CONSTEXPR

Программы времени компиляции и метапрограммы

К. Владимиров, Intel, 2022
mail-to: konstantin.vladimirov@gmail.com

- Метапрограммирование

- Constexpr функции

- Мета-ООП

- Concepts

Идея "рекурсивного" раскрытия

- Вспомним функцию print_all, которая была написана нами ранее.

```
void print_all() { return 0; }

template <typename T, typename... Args>
void print_all(T first, Args... args) {
  std::cout << first << " ";
  print_all(args...);
}
```

- Здесь порождается цепочка экземпляров шаблонной функции.

```
print_all(1, 1.0, 1u);
// → print_all<double, unsigned>(1.0, 1u)
// → print_all<unsigned>(1u) → print_all()
```

Обсуждение

- На самом деле никакой рекурсии здесь нет: цепочка инстанцирований порождает разные инстанцирования
- Но сама схожесть процессов наводит на мысли
- Первым кого она навела на мысли был Эрвин Анрух в 1994-м году
- И эти мысли без малого обусловили успех C++

Открытие метапрограммирования

```
template <int i> struct D {
  D(void*); // 0 is ok, but not 1
  operator int();
};

template <int p, int i> struct is_prime {
  enum { prim = (p % i) && is_prime<(i > 2 ? p : 0), i - 1>::prim };
};

template <int i> struct Prime_print {
  Prime_print<i - 1> a;
  enum { prim = is_prime<i, i - 1>::prim };
  void f() { D<i> d = prim; } // error: Type enum can't be converted to type D<3>
};

struct is_prime<0, 0> { enum { prim = 1 }; };
struct is_prime<0, 1> { enum { prim = 1 }; };
struct Prime_print<2> { enum { prim = 1 }; void f() { D<2> d = prim; } };

int main () { Prime_print<30> a; }
```


Primes by Erwin Unruh, unruh.cc

Факториал

- Идея лежит на поверхности: что если развернуть систематическое sfinae от типов на целые числа?

```
template<size_t N> struct fact :
  integral_constant<size_t, N * fact<N - 1>{}> {};

template<> struct fact<0> : integral_constant<size_t, 1> {};

std::cout << fact<5>::value << std::endl;
```

- Вычисления итогового значения выполняются на этапе компиляции
- Наследование играет роль рекурсивного вызова

fact.cc

Числа Фибоначчи

- С той же лёгкостью можно вычислять на этапе компиляции числа Фибоначчи.

```
template <int N> struct fibonacci :
     std::integral_constant<int,
                            fibonacci<N - 1>{} +
                            fibonacci<N - 2>{}> {};

template <> struct fibonacci<1> :
     std::integral_constant<int, 1> {};

template <> struct fibonacci<0> :
     std::integral_constant<int, 0> {};
```

- Не смущает ли нас здесь двойная "рекурсия"?

fib.cc

Две модели вычислений

- "Императивная"

```
int fact (int x) {
  int i = 2, res = 1;
  for (; i <= x; ++i)
    res *= i;
  return res;
}
```

- Временные переменные
- Циклы
- Изменяемая память

- "Функциональная"

```
int fact(const int x) {
  if (x < 2)
    return x;
  else
    return x * fact(x - 1);
}
```

- Вызовы функций
- Рекурсия
- "Чистые" вычисления

Целочисленный квадратный корень

- Чтобы делать такие сложные вещи на шаблонах, полезно сначала просто написать программу в функциональном стиле.

```
int isqrt (int N, int lo = 1, int hi = N) {
  int mid = (lo + hi + 1) / 2;
  if (lo == hi) // это похоже на специализацию
    return lo;
  else {
    if (N < mid * mid) // как организовать if?
      return isqrt (N, lo, mid - 1);
    else
      return isqrt (N, mid, hi);
  }
}
```

Условный тип

- Вспомним уже известный нам условный тип:

```
template <bool B, typename T, typename F>
struct conditional { using type = T; }

template <typename T, typename F>
struct conditional<false, T, F> { using type = F; }

template <bool B, typename T, typename F>
using conditional_t = typename conditional<B, T, F>::type;
```

