

Концепты в C++

Обзор предложения Concepts Lite TS
и его реализации в GCC 6.1

Константин Владимиров, SMWare, 2016

mail-to: konstantin.vladimirov@gmail.com

Концепты В С++

→ **Контроль полиморфизма**

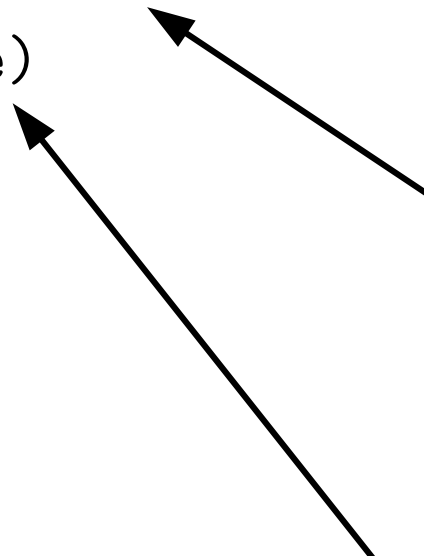
- ♦ Простые ограничения
- ♦ Сложные ограничения
- ♦ Простые концепты
- ♦ Вариабельные концепты
- ♦ Частичное упорядочение
- ♦ Критика концептов

Обобщенный код

```
template<typename R, typename T> bool  
in (R const& range, T const& value)  
{  
    for (auto const& x : range)  
        if (x == value)  
            return true;  
    return false;  
}
```

Обобщенный код: неявные обещания

```
template<typename R, typename T> bool  
in (R const& range, T const& value)  
{  
    for (auto const& x : range)  
        if (x == value)  
            return true;  
    return false;  
}
```



```
R::begin()  
R::end()  
typename R::iterator  
R::iterator::operator++()  
R::iterator::operator!=()  
R::iterator::operator*()
```

```
typename R::elemT  
bool ::operator==(R::elemT, T)
```

Обобщенный код: нарушение

```
template<typename R, typename T> bool  
in (R const& range, T const& value)  
{  
    for (auto const& x : range)  
        if (x == value)  
            return true;  
    return false;  
}
```

bool operator==(string, int)?

```
vector<string> v { "0", "1", "2" };  
bool is_in = in (v, 0);
```

Кара нарушителю

...

In file included from /home/tilir/Applications/gcc-5.2/include
/c++/5.2.0/bits/stl_algobase.h:67:0,

from /home/tilir/Applications/gcc-5.2/include
/c++/5.2.0/vector:60,

from constr1.cc:1:

/home/tilir/Applications/gcc-5.2/include/c++/5.2.0/bits

/stl_iterator.h:820:5: note: candidate:

template<class _IteratorL, class _IteratorR, class _Container>

bool __gnu_cxx::operator==(const __gnu_cxx::__normal_iterator

<_IteratorL, _Container>&, const __gnu_cxx::__normal_iterator

<_IteratorR, _Container>&)

operator==(const __normal_iterator

<_IteratorL, _Container>& __lhs,

^

/home/tilir/Applications/gcc-5.2/include/c++/5.2.0/bits

/stl_iterator.h:820:5: note: template argument deduction

/substitution failed:

constr1.cc:10:11: note: 'const std::__cxx11::basic_string<char>'

is not derived from 'const __gnu_cxx::__normal_iterator

<_IteratorL, _Container>'

if (x == value)

...

Контроль полиморфизма

- Отсекает лишние инстанцирования
- Улучшает диагностику ошибок
- Позволяет дополнительно разграничивать семейства полиморфных классов (пример: `complex<int-like>` VS `complex<double-like>`)
- Неужели никто никогда до него не додумался?

