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

```
// TU1.cpp

#include "MyLibrary.h"

void bar()
{
  foo(....);
}
```

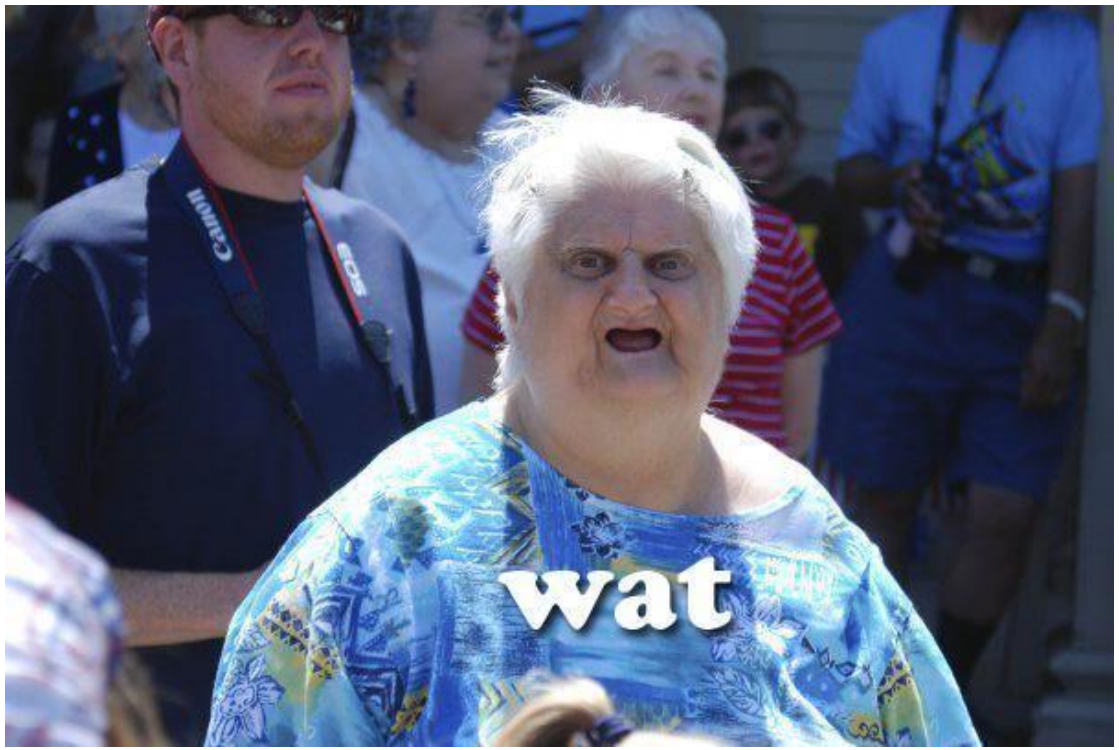
```
// TU2.cpp

#include "MyLibrary.h"

void foobar()
{
  foo(....);
}
```

```
1>TU1.cpp
1>TU2.cpp
1>TU2.obj : error LNK2005: "void __cdecl foo(void)"
(?foo@@YAXXZ) already defined in main.obj
1>TestExamples.exe : fatal error LNK1169: one or more multiply
defined symbols found
```

```
1>TU1.cpp
1>TU2.cpp
1>TU2.obj : error LNK2005: "void __cdecl foo(void)"
(?foo@@YAXXZ) already defined in main.obj
1>TestExamples.exe : fatal error LNK1169: one or more multiply
defined symbols found
```



```
// MyLibrary.h

#pragma once

inline void foo(....)
{
   ....
}
```

```
// MyLibrary.h

#pragma once

int foobar = ....;
```

```
// TU1.cpp

#include "MyLibrary.h"

void bar()
{
  foobar = 0;
}
```

```
// TU2.cpp

#include "MyLibrary.h"

void foobar()
{
  assert(foobar == 0);
}
```

```
// MyLibrary.h

#pragma once

inline int foobar = ....;
```

