Филипп Хандельянц

Лекция 12/12

Сборка C/C++ проектов и ее оптимизация

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

Внесенные изменения в C++20

- Concepts
- Ranges
- Coroutines
- Modules
- ~~Contracts~~
- operator <=>
- Attributes
- Range-base for with init
- constexpr all the things!
- <bit>
- Calendar and timezone
- std::span
- starts_with, ends_with
- Heterogeneous lookup for unordered containers
- 'contains' for associative containers
- std::lerp, std::midpoint
- std::syncbuf
- std::atomic_ref

Concepts

```
template <class T>
void PrintContainer(const T &c)
{
  if (c.empty())
  {
    return;
  }

  std::cout << '(' << c.front();
  for (auto it = std::next(c.begin()); it != c.end(); ++it)
  {
    std::cout << ", " << *it;
  }

  std::cout << ')';
}

PrintContainer(std::vector<int> { 0, 1, 2 }); // ok
PrintContainer(std::list<int> { 0, 1, 2 });   // ok

PrintContainer(std::stack<int> { std::deque<int> { 0, 1, 2 } }); // compile-time error
```

```
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;
  nonesuch(const nonesuch &) = delete;
  nonesuch(nonesuch &&) = delete;
  nonesuch& operator=(const nonesuch &) = delete;
  nonesuch& operator=(nonesuch &&) = delete;
};
```

```
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;
```

```
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>
using increment = decltype(++std::declval<T&>());

template <class T>
using dereference = decltype(*std::declval<T&>());

template <class T, class U>
using equal = decltype(std::declval<T>() == std::declval<U>());

template <class T, class U>
using not_equal = decltype(std::declval<T>() != std::declval<U>());
```

```
template <class T>
inline constexpr bool is_iterable_v =
                         is_detected_v<free_begin, T>
                      && is_detected_v<free_end, T>
                      && is_detected_v<increment, detected_t<free_begin, T>>
                      && is_detected_v<dereference, detected_t<free_begin, T>>

                      && is_detected_v<equal,
                                       detected_t<free_begin, T>,
                                       detected_t<free_end, T>>

                      && is_detected_v<not_equal,
                                       detected_t<free_begin, T>,
                                       detected_t<free_end, T>>;
```

```
template <class T>
inline constexpr bool is_iterable_v =
                        is_detected_v<free_begin, T>
                     && is_detected_v<free_end, T>
                     && is_detected_v<increment, detected_t<free_begin, T>>
                     && is_detected_v<dereference, detected_t<free_begin, T>>

                     && is_detected_v<equal,
                                      detected_t<free_begin, T>,
                                      detected_t<free_end, T>>

                     && is_detected_v<not_equal,
                                      detected_t<free_begin, T>,
                                      detected_t<free_end, T>>;

template <class T, std::enable_if_t<is_iterable_v<T>, int> = 0>
void PrintContainer(const T &c)
{
  ....
}
```

```
template <class T>
concept Incrementable = requires(T t)
{
  { ++t; } -> T&
  { t++; } -> T
};

template <class T>
concept Dereferencable = requires(T t) { { *t; } };

template <class T, class U>
concept EqualityComparable = requires(T t, U u)
{
  { t == u; } -> bool;
  { t != u; } -> bool;
  { u == t; } -> bool;
  { u != t; } -> bool;
};
```

```
template <class T>
concept Iterable = requires(T c)
{
  std::begin(c);
  std::end(c);

  requires Incrementable<decltype(std::begin(c))>;

  requires Dereferencable<decltype(std::begin(c))>;
  requires Dereferencable<decltype(++std::begin(c))>;
  requires Dereferencable<decltype(std::begin(c)++)>;

  requires EqualityComparable<decltype(std::begin(c)), decltype(std::end(c))>;
  requires EqualityComparable<decltype(++std::begin(c)), decltype(std::end(c))>;
  requires EqualityComparable<decltype(std::begin(c)++), decltype(std::end(c))>;
};
```

```
template <class T> requires Iterable<T> // 1
void PrintContainer(const T &c)
{
  ....
}
```

```
template <class T> requires Iterable<T> // 1
void PrintContainer(const T &c)
{
  ....
}

template <Iterable T> // 2
void PrintContainer(const T &c)
{
  ....
}
```

```
template <class T> requires Iterable<T> // 1
void PrintContainer(const T &c)
{
  ....
}

template <Iterable T> // 2
void PrintContainer(const T &c)
{
  ....
}

void PrintContainer(const Iterable auto &c) // 3
{
  ....
}

// template <class T> requires Iterable<T>
// void PrintContainer(const T &c)
// {
//   ....
// }
```