Упрощенный пример

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

Неявный контракт из одного пункта

`operator== (T, U);`

Упростилась и ошибка

```
eqcomp00.cc: In instantiation of 'bool check(T&&, U&&)
[with T = int; U = noeq]':
```

```
eqcomp00.cc:26:27:   required from here
```

```
eqcomp00.cc:16:15: error: no match for 'operator=='
(operand types are 'int' and 'noeq')
```

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

```
~~~~~^~~~~~
```

Проверка на существование (==)

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

// I feel really C++ here

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

Первая попытка: static assert

```
template <typename T, typename U> bool  
check_eq (T &&lhs, U &&rhs) {  
    static_assert (  
        is_equality_comparable<T, U>::value,  
        "Comparable types required");  
    return (lhs == rhs);  
}
```

Стало ли лучше?

Увы, стало хуже

```
eqcomp00.cc: In instantiation of 'bool check(T&&, U&&)
[with T = int; U = noeq]':
```

```
eqcomp00.cc:26:27:   required from here
```

```
eqcomp00.cc:14:3: error: static assertion failed:
```

```
Comparable types required
```

```
    static_assert (is_equality_comparable<T, U>::value,
    ^~~~~~
```

```
eqcomp00.cc:16:15: error: no match for 'operator=='
```

```
(operand types are 'int' and 'noeq')
```

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

```
    ~~~~~^~~~~~
```

Вторая попытка: enable if

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

Теперь стало лучше

eqcomp01.cc:27:20: error:

```
no matching function for call to 'check(int, noeq)'  
  check (1, noeq(1));
```

eqcomp01.cc:18:6: note: candidate:

```
template<class T, class U, class> bool check(T&&, U&&)  
bool check (T &&lhs, U &&rhs);  
    ^
```

eqcomp01.cc:18:6: note:

```
template argument deduction/substitution failed
```

Проблемы с enable if

- Усложненный синтаксис (тяжело читать и поддерживать такой код)
- Введение дополнительного шаблонного аргумента (или сложное изменение существующего)
- Все ещё не слишком очевидная диагностика

Концепты В С++

- ♦ Контроль полиморфизма
- ➔ **Простые ограничения**
- ♦ Сложные ограничения
- ♦ Простые концепты
- ♦ Вариабельные концепты
- ♦ Частичное упорядочение
- ♦ Критика концептов

Requires: явные требования

```
template <typename T, typename U> bool
```

```
requires
```

```
is_equality_comparable<T, U>::value
```

```
check_eq (T &&lhs, U &&rhs) {
```

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

```
}
```

Теперь стало ещё лучше

```
eqcomp01s.cc: In function 'int main()':
```

```
eqcomp01s.cc:27:20: error: cannot call function
```

```
    'bool check(T&&, U&&) [with T = int; U = noeq]'
```

```
    check (1, noeq(1));
```

```
        ^
```

```
eqcomp01s.cc:18:6: note:    constraints not satisfied
```

```
    bool check (T &&lhs, U &&rhs);
```

```
    ^~~~~
```

```
eqcomp01s.cc:18:6: note:
```

```
    'is_equality_comparable<T, U>::value'
```

```
    evaluated to false
```

Пример: перегрузка конструкторов

```
struct Foo {  
    enum {int_like, float_like} type_;  
    template <typename Int,  
              typename = std::enable_if_t<  
                  std::is_integral<Int>::value> >  
    Foo (Int x) : type_(int_like) {  
        std::cout << "int like: " << x << std::endl;  
    }  
    template <typename Float,  
              typename = std::enable_if_t<  
                  std::is_floating_point<Float>::value> >  
    Foo (Float x) : type_(float_like) {  
        std::cout << "float like: " << x << std::endl;  
    }  
};
```

Проблемы: такой код не работает

```
int main () {  
    Foo (1);  
    Foo (5.0);  
    return 0;  
}
```

ctorex01.cc:13:3: error:

```
    'template<class Float, class> Foo::Foo(Float)'  
    Foo (Float x) : type_(float_like)  
    ^~~
```

ctorex01.cc:9:3: error: with

```
    'template<class Int, class> Foo::Foo(Int)'  
    Foo (Int x) : type_(int_like)
```

Перегрузка конструкторов - 2

```
template <int> struct dummy { dummy(int) {} };

struct Foo {
// .....
    template <typename Int, typename = std::enable_if_t<
        std::is_integral<Int>::value> >
    Foo (Int x, dummy<0> = 0);
// .....
    template <typename Float, typename = std::enable_if_t<
        std::is_floating_point<Float>::value> >
    Foo (Float x, dummy<1> = 0);
};
```

Перегрузка конструкторов - 3