- Это отображение `{true, false}` на `{T, F}`

std::conditional

Целочисленный квадратный корень

- Здесь `std::conditional_t` вполне сработает в качестве meta-if

```
template <int N, int L = 1, int H = N, int mid = (L + H + 1) / 2>
struct Sqrt : std::integral_constant<int,
                  std::conditional_t<(N < mid * mid),
                                     Sqrt<N, L, mid - 1>,
                                     Sqrt<N, mid, H>>{}> {};

template <int N, int S> struct Sqrt <N, S, S, S> :
  std::integral_constant<int, S> {};
```

- Домашняя наработка: попробуйте найти N-е простое число на этапе компиляции

isqrt.cc

Квадранты вычислений

- Runtime computations
- Compile-time computations
- Type-level computations

```
template <typename T> struct add_const_pointer {
  using type = const T*;
};

using types = mpl::vector<int, char, float, void>;
using pointers = mpl::transform<types,
   add_const_pointer<mpl::_1>>::type;
```

- Heterogenious computations

Квадранты вычислений

- Runtime computations
- Compile-time computations
- Type-level computations
- Heterogenious computations

```
auto to_string = [](auto t) {
  std::stringstream ss; ss << t; return ss.str();
};

fusion::vector<int, std::string, float> seq{1, "abc", 3.4f};

auto strings = fusion::transform(seq, to_string);
```

Обсуждение

- Поговорим о вычислениях времени компиляции.
- Допустим я хочу предвычислить на этапе компиляции первые двадцать чисел Фибоначчи и использовать их на этапе исполнения.

- Рекурсивные инстанцирования
  - Constexpr функции
- Мета-ООП
- Concepts

Константность

- В чём смысл следующей конструкции и где она может быть применима?

```
uint8_t const volatile * const p_latch_reg = (uint8_t *) 0x42;
```

Константность

- В чём смысл следующей конструкции и где она может быть применима?

```
uint8_t const volatile * const p_latch_reg = (uint8_t *) 0x42;
```

- Это проводок с заданным адресом, с которого можно считать данные но не изменить их
- При этом сами данные могут непредсказуемо изменится, так что доступ к ним нельзя оптимизировать

```
data = *p_latch_reg; // считали значение
.....
data = *p_latch_reg; // снова считали значение
```

- Этот пример показывает, что `const` означает `readonly`

Что известно на этапе компиляции

- Литералы (`1, "hello", 'c', 1.0, 1ull`) и члены `enum`
- Параметры шаблонов и результаты `sizeof` над типами
- `constexpr` переменные

```
template <typename T> struct my_numeric_limits;

template <> struct my_numeric_limits<char> {
  static constexpr size_t max() { return CHAR_MAX; }
};

constexpr size_t arrsz = my_numeric_limits<char>::max();
int arr[arrsz]; // OK
```

Ограничение на constexpr переменные

- `constexpr` переменная должна иметь литеральный тип
- Использовать constexprs с плавающей точкой можно, но не рекомендуется

```
constexpr float ct = 1.0f / 3.0f;

assert (x == 1.0f && y == 3.0f);
float rt = x / y;

assert (rt == ct); // ORLY?
```

CONSTEXPR означает CONST?

- Следующий случай может быть несколько не очевиден:

```
constexpr int arr[] = {2, 3, 5, 7, 11};

constexpr int * x = &arr[3]; // всё хорошо?
```

- Тут зависит от того, к чему относится `constexpr` во второй строчке. Варианта, собственно, два

1. `constexpr int * x` → `const int * x`
2. `constexpr int * x` → `int * const x`

- Обсуждение: давайте проголосуем?

CONSTEXPR означает CONST?