Ranges

```
std::vector<std::string> split(const std::string &str, const char sep)
{
  std::vector<std::string> res;

  auto prev = str.begin(), it = std::find(str.begin(), str.end(), sep);

  for (; it != str.end(); prev = it + 1, it = std::find(it + 1, str.end(), sep))
  {
    if (it - prev == 0)
    {
      continue;
    }

    res.emplace_back(prev, it);
  }

  res.emplace_back(prev, it);

  return res;
}
```

```
std::vector<std::string> split(const std::string &str, const char sep)
{
  std::vector<std::string> res;

  auto prev = str.begin(), it = std::find(str.begin(), str.end(), sep);

  for (; it != str.end(); prev = it + 1, it = std::find(it + 1, str.end(), sep))
  {
    if (it - prev == 0)
    {
      continue;
    }

    res.emplace_back(prev, it);
  }

  res.emplace_back(prev, it);

  return res;
}


auto res = split("Hello World", ' ');
```

```
std::vector<std::string> split(std::string_view str, const char sep)
{
  std::vector<std::string> res;

  auto prev = str.begin(), it = std::find(str.begin(), str.end(), sep);

  for (; it != str.end(); prev = it + 1, it = std::find(it + 1, str.end(), sep))
  {
    if (it - prev == 0)
    {
      continue;
    }

    res.emplace_back(prev, it);
  }

  res.emplace_back(prev, it);

  return res;
}


auto res = split("Hello World", ' ');
```

```
std::vector<std::string_view> split(std::string_view str, const char sep)
{
  std::vector<std::string_view> res;

  auto prev = str.begin(), it = std::find(str.begin(), str.end(), sep);

  for (; it != str.end(); prev = it + 1, it = std::find(it + 1, str.end(), sep))
  {
    if (it - prev == 0)
    {
      continue;
    }

    res.emplace_back(&*prev, it - prev);
  }

  res.emplace_back(&*prev, it - prev);

  return res;
}


auto res = split("Hello World", ' ');
```

```
template <class InputIt, class T>
InputIt find (InputIt first, InputIt last, const T &value);
```

```
template <class InputIt, class T>
InputIt find (InputIt first, InputIt last, const T &value);


namespace std::ranges
{

  template <InputIterator I, Sentinel<I> S, class T, class Proj = identity>
    requires IndirectRelation<ranges::equal_to<>, projected<I, Proj>, const T*>

  constexpr I find(I first, S last, const T &value, Proj proj = {}); // since C++20

}
```

```
template <class InputIt, class T>
InputIt find (InputIt first, InputIt last, const T &value);


namespace std::ranges
{

  template <InputIterator I, Sentinel<I> S, class T, class Proj = identity>
    requires IndirectRelation<ranges::equal_to<>, projected<I, Proj>, const T*>

  constexpr I find(I first, S last, const T &value, Proj proj = {}); // since C++20

}


void foo(const char *str)
{
  auto it = std::ranges::find(str, value_sentinel { '\0' }, ' ');
}
```

```
template <class InputIt, class T>
InputIt find (InputIt first, InputIt last, const T &value);


namespace std::ranges
{

  template <InputIterator I, Sentinel<I> S, class T, class Proj = identity>
    requires IndirectRelation<ranges::equal_to<>, projected<I, Proj>, const T*>

  constexpr I find(I first, S last, const T &value, Proj proj = {}); // since C++20

}


void foo(std::string_view name)
{
  std::map<uintmax_t, std::string> hashedNames = ....;

  auto it = std::ranges::find(hashedNames.begin(), hashedNames.end(), name,
                              [](const auto &v) -> std::string_view { return v.second; });
}
```

```
template <class InputIt, class T>
InputIt find (InputIt first, InputIt last, const T &value);


namespace std::ranges
{

  template <InputRange R, class T, class Proj = identity>
    requires IndirectRelation<ranges::equal_to<>, projected<iterator_t<R>, Proj>, const T*>

  constexpr safe_iterator_t find(R &&r, const T &value, Proj proj = {}); // since C++20

}
```

```
template <class InputIt, class T>
InputIt find (InputIt first, InputIt last, const T &value);


namespace std::ranges
{

  template <InputRange R, class T, class Proj = identity>
    requires IndirectRelation<ranges::equal_to<>, projected<iterator_t<R>, Proj>, const T*>

  constexpr safe_iterator_t find(R &&r, const T &value, Proj proj = {}); // since C++20

}

void foo(std::string_view name)
{
  std::map<uintmax_t, std::string> hashedNames = ....;

  auto it = std::ranges::find(hashedNames, name,
                              [](const auto &v) -> std::string_view { return v.second; });
}
```

Views

```
namespace std::views
{
  inline constexpr /* unspecified */       all = /* unspecified */;
  inline constexpr /* unspecified */ transform = /* unspecified */;
  inline constexpr /* unspecified */    filter = /* unspecified */;
  inline constexpr /* unspecified */      join = /* unspecified */;
  inline constexpr /* unspecified */     split = /* unspecified */;
  inline constexpr /* unspecified */      iota = /* unspecified */;
  inline constexpr /* unspecified */   reverse = /* unspecified */;
  inline constexpr /* unspecified */   counted = /* unspecified */;
}
```

```
void foo(const std::string &str)
{
  for (auto symb : str) // direct order
  {
    ....
  }
}
```

```
void foo(const std::string &str)
{
  for (auto symb : str) // direct order
  {
    ....
  }

  for (auto symb : std::views::reverse(str)) // reverse order since C++20
  {
    ....
  }

}
```

```
void foo(const std::string &str)
{
  for (auto symb : str) // direct order
  {
    ....
  }

  for (auto symb : std::views::reverse(str)) // reverse order since C++20
  {
    ....
  }

  using namespace std::views;
  for (auto symb : str | reverse) // pipe syntax since C++20
  {
    ....
  }
}
```

```
void foo(const std::string &str)
{
  for (auto symb : str) // direct order
  {
    ....
  }

  for (auto symb : std::views::reverse(str)) // reverse order since C++20
  {
    ....
  }

  using namespace std::views;
  for (auto symb : str | reverse) // pipe syntax since C++20
  {
    ....
  }

  for (auto splitted : str | transform([](unsigned char s) { return std::toupper(s); })
                           | split(' ')) // pipe syntax since C++20
  {
    ....
  }
}
```

