Concepts Lite in Practice

Roger Orr

OR/2 Limited

The Concepts TS is published and implemented in gcc

What does *using* concepts look like in practice: do we get what we hoped for?

– ACCU 2016 –

How, and why, did we get here?

- C++ is a rich language and supports polymorphic behaviour at both run time and compile time
  - Run-time: class hierarchy and O-O practices
  - Compile time: templates
- One major difference between these two is that the first one is tightly constrained by the inheritance hierarchy of the objects involved but the second one can be applied to unrelated types.

Run-time polymorphism

- Run-time polymorphism is a key component in object oriented design. The function signature to use is decided at compile time based on the static type of the target object; the implementation used is based on the run-time type of the object
- This has been a fundamental part of C++ for a very long time (about as long as it has been called C++) and has been essentially unchanged for over 30 years
- **Inheritance** means “if it compiles it runs”
- Nothing to see here … move along please

Compile time polymorphism

- This is also known as “generic programming” and it too has been in the language for a very long time
- A *template* is written which is used to generate code at **compile** time
- Simple uses include type-safe containers; when coupled with non-type template arguments, overloading, and tag-dispatching the result is very expressive

Compile time polymorphism

- This is also known as "*%$!!** programming" and it too has been in the language for a very long time
- A *template* is written which is used to generate code at **compile** time
- Simple uses include type-safe containers; when coupled with non-type template arguments, overloading, and tag-dispatching the result is very expressive
- … but can also be very hard to debug

Template troubles

- Templated code is fragile – whether the code is **valid** depends on the template arguments provided by the user when the template is instantiated
- The writer of a template has (usually) tested at least one instantiation of their code*, but they may have, possibly unconscious, assumptions about the template arguments

- *If there is **no** valid instantiation the program is ill-formed, but a diagnostic is not required - it is a “hard problem”

Template troubles

- Library writers rely on documenting their assumptions about template parameters
  - It is hard to get this right
  - Compilers don't read documentation
  - Diagnosis of failure is painful (for the user)
- Enter concepts!
- “Concepts introduce a type system for templates that makes templates easier to use and easier to write.” - N2081, Sep 2006

How, and why, did we get here?

- One of the earliest papers on Concepts was Bjarne's paper “Concept Checking - A more abstract complement to type checking” from Oct 2003
- The fundamental problem that makes it hard in C++ is that templates are not constrained, so by default *any* possible type may be used to instantiate a template
- Type checking occurs when some part of the **instantiation** fails, not the **signature**

How, and why, did we get here?

```
template <typename T>
T sum(T a, T b);
```

The template declaration tells you nothing about what the characteristics are for appropriate types to use with this template

There are two different levels of characteristics of course: **syntax** and **semantics**

Syntax answers the question “will it compile?”

Semantics answers the question “what does it *do*?”

How, and why, did we get here?

```
template <typename T>
T sum(T a, T b);

class X { /*...*/ };

void test(X x1, X x2)
{
   auto val = sum(x1, x2); // Valid?
   // ...
}
```

We can't answer the question about validity without knowing the *implementation* of the `sum` template – this breaks encapsulation and also makes it hard to discover the constraints on the template as the implementation contains more detail than we need

For a standard library, with *different* implementations, answering the question is even harder

How, and why, did we get here?

- Currently one way forward is to document the restrictions, for example the C++ standard library documents a lot of syntactic, and semantic, requirements, eg:

Table 18 — LessThanComparable requirements [lessthancomparable]

Expression	Return Type	Requirement
a < b	convertible to bool	< is a strict weak ordering relation* * that is: !(x < x) (a < b) && (b < c) => (a < c)

How, and why, did we get here?

- As documentation *cannot* be read by the compiler … and *may* not be read by the programmer, the result is typically horrendous compiler errors – e.g.

```
class X{};
std::set<X> x;
x.insert(X{});
```

- I get 50 – 100 lines of error text from this!
- I'm missing the '<' operator for X* so it does not satisfy LessThanComparable

  (* with *current* versions of C++: this may change)

Let's build up from a trivial example

A type-specific function declaration:

```
bool check(int lhs, int rhs);
```

The function takes two `int` values and the validity of the calling code is decided without any need to see the implementation of the function.

```
int main()
{
  int i = 1;
  int j = 2;
  return check(i, j); // Valid!
}
```

A possible function definition could be:

```
bool check(int lhs, int rhs) { return lhs == rhs; }
```

What about type conversion?

```
int main()
{
  double e = 2.71828;
  double pi = 3.14159;
  return check(e, pi);
}
```

The call converts the values to `int` and operates on those. (You *may* get a compiler warning.)