- Следующий случай может быть несколько не очевиден:

```
constexpr int arr[] = {2, 3, 5, 7, 11};

constexpr const int * x = &arr[3]; // теперь всё хорошо
```

- Тут зависит от того, к чему относится constexpr во второй строчке. Варианта, собственно, два

1. `constexpr int * x` → `const int * x`
2. `constexpr int * x` → `int * const x`

- Обсуждение: давайте проголосуем?
- Второй вариант семантически консистентен: мы объявили constexpr pointer

C++17: constexpr control flow

- Возможность использования выражений времени компиляции делает интересным вопрос переключения по ним

```
if constexpr (b) {
  // тут много кода
}
else {
  // эта ветка не участвует в инстанцировании
}
```

- Начиная с C++17 такое ленивое поведение предоставляет `if constexpr`
- Обратите внимание: основное использование этой конструкции это выбрасывание веток инстанцирований

Некоторые альтернативы SFINAE

```
template <typename T> enable_if_t<(sizeof(T) > 4)>
foo (T x) { сделать что-то с x }

template <typename T> enable_if_t<(sizeof(T) <= 4)>
foo (T x) { сделать что-то ещё с x }
```

- Кажется, теперь появился иной вариант

```
template <typename T> void
foo (T x) {
  if constexpr (sizeof(T) > 4) { сделать что-то с x }
  else { сделать что-то ещё с x }
}
```

- Но это выглядит немного интрузивно. Скоро мы увидим ещё лучшие опции

If constexpr для вариабельных шаблонов

- В случае вариабельных шаблонов тоже можно избежать специализаций

```
template <typename Head, typename... Tail>
void print (Head head, Tail... tail) {
  std::cout << head;
  if constexpr(sizeof...(tail) > 0) {
    std::cout << ", ";
    print(tail...);
  }
}
```

- Вы понимаете почему это работает?

print_all.cc

Снова о метапрограммах

- Простая задача: возведение в квадрат времени компиляции

```
template <size_t n> square: integral_constant <size_t, n * n>;
int arr[square<5>{}]; // arr[25]
```

- Тут угадать, что `square` на самом деле функтор – довольно сложно

Снова о метапрограммах

- Простая задача: возведение в квадрат времени компиляции

```
template <size_t n> square: integral_constant <size_t, n*n>;
int arr[square<5>{}]; // arr[25]
```

- Тут угадать, что `square` на самом деле функтор – довольно сложно

```
constexpr int square(int x) { return x * x; }
int arr[square(5)]; // ok, arr[25]
```

- Теперь очевидно, что мы вызываем функцию времени компиляции
- Стандарт накладывает некоторые ограничения на тела таких функций

Ограничения в C++14

- new и delete
- Генерация исключений через throw
- Вызов не-constexpr функций
- Использование goto
- Лямбда выражения
- Преобразования const_cast и reinterpret_cast
- Преобразования void* в object*
- Модификация нелокальных объектов
- Неинициализированные данные
- Сравнения с unspecified результатом
- Вызов type_id для полиморфных классов и dynamic_cast
- Блоки try для обработки исключений
- Операции с undefined behavior
- Инлайн ассемблер во всех разновидностях
- Большая часть операций с this

Пример: целочисленный логарифм

```
constexpr size_t int_log (size_t N) {
  size_t pos = sizeof(size_t) * CHAR_BIT, mask = 0;

  // throw idiom
  if (N == 0) throw "N == 0 not supported";

  do {
    pos -= 1;
    mask = 1ull << pos;
  } while ((N & mask) != mask);

  if (N != mask) pos += 1;

  return pos;
}
```

Не всегда constexpr

- Логичный вопрос: можно ли перегрузить функцию по `constexpr`, чтобы иметь и статический и нестатический вариант `int_log`?
- Ответ немного удивителен: это просто не нужно. Статический вариант уже может быть использован с неизвестным на этапе компиляции аргументом

```
std::cin >> x;

std::cout << int_log (x) << std::endl;
```

- Поэтому `constexpr` не входит в тип функции и не может аннотировать параметры

Обсуждение

- Можем ли мы каким-то образом гарантировать, что constexpr функция выполнилась во время компиляции?

```
int t = int_log(5);
```