__has_include

```
#if __has_include(<optional>)
  #include <optional>
  #define have_optional 1
#elif __has_include(<experimental/optional>)
  #include <experimental/optional>
  #define have_optional 1
  #define experimental_optional 1
#else
  #define have_optional 0
#endif
```

Dynamic allocation of over-aligned types

```
struct alignas(32) S
{
  int a;
  char c;
};

int main()
{
  S *objects = new S[10];
  ....

  detele[] objects;
  return 0;
}
```

static_assert without message

```
static_assert ( constant-expression , string-literal ) ;

static_assert(a == 42, "a must be equal to 42");


static_assert ( constant-expression ) ;

static_assert(a == 42);


<source>:57:3: error: static_assert failed
  static_assert(a == 42);
  ^             ~~~~~

1 error generated.
```

std::string_view

```
// before C++17

void foo(const std::string &str) { .... }
```

```
// before C++17

void foo(const std::string &str) { .... }
void foo(const char *str) { .... }
void foo(const char *str, size_t len) { .... }

template <size_t N>
void foo(const char (&str)[N]) { .... }
```

```
// before C++17

void foo(const std::string &str) { .... }
void foo(const char *str) { .... }
void foo(const char *str, size_t len) { .... }

template <size_t N>
void foo(const char (&str)[N]) { .... }

// since C++17

void foo(std::string_view str) { .... }
```

```
#include <string_view>

template <typename CharT, typename Traits = char_traits<CharT>>
class basic_string_view;

using string_view = basic_string_view<char>;
using u16string_view = basic_string_view<char16_t>;
using u32string_view = basic_string_view<char32_t>;
using wstring_view = basic_string_view<wchar_t>;
```

```
#include <string_view>

template <typename CharT, typename Traits = char_traits<CharT>>
class basic_string_view;

using string_view = basic_string_view<char>;
using u16string_view = basic_string_view<char16_t>;
using u32string_view = basic_string_view<char32_t>;
using wstring_view = basic_string_view<wchar_t>;

template <typename CharT>
void foo(std::basic_string_view<CharT> str) { .... }
```

```
std::vector<std::string> vec;
....

std::sort(vec.begin(), vec.end(),
          [](const std::string &lhs, const std::string &rhs)
          {
            return lhs.substr(3) < rhs.substr(3);
          });
```

```
std::vector<std::string> vec;
....

std::sort(vec.begin(), vec.end(),
          [](std::string_view lhs, std::string_view rhs)
          {
            return lhs.substr(3) < rhs.substr(3);
          });
```

std::to_chars / std::from_chars

```
// returns 0, if fails

int atoi(const char *str);
long atol(const char *str);
long long atoll(const char *str);

// returns 0 if no conversion can be performed
// sets errno to ERANGE if converted value is out of range
// sets str_end pointer to the last incorrect symbol

long strtol(const char *str, char **str_end, int base);
long long strtoll(const char *str, char **str_end, int base);
```

```
// throws std::invalid_argument if no conversion can be performed
// throws std::out_of_range if converted value is out of range

// sets size_t value addressed by pointer to the last incorrect
// symbol

// returns value if conversion can be performed

int stoi(const std::string &str, size_t *pos = 0, int base = 10);
long stol(const std::string &str, size_t *pos = 0, int base = 10);
long long stoll(const std::string &str, size_t *pos = 0, int base = 10);
```

```cpp
#include <sstream>

template <typename T>
T to_integral(const std::string &str)
{
  T res {};
  std::ostringstream converter { str };
  converter >> res;
  return res;
}
```

```cpp
#include <charconv>

template <typename T>
T to_integral(std::string_view str)
{
  T num {};
  auto res = std::from_chars(str.data(), str.data() + str.size(), num);
  return num;
}
```