Easily fixed ...

Generalising the function

Write **two** function declarations:

```
bool check(int lhs, int rhs);
bool check(double lhs, double rhs);
```

Two function definitions:

```
bool check(int lhs, int rhs) { return lhs == rhs; }
bool check(double lhs, double rhs) { return lhs == rhs; }
```

We have generalised our function to a *predefined* set of types but *duplicated* the implementation.

```
{
  bool b1 = check(1, 2); // works with int
  bool b2 = check(e, pi); // works with double
}
```

~~General~~ Templatising the function

One function declaration:

```
template <typename T>
bool check(T lhs, T rhs);
```

One function definition:

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

We have produced a template for a **set** of functions for an *unbounded* number of types.

```
{
  bool b1 = check(1, 2); // works with int
  bool b2 = check(e, pi); // works with double
}
```

Aside: value, const ref, or ref ref?

This:

```
template <typename T>
bool check(T lhs, T rhs);
```

Or this:

```
template <typename T>
bool check(T const & lhs, T const & rhs);
```

Or this:

```
template <typename T>
bool check(T && lhs, T && rhs);
```

The best answer may depend on the type, e.g. is T a built-in type or a user-defined one?
We'll mention this briefly again, later on.

Further extending the template

We can further generalise to take two *different* types:

```
template <typename T, typename U>
bool check(T lhs, U rhs);
```

Now, without any further changes to the implementation, the function can deal with *any* two types for which equality is defined

As we mentioned earlier though, compilation errors are reported during the *instantiation* of the function template

```
Basic_Template_Failure.cpp: In instantiation of
 'bool check(T&&, U&&) [with T = int&; U = int*]':
... comparison between pointer and integer [-fpermissive]
 bool check(T && lhs, U && rhs) { return lhs == rhs; }
                                          ~~~~^~~~~~
```

Further extending the template

We can further generalise to take two *different* types:

```
template <typename T, typename U>
bool check(T lhs, U rhs);
```

Now, without any further changes to the implementation, the function can deal with *any* two types for which equality is defined

As we mentioned earlier though, compilation errors are reported during the *instantiation* of the function template

We could do better ...

Constraining the template

- When a function template is called the compiler goes through two main stages
  1. Overload resolution – find the best possible **declaration** match for the arguments
  2. Then **instantiate** the best candidate
- In stage 1, if substituting in a template argument would result in ill-formed code, the compilation of the program does *not* fail. Instead **that specialization** is removed from the overload set

“SFINAE”

The technique of using substitution failure to remove unwanted Overloads from the overload set is known as SFINAE.

This stands for “**S**ubstitution **F**ailure **I**s **N**ot **A**n **E**rror”

C++ programmers know how to pick a snappy acronym...

This is the basis for `enable_if` and other similar techniques

Near-trivial SFINAE example

```
template <typename T>
void f(T t, typename T::value_type u);

template <typename T>
void f(T t, T u);

int main()
{
   f(1, 2); //  first f() non-viable when substituting int

   std::vector<int> v;
   f(v, 2); //  second f() not a match as types differ
}
```

Constraining the template

```
#include "magic.h" // (we'll come to this next)

template <typename T, typename U,
  typename = std::enable_if_t<
    is_equality_comparable<T, U>::value>>
bool check(T && lhs, U && rhs);
```

This has the desired effect of removing the function from the overload set if the two types are not equality comparable:

```
Basic_Enable_Failure.cpp:40:21: error: no matching function for call to
'check(int&, int*)'
   return check(i, &j);
                    ^
Basic_Enable_Failure.cpp:34:6: note: candidate: template<class T, class U,
class> bool check(T&&, U&&)
 bool check(T && lhs, U && rhs) { return lhs == rhs; }
      ^~~~~
Basic_Enable_Failure.cpp:34:6: note:   template argument
deduction/substitution failed:
```

"magic.h"

```
#include <type_traits>

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

template<typename T, typename U>
struct is_equality_comparable<T,U,
    typename std::enable_if<
        true,
        decltype(std::declval<T&>() == std::declval<U&>(), (void)0)
        >::type
    > : std::true_type
{
};
```

Not for the fainthearted: hard to write, hard to read, and slow to compile

"magic.h"

Apart from the complexity, there is a problem with the disjoint between the **check** and the **expression** being checked

```
decltype(std::declval<T&>() == std::declval<U&>())
```

The expression we want to detect is "`lhs == rhs`" but we have to write a more complicated expression, subject to the various rules for SFINAE, which acts as a proxy for the check we actually wanted.