- Законных оснований надеяться на это здесь у нас нет

Обсуждение

- Можем ли мы каким-то образом гарантировать, что constexpr функция выполнилась во время компиляции?
- Решение: использовать в compile-time контексте (положить в constexpr переменную, сделать размером массива, параметризовать шаблон)

```
constexpr int logval = int_log(5);

int t = logval;
```

- Теперь мы уверены, что вызов состоялся на этапе компиляции

static_log.cc

C++20, введение consteval и constinit

- Функции, помеченные `consteval` обязаны быть выполнены именно и конкретно на этапе компиляции

```
consteval int ctsqr(int n) { return n*n; }

constexpr int r = ctsqr(100); // OK

int x = 100; int r2 = ctsqr(x); // Ошибка: не ct const
```

- Для того чтобы гарантировать только константную инициализацию `constexpr` наоборот слишком сильная гарантия и достаточно `constinit`

```
constinit int x = 1000; // запрещено для локальных переменных

++x; // OK
```


https://godbolt.org/z/vG8nPnMKM

Не везде constexpr

- Двойная природа constexpr функций имеет обратную сторону

```
template <typename T>
constexpr size_t ilist_sz(std::initializer_list<T> init) {
  constexpr size_t init_sz = init.size();
  return init_sz;
}
```

- Это ошибка. Компилятор тут не может дать гарантию константности для переменной (хотя сама функция и constexpr)
- Как вы думаете, изменится ли ситуация если я заменю на consteval?
- А если я уберу отмеченное красным?

https://godbolt.org/z/sd11ETEnf

Обсуждение

- Имеют ли смысл нестатические constexpr методы в классах?

- Рекурсивные инстанцирования
- Constexpr функции
- Мета-ООП
- Concepts

Пользовательские литеральные типы

- Чтобы сделать пользовательский тип литеральным, ему нужен constexpr конструктор

```
struct Complex{
 constexpr Complex(double r, double i) : re(r), im(i) { }
 constexpr double real() const { return re;}
 constexpr double imag() const { return im;}
private:
 double re, im;
};

constexpr Complex c{0.0, 1.0}; // это литеральное значение
```

Арифметика

- Для таких объектов становится возможной арифметика времени компиляции

```
constexpr Complex Complex::operator+= (Complex rhs) {
    re += rhs.re; im += rhs.im; return *this;
}

constexpr Complex operator+ (Complex lhs, Complex rhs){
  lhs += rhs; return lhs;
}
```

- Использование:

```
constexpr Complex c{0.0, 1.0}, d{1.0, 2.0};
constexpr Complex e = c + d;
```

Обсуждение

- Литералы такого класса выглядят как `Complex{1.0, 1.0}`
- Хотелось бы более привычной формы `1.0 + 1.0_i`
- Для сложения у нас есть выход, но как приделать суффикс?
- Удивительно, но для этого мы тоже используем перегрузку очень специального оператора

Пользовательский суффикс

- И этот оператор это оператор кавычки

```
struct Complex{
 constexpr Complex(double r, double i) : re(r), im(i) { }
 // и так далее
};

constexpr Complex operator "" _i (long double arg){
  return Complex{0.0, arg};
}

constexpr Complex c = 0.0 + 1.0_i; // ok, arg_i → ""_i(arg)
```

- Здесь суффикс определён с параметром типа double

https://godbolt.org/z/xxGgabvYo

Внезапная проблема

- Допустим, хочется переопределить суффикс _binary для бинарных констант
- Но уже даже довольно маленькая константа: `1010101010101_binary` не влазит в `unsigned long long` параметр
- Решение: синтаксис с вариабельным суффиксом

```
template<char... Chars>
constexpr unsigned long long operator "" _binary() {
  // и что мы напишем здесь?
}
```