```
struct Foo {  
    enum {int_like, float_like} type_;  
  
    template <typename Int>  
    requires std::is_integral<Int>::value  
    Foo (Int x);  
  
    template <typename Float>  
    requires std::is_floating_point<Float>::value  
    Foo (Float x);  
};
```

Чуть больше о простых ограничениях

Позволяют проверять всё, что можно вычислить на этапе компиляции, включая:

- ♦ Вызовы constexpr функций

```
template <typename T, typename U>  
requires (somepred<T>() == 14) ||  
         (somepred<U>() == 42)  
bool check (T &&lhs, U &&rhs);
```

- ♦ Вычисления sizeof

```
void f () requires sizeof(int) == 4
```

Недостатки простых ограничений

- Требуют сложного кода для вычислений времени компиляции (см. определение `is_equality_comparable`)
- Шаблонные ограничения в свою очередь могут содержать неявные контракты
- Сложно проверять корректность выведенных типов для выражений

Концепты В C++

- ♦ Контроль полиморфизма
- ♦ Простые ограничения
- ➔ **Сложные ограничения**
- ♦ Простые концепты
- ♦ Вариабельные концепты
- ♦ Частичное упорядочение
- ♦ Критика концептов

Возвращение к check_eq

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

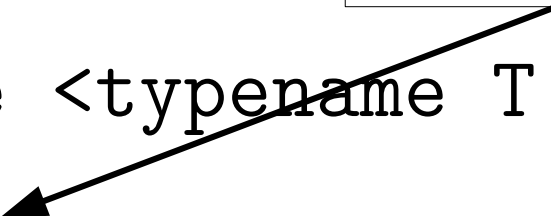
Сложные ограничения

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

Сложные ограничения

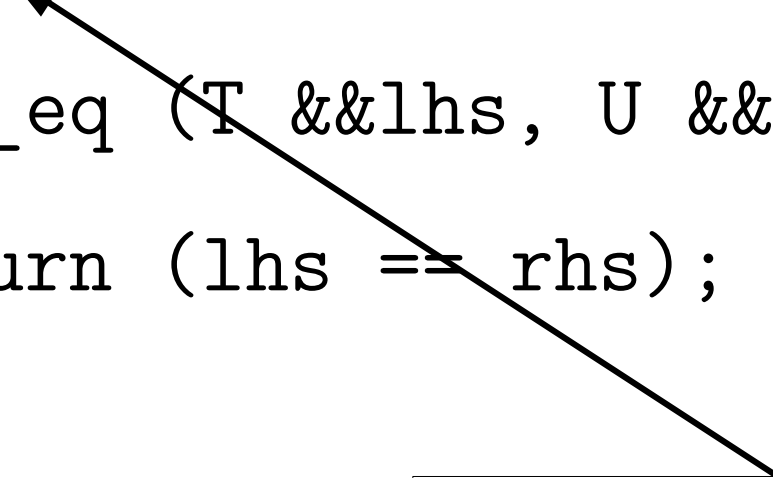
requires clause

```
template <typename T, typename U> bool  
requires
```



```
requires(T t, U u) { t == u; }
```

```
check_eq (T &&lhs, U &&rhs) {  
    return (lhs == rhs);  
}
```



requires expression

Различия между простыми и сложными ограничениями

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

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

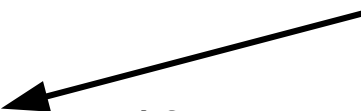
template <typename T>
requires requires (T t) {
    somepred<T>() == 42;
}
bool bar (T&& lhs, U&& rhs);
```

Различия между простыми и сложными ограничениями

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

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


Вычисляется
(проверка не
пройдет)



```
template <typename T>
requires requires (T t) {
    somepred<T>() == 42;
}

bool bar (T&& lhs, U&& rhs);
```

Проверяются типы
(проверка пройдет)



Сложные атомарные ограничения

- Базовые

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

- Для типов

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

- Составные

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

Комбинации сложных ограничений

```
template <typename T>
requires requires(T x) {
    { *x } -> typename T::inner;
} &&
requires() {
    typename T::inner;
} ||
requires (T a, T b) {
    a + b;
}

T test_complex (T, T);
```


Вопрос на понимание

```
struct HasInner { using inner = int;};
```

```
struct HasDeref { using inner = int;  
                  inner operator*(); };
```

```
struct HasPlus { using inner = int;  
                 inner operator*();  
                 void operator+(HasPlus x); };
```

(1) test_complex (HasInner{}, HasInner{});