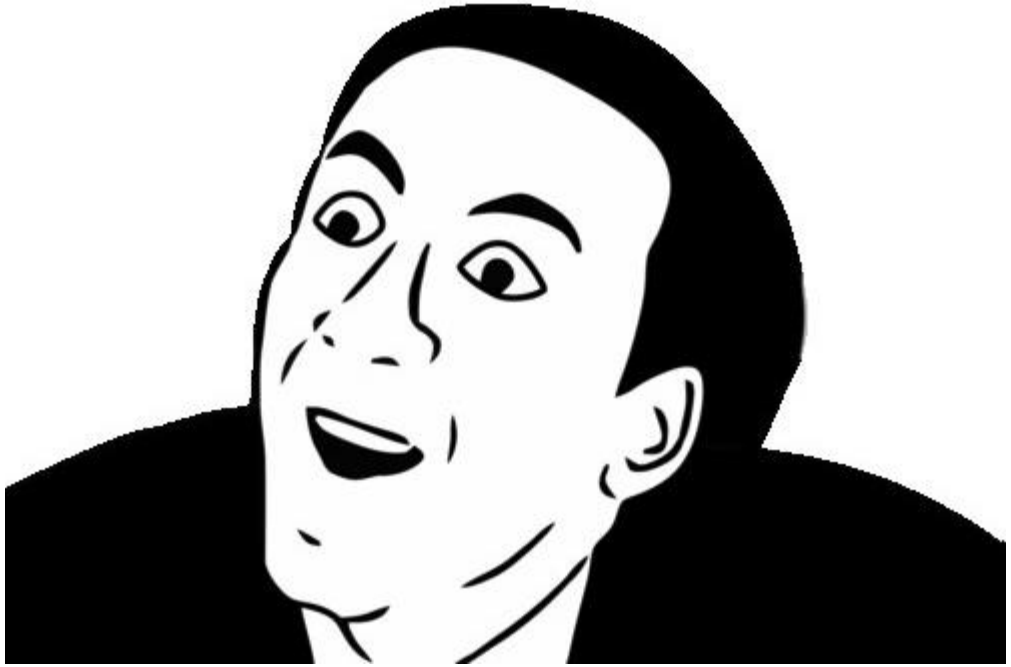
Coroutines

Что такое корутина (сопрограмма)?

- Сопрограмма – обобщение понятия подпрограммы

Что такое корутина (сопрограмма)?

- Сопрограмма – обобщение понятия подпрограммы



Что такое корутина (сопрограмма)?

- Сопрограмма – обобщение понятия подпрограммы
- Подпрограмма – часть кода, которая:
  - может быть многократно выполнена «вызывающим кодом»
  - может вернуть поток управления обратно «вызывающему коду»

Что такое корутина (сопрограмма)?

- Сопрограмма – обобщение понятия подпрограммы
- Подпрограмма – часть кода, которая:
  - может быть многократно выполнена «вызывающим кодом»
  - может вернуть поток управления обратно «вызывающему коду»

- Сопрограмма делает то же, что и подпрограмма, а также:
  - умеет приостанавливаться и возвращать управление «вызывающему коду»
  - «вызывающий код» может позвать сопрограмму снова, и она продолжит свое выполнение с места остановки

Что такое корутина (сопрограмма)?

- Сопрограммой является функция, содержащая что-либо из:
  - co_return
  - co_yield
  - co_await

Что такое корутина (сопрограмма)?