std::optional

```cpp
#include <charconv>

template <typename T>
T to_integral(std::string_view str)
{
  T num {};
  auto res = std::from_chars(str.data(), str.data() + str.size(), num);
  return num;
}
```

```cpp
#include <charconv>
#include <optional>

template <typename T>
std::optional<T> to_integral(std::string_view str)
{
  T num {};
  auto res = std::from_chars(str.data(), str.data() + str.size(), num);

  return res.ec == std::errc {}
           ? std::optional { num }
           : std::nullopt;
}
```

```
void foo(const std::string &str)
{
  const auto num = to_integral<uint64_t>(str); // std::optional

  auto tmp1 = *num;          // UB if num == std::nullopt

  auto tmp2 = num.value(); // throws exception
                           // if num == std::nullopt

  if (num) // num.has_value()
  {
    ....
  }
  else // std::nullopt
  {
    ....
  }
}
```

```
void foo(const std::string &str)
{
  const auto num = to_integral<uint64_t>(str); // std::optional

  auto tmp = num.value_or({}); // value from std::optional
                               // or default value
                               // if num == std::nullopt

  if (num < 10) // returns true if num == std::nullopt
                // or compares underlying value
                // with another value
  {
    ....
  }
}
```

```
class SomeClass { public: void bar() { .... } };

void foo()
{
  std::optional obj { SomeClass {} }; // std::optional<SomeClass>

  obj->bar(); // SomeClass::bar
}
```

std::variant

```cpp
#include <variant>

using Type = std::variant<size_t, double, std::string>;

Type var; // size_t by default

var = "Hello, I'm variant"; // std::string
var = size_t {0};           // size_t
var = 3.14;                 // double

var = Type { std::in_place_index<2>, 3, 'A' };          // "AAA"
var = Type { std::in_place_type<std::string>, 3, 'A'}; // "AAA"
```

```cpp
#include <variant>

std::variant<size_t, double, std::string> var; // size_t

var = "Hello, I'm variant"; // std::string

auto str = std::get<2>(var);             // std::string
auto str = std::get<std::string>(var); // std::string

auto d = std::get<1>(var); // throws exception
```

```cpp
#include <variant>

std::variant<size_t, double, std::string> var; // size_t

var = "Hello, I'm variant"; // std::string

auto str = std::get<2>(var);             // std::string
auto str = std::get<std::string>(var); // std::string

auto d = std::get<1>(var); // throws exception

auto f = std::get<float>(var); // compile-time error
```

```cpp
#include <variant>

std::variant<size_t, double, std::string> var; // size_t

var = "Hello, I'm variant"; // std::string

auto str = *std::get_if<2>(&var);              // std::string
auto str = *std::get_if<std::string>(&var); // std::string

if (std::get_if<double>(&var) == nullptr) // true
{
  ....
}
```

```cpp
#include <variant>

struct S
{
    S(int i) : m_i { i } {}
    int m_i;
};

struct U
{
  U(double d) : m_d { d } {}
  double m_d;
};

std::variant<S, U> var1;                    // compile-time error
std::variant<std::monostate, S, U> var2; // ok
```

```
struct Shape
{
  virtual void draw() const noexcept = 0;
};

struct Circle : Shape
{
  virtual void draw const noexcept override { .... }
};

struct Rectangle : Shape
{
   virtual void draw const noexcept override { .... }
};
```

```
std::vector<std::unique_ptr<Shape>> shapes;

shapes.push_back(std::make_unique<Circle>(....));
shapes.push_back(std::make_unique<Rectangle>(....));
....

for (auto &&shape : shapes)
{
  shape->draw();
}
```

```
struct Shape
{
  virtual void draw() const noexcept = 0;
};

struct Circle : Shape
{
  virtual void draw const noexcept override { .... }
};

struct Rectangle : Shape
{
   virtual void draw const noexcept override { .... }
};

using Shape = std::variant<Circle, Rectangle>;
```

```
std::vector<Shape> shapes;

shapes.push_back(Circle { .... });
shapes.push_back(Rectangle { .... });
....

struct Printer
{
  void operator()(const Circle &obj) { obj.draw(); }
  void operator()(const Rectangle &obj) { obj.draw(); }
};

for (auto &&shape : shapes)
{
  std::visit(Printer {}, shape);
}
```

```
std::vector<Shape> shapes;

shapes.push_back(Circle { .... });
shapes.push_back(Rectangle { .... });
....

for (auto &&shape : shapes)
{
  std::visit([](const auto &obj) { obj.draw(); }, shape);
}
```