(2) test_complex (HasDeref{}, HasDeref{});

(3) test_complex (HasPlus{}, HasPlus{});

Вопрос на понимание

```
struct HasInner { using inner = int;};
```

```
struct HasDeref { using inner = int;  
                  inner operator*(); };
```

```
struct HasPlus { using inner = int;  
                 inner operator*();  
                 void operator+(HasPlus x); };
```

(1) `test_complex (HasInner{}, HasInner{});` *// fail*

(2) `test_complex (HasDeref{}, HasDeref{});` *// ok*

(3) `test_complex (HasPlus{}, HasPlus{});` *// ok*

Недостатки сложных ограничений

- Сложный синтаксис
- Усложняют объявления функций
- Каждый раз надо писать заново
- Сложно переиспользовать

Концепты В C++

- ♦ Контроль полиморфизма
- ♦ Простые ограничения
- ♦ Сложные ограничения
- ➔ **Простые концепты**
- ♦ Вариабельные концепты
- ♦ Частичное упорядочение
- ♦ Критика концептов

Неявные требования к FI Seq

```
Последовательность ::= {  
T::element_type  
size_t size()  
bool empty()  
T::element_type back()  
push_back(T::element_type)  
}
```

Явные требования к FI Seq

```
template <typename T>
concept bool Sequence() {
    return
        requires { typename T::element_type; } &&
        requires (T t, typename T::element_type x) {
            { t.size() } -> size_t;
            { t.empty() } -> bool;
            { t.back() } -> typename T::element_type;
            { t.push_back(x) }
        };
}
```

Использование концептов

(1) Базовый синтаксис

```
template <typename T>  
requires Sequence<T>  
void fill_with_random (T &x, int n);
```

(2) Упрощенный синтаксис

```
template <Sequence T>  
void fill_with_random (C &x, int n);
```

(3) Вывод типов

```
void fill_with_random (Sequence &x, int n);
```

Лирическое отступление - 1: ВЫВОД ТИПОВ аргументами

(1) Полная форма функций

→ `template<typename T>
void foo (T x);`

→ `template<typename T, typename U, typename V>
void bar(T (U::*)(V));`

(2) Упрощенный синтаксис в C++17

→ `void foo (auto x);`

→ `void bar(auto (auto::*)(auto));`

Лирическое отступление - 2: концепты для лямбда-функций

(1) Обобщенная лямбда

```
→ find_if(v, [str](const auto& x)  
           { return str == x; }));
```

(2) Ограниченная обобщенная лямбда

```
→ find_if(v, [str](const String& x)  
           { return str == x; }));
```

Ещё пример концепта

```
template <typename It>
concept bool InputIterator () {
    return
        requires(const It iconst, const It jconst, It i) {
            typename std::iterator_traits<It>::reference;
            typename std::iterator_traits<It>::value_type;
            { iconst == jconst } -> bool;
            { iconst != jconst } -> bool;
            { *i } -> typename std::iterator_traits<It>::reference;
            { ++i } -> It&;
            { *i++ } -> typename std::iterator_traits<It>::value_type;
        };
};
```

Требования к концептам

- ✓ Всегда возвращают bool
- ✓ Не принимают аргументов
- ✓ Состоят из одного return
- ✓ Не могут быть отдельно объявлены
- ✓ Не могут быть членами или друзьями
- ✓ Запрещена рекурсия

Требования к концептам

- ✓ Всегда возвращают bool
- ✓ Не принимают аргументов
- ✓ Состоят из одного return
- ✓ Не могут быть отдельно объявлены
- ✓ Не могут быть членами или друзьями
- ✓ Запрещена рекурсия (увы, мы все так надеялись на concept MP)

Ограничения ограничений

- × Концепты не проверяют семантику

```
t.empty() == (t.size() == 0) // ←----- NOPE
```

- × Концепты не проверяют реализацию

```
template<Range R, typename T> bool  
requires RangeEqComparable<R, T>()  
in (R const& range, T const& value) {  
    for (auto const& x : range)  
        if (x != value) // ←----- OOPS  
            return false;  
    return true;  
}
```