Небольшая метапрограмма

```
template <int Sum, char... Chars> struct binparser;

template <int Sum, char... Rest> struct binparser<Sum, '0', Rest...>
  { static constexpr int value = binparser<Sum * 2, Rest...>::value; };

template <int Sum, char... Rest> struct binparser<Sum, '1', Rest...>
  { static constexpr int value =
                            binparser<Sum * 2 + 1, Rest...>::value; };

template <int Sum> struct binparser<Sum>
  { static constexpr int value = Sum; };

template<char... Chars> constexpr int operator "" _binary() {
  return binparser<0, Chars...>::value;
}
```


https://godbolt.org/z/W6sPsba6a

Ладно, это была шутка

```
template<char... Chars> constexpr int operator "" _binary() {
  std::array<int, sizeof...(Chars)> arr { Chars... };
  int sum = 0;
  for (auto c : arr)
    switch(c) {
      case '0': sum = sum * 2; break;
      case '1': sum = sum * 2 + 1; break;
      default: throw "Unexpected symbol";
    }
  return sum;
}
```

- Но как мы использовали в программе времени компиляции `std::array`?

https://godbolt.org/z/sTq76qTvd

Constexpr all the things!

- После их появления, constexpr-ctors начали торжественно расползаться по стандартной библиотеке
- Очевидно сразу появились constexpr-контейнеры std::array и std::bitset
- Точно так же сразу появились constexpr-алгоритмы.
- Постепенно контейнеров и алгоритмов (с некоторыми ограничениями) становится больше и больше.
- Первоначально написание дуального кода было связано с некоторыми проблемами.

https://godbolt.org/z/4z3hd1x84

Case study: замена vector на array

- Попробуем перейти от

```
template <typename T> class PermLoop {
  std::vector<T> loop_;
....
PermLoop(std::initializer_list<T> ls): loop_(ls) { reroll(); }
```

- К чему-то вроде (в таком виде это работать не будет).

```
template <typename T, size_t N> class PermLoop {
  std::array<T, N> loop_;
....
constexpr PermLoop(std::initializer_list<T> ls): loop_(ls) {
```


https://godbolt.org/z/cfo6obs5f

Обсуждение

```
template <typename T, size_t N> class PermLoop {
  std::array<T, N> loop_;
....
constexpr PermLoop(std::initializer_list<T> ls): loop_(ls) {
```

- Что будем делать?

Index sequences

- Удивительно полезный класс integer_sequence:

```
template <class T, T... Ints> class integer_sequence;
```

- Его синоним если нам нужны индексы:

```
template<size_t... Ints>
using index_sequence = std::integer_sequence<size_t, Ints...>;
```

- Мы можем писать `std::make_index_sequence<3>`
- Типом этого выражения является `integer_sequence<size_t, 1, 2, 3>`
- Теперь у нас есть инструменты чтобы подступиться к созданию array.

Переход от вектора к массиву

```
template <typename T, size_t N, size_t... Ns>
constexpr std::array<T, N>
make_array_impl(std::initializer_list<T> t,
                std::index_sequence<Ns...>) {
  return std::array<T, N>{*(t.begin() + Ns)...};
}

template <typename T, size_t N>
constexpr std::array<T, N>
make_array(std::initializer_list<T> t) {
  return make_array_impl<T, N>(t,
                               std::make_index_sequence<N>());
}
```

C++20: constexpr vector и string!

- Казалось бы мучений с заменой на array больше не надо?

```
struct S {
  std::vector<int> arr;
  constexpr S(std::initializer_list<int> il) : arr(il) {}
};
```

- Увы это (пока?) не работает даже с последним клангом: non-constexpr constructor 'vector' cannot be used in a constant expression.
- Интересно, конечно, как это гипотетически должно работать….

https://godbolt.org/z/K66n9GsP3

Core constant expression…

- Всё, что касается constexpr, полно сложных и странных сюрпризов.

```
struct S {
  int n_;
  S(int n) : n_(n) {} // non-constexpr ctor!
  constexpr int get() { return 42; }
};

int main() {
  S s{2};
  constexpr int k = s.get();
}
```


https://godbolt.org/z/ceMTj9vT7

Обсуждение

- Если у нас есть возможность считать вещи на этапе компиляции...
- Почему бы нам не считать также SFINAE характеристики?