- Сопрограммой является функция, содержащая что либо-из:
  - co_return
  - co_yield
  - co_await

```
std::future<size_t> calc()
{
  return std::async([]
  {
    return ....;
  });
}
```

```
std::future<size_t> calc()
{
  auto res = co_await std::async([]
  {
    return ....;
  });

  co_return res;
}
```

```
#include <generator>

generator<T> sequence()
{
  for (T value {}; ; ++value)
  {
    co_yield value;
  }
}

void foo()
{
  std::vector<size_t> v;
  v.reserve(10);

  std::generate_n(std::back_inserter(v), 10, sequence<size_t>());
}
```

```
#include <generator>

generator<T> sequence()
{
  auto *__pContext = new __coroutine_context { .... };
  co_await pContext->promise.initial_suspend();

  for (T value {}; ; ++value)
  {
    co_yield value;
  }

final_suspend:
  co_await pContext->promise.final_suspend();
  delete pContext;
}
```

```
co_await expr;
```

```
auto &&__tmp = expr;
if (!__tmp.await_ready())
{
  __tmp.await_suspend(this_coroutine_handle);
}

__tmp.await_resume();
```

```
co_yield expr;
```

```
co_await __pContext->promise.yield_value(expr);
```

```
co_return expr;
```

```
__pContext->promise.return_value(expr);
goto final_suspend;
```

```
co_return;
```

```
__pContext->promise.return_void();
goto final_suspend;
```

```
template <class T>
struct generator
{
  struct promise_type { .... };

  coroutine_handle<promise_type> m_ch;

  ....
};
```

```
template <>
struct coroutine_handle<void>
{
  .... // constructors

  static coroutine_handle from_address(void *addr) noexcept;

  void* address() const noexcept;

  void resume() const;

  void operator()() const noexcept { resume(); }

  explicit operator bool() const noexcept;

  void destroy();

  bool done() const;
};
```

```
template <class Promise>
struct coroutine_handle : coroutine_handle<>
{
  using coroutine_handle<>::coroutine_handle; // constructors

  static coroutine_handle from_promise(Promise &prom) noexcept;

  static coroutine_handle from_address(void *addr) noexcept;

  Promise& promise() const noexcept;
};
```

```
template <class T>
struct generator
{
  struct promise_type { .... };

  coroutine_handle<promise_type> m_ch;

  explicit generator(promise_type & prom)
    : m_ch(coroutine_handle<promise_type>::from_promise(prom)) {}

   generator() = default;

   generator(const generator &) = delete;

   generator& operator=(const generator &) = delete;

   generator(generator &&other) : m_ch { std::exchange(other.m_ch, nullptr) } { }

   generator& operator=(generator&& _Right)
     : m_ch { std::exchange(other.m_ch, nullptr) } { }

   ~generator()
   {
     if (m_ch)
     {
       m_ch.destroy();
     }
   }
};
```

```
template <class T>
struct generator
{
  struct promise_type
  {
    const T *m_pCurr;

    generator get_return_object() const noexcept
    {
      return generator { coroutine_handle<promise_type>::from_promise(*this); };
    }

    suspend_always initial_suspend() const noexcept { return {}; }
    suspend_always   final_suspend() const noexcept { return {}; }

    suspend_always yield_value(T &&value) noexcept
    {
      m_pCurr = std::addressof(value);
      return {};
    }
  };

  ....
};
```

```
#include <generator>

generator<T> sequence()
{
  auto *__pContext = new __coroutine_context { .... };
  co_await pContext->promise.initial_suspend();

  for (T value {}; ; ++value)
  {
    co_yield value; // co_await __pContext->promise.yield_value(value);
  }

final_suspend:
  co_await pContext->promise.final_suspend();
  delete pContext;
}
```

```
struct suspend_always
{
  bool await_ready() const noexcept { return false; }
  void await_suspend(coroutine_handle<>) const noexcept { }
  void await_resume() const noexcept { }
};

struct suspend_never
{
  bool await_ready() const noexcept { return true; }
  void await_suspend(coroutine_handle<>) const noexcept { }
  void await_resume() const noexcept { }
};

struct suspend_if
{
  bool m_cond;

  explicit suspend_if(bool cond) : m_cond { cond } { }
  bool await_ready() const noexcept { return m_cond; }
  void await_suspend(coroutine_handle<>) const noexcept { }
  void await_resume() const noexcept { }
};
```

```
template <class T>
struct generator
{
  struct iterator
  {
    using iterator_category = input_iterator_tag;
    using difference_type   = ptrdiff_t;
    using value_type        = T;
    using reference         = const T &;
    using pointer           = const T *;

    coroutine_handle<promise_type> m_ch = nullptr;

    iterator() = default;
    iterator(nullptr_t) : m_ch(nullptr) {}

    iterator(coroutine_handle<promise_type> ch) : m_ch(ch) {}
  };

  ....
};
```

```
template <class T>
struct generator
{
  struct iterator
  {
    reference operator*() const noexcept
    {
      return *m_ch.promise().m_pCurr;
    }

    pointer operator->() const noexcept
    {
      return m_ch.promise().m_pCurr;
    }

    iterator& operator++()
    {
      m_ch.resume();
      if (m_ch.done())
      {
        m_ch = nullptr;
      }

      return *this;
    }
  };
```

```
template <class T>
struct generator
{
  ....

  iterator begin() noexcept
  {
    if (m_ch)
    {
      m_ch.resume();
      return { !m_ch.done() ? m_ch : nullptr };
    }

    return { m_ch };
  }

  iterator end() const noexcept
  {
    return { nullptr };
  }
};
```

```
template <typename T>
generator<T> seq()
{
  for (T i = {}; ; ++i)
    co_yield i;
}

template <typename T>
generator<T> take_until(generator<T> &g,
                        T limit)
{
  for (auto&& v: g)
    if (v < limit) co_yield v;
    else break;
}

template <typename T>
generator<T> multiply(generator<T> &g,
                      T factor)
{
  for (auto&& v: g)
    co_yield v * factor;
}
```

```
template <typename T>
generator<T> add(generator<T>& g, T adder)
{
  for (auto&& v: g)
    co_yield v + adder;
}

int main()
{
  auto s = seq<int>();
  auto t = take_until(s, 10);
  auto m = multiply(t, 2);
  auto a = add(m, 110);
  return std::accumulate(a.begin(),
                         a.end(), 0);
}
```

```cpp
template <typename T>
generator<T> seq() {
  for (T i = {};; ++i)
    co_yield i;
}

template <typename T>
generator<T> take_until(generator<T>& g, T limit) {
  for (auto&& v: g)
    if (v < limit) co_yield v;
    else break;
}

template <typename T>
generator<T> multiply(generator<T>& g, T factor) {
  for (auto&& v: g)
    co_yield v * factor;
}

template <typename T>
generator<T> add(generator<T>& g, T adder) {
  for (auto&& v: g)
    co_yield v + adder;
}

int main() {
  auto s = seq<int>();
  auto t = take_until(s, 10);
  auto m = multiply(t, 2);
  auto a = add(m, 110);
  return std::accumulate(a.begin(), a.end(), 0);
}
```