Концепты в терминах концептов

```
template <typename T, typename U>
concept bool Weak_equality_comparable() {
    return requires(T t, U u) {
        {t == u} -> bool;
    };
}
```

```
template <typename T, typename U = T>
concept bool Equality_comparable() {
    return Weak_equality_comparable<T, U>() &&
        Weak_equality_comparable<U, T>();
}
```

Введение шаблонов

(1) Упрощенный синтаксис (один аргумент концепта)

- `template <Equality_comparable T> bool check(T, T);`
- `template <Equality_comparable T,
Equality_comparable U> bool check(T, U);`

(2) Вывод типов (один аргумент, два раза одинаковый)

```
bool check(Equality_comparable, Equality_comparable);
```

(3) Введение шаблона (два аргумента)

```
Equality_comparable{T, U} bool check(T, U);
```

Функции и переменные концепты

(1) Функция-концепт

```
template <typename C>
concept bool isInt() {
    return std::is_integral<C>::value;
}
```

(2) Переменная-концепт

```
template <typename C>
concept bool Int = std::is_integral<C>::value;
```

Переменную можно использовать для вывода типа

```
Int x = f(x); // auto x = f(x);
```


Перегрузка конструкторов - 4

```
template <typename T>
concept bool Int = is_integral_v<T>;

template <typename T>
concept bool Float = is_floating_point_v<T>;

struct Foo {
    enum {int_like, float_like} type_;
    Foo (Int x);
    Foo (Float x);
};
```

Перегрузка классов

```
template <typename T>  
class complex;
```

```
template <Float T>  
class complex { /* complex numbers */ };
```

```
template <Int T>  
class complex { /* gaussian integers */ };
```

Концепты В С++

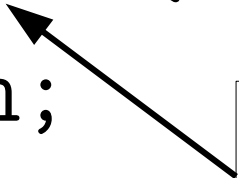
- ♦ Контроль полиморфизма
- ♦ Простые ограничения
- ♦ Сложные ограничения
- ♦ Простые концепты
- ➔ **Вариабельные концепты**
- ♦ Частичное упорядочение
- ♦ Критика концептов

Вопрос: чем плох код на слайде?

```
template <Sequence S, typename P>
int count (S const& seq, P pred) {
    int n = 0;
    for (auto const &x : seq)
        if (pred(x)) ++n;
    return n;
}
```

Ответ: не ограниченный P

```
template <Sequence S, typename P>
int count (S const& seq, P pred) {
    int n = 0;
    for (auto const &x : seq)
        if (pred(x)) ++n;
    return n;
}
```



Тут произойдет
что угодно!

Идея для ограничения

```
template <Sequence S, typename P>
requires
Predicate<P, typename S::element_type>()
int count (S const& seq, P pred) {
    int n = 0;
    for (auto const &x : seq)
        if (pred(x)) ++n;
    return n;
}
```

Идея для ограничения

Предикат

Аргументы

```
template <Sequence S, typename P>
requires
Predicate<P, typename S::element_type>()
int count (S const& seq, P pred) {
    int n = 0;
    for (auto const &x : seq)
        if (pred(x)) ++n;
    return n;
}
```

Концепт для предиката

```
template <typename P, typename ... Args>
concept bool Predicate() {
    return
    requires (P pred, Args ... args) {
        { pred(args...) } -> bool;
    };
}
```


Введение переменного шаблона

```
template<typename T, int N, typename... Xs>  
concept bool C1 = true;
```

Допустим любой из вариантов:

```
1. C1{A, B, ...C} struct S1;
```

```
2. template<typename A, int B, typename... C>  
    requires C1<A, B, C...>  
    struct S1;
```

Концепты В С++

- ♦ Контроль полиморфизма
- ♦ Простые ограничения
- ♦ Сложные ограничения
- ♦ Простые концепты
- ♦ Вариабельные концепты
- **Частичное упорядочение**
- ♦ Критика концептов

Сравнение концептов: нормализация

- Полная подстановка подвыражений
- Формирование логических цепочек через `&&` и `||` из атомарных концептов

```
template<typename T>
concept bool Subsumed() {
    return requires () { typename T::type1; };
}

template<typename T>
concept bool Subsuming() {
    return Subsumed<T>()
        && requires () { typename T::type2; };
}
```

Сравнение концептов: нормализация

```
template<typename T>
concept bool Subsumed() {
    return
        requires () { typename T::type1; };
}

template<typename T>
concept bool Subsuming() {
    return
        requires () { typename T::type1; } &&
        requires () { typename T::type2; };
}
```