- Рекурсивные инстанцирования
- Constexpr функции
- Мета-ООП
- Concepts

Обсуждение

- Говорят, что интерфейсы в статическом полиморфизме являются неявными
- Хорошо ли, что они неявные?
- Должны ли они быть неявными?
- Что если взять пример попроще и, находясь в реалиях C++17, попробовать сформулировать явный интерфейс в терминах типов?

Пример: проверка равенства

- В следующей функции неявный контракт состоит из одного пункта: равенство

```
template <typename T, typename U>
bool check_eq (T &&lhs, U &&rhs) { return (lhs == rhs); }
```

- Разумеется, это требование можно сформулировать явно

```
template <typename T, typename U, typename = void>
struct is_equality_comparable : false_type {};

template <typename T, typename U>
struct is_equality_comparable <T, U,
  void_t<decltype(declval<T>() == declval<U>())>>: true_type {};
```

- Вопрос в том, как его лучше всего проверить?

https://godbolt.org/z/6GaKr69G1

Пример: проверка равенства

- В следующей функции неявный контракт состоит из одного пункта: равенство

```
template <typename T, typename U>
bool check_eq (T &&lhs, U &&rhs) { return (lhs == rhs); }
```

- Опция по умолчанию в таких случаях это enable_if

```
template <typename T, typename U,
  typename = enable_if_t <is_equality_comparable<T, U>::value>>
bool check_eq (T &&lhs, U &&rhs) { return (lhs == rhs); }
```

- Теперь сообщение будет выглядеть как-то так:

error: no matching function for call to 'check_eq'

https://godbolt.org/z/6GaKr69G1

Обсуждение

```
template <typename T, typename U,
  typename = enable_if_t <is_equality_comparable<T, U>::value>>
bool check_eq (T &&lhs, U &&rhs) { return (lhs == rhs); }
```

- Какие проблемы вы здесь видите?

Обсуждение

```
template <typename T, typename U,
  typename = enable_if_t <is_equality_comparable<T, U>::value>>
bool check_eq (T &&lhs, U &&rhs) { return (lhs == rhs); }
```

- Какие проблемы вы здесь видите?
- Используется шаблонный параметр, которого на самом деле не существует

```
check_eq<int, std::string, void>(1, "1"); // oops, 157 err lines
```

- В случае проблемы будет выдано сообщение, что нет такой функции, но не будет ничего или почти ничего сказано о том почему её нет

Интересная идея

- Заслуживает внимания идея if constexpr + static assert

```
template <typename T, typename U>
bool check_eq (T &&lhs, U &&rhs) {
  if constexpr (!is_equality_comparable<T, U>::value>) {
    static_assert(0 && "equality comparable expected");
  }

  return (lhs == rhs);
}
```

- Стало лучше?
- Но мне не нравится эта идея. Почему?

https://godbolt.org/z/fKfevEv89

Интересная идея

- Заслуживает внимания идея if constexpr + static assert

```
template <typename T, typename U>
bool check_eq (T &&lhs, U &&rhs) {
  if constexpr (!is_equality_comparable<T, U>::value>) {
    static_assert(0 && "equality comparable expected");
  }

  return (lhs == rhs);
}
```

- Перенося проверку корректности из констекста подстановки в тело функции мы меняем SFINAE-out на ошибку. Но часто мы хотим именно SFINAE-out

Загадочный distance

- Вспомним наши мучения с самописным итератором где мы нечто забыли…

```
int main () {
  int arr[10];
  junk_iter_t fst(arr), snd(arr + 3);
  auto dist = std::distance(fst, snd);
}
```

- Он выдаёт ошибку

error: no matching function for call to 'distance(junk_iter_t&, junk_iter_t&)'