For some expressions it can be rather challenging to find an equivalent SFINAE check, and sometimes there can be subtle differences.

Concepts to the rescue

- We have seen some of what can be done to constrain the parameters used to instantiate templates within the existing grammar rules.
- How do solutions using concepts compare?
- There are two main use cases:
  - Writing function templates
  - Using them
- While both are important, templates are typically **used** more often than they are **written**.

Using 'requires'

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

- The **declaration** shows the constraint
- Calling `check()` with `int` and `char*` gives

```
Basic_Requires.cpp:19:6: note:   constraints not satisfied
 bool check(T && lhs, U && rhs) { return lhs == rhs; }
```

- The syntax of the requirement matches the code used in the definition
- Could we do better?

Using a named concept

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

- The concept is **named** and can be **shared** by multiple template declarations
- Naming also allows us to express some of the **semantic** constraints on the type.

Using a named concept

```
template <typename T, typename U>
requires Equality_comparable<T, U>()
bool check(T && lhs, U && rhs);
```

- Use the concept, with appropriate template parameters, in the **requires** clause
- Calling `check()` with `int` and `char *` gives

```
Basic_Concept.cpp:33:29: error: cannot call function
'bool check(T&&, U&&) [with T = int&; U = char*&]'
   return check(argc, argv[0]);

Basic_Concept.cpp:29:6: note:   concept
  'Equality_comparable<int&, char*&>()' was not satisfied
```

Using a named variable concept

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

template <typename T, typename U>
requires Equality_comparable<T, U>
bool check(T && lhs, U && rhs);
```

- Similar syntax to function template form
- One restriction is that you cannot overload

```
Basic_Variable_Concept.cpp:29:6: note:   concept
  'Equality_comparable<int&, char*&>' was not satisfied
```

Named concepts

- The **Equality_comparable** concept is asymmetric as it only checks `t == u`.
- This matches our only use case but is not suitable for use as a *general* concept.
- The ranges TS defines something like this:

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

Named concepts

- If we use this we have the *reverse* problem: you can argue our template is now *over*-constrained

```
struct Test{
  bool operator==(Test);
};

bool foo(Test v, Test v2)
{
  return check(v, v2);
}

Basic_Concept_Failure.cpp: In function 'int main()'
Basic_Concept_Failure.cpp:39:20: error:
cannot call function 'bool check(T&&, U&&)'
   return check(v, v2);
... concept 'Equality_comparable<Test&, Test&>()'
```

Named concepts

- In order to call our `check` function we have to declare, but not necessarily define, an extra function.

```
struct Test{
  bool operator==(Test);
  bool operator!=(Test);
};

- or --

bool operator!=(Test, Test);

bool foo(Test v, Test v2)
{
  return check(v, v2); // Ok
}
```

Named concepts

- If the arguments to operator== differ in type we have a little more work to do.

```
struct Test{
  bool operator==(int);
};

bool foo(Test v)
{
  return check(v, 0);
}
```

```
Basic_Concept_Failure2.cpp: In function 'int main()'
Basic_Concept_Failure2.cpp:39:20: error:
cannot call function 'bool check(T&&, U&&)'
   return check(v, 0);
... concept 'Equality_comparable<Test&, int>()'
```

Named concepts

- In order to call our `check` function we now have to declare **three** extra functions.

```
struct Test{
  bool operator==(int);
  bool operator!=(int);
};

// These two cannot be defined in-class
bool operator==(int, Test);
bool operator!=(int, Test);

bool foo(Test v)
{
  return check(v, 0); // Ok
}
```

Named concepts

- In order to call our `check` function we now have to declare **three** extra functions.
- This is arguably a good idea as it encourages/enforces more consistent operator declarations for types.
- It can increase the work needed to adapt a class to comply with a concept.
- The 'requires' use is less affected by this issue as each function template has its own clause – but it raises the complexity of **each** function declaration, and breaks DRY.

Concepts – first reflection

- Simpler to write than the alternatives
- Error messages seem to be clearer
- Danger of under or over constraining
- The syntax is slightly awkward

```
template <typename T> requires requires(T t) {…}
```

- “requires requires” - can we be serious?