A ▾ ☐ 11010 ☐ ./a.out ☑ .LX0: ☐ lib.f: ☑ .text ☑ // ☐ \s+

```
main:                                   # @main
        mov     eax, 1190
        ret
```

```cpp
template <typename T>
generator<T> seq() {
  for (T i = {};; ++i)
    co_yield i;
}

template <typename T>
generator<T> take_until(generator<T>& g, T limit) {
  for (auto&& v: g)
    if (v < limit) co_yield v;
    else break;
}

template <typename T>
generator<T> multiply(generator<T>& g, T factor) {
  for (auto&& v: g)
    co_yield v * factor;
}

template <typename T>
generator<T> add(generator<T>& g, T adder) {
  for (auto&& v: g)
    co_yield v + adder;
}

int main() {
  auto s = seq<int>();
  auto t = take_until(s, 10);
  auto m = multiply(t, 2);
  auto a = add(m, 110);
  return std::accumulate(a.begin(), a.end(), 0);
}
```

A ☐ 11010 ☐ ./a.out ☑ .LX0: ☐ lib.f: ☑ .text ☑ // ☐ \s+

```asm
main:                                   # @main
        mov     eax, 1190
        ret
```

Modules

```
// A.cpp

#include <vector>
#include <list>
....

// all macros and symbols are available
```

```
// B.cpp

#define _ITERATOR_DEBUG_LEVEL 0

#include <vector>
#include <list>
....

// all macros and symbols are available

// "A.cpp" and "B.cpp" are different

// includes are sensitive to previous macro definitions
```

```
// B.cpp

#define _ITERATOR_DEBUG_LEVEL 0

// modules for legacy code, export macros and symbols
import <vector>;
import <list>;

// new cool modules, export symbols that you want, don't export macros
import std.vector;
import std.list;
....

// "A.cpp" and "B.cpp" are the same

// imports aren't sensitive to previous macro definitions!!!
```

```
// Interface part

// Starts with preamble

export module M;

export import N; // imports
                 // module N
                 // and exports

void foo(); // doesn't export

export void bar(); // exports
```

```
// Implementation part

// Starts with preamble

module M;

void foo() { .... }

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

Implementations

- MSVC
- GCC
- Clang

Contracts

~~Contracts~~

```
int foo(int *p)
{
  ....

  p->... // derefence without check
}

void bar()
{
  int res = foo(nullptr); // null dereference
}
```

[[contract-attribute modifier$_{(optional)}$ identifier$_{(optional)}$: cond-expr]]

contract-attribute ::= "expects", "ensures", "assert"

modifier ::= "default", "audit", "axiom"

cond-expr -> bool

```
int foo(int *p)
[[ expects: p != nullptr ]]
{
  ....

  p->...
}

void bar()
{
  int res = foo(nullptr); // contract violation!
}
```

```
int foo(int *p)
[[ expects: p != nullptr ]]
[[ ensures res: res >= 0 && res <= 10 ]]
{
  ....


  p->...;

  return 11; // contract violation!
}

void bar()
{
  int res = foo(nullptr); // contract violation!
}
```

```
int foo(int *p)
[[ expects: p != nullptr ]]
[[ ensures res: res >= 0 && res <= 10 ]]
{
  ....


  p->...;

  return 10; // ok
}

void bar()
{
  int res = foo(...); // ok

  if (res >= 0 && res <= 10) // always true, optimized by compiler
  {
    ....
  }
  else // unreachable code
  {

  }
}
```

```
int foo(int *p)
[[ expects: p != nullptr ]]
[[ ensures res: res >= 0 && res <= 10 ]]
{
  ....


  auto i = p->...;

  [[ assert: i > 0 ]];

  return 11; // contract violation!
}

void bar()
{
  int res = foo(nullptr); // contract violation!
}
```

```
void(const std::contract_violation &info);

void(const std::contract_violation &info) noexcept;

struct contract_violation
{
  uint_least32_t   line_number()     const noexcept;
  std::string_view file_name()       const noexcept;
  std::string_view function_name()   const noexcept;
  std::string_view comment()         const noexcept;
  std::string_view assertion_level() const noexcept;
};
```

```
void (const std::contract_violation &info)
{
  std::cerr << "line_number     : " << info.line_number()     << '\n';
  std::cerr << "file_name       : " << info.file_name()       << '\n';
  std::cerr << "function_name   : " << info.function_name()   << '\n';
  std::cerr << "comment         : " << info.comment()         << '\n';
  std::cerr << "assertion_level : " << info.assertion_level() << '\n';
}