- Вы помните с лекции по итераторам в чём тут было дело?

https://godbolt.org/z/5b8GjE8Y8

Констрейнты

- Констрейнты были введены чтобы сделать статические интерфейсы явными

```
template <typename T, typename U> bool
  requires is_equality_comparable<T, U>::value
check_eq (T &&lhs, U &&rhs) { return (lhs == rhs); }
```

- Больше нет мусорного параметра шаблона. Языковые средства используются для того, для чего должны
- Сообщение об ошибке куда как лучше

```
'is_equality_comparable<T, U, void>::value' evaluated to false
```

- Внутри requires может быть что угодно, вычислимое на этапе компиляции

Полное покрытие

- Все помнят почему не работает очевидный SFINAE подход к разграничению?

```
template <typename T, typename = enable_if_t<(sizeof(T) > 4)>>
void foo (T x) { сделать что-то с x }

template <typename T, typename = enable_if_t<(sizeof(T) <= 4)>>
void foo (T x) { сделать что-то ещё с x }
```

- Очевидный подход через констрейнты вполне работает

```
template <typename T> requires (sizeof(T) > 4)
void foo (T x) { сделать что-то с x }

template <typename T> requires (sizeof(T) <= 4)
void foo (T x) { сделать что-то ещё с x }
```


https://godbolt.org/z/8KoGbbabK

Недостатки sfinae-constraints

- Увы, SFINAE определители не упорядочены в отношении ограниченности

```
template <typename It>
struct is_input_iterator: std::is_base_of<
   std::input_iterator_tag,
   typename std::iterator_traits<It>::iterator_category>{};

template <typename It>
struct is_random_iterator: std::is_base_of<
    std::random_access_iterator_tag,
    typename std::iterator_traits<It>::iterator_category>{};
```

- Это просто два разных шаблона. И это приводит к проблемам, когда мы пытаемся исправить `distance`

Недостатки sfinae-constraints

- Увы, SFINAE определители не упорядочены в отношении ограниченности

```
template <typename Iter>
  requires is_input_iterator<Iter>::value
int my_distance(Iter first, Iter last) {
  int n = 0; while (first != last) { ++n; ++first; } return n;
}

template <typename Iter>
  requires is_random_iterator<Iter>::value
int my_distance(Iter first, Iter last) { return last - first; }
```

- При реальном использовании здесь будет неоднозначность для `std::vector`

https://godbolt.org/z/8KoGbbabK

Сложные ограничения

- Вернёмся к простому примеру

```
template <typename T, typename U> bool
  requires is_equality_comparable<T, U>::value
check_eq (T &&lhs, U &&rhs) { return (lhs == rhs); }
```

- То же самое можно записать через requires-expression

```
template <typename T, typename U> bool
  requires requires(T t, U u) { t == u; }
check_eq (T &&lhs, U &&rhs) { return (lhs == rhs); }
```

- Да, requires-requires может смущать. Но вспомните noexcept-clause и noexcept-expression

Ещё лучше диагностика

```
template <typename T, typename U> bool
  requires requires(T t, U u) { t == u; }
check_eq (T &&lhs, U &&rhs) { return (lhs == rhs); }
```

- Выражение

```
check_eq(std::string{"1"}, 1);
```

- Даёт

```
note: the required expression '(t == u)' would be ill-formed
```

- Здесь сказано не только название констрейнта, но ещё и конкретный ill-formed expression в нём

https://godbolt.org/z/cK1qzqKnb

Главное отличие сложных ограничений

- Простые ограничения вычисляются на этапе компиляции

```
template <typename T> constexpr int somepred() { return 14; }

template <typename T>
  requires (somepred<T>() == 42)
bool foo (T&& lhs, U&& rhs);
```

- В сложных ограничениях проверяется синтаксическая валидность выражения

```
template <typename T>
  requires requires (T t) { somepred<T>() == 42; }
bool bar (T&& lhs, U&& rhs);
```