Сравнение концептов: поглощение

- Как определить, что P поглощает Q ?
(сокращенно $P \Rightarrow Q$)
- Пусть $P = P_1 \ || \ P_2 \ || \ \dots$ (ДНФ)
 $Q = Q_1 \ \&\& \ Q_2 \ \&\& \ \dots$ (КНФ)
- Тогда $P_i \Rightarrow Q_j$ если
 $\text{forany } k, \text{ forall } n \mid P_{ik} \Rightarrow Q_{jn}$
- И далее $P \Rightarrow Q$ если
 $\text{forany } k, \text{ forall } n \mid P_k \Rightarrow Q_n$

Пример: поглощающие концепты

```
template<typename T>  
struct TM;
```

```
template<Subsumed T>  
struct TM<T> {  
    TM() {std::cout << "Subsumed!\n";}  
};
```

```
template<Subsuming T>  
struct TM<T> {  
    TM() {std::cout << "Subsuming!\n";}  
};
```

Вопрос: что на экране?

```
struct M {  
    using type1 = int;  
    using type2 = int;  
};
```

```
struct L {  
    using type1 = int;  
}
```

```
TM<M> X{};
```

```
TM<L> Y{};
```

Вопрос: что на экране?

```
struct M {  
    using type1 = int;  
    using type2 = int;  
};
```

```
struct L {  
    using type1 = int;  
}
```

```
TM<M> X{}; // Subsuming
```

```
TM<L> Y{}; // Subsumed
```


Концепты В С++

- ♦ Контроль полиморфизма
- ♦ Простые ограничения
- ♦ Сложные ограничения
- ♦ Простые концепты
- ♦ Вариабельные концепты
- ♦ Частичное упорядочение
- ➔ **Критика концептов**

Критика концептов: консервативность комитета

- Предложение опубликовано менее года назад
- Спецификация реализована только в одном компиляторе и реализована тем же человеком, который писал документ спецификации
- Сейчас очень мало кода, который полагался бы на концепты или хоть как-то их использовал.

Критика концептов: технические возражения

- Синтаксические неоднозначности
`void f(X x); // function or template?`
- Дополнительное значение для `{}`
- Уже утверждена возможность делать так:
`template<auto V>`
`constexpr auto v = V*2;`
И она будет конфликтовать с концептами
- Ошибки проверки концептов часто проявляются как ошибки разрешения перегрузки

Критика концептов: они не то, о чем все мечтали

```
concept Comparable<typename T> {  
    // syntax of equality  
    requires constraint Equal<T>;  
    // semantics of equivalence  
    requires axiom Equivalence_relation<Equal<T>, T>;  
    // if  $x == y$  then for any Predicate  $p$ ,  $p(x) == p(y)$   
    template<Predicate P>  
    axiom Equality(T x, T y, P p) {  
         $x == y \Rightarrow p(x) == p(y)$ ;  
    }  
    // inequality is the negation of equality  
    axiom Inequality(T x, T y) {  
         $(x != y) == !(x == y)$ ;  
    }  
}
```

Для дальнейшего исследования

Paper by Stroustrup'2003

<http://www.stroustrup.com/N1522-concept-criteria.pdf>

Paper by Sutton and Stroustrup'2011

<http://www.stroustrup.com/sle2011-concepts.pdf>

Paper by Sutton, Stroustrup and Reis'2013

<http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3701.pdf>

Concepts Lite TS'2014

<http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n3889.pdf>

Talks:

- Sutton, CppCon'14 (1): <https://www.youtube.com/watch?v=qwXq5MqY2ZA>
- Sutton, CppCon'14 (2): <https://www.youtube.com/watch?v=NZeTAnW5LL0>
- Sutton, C++Now'15: https://www.youtube.com/watch?v=_rBhX-FJCdg
- Фокин, C++ Siberia'15: <https://www.youtube.com/watch?v=482JCDghZ8s>
- Niebler, C++Siberia'15: <https://www.youtube.com/watch?v=g0KHcQad7xE>
- Orr, ACCU'16: <https://www.youtube.com/watch?v=S1Z-RbygAlw>
- Github page: <http://cplusplus.github.io/concepts-ts/>