// calls std::abort after violation_handler has returned
```

```
void (const std::contract_violation &info)
{
  std::cerr << "line_number     : " << info.line_number()     << '\n';
  std::cerr << "file_name       : " << info.file_name()       << '\n';
  std::cerr << "function_name   : " << info.function_name()   << '\n';
  std::cerr << "comment         : " << info.comment()         << '\n';
  std::cerr << "assertion_level : " << info.assertion_level() << '\n';

  throw std::runtime_error { "..." };
}
```

```
void (const std::contract_violation &info) noexcept
{
  std::cerr << "line_number     : " << info.line_number()     << '\n';
  std::cerr << "file_name       : " << info.file_name()       << '\n';
  std::cerr << "function_name   : " << info.function_name()   << '\n';
  std::cerr << "comment         : " << info.comment()         << '\n';
  std::cerr << "assertion_level : " << info.assertion_level() << '\n';

  throw std::runtime_error { "..." };
}
```

Spaceship operator

Spaceship operator

```
struct Point3d
{
  intmax_t x = 0, y = 0, z = 0;
};
```

```
struct Point3d
{
  intmax_t x = 0, y = 0, z = 0;
};

bool operator==(const Point3d &l, const Point3d &r) noexcept
{
  return l.x == r.x && l.y == r.y && l.z == r.y;
}

bool operator!=(const Point3d &l, const Point3d &r) noexcept
{
  return !(l == r);
}
```

```
struct Point3d
{
  intmax_t x = 0, y = 0, z = 0;
};

bool operator==(const Point3d &l, const Point3d &r) noexcept
{
  return l.x == r.x && l.y == r.y && l.z == r.y;
}

bool operator!=(const Point3d &l, const Point3d &r) noexcept
{
  return !(l == r);
}
```

```
struct Point3d
{
  intmax_t x = 0, y = 0, z = 0;
};

bool operator==(const Point3d &l, const Point3d &r) noexcept
{
  return l.x == r.x && l.y == r.y && l.z == r.z;
}

bool operator!=(const Point3d &l, const Point3d &r) noexcept
{
  return !(l == r);
}
```

```
struct Point3d
{
  intmax_t x = 0, y = 0, z = 0;

  friend constexpr auto operator<=>(const Point3d &, const Point3d &) = default;
};
```

```
struct Point3d
{
  intmax_t x = 0, y = 0, z = 0;

  friend constexpr auto operator<=>(const Point3d &, const Point3d &) = default;
};
```

```
struct Point3d
{
  intmax_t x = 0, y = 0, z = 0;

  friend constexpr auto operator<=>(const Point3d &, const Point3d &) = default;

  // friend constexpr bool operator< (const Point3d &, const Point3d &) { .... }
  // friend constexpr bool operator<=(const Point3d &, const Point3d &) { .... }
  // friend constexpr bool operator> (const Point3d &, const Point3d &) { .... }
  // friend constexpr bool operator>=(const Point3d &, const Point3d &) { .... }
  // friend constexpr bool operator==(const Point3d &, const Point3d &) { .... }
  // friend constexpr bool operator!=(const Point3d &, const Point3d &) { .... }
};


static_assert(Point3d { 0, 0, 0 } == Point3d { 1, 1, 1 }); // false
```

```cpp
#include <compare>

struct Point3d
{
  intmax_t x = 0, y = 0, z = 0;

  friend constexpr std::weak_ordering operator<=>(const Point3d &, const Point3d &) = default;

  // friend constexpr bool operator==(const Point3d &, const Point3d &) { .... }
  // friend constexpr bool operator!=(const Point3d &, const Point3d &) { .... }
};
```

Возвр. тип	Операторы	Свойства	Несравнимые типы
`std::weak_ordering`	`'==', '!='`	`a == b` `f(a) != f(b)`	Не разрешены
`std::strong_ordering`	`'==', '!='`	`a == b` `f(a) == f(b)`	Не разрешены
`std::partial_ordering`	`'==', '!=', '<',` `'<=', '>', '>='`	`a == b` `f(a) != f(b)`	Разрешены
`std::weak_ordering`	`'==', '!=', '<',` `'<=', '>', '>='`	`a == b` `f(a) != f(b)`	Не разрешены
`std::strong_ordering`	`'==', '!=', '<',` `'<=', '>', '>='`	`a == b` `f(a) == f(b)`	Не разрешены

```cpp
#include <compare>

struct Point3d
{
  intmax_t x = 0, y = 0, z = 0;

  friend constexpr std::weak_ordering operator<=>(const Point3d &l, const Point3d &r) noexcept
  {
    if (auto cmp = l.z <=> r.z; cmp != 0) return cmp;
    if (auto cmp = l.x <=> r.x; cmp != 0) return cmp;
    return l.y <=> r.y;
  }


  // friend constexpr bool operator==(const Point3d &, const Point3d &) { .... }
  // friend constexpr bool operator!=(const Point3d &, const Point3d &) { .... }
};
```

```cpp
#include <compare>

struct Point3d
{
  intmax_t x = 0, y = 0, z = 0;

  friend constexpr std::weak_ordering operator<=>(const Point3d &l, const Point3d &r) noexcept
  {
    if (auto cmp = l.z <=> r.z; is_neq(cmp)) return cmp;
    if (auto cmp = l.x <=> r.x; is_neq(cmp)) return cmp;
    return l.y <=> r.y;
  }


  // friend constexpr bool operator==(const Point3d &, const Point3d &) { .... }
  // friend constexpr bool operator!=(const Point3d &, const Point3d &) { .... }
};
```

Attributes

- [[likely]], [[unlikely]]
- [[no_unique_address]]
- [[nodiscard]]

[[likely]], [[unlikely]]