- В итоге вызов foo будет ошибкой, а вызов bar нет

https://godbolt.org/z/WY6j5bMTx

Что проверяют сложные ограничения

- Сложные ограничения могут проверять валидность выражений

```
requires requires(T a, T b) { a + b; }
```

- Либо они могут проверять существование типов

```
requires requires() { typename T::inner; }
```

- Есть специальный синтаксис для noexcept

```
requires requires(T t) {
  { ++t } noexcept;
}
```

- Они могут комбинироваться друг с другом и с простыми ограничениями

Пример: convertible_to

- Чтобы выделять системы ограничений, в C++20 введено специальное ключевое слово concept
- Простейший концепт который определён в хедере concepts и часто используется как вспомогательный

```
template<class From, class To>
concept convertible_to =
  std::is_convertible_v<From, To> &&
  requires(From (&f)()) { static_cast<To>(f()); };
```

- Он состоит и из старых SFINAE определителей и из новых концептов

Синтаксический сахар

- Чтобы немного проще записывать одновременное требование к выражению и типу:

```
requires requires(T x) {
  *x;
  requires convertible_to<decltype(*x), typename T::inner>;
}
```

- Существует более приятная форма записи со стрелочкой.

```
requires requires(T x) {
  {*x} -> convertible_to<typename T::inner>;
}
```

Концепты

- На основе простых концептов можно строить более сложные.

```
template <typename T, typename U>
concept WeaklyEqualityComparableWith =
  requires(const std::remove_reference_t<T>& t,
           const std::remove_reference_t<U>& u) {
    { t == u } -> convertible_to<bool>;
    { t != u } -> convertible_to<bool>;
    { u == t } -> convertible_to<bool>;
    { u != t } -> convertible_to<bool>;
  };
```

- Концепт это предикат, выполняющийся на этапе компиляции.

https://godbolt.org/z/Ez981d41P

Концепты

- Теперь при наличии концепта, довольно легко ограничить функцию

```
template <typename T, typename U>
 requires WeaklyEqualityComparableWith<T, U>
bool foo(T x, U y);
```

- Это также просто как использовать обычный предикат времени компиляции
- Можно определять одни концепты в терминах других

```
template <typename T>
concept EqualityComparable = WeakEqualityComparableWith<T, T>;
```

Отношение subsumes

- Сложные концепты можно написать так, чтобы они участвовали в отношениях большей или меньшей ограниченности

P subsumes Q if it can be proven that P implies Q

- Если в концепте P присутствуют все атомарные ограничения из Q, в таких же логических связях, то между ними есть это отношение
- Самое простое это прямое включение

```
template <typename T>
concept P = Q<T> && R<T>; // P subsumes Q and R

template <typename T>
concept P = Q<T> || sizeof(T) == 4; // P not subsumes Q
```

Теперь перегрузка работает

```
template <std::input_iterator Iter>
int my_distance(Iter first, Iter last) {
  int n = 0;
  while (first != last) { ++first; ++n; }
  return n;
}

template <std::random_access_iterator Iter>
int my_distance(Iter first, Iter last) {
  return last - first;
}
```

- Благодаря тому, что `InputIterator` является менее общим (он входит как подусловие в `RandomAccessIterator`) тут нет неоднозначности

https://godbolt.org/z/zrrdzjKeG

Литература

- Information technology – Programming languages – C++, ISO/IEC 14882, 2017
- Bjarne Stroustrup – The C++ Programming Language (4th Edition)
- Davide Vandevoorde, Nicolai M. Josuttis – C++ Templates. The Complete Guide, 2nd edition, Addison-Wesley Professional, 2017
- Scott Shurr, "Constexpr Introduction" and "Constexpr Applications", CppCon'15
- Dietmar Kuhl, "Constant fun", CppCon'16
- Ben Deane, Jason Terner, "Constexpr all the things", CppCon'17
- Arne Mertz, "Constexpr Additions in C++17", 2017
- Andrew Sutton – Concepts in 60: everything you need to know about concepts, CppCon'18