```
concept bool x = requires(...) {}
```

- The type **must** be specified and **must** be `bool`
- Function and variable form seem overkill

Constraints and Interfaces

- One main difference between C++ templates and C#/Java generics is that the latter operate in terms of **interfaces**.
- This more closely ties the instantiating classes together and also prevents compile time optimisations as the net result is a type-safe call to a single shared implementation*
- Constraints define an interface expressed in terms of syntax, rather than inheritance.

(* ignoring special cases for primitive types)

Constraints and Interfaces

- The C#/Java generics model is “opt-in”: you need to mark the class to indicate that it supports the interface; simply having methods with matching signatures is not enough.
- If I add a new method to an interface the compiler forces me to implement the new method.
- If I add a new requirement to a constraint then classes not satisfying this *silently* fail to satisfy the constraint – the program fails only if this removes **all** viable overloads.

Simplifying complex overloads

- While `enable_if` and similar techniques do provide ways to constrain function templates it can get quite complicated.
- One recurring problem is the ambiguity of template argument types solely constrained by `enable_if`
- Perhaps an example will help explain this problem (or then again, we might be past that stage)

Simplifying complex overloads

- Consider this attempt to use `enable_if`

```
struct V {
    enum { int_t, float_t } m_type;

    // Constructor from 'Int' values
    template <typename Int,
              typename = std::enable_if_t<
                  std::is_integral<Int>::value>>
    V(Int) : m_type(int_t) { /* … */ }

    // Constructor from 'Float' values
    template <typename Float,
              typename = std::enable_if_t<
                  std::is_floating_point<Float>::value>>
    V(Float) : m_type(float_t) { /* ... */ }
};
```

Simplifying complex overloads

- Unfortunately this example does not compile - we have two overloads of the same constructor with the same type (that of the first template argument)

```
ambiguity_with_enable_if.cpp:25:5: error:
'template<class Float, class> V::V(Float)' cannot
be overloaded

     V(Float) : m_type(float_t) {}

     ^

ambiguity_with_enable_if.cpp:20:5: error: with
'template<class Int, class> V::V(Int)'

     V(Int) : m_type(int_t) {}

     ^
```

Simplifying complex overloads

- The compiler never gets to try overload resolution as declaring the two overloads is a syntax error.
- If we add a *second* argument we can resolve the ambiguity and allow overload resolution to take place; SFINAE will then remove the case(s) we do not want.
- We can give the extra argument a default value to avoid the caller needing to be concerned with it.

Simplifying complex overloads

```
enum { int_t, float_t } m_type;

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

// Constructor from 'Int' values
template <typename Int,
          typename = std::enable_if_t<
              std::is_integral<Int>::value>
>
V(Int, dummy<0> = 0) : m_type(int_t) { /* ...  */}

// Constructor from 'Float' values
template <typename Float,
          typename = std::enable_if_t<
              std::is_floating_point<Float>::value>
>
V(Float, dummy<1> = 0) : m_type(float_t) { /* ... */ }
```

Simplifying complex overloads

- The two types `dummy<0>` and `dummy<1>` are different types. Now the two constructors have *different* argument lists and so both functions participate in overload resolution.
- This addition of dummy arguments that take no other part in the function call adds needless complexity.
- Surely there must be a better way?

Concepts simplifying overloads

- With concepts the **constrained** template arguments are of *different* types and both functions participate in overload resolution without any need for an extra argument.
- This seems to me a clear demonstration of the benefit of concepts as part of the **language**.

Concepts simplifying overloads

- The concept solution is straightforward

```
struct T {
    enum { int_t, float_t } m_type;

    // Constructor from 'Int' values
    template <typename Int>
        requires std::is_integral<Int>::value
    V(Int) : m_type(int_t) { /* ... */ }

    // Constructor from 'Float' values
    template <typename Float>
        requires std::is_floating_point<Float>::value
    V(Float) : m_type(float_t) { /* ... */ }
};
```

Concepts simplifying overloads

- Or, using proposed C++17 syntax

```
struct T {
    enum { int_t, float_t } m_type;

    // Constructor from 'Int' values
    template <typename Int>
        requires std::is_integral_v<Int>
    V(Int) : m_type(int_t) { /* ... */ }