```
int f(int i)
{
  switch(i)
  {
  case 1: [[fallthrough]];

  [[likely]] case 2: return 1;
  }

  return 2;
}
```

[[no_unique_address]]

```
struct Empty {}; // empty class

struct Foo
{
  int i;
  Empty e;
};

static_assert(sizeof(Empty) == 0);
static_assert(sizeof(Foo)   == 4);
```

[[no_unique_address]]

```
struct Empty {}; // empty class

struct Foo
{
  int i;
  Empty e;
};

static_assert(sizeof(Empty) == 0); // asserts
static_assert(sizeof(Foo)   == 4); // asserts

static_assert(sizeof(Empty) == 1); // ok
static_assert(sizeof(Foo)   == 8); // ok
```

[[no_unique_address]]

```
struct Empty {}; // empty class

struct Foo : Empty
{
  int i;
};

static_assert(sizeof(Empty) == 1); // ok
static_assert(sizeof(Foo)   == 4); // ok
```

[[no_unique_address]]

```
struct Empty {}; // empty class

struct Foo
{
  int i;
  [[no_unique_address]] Empty e;
};

static_assert(sizeof(Empty) == 1); // ok
static_assert(sizeof(Foo)   == 4); // ok
```

[[nodiscard]]

```
[[nodiscard]] // since C++17

[[nodiscard(string-literal)]] // since C++20


template <class T, class Alloc = std::allocator<T>>
class vector
{
....

public:

  [[nodiscard("\"empty\" function doesn't clear vector")]]
  bool empty() const noexcept { .... }
};
```

Constexpr all the things!

- constexpr standard algorithms

- constexpr standard algorithms
- constexpr iterator

- constexpr standard algorithms
- constexpr iterator
- constexpr allocator

- constexpr standard algorithms
- constexpr iterator
- constexpr allocator
- constexpr std::vector, std::string

- constexpr standard algorithms
- constexpr iterator
- constexpr allocator
- constexpr std::vector, std::string
- consteval

consteval

```
consteval intmax_t sqr(intmax_t n)
{
  return n * n;
}

void foo(intmax_t n)
{
  constexpr auto a = sqr(3); // ok
  constexpr auto b = sqr(n); // compile-time error
}
```

- constexpr standard algorithms
- constexpr iterator
- constexpr allocator
- constexpr std::vector, std::string
- consteval
- Relaxed constexpr constraints

Small features

- char8_t

char8_t

```
#include <string_view>

bool foo(std::string_view data, int& result)
{
  result += data[0] - '0';
  return data[0] != '0';
}

bool foo(std::u8string_view data, int& result)
{
  result += data[0] - u8'0';
  return data[0] != u8'0';
}
```

- char8_t
- Feature test macros

- char8_t
- Feature test macros
- std::is_constant_evaluated

- char8_t
- Feature test macros
- std::is_constant_evaluated
- typename

- char8_t
- Feature test macros
- std::is_constant_evaluated
- typename
- Designated initialization

Designated initialization

```
struct Point3d
{
  intmax_t x = 0, y = 0, z = 0;
};