std::any

```
std::any a = 1; // int
a = 3.14;       // double

std::cout << std::any_cast<float>(a) << '\n';  // throws exception
std::cout << std::any_cast<double>(a) << '\n'; // ok

const auto &typeInfo = a.type(); // RTTI

if (a.has_value())
{
  ....
}

a.reset();
```

std::filesystem

```
#include <filesystem>

int main(int argc, char *argv[])
{
  namespace fs = std::filesystem;
  for(auto &&entity: fs::directory_iterator(argv[0]))
  {
    switch (entity.status().type())
    {
    case file_type::regular:
      ....
    case file_type::directory:
      ....
    case file_type::symlink:
      ....
    }
  }
}
```

```
#include <filesystem>

explicit directory_iterator(const path &p);

directory_iterator(const path &p, error_code &ec);
```

Parallel algorithms

```
#include <algorithm>
#include <vector>


std::for_each(std::execution::par, vec.begin(), vec.end(),
              [](auto &e) { e += 42; });



inline constexpr sequenced_policy seq { /* unspecified */ };

inline constexpr parallel_policy par { /* unspecified */ };

inline constexpr parallel_unsequenced_policy par_unseq
{ /* unspecified */ };
```

Removed things

Trigraphs

- Trigraphs
- **'register' keyword**

- Trigraphs
- 'register' keyword
- **operator++ for bool type**

- Trigraphs
- 'register' keyword
- operator++ for bool type
- **Exception specification**

- Trigraphs
- 'register' keyword
- operator++ for bool type
- Exception specification
- **std::auto_ptr**

- Trigraphs
- 'register' keyword
- operator++ for bool type
- Exception specification
- std::auto_ptr
- **std::random_shuffle**

- Trigraphs
- 'register' keyword
- operator++ for bool type
- Exception specification
- std::auto_ptr
- std::random_shuffle
- **std::function allocators**

Other features

- **Nested namespaces**
  `(namespace std::detail { .... })`

- Nested namespaces
  (`namespace std::detail { .... }`)
- **constexpr lambdas**

- Nested namespaces
  (`namespace std::detail { .... }`)
- constexpr lambdas
- **std::*_v<T> traits**

- Nested namespaces
  (`namespace std::detail { .... }`)
- constexpr lambdas
- std::*_v<T> traits
- **std::byte**

- Nested namespaces
  (`namespace std::detail { .... }`)
- constexpr lambdas
- std::*_v<T> traits
- std::byte
- **Searcher function objects**

- Nested namespaces
  (`namespace std::detail { .... }`)
- constexpr lambdas
- std::*_v<T> traits
- std::byte
- Searcher function objects
- **std::invoke, std::apply, std::make_from_tuple**

- Nested namespaces
  (`namespace std::detail { .... }`)
- constexpr lambdas
- std::*_v<T> traits
- std::byte
- Searcher function objects
- std::invoke, std::apply,
  std::make_from_tuple
- **std::as_const**

- Nested namespaces
  (`namespace std::detail { .... }`)
- constexpr lambdas
- std::*_v<T> traits
- std::byte
- Searcher function objects
- std::invoke, std::apply,
  std::make_from_tuple
- std::as_const
- **std::clamp**

- Nested namespaces
  (`namespace std::detail { .... }`)
- constexpr lambdas
- std::*_v<T> traits
- std::byte
- Searcher function objects
- std::invoke, std::apply,
  std::make_from_tuple
- std::as_const
- std::clamp
- **try_emplace, insert_or_assign for associative containers**