    // Constructor from 'Float' values
    template <typename Float>
        requires std::is_floating_point_v<Float>
    V(Float) : m_type(float_t) { /* ... */ }
};
```

Concepts simplifying overloads

- This also makes it much easier to overload between pass-by-value and pass-by-reference using concepts than when using `enable_if` style of programming
- For example, there might be one overload using pass-by-value that constrains the argument with `std::is_primitive` and another one using pass-by-reference that constrains the argument with `std::is_object`

Concept introducer syntax

- The concepts TS supports a 'concept introducer' syntax and an abbreviated syntax which I have not yet demonstrated, so we can modify the example to do so.
- A lot of the complexity in the wording of the Concepts TS is to ensure that the specification of a constrained function using this additional syntax is equivalent to the `requires` form

Concept introducer syntax

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

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

struct V {
    enum { int_t, float_t } m_type;

    // Constructor from 'Int' values
    template <Int T>
    V(T) : m_type(int_t) { /* ... */ }

    // Constructor from 'Float' values
    template <Float F>
    V(F) : m_type(float_t) { /* ... */ }
};
```

Pure syntactic sugar

```
template <Int T>
bool check(T value);
```

is equivalent to:

```
template <typename T>
requires Int<T>
bool check(T value);
```

The translation between the introducer syntax and that using `requires` is fairly simple and unlikely to cause confusion. Note though that, as they are equivalent, both forms can occur in the same translation unit – and that the 'T' could be different.

It does save characters: in this case 37 vs 58.

Abbreviated syntax

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

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

struct V {
    enum { int_t, float_t } m_type;

    // Constructor from 'Int' values
    V(Int) : m_type(int_t) { /* ... */ }

    // Constructor from 'Float' values
    V(Float) : m_type(float_t) { /* ... */ }
};
```

Abbreviated syntax

- This syntax allows the declaration of templates without using <>
- Is this a good thing? There seem to be three main answers:
  - Yes
  - No
  - Maybe

Is it a template or not?

```
bool check(Int value);

void test(Float f)
{
    if (check(f))
    {
        // do something
    }
}
```

- Will `check` get called?
  - If `Int` is a type we look for conversions
- Where can we define `check`?
  - If `Int` is not a type we need the definition

What does it all mean, anyway?

```
bool check(Int value);
```

This is equivalent to:

```
template <Int A>
bool check(A value);
```

Which is itself equivalent to:

```
template <typename C> requires Int<C>
bool check(C value);
```

And all of these are *functionally* equivalent to:

```
template <typename D>
requires std::is_integral_v<D>
bool check(D value);
```

The abbreviated form is less xprsiv

```
bool check(Int value, Int other);
```

This is equivalent to:

```
template <Int T>
bool check(T value, T other);
```

Note that the two variables will always have the same type – we cannot* use the short form syntax to replace:

```
template <Int T, Int U>
bool check(T value, U other);
```

(* there were some tentative proposals ...)

Using auto to declare a template

- The concepts TS also allows using auto to introduce an unconstrained template parameter

```
bool check(auto value);

void test(Float f)
{
    if (check(f))
    {
        // do something
    }
}
```

- I see this as less problematic than the constrained case – it mirrors the existing use of auto for declaring a polymorphic lambda

Concept introducer syntaxes

- What do **you** think?
- I am not persuaded that we need **both** forms of the concept introducer.
- I am fairly happy, myself, with using the `requires` form and it does ensure a clear separation between the template arguments and their constraints.
- Mixing concept introducers with `requires` clauses is valid, but may be hard to read.

The Ranges TS experience

- Thanks to Eric Niebler and Casey Carter we have a working paper for the “Ranges” TS.
- This re-specifies iterators and algorithms using concepts rather than documentation.
- The end goal is “STL2.0”
- This is probably at present the biggest single use of concepts anywhere.
- It does not use the abbreviated syntax; earlier versions of the proposal did but the author was requested to remove these uses during review by the library working group.

The Ranges TS: `count()`

- An example algorithm: `count`
- The existing C++ specification is:

```
template<class InputIterator, class T>
  typename iterator_traits<InputIterator>::difference_type
    count(InputIterator first, InputIterator last,
      const T& value);
```

- “If an algorithm’s template parameter is named InputIterator ... the template argument shall satisfy the requirements of an input iterator.”

The Ranges TS: `count()`

- An example algorithm: `count`
- The Ranges TS specification is:

```
template<InputIterator I, Sentinel<I> S, class T>
  requires IndirectCallableRelation<equal_to<>,
    I, const T*>()
  difference_type_t<I>
    count(I first, S last, const T& value);
```

- The highlighted terms are all concepts readable by both the compiler and the user

The Ranges TS: count()

- I lied, for simplicity.
- The full Ranges TS specification is:

```
template<InputIterator I, Sentinel<I> S, class T,
class Proj = identity>
  requires IndirectCallableRelation<equal_to<>,
    projected<I, Proj>, const T*>()
  difference_type_t<I>
    count(I first, S last, const T& value,
      Proj proj = Proj{});
```

- This includes the new 'projection' feature the TS adds to permit on-the-fly data transformations.

The Ranges TS: function form

- Early drafts of the Ranges TS used the variable form for concepts when possible and the function form when not – typically when overloading was required.
- It was changed after review to use the function form only.
  - More consistent using a single style
  - If you need to add overloading to a variable concept, the change to the function form requires changes to the places where the concept is **used**

The Ranges TS: semantics

- Concepts cover the syntax well but there is still a need to describe the **semantics**
- For example there is a section, describing the meaning of “Equality Preservation”, that includes this sentence:
- Expressions declared in a requires-expression in this document are required to be equality preserving, except for those annotated with the comment “not required to be equality preserving.”
- All clear, then?

Constraint failure with overload

- If we add `bool check(...)` to our earlier example (where we were missing a required `operator !=` to satisfy with the constraint.)

```
bool check(...);

template <typename T, typename U>
requires Equality_comparable<T, U>()
bool check(T && lhs, U && rhs);

{
  Test t, u;
  return check(t, u); // Ok: now finds check(...)
}
```

Constraint failure with overload

- Matching `bool check(...)` is a poor match, but the 'better' match was omitted from the overload set.
- We get no warning that we failed to get the overload we (probably) expected.
- A direct consequence of the *implicit* nature of concepts.

Constraint checking

- The concepts TS does not include concepts checking. Is this a good or a bad thing?
- The original concepts design, which was eventually dropped from C++0x, enforced that the **implementation** of the function only used methods that were explicitly allowed by the constraints on the template parameters.
- However, the concepts TS does not *preclude* a future change to add this.

Constraint checking

- The **advantage** is that this ensures that the template instantiation **will** compile for any types that satisfy the constraints.
- The **disadvantage** is that this requires that **every single operation** used in the constrained template is explicitly listed in the constraints.
- Some people see concepts TS as a step towards 'full concepts', others have no desire to see checking added
- Time will tell...

Constraint cost at compile time

- There are several different factors involved in the effect of concepts on compile times
  - The cost of concepts *when replacing existing constraint checking*
  - The cost of *adding* concepts to existing functions, now using documentation
  - The cost of disjunctions:

```
template <typename T>
    requires std::is_integral<T>::value ||
             std::is_floating_point<T>::value
T calculate(T a, T b);
```

Constraint cost at compile time

- I've not measured any of these costs myself.
- Andrew Sutton presented some figures showing double-digit percentage speed up when *replacing* `enable_if` with concepts.
- There is a fix going into gcc to improve the compilation times with disjunctions, but the current wording still seems to leave the potential for a performance problem.
  - Users of concepts will need to take care with use of disjunctions, e.g placing them later

Concepts are still a TS*

- As you may know, at the recent standards meeting in Jacksonville the committee voted against adopting the Concepts TS into the C++17 standard
- This has caused some over-reaction...
- There is a TS and it is included in gcc 6, enabled with `-fconcepts`
- I don't think making it part of C++17 would have had much effect on the level of compiler support becoming available
- A TS (“Technical Specification”) is *optional* and roughly translates to “if you're going to do this, here's how to”

Concepts are still a TS

- What **might** change before adoption into C++?
  - Minor changes to support possible future concept checking
  - Removal of one or more of the shortened syntaxes
  - Coalescing of the function and variable form of constraints in some way
  - Change to disjunction (for performance)
- Making changes to the basic principles of the language feature is extremely unlikely at this stage

Concepts are still a TS

- If you want to read more Tom Honermann wrote a good summary at: honermann.net/blog/?p=3
- (I received the BSI Concepts TS 2016-02-23)
- Conversely, if you want to see what you can do *without* concepts, there are many examples, such as this recent blog post:
- https://akrzemi1.wordpress.com/2016/03/21/concepts-without-concepts/
- Generally, while admiring the cleverness of some of the techniques, I prefer using a language feature designed for this purpose

Some conclusions

- Concepts does deliver
  - Better compiler errors, for users
  - Easier to constrain functions, for writers
  - Constraints express intent in code
- The syntax still has some rough edges and redundancy
- Now is the time to *use* concepts and to provide feedback to the standards body