void foo()
{
  Point3d p1 { 0, 0, 0 };                  // ok since C++14

  Point3d p2 { .x = 1, .y = 1, .z = 1 }; // ok since C++20

  Point3d p3 { .z = 1 };                   // ok since C++20

  Point3d p4 { .z = 1, .x = 0 };           // compile-time error
}
```

- char8_t
- Feature test macros
- std::is_constant_evaluated
- typename
- Designated initialization
- [=, this] lambda capture

- char8_t
- Feature test macros
- std::is_constant_evaluated
- typename
- Designated initialization
- [=, this] lambda capture
- explicit(bool)

- char8_t
- Feature test macros
- std::is_constant_evaluated
- typename
- Designated initialization
- [=, this] lambda capture
- explicit(bool)
- Prohibited aggregate initialization with user-defined constructor

Standard library changes


bit

‹bit›

Calendar and timezone

- ‹bit›
- Calendar and timezone
- std::span

- ‹bit›
- Calendar and timezone
- std::span
- starts_with, ends_with

- ‹bit›
- Calendar and timezone
- std::span
- starts_with, ends_with
- Heterogeneous lookup for unordered containers

- ‹bit›
- Calendar and timezone
- std::span
- starts_with, ends_with
- Heterogeneous lookup for unordered containers
- 'contains' for associative containers

- ‹bit›
- Calendar and timezone
- std::span
- starts_with, ends_with
- Heterogeneous lookup for unordered containers
- 'contains' for associative containers
- std::lerp, std::midpoint

- ‹bit›
- Calendar and timezone
- std::span
- starts_with, ends_with
- Heterogeneous lookup for unordered containers
- 'contains' for associative containers
- std::lerp, std::midpoint
- std::syncbuf

- ‹bit›
- Calendar and timezone
- std::span
- starts_with, ends_with
- Heterogeneous lookup for unordered containers
- 'contains' for associative containers
- std::lerp, std::midpoint
- std::syncbuf
- std::atomic_ref

Nearest future

Reflection

- Reflection
- Executors

- Reflection
- Executors
- Networking library

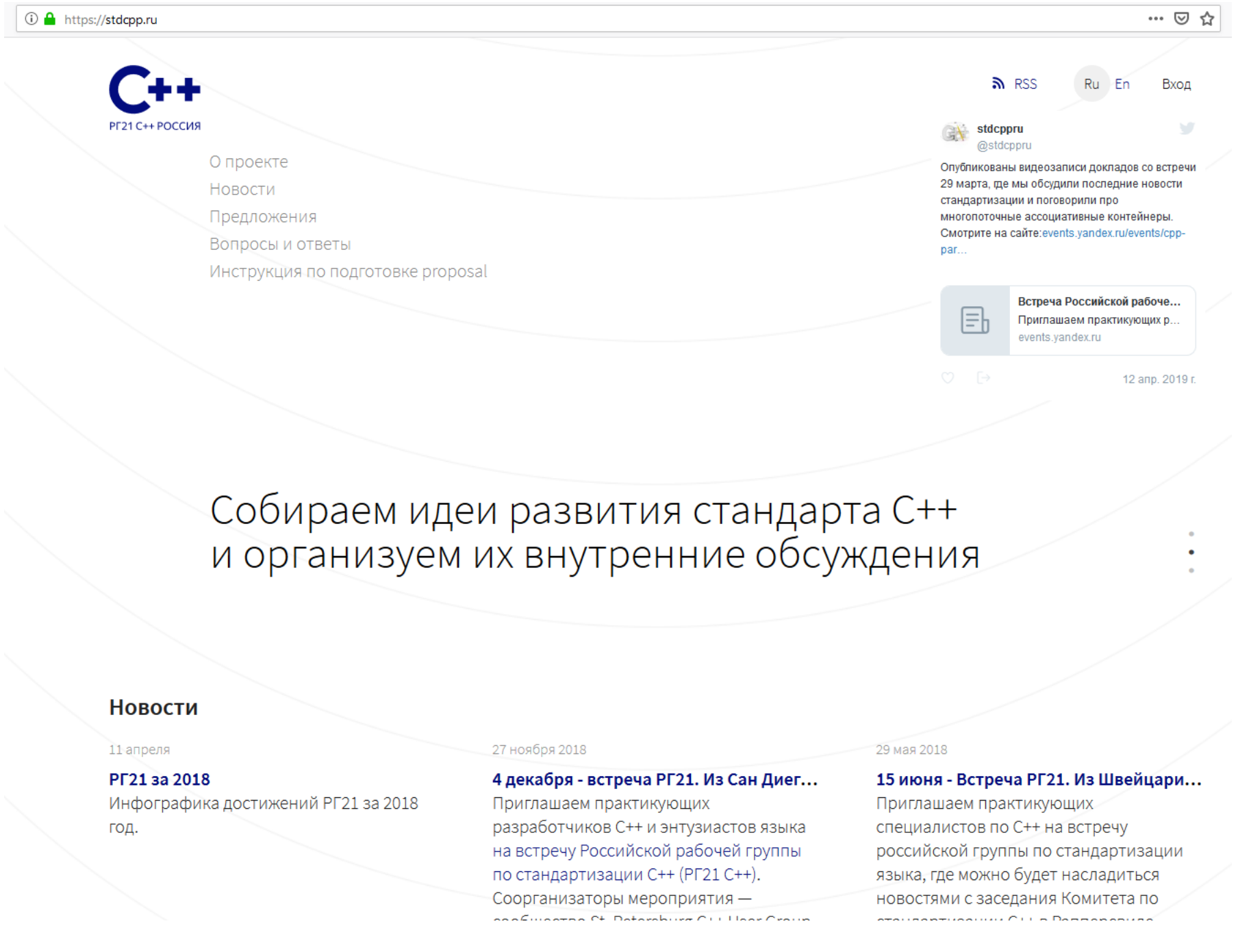
- Reflection
- Executors
- Networking library
- Contracts

- Reflection
- Executors
- Networking library
- Contracts
- Metaclasses

- Reflection
- Executors
- Networking library
- Contracts
- **Metaclasses**

You have an improvement idea?

You have an improvement idea?
Write a proposal ☺


https://stdcpp.ru
C++
РГ21 С++ РОССИЯ
RSS
Ru
En
Вход
О проекте
Новости
Предложения
Вопросы и ответы
Инструкция по подготовке proposal
stdcppru
@stdcppru
Опубликованы видеозаписи докладов со встречи 29 марта, где мы обсудили последние новости стандартизации и поговорили про многопоточные ассоциативные контейнеры. Смотрите на сайте:events.yandex.ru/events/cpp-par...
Встреча Российской рабоче...
Приглашаем практикующих р...
events.yandex.ru
12 апр. 2019 г.
Собираем идеи развития стандарта С++
и организуем их внутренние обсуждения
Новости
11 апреля
РГ21 за 2018
Инфографика достижений РГ21 за 2018 год.
27 ноября 2018
4 декабря - встреча РГ21. Из Сан Диег...
Приглашаем практикующих разработчиков С++ и энтузиастов языка на встречу Российской рабочей группы по стандартизации С++ (РГ21 С++). Соорганизаторы мероприятия —
29 мая 2018
15 июня - Встреча РГ21. Из Швейцари...
Приглашаем практикующих специалистов по С++ на встречу российской группы по стандартизации языка, где можно будет насладиться новостями с заседания Комитета по

END

Q&A