- Nested namespaces (`namespace std::detail { .... }`)
- constexpr lambdas
- std::*_v<T> traits
- std::byte
- Searcher function objects
- std::invoke, std::apply, std::make_from_tuple
- std::as_const
- std::clamp
- try_emplace, insert_or_assign for associative containers

- **std::extract, std::insert, std::merge for associative containers**

- Nested namespaces (`namespace std::detail { .... }`)
- constexpr lambdas
- std::*_v<T> traits
- std::byte
- Searcher function objects
- std::invoke, std::apply, std::make_from_tuple
- std::as_const
- std::clamp
- try_emplace, insert_or_assign for associative containers
- std::extract, std::insert, std::merge for associative containers
- **Special math functions**

- Nested namespaces (`namespace std::detail { .... }`)
- constexpr lambdas
- std::*_v<T> traits
- std::byte
- Searcher function objects
- std::invoke, std::apply, std::make_from_tuple
- std::as_const
- std::clamp
- try_emplace, insert_or_assign for associative containers
- std::extract, std::insert, std::merge for associative containers
- Special math functions
- **std::not_fn**

- Nested namespaces (`namespace std::detail { .... }`)
- constexpr lambdas
- std::*_v<T> traits
- std::byte
- Searcher function objects
- std::invoke, std::apply, std::make_from_tuple
- std::as_const
- std::clamp
- try_emplace, insert_or_assign for associative containers
- std::extract, std::insert, std::merge for associative containers
- Special math functions
- std::not_fn
- **non-const std::string::data**

- Nested namespaces (`namespace std::detail { .... }`)
- constexpr lambdas
- std::*_v<T> traits
- std::byte
- Searcher function objects
- std::invoke, std::apply, std::make_from_tuple
- std::as_const
- std::clamp
- try_emplace, insert_or_assign for associative containers
- std::extract, std::insert, std::merge for associative containers
- Special math functions
- std::not_fn
- non-const std::string::data
- **std::size, std::data, std::empty**

- Nested namespaces (`namespace std::detail { .... }`)
- constexpr lambdas
- std::*_v<T> traits
- std::byte
- Searcher function objects
- std::invoke, std::apply, std::make_from_tuple
- std::as_const
- std::clamp
- try_emplace, insert_or_assign for associative containers
- std::extract, std::insert, std::merge for associative containers
- Special math functions
- std::not_fn
- non-const std::string::data
- std::size, std::data, std::empty
- **std::shared_ptr<T[]>**

- Nested namespaces (`namespace std::detail { .... }`)
- constexpr lambdas
- std::*_v<T> traits
- std::byte
- Searcher function objects
- std::invoke, std::apply, std::make_from_tuple
- std::as_const
- std::clamp
- try_emplace, insert_or_assign for associative containers
- std::extract, std::insert, std::merge for associative containers
- Special math functions
- std::not_fn
- non-const std::string::data
- std::size, std::data, std::empty
- std::shared_ptr<T[]>
- **std::gcd, std::lcm**

- Nested namespaces (`namespace std::detail { .... }`)
- constexpr lambdas
- std::*_v<T> traits
- std::byte
- Searcher function objects
- std::invoke, std::apply, std::make_from_tuple
- std::as_const
- std::clamp
- try_emplace, insert_or_assign for associative containers
- std::extract, std::insert, std::merge for associative containers
- Special math functions
- std::not_fn
- non-const std::string::data
- std::size, std::data, std::empty
- std::shared_ptr<T[]>
- std::gcd, std::lcm
- **std::scoped_lock**

- Nested namespaces (`namespace std::detail { .... }`)
- constexpr lambdas
- std::*_v<T> traits
- std::byte
- Searcher function objects
- std::invoke, std::apply, std::make_from_tuple
- std::as_const
- std::clamp
- try_emplace, insert_or_assign for associative containers
- std::extract, std::insert, std::merge for associative containers
- Special math functions
- std::not_fn
- non-const std::string::data
- std::size, std::data, std::empty
- std::shared_ptr<T[]>
- std::gcd, std::lcm
- std::scoped_lock
- **T& emplace_back(....);**

END

